NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

AY2021/2022 Semester 1

CE3006 - Digital Communications

Topic: Course Project Report

Done by:

Team Members	**Matriculation Number**
Ang Kai Jun	U1921062L
Mervyn Chiong Jia Rong	U1921023K
Yeo Li Ting	U1921466G
Lee Zhe Ren	U1920248G
Elayne Tan Hui Shan	U1921730C

1 Introduction 4
2 Phase 1: Data Generation 5
2.1 Objective 5
2.2 Phase 1 5
2.3 Graphical Analysis 6
2.4 SNRToErrorRate(N, dBSNR) 6
2.5 gaussian(N) 7
2.6 noise(N, dBSNR) 7
2.7 signalAdd(sig1, sig2) 7
2.8 threshold(receivedSig, thresholdValue) 8
2.9 checkBitErrorRate(thresholdOut, data) 8
3 Phase 2: Modulation for communication 9
3.1 OOK 9
3.2 BPSK 10
3.3 Phase 2 11
3.4 stretchData (data, numSample, dataRate, fs) 12
3.5 OOK (dataStream, carrier) 12
3.6 BPSK (dataStream, carrier) 12
3.7 Post-Modulation 13
3.7.1 Code Explanation 13
3.8 demod (receivedSig, carrier) 13
3.9 Graph Analysis 14
3.10 Additional Analysis 15
4 Phase 3: Basic error control coding to improve the performance 17
4.1 Objective 17
4.2 Phase 3 parameter initialisation 17
4.2.1 Code Explanation 17
4.3 Encoding 18
4.3.1 Code Explanation 18
4.4 Phase 3 error code modulation and demodulation 18
4.5 demod_phase3 (receivedSig, encoded_bits, fs, dataRate, carrier) 19
4.5.1 Code Explanation 19
4.6 Graph Plotting 19

4.7 Comparison of Linear block codes 20
4.8 Comparison of Hamming error correction 20
4.9 Comparison of Cyclic error correction 21
5 Additional Analysis - BFSK Exploration 22
5.1 Non-coherent Detection 22
5.2 Bandpass filter specific to frequency 22
5.3 Datastream Plot 23
5.4 BFSK Modulation 23
5.5 Filtered BFSK Signal 24
5.6 Bit Error Rate for Bandpass Filtration 24
5.7 Low and High pass filter Exploration 25
5.8 Demodulation using Square Law 25
5.9 Plot Analysis 26
5.10 Overall Results comparison 26
6 Conclusion 27

1 Introduction

A digital communication system is a system that transmits a digital signal, modulates it at the transmitter and demodulates it at the receiver. This system is commonly used today mainly for its good tolerance to noise and distortion, simple and reliable circuit design, low-cost components, and near-perfect regenerative capabilities.

As shown in Fig 1.1, a digital communication system consists of a transmitter, channel and receiver blocks. The input signals are passed to the transmitter block. In the transmitter block, the source coder takes sampled signals from the input signals and compresses the information to reduce the required bit rate. This is to minimize the amount of bandwidth used per information source, ensuring efficiency and reliability before being passed on to the communication channel. The channel is inherently noisy due to its physical medium property and the received signal at receiver is often corrupted with these distortions. At the receiver blocks, the corresponding source decoder would then decompress the data and reconstruct the original source information.

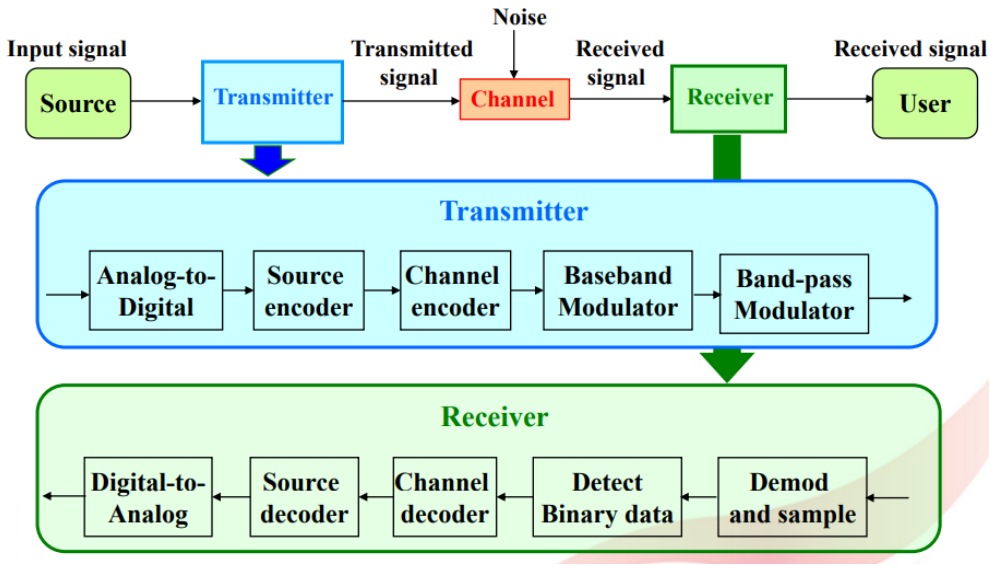

Fig 1.1: Model of Communication System

This project requires us to design and develop a basic digital communication system to study the working principles of each sub-block in transmission and receiver blocks and the effect of noise in communication. Hence, we have broken down the system design into three phases:
Phase 1: Data generation
Phase 2: Modulation for communication
Phase 3: Basic error control coding to improve the performance
to stimulate the various building blocks in Fig 1.1 with the help of MATLAB.

Implementations and results will be attached in the report to study how data can be transferred over a noisy channel to the recipient.

2 Phase 1: Data Generation

2.1 Objective

The main objective of phase 1 is to observe the effect of Additive White Gaussian Noise on the transmitted signals.

In phase 1, we send a signal through a channel with the addition of Additive White Gaussian Noise, which will result in the received signal being corrupted with the noise. We then use this signal to calculate the bit error rate.

For ease of usage, we have separated most interactions into functions to be reused when we require it later.

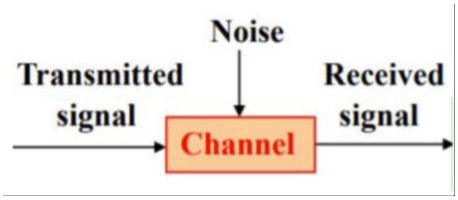

Fig 2.1: Channel Model of Communication System

2.2 Phase 1

```
N = 1024;
SNR = zeros(1,10,'double');
error = zeros(1,10,'double');

for i = 1:11
    SNR(i) = (i-1)*5;
    error(i) = SNRToErrorRate(N,SNR(i));
end

plot(SNR, error)
title("Plot of BER against SNR(dB)")
xlabel("SNR(dB)");
ylabel("BER");
grid on
```

Fig 2.2: Phase1

With reference to Fig 2.2,

- N refers to the randomly generated transmitted signal, with 1024 bits
- SNR and error are arrays initialised to store ‘double’ values of zeros
- Since we will be plotting the graph from 0-50dB at multiples of 5dB, we then run a for loop 10 times
- In the for loop, SNR is initialized with multiples of 5 in the range 0-50
- SNR value for the loop will be fed into our user defined function, SNRToErrorRate(N,SNR(i)) to generate the error rates for the respective SNR values
- We will then generate the graph (Fig 2.3) of Bit Error Rate (BER) against SNR

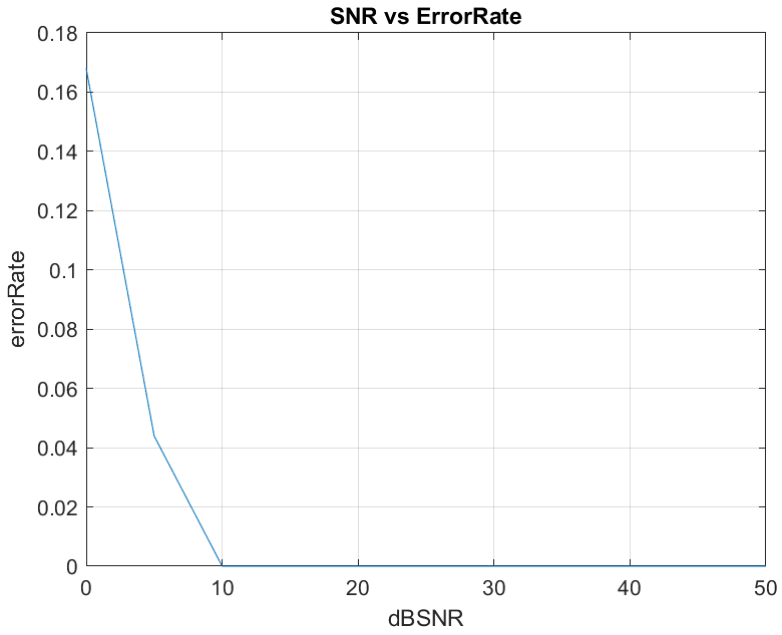

Fig 2.3: SNR vs ErrorRate Graph

2.3 Graphical Analysis

From the graph we attained, we noticed that BER is inversely proportional to SNR; as BER decreases, SNR increases. Due to the noise, the closer the SNR is to the noise floor[1], the more unstable and unpredictable the SNR is, hence the greater the BER.

2.4 SNRToErrorRate(N, dBSNR)

```
function errorRate = SNRToErrorRate(N,dBSNR)
% Call functions to perform phase 1
% calculate error rate based on given SNR
thresholdValue = 0;

% generate data and noise
data = gaussian(N);
noiseData = noise(N, dBSNR);

%add noise to signal
receivedSig = signalAdd(data, noiseData);

thresholdOut = threshold(receivedSig, thresholdValue);

errorRate = checkBitErrorRate(thresholdOut, data);
end
```

Fig 2.4: SNRToErrorRate(N, dBSNR)

With reference to Fig 2.4, SNRToErrorRate is a function to incorporate all functions given N and SNR. It provides an overview of the error rate calculation.

[1]noise floor: The ambient power present in the radio frequency spectrum for your location, frequency, temperature and bandwidth. Understanding the noise floor is important when modelling Bit Error Rate as it is subjected to change and will determine your SNR.

2.5 gaussian(N)

```
function data = gaussian(N)

data = randi([0,1], [1,N]);

data(data==0) = [-1];

end
```

Fig 2.5: gaussian(N)

With reference to Fig 2.5, we initialise the random binary data to be used with MATLAB's predefined randi() function. To ensure that the data consists of only -1 or 1, we will run it through an if statement to convert 0 to -1. We then feed this data back.

2.6 noise(N, dBSNR)

```
function noiseData = noise(N, dBSNR)
% Generate noise with zero mean and required SNR variance
% Phase 1, 2
    S = 1; %Signal Value

    SNR = 10^(dBSNR/10); % dBSNR = 10log10(SNR)
    noiseVariance = S/SNR; % SNR = S/N

    noiseData = randn(1,N) * sqrt(noiseVariance);

end
```

Fig 2.6: noise(N, dBSNR)

With reference to Fig 2.6, we generate noise variance given a SNR value. First, we convert dBSNR to SNR, using $SNR = 10log_{10}\frac{S}{N}dB$, and calculate the noise variance by taking the signal value divided by SNR. Assuming input data has unit power, the signal value S is 1. The noise data has unit variance and zero mean. We then calculate the noise data to be returned.

2.7 signalAdd(sig1, sig2)

```
function addOut = signalAdd(sig1,sig2)
% Add signals together

    addOut = sig1 + sig2;

end
```

Fig 2.7: signalAdd(sig1, sig2)

With reference to Fig 2.7, signalAdd is an adder function to add noise data with signal data, where addOut will be the received signal with Additive White Gaussian Noise.

2.8 threshold(receivedSig, thresholdValue)

```
function thresholdOut = threshold(receivedSig, thresholdValue)

thresholdOut(receivedSig >= thresholdValue) = 1;
thresholdOut(receivedSig < thresholdValue) = 0;

end
```

Fig 2.8: threshold(receivedSig, thresholdValue)

With reference to Fig 2.8, the received signal is fed through if statements. Considering a threshold logic, if the received signal is larger than or equals to the threshold level, we take it as 1, else we take it as 0.

2.9 checkBitErrorRate(thresholdOut, data)

```
function errorRate = checkBitErrorRate(thresholdOut, data)
% Calculate error rate
N = length(data);

if( N ~= length(thresholdOut))
   disp("Data lengths do not match");
   return;
end
error = 0;
for i = 1:N
    if(data(i) > 0 && thresholdOut(i) == 0) || (data(i) <= 0 && thresholdOut(i) == 1)
        error = error + 1;
    end
end

errorRate = error/N;

end
```

Fig 2.9: checkBitErrorRate(thresholdOut, data)

With reference to Fig 2.9, checkBitErrorRate computes the BER. First, we check if the data lengths tally, to weed out errors should we require a rerun. For data, values 1 and -1 represent 1 and 0 respectively. With that in mind, we then compare the threshold values against the data values, and we increment the error counter when the values do not match. After running through the for loop to determine the total amount of errors, we then derive the BER using

Bit Error Rate = $\frac{number\ of\ errors\ during\ transmission}{total\ number\ of\ bits\ for\ transmission}$

This value is then fed back to Phase1.m to generate the graph.

3 Phase 2: Modulation for communication

Referencing back to Fig 1.1, Band pass modulation and band pass demodulation techniques such as On-Off Keying (OOK) and Binary Phase Shift Keying (BPSK) will be applied to the received signal.

3.1 OOK

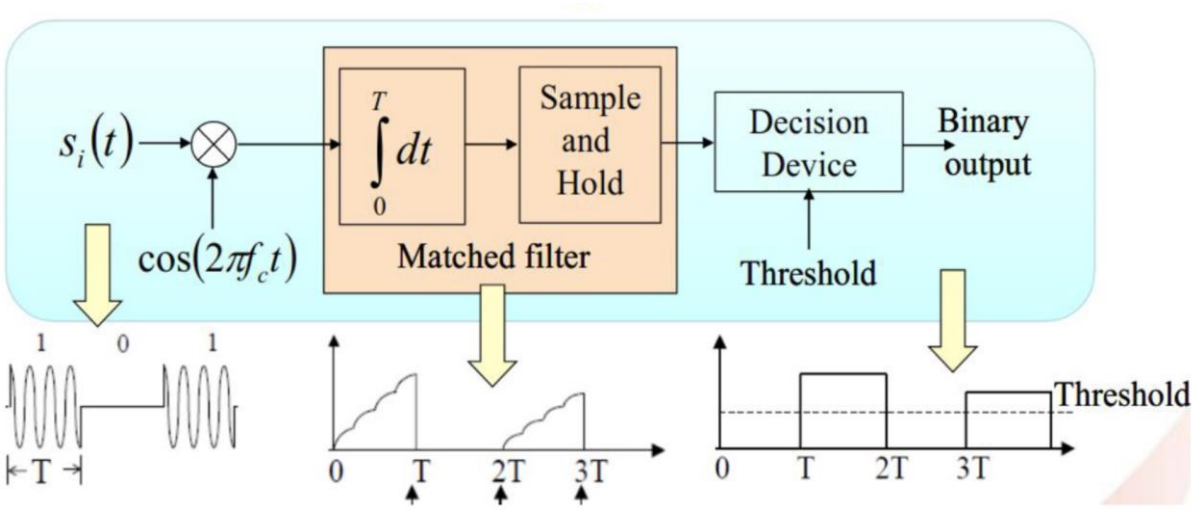

Fig 3.1: Coherent detection for Amplitude Shift Keying

OOK is the simplest form of Amplitude Shift Keying (ASK), where the amplitude is altered to signify a change in levels. Like Fig 3.1, we will be using a coherent detection method instead of non-coherent detection as coherent detection has better performance. This will involve the mixing of the incoming signal ($S_i(t)$) with a locally generated carrier signal($\cos(2\pi f_c t)$). The resultant signal is then put through a Butterworth filter using MATLAB's predefined function butter(). This is to remove all high frequency components present in the signal. The filtered signal is then brought to a decision threshold. We perform this in MATLAB by using if statements to decide the output value.

3.2 BPSK

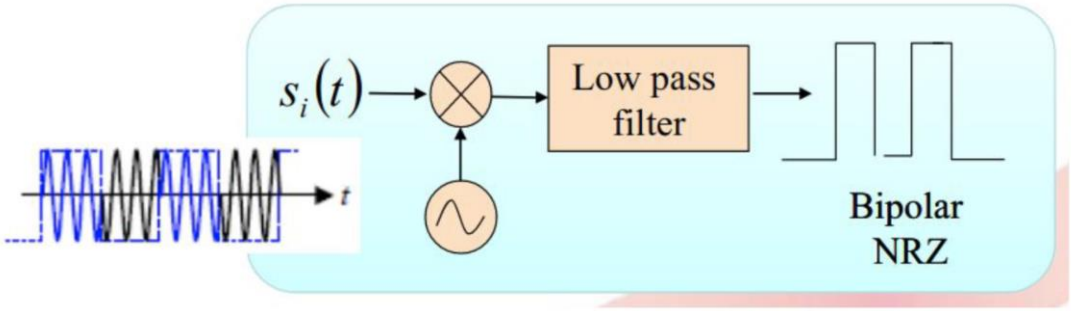

Fig 3.2.1: Coherent detection of Phase Shift Keying (PSK)

Binary Phase Shift Keying (BPSK) is the simplest form of PSK, where a shift in the phase will signify a change in levels. If we extend PSK to a M-ary scheme, BPSK will have an M value of 2. Similar to OOK and taking reference from Fig 3.2.1, we will be using a coherent detection method. As such, we mix the incoming signal ($S_i(t)$) with a locally generated carrier signal ($\cos(2\pi f_c t)$) and pass it through the Butterworth filter. In addition to removing high frequency elements, it will also prevent spectral spreading. The filtered signal is then put through a Bipolar Non Return Zero (NRZ) encoder. The bipolar scheme is an alternative to NRZ.
This scheme has the same signal rate as NRZ, but there is no DC component as one bit is represented by voltage zero and the other alternates.

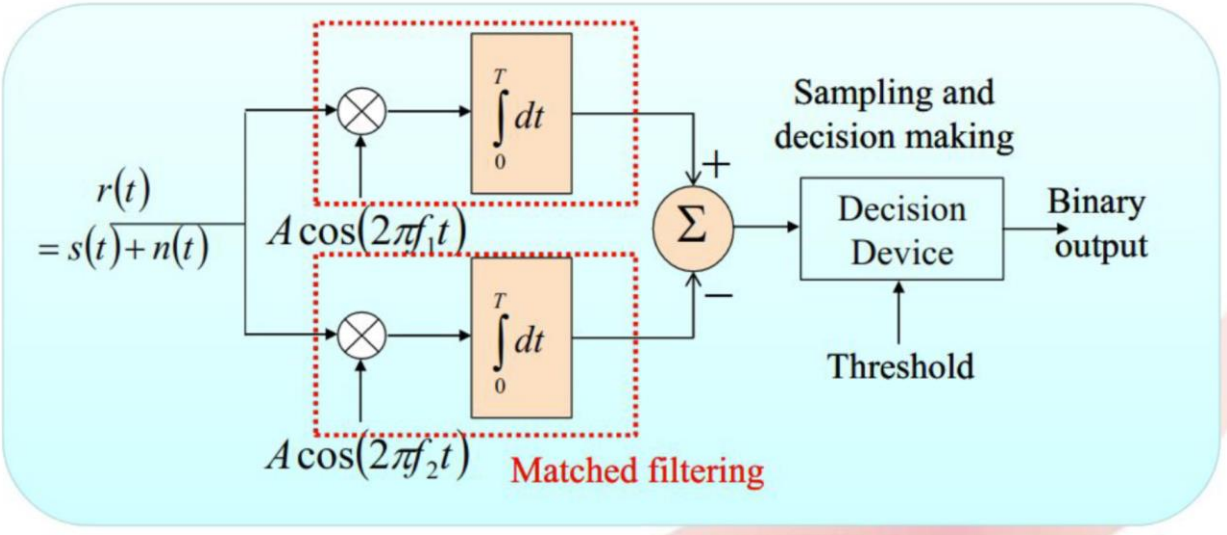

Fig 3.2.2: Matched filtering

With reference to Fig 3.2.2, if we were to use a matched filter instead, this is how it would be represented. The combined signal would have to be integrated and then summed up before going through the decision device where threshold logic will take place.

3.3 Phase 2

```
N = 1024;

data = randi([0,1], [1,N]);

fc = 10000; %carrier freq
dataRate = 1000;
fs = fc * 16; %sampling freq
%to start at where sampling interval starts
t = 1/(2*fs):1/fs:N/dataRate;
carrier = cos(2*pi*fc*t);
numSample = fs*N/dataRate;
```

Fig 3.3.1: Parameter Initialisation

```
dBSNR = zeros(1,10,'double');
OOK_error = zeros(1,10,'double');
BPSK_error = zeros(1,10,'double');
```

Fig 3.3.2: Array Initialisation

With reference to Figs 3.3.1 and 3.3.2, we initialise the data:

- Like Phase1, N = 1024
- Carrier frequency, f_c = 10000 (10kH_z)
- Baseband data rate, dataRate = 1000 (1kbps)
- Sampling frequency, f_s = f_c*16 (carrier frequency is 16 times oversampled, based on the assumption given)
- t = 1/(2*f_s):1/f_s:N/data rate (sampling intervals starts at 1/(2*f_s) and hence we start from this value instead of 0, which will give us 163840 intervals)
- carrier = $\cos(2\pi f_c\, t)$
- numSample = f_s*N/dataRate (Generate the number of samples from the predefined parameters)
- dBSNR, OOK_error and BPSK_error are array initialisations to store ‘double’ values of zeros

Data Modulation

```
dataStream = stretchData(data, numSample, dataRate, fs);
OOK_mod_signal = OOK(dataStream, carrier);
BPSK_mod_signal = BPSK(dataStream, carrier);
```

Fig 3.3.3: Obtaining OOK and BPSK data in manipulatable format

We then obtain the modulated data for both OOK and BPSK.

3.4 stretchData(data, numSample, dataRate, fs)

```
function dataStream = stretchData(data, numSample, dataRate, fs)
% Extend original data by 160 to correctly reflect values
% represent total number of intervals (163840) correctly\
    dataStream = zeros(1, numSample);
    for k = 1: numSample
        dataStream(k) = data(ceil(k*dataRate/fs));
    end
end
```

Fig 3.4: stretchData function

stretchData function enables the original data to be compatible for array manipulation techniques used later. We extend the data by 160 times to obtain the same number of intervals (163840).

3.5 OOK(dataStream, carrier)

```
function signal = OOK(dataStream, carrier)
%OOK Modulation
    signal = dataStream .* carrier;
end
```

Fig 3.5: OOK function to get modulated signal

With reference to Fig 3.5, we generate the modulated signal for OOK by employing array multiplication on the carrier signal with the stretched data.

3.6 BPSK(dataStream, carrier)

```
function signal = BPSK(dataStream, carrier)
% BPSK Modulation
% Convert 0s to -1, 1s remains as 1s
dataStream(dataStream==0) = [-1];
signal = dataStream .* carrier;
end
```

Fig 3.6: BPSK function to get modulated signal

With reference to Fig 3.6, for BPSK we convert 0s in the original data to -1s, with no changes made to the 1s. The result is then multiplied with the carrier signal to obtain the BPSK modulated signal.

3.7 Post-Modulation

```
for i = 1:11
    dBSNR(i) = (i-1)*5;
    noiseData = noise(numSample,dBSNR(i));
    OOK_rx = signalAdd(OOK_mod_signal, noiseData);
    BPSK_rx = signalAdd(BPSK_mod_signal, noiseData);

    OOK_demod_data = demod(OOK_rx, carrier);
    BPSK_demod_data = demod(BPSK_rx, carrier);

    OOK_error(i) = checkBitErrorRate(OOK_demod_data,dataStream);
    BPSK_error(i) = checkBitErrorRate(BPSK_demod_data,dataStream);
end
```

Fig 3.7: Post Modulation process

3.7.1 Code Explanation

Like phase 1, since we are working with intervals of 5dB, we only need to execute a for loop for 10 times.

- Additive White Gaussian Noise (AWGN) is initialised at each iteration with the same function used in phase 1.
- OOK_rx = array addition of OOK modulated signal with noise data generated
- BPSK_rx = array addition of BPSK modulated signal with noise data generated
- After obtaining signals for OOK and BPSK with AWGN, we then demodulate both signals.
- The result of the demodulation is then run through the same checkBitErrorRate function and is stored in an array that keeps track of the error rate from each iteration.

3.8 demod(receivedSig, carrier)

```
function data = demod(receivedSig, carrier)
% Demodulation
    % sigA is the demodulated signal
    sigA = receivedSig .* 2 .* carrier;
    % pass through low pass filter - 6th order, 0.2 cutoff freq
    [b,a] = butter(6, 0.2);
    filteredSig = filtfilt(b, a, sigA);
    % generate output data to be compared, using threshold
    data = threshold(filteredSig, 0.5);
end
```

Fig 3.8: Demodulate the input signal and apply 6th order Butterworth filter

Based on the assumption given to us, we generate a low pass filter by using a 6th order Butterworth filter with a cut-off frequency of 0.2. We utilise filtfilt to pass the demodulated signal through the low pass filter, thus obtaining our filtered signal. We then generate output data for comparison by running the filtered signal through threshold.

3.9 Graph Analysis

```
hold on
title("SNR vs ErrorRate")
xlabel("dBSNR");
ylabel("errorRate");
semilogy(dBSNR, OOK_error)
semilogy(dBSNR, BPSK_error)
hold off
legend('OOK','BPSK');
```

Fig 3.9.1: Graph Function

With reference to Fig 3.9.1, once all iterations are complete, we will generate the graph from the stored Bit Error Rates (BERs) with ranging dBSNR values. We call variables OOK_error and BPSK_error to plot the stored BERs of OOK and BPSK on the same graph.

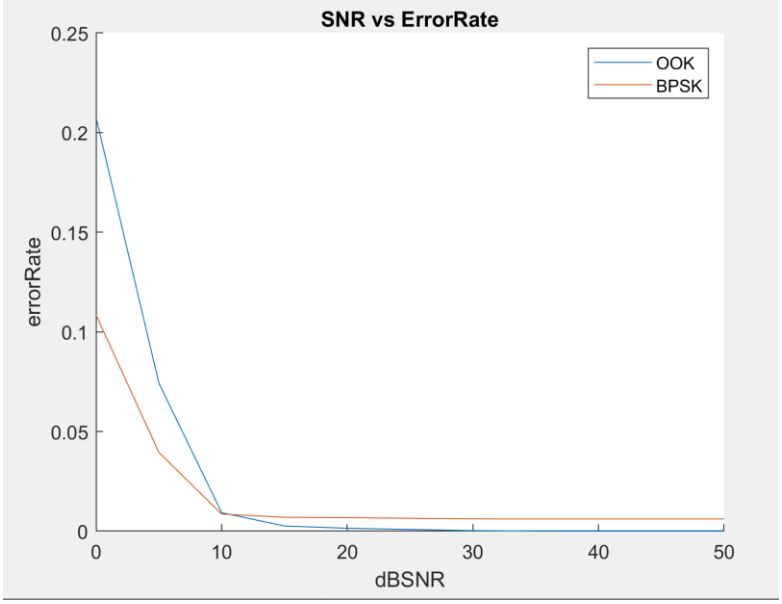

Fig 3.9.2: Graph of OOK BER and BPSK BER against dBSNR

From Fig 3.9.2, OOK has a higher initial BER of 0.2, whereas BPSK's initial BER is around 0.1. However, OOK overtakes BPSK at dBSNR value of 10 and eventually has a near zero BER, as compared to BPSK's BER which converges towards 0 but never reaches 0. This is due to the lack of a Voltage Controlled Oscillator (VCO) to prevent phase discontinuity. As the dBSNR increases, the signal to noise ratio increases, and hence the noise should cause less distortion to the signals, which will mean less errors, eventually resulting in a lower BER. This explains why as dBSNR increases, both BERs decrease.

3.10 Additional Analysis

This section will show all signals plotted by executing the following code

```
ax1 = nexttile([1,2]);
title(ax1,"Data Waveform")
plot(ax1, dataStream)
```

Fig 3.10.1: Waveform generator code

nexttile() function will generate the axes for the chart layout. The chart plot is then plotted against the specific wave form.

Fig 3.10.2: Datastream signal

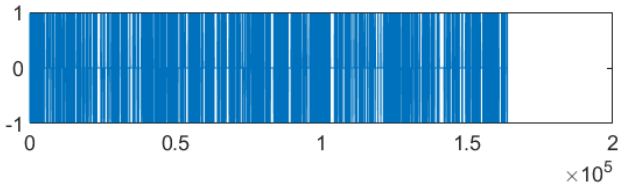

Fig 3.10.3: Modulated OOK Signal

Fig 3.10.4: Modulated BPSK Signal

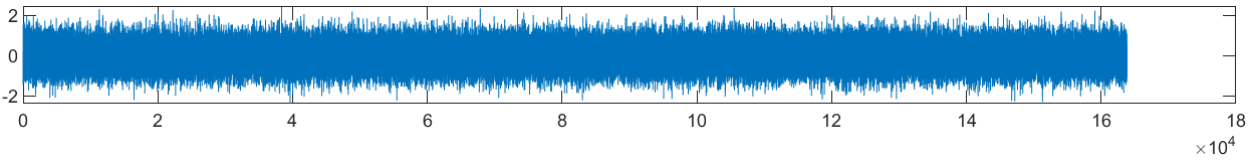

Fig 3.10.5: Noise Signal

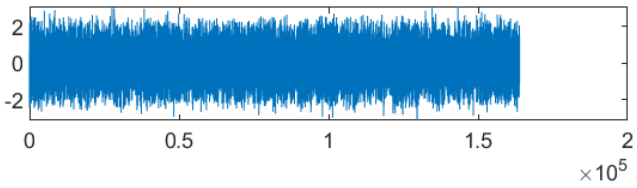

Fig 3.10.6: Modulated OOK Signal added with Noise Signal

When the noise signal is added with the modulated OOK signal, we observe that the amplitude increases to 2. It is also evident that there is no change in the phase or frequency components post-modulation.

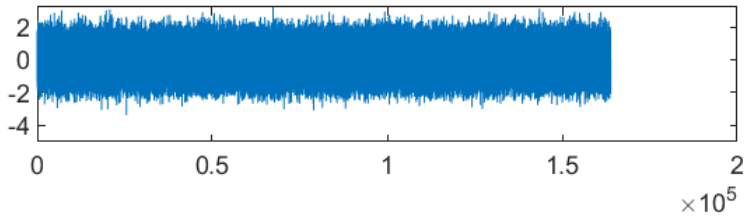

Fig 3.10.7: Modulated BPSK Signal added with Noise Signal

As observed in Figs 3.10.6 and 3.10.7's jagged waveforms, the addition of noise signals creates distortions on the signals.

Fig 3.10.8: OOK Signal Demodulated

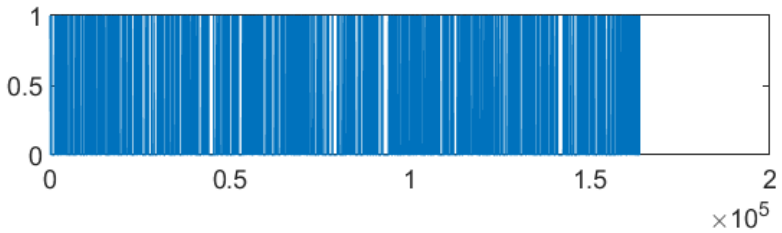

Fig 3.10.9: BPSK Signal Demodulated

Based on Figs 3.10.8 and 3.10.9, we observe that both demodulated signals are extremely similar to the signals in Figs 3.10.3 and 3.10.4 for OOK and BPSK respectively. This suggests that there are little/no errors and therefore aligns with the previous graphical observation of the low BERs.

4 Phase 3: Basic error control coding to improve the performance

4.1 Objective

The main objective of phase 3 is to introduce basic error control coding into the pre-existing communication system to improve the performance of the system. In our case, we will be making use of MATLAB's predefined functions in encoding and decoding functions. This will include the use of linear block code and 2 convolutional techniques, namely cyclic and hamming error correction. We will employ these error correction methods on both BPSK and OOK.

4.2 Phase 3 parameter initialisation

```
[OOK_error, BPSK_error] = Phase2();

% Linear - OOK and BPSK
N = 1024;

data = randi([0,1], [1,N]);

fc = 10000; %carrier freq
dataRate = 1000;
fs = fc * 16; %sampling freq
encoded_bits=N*7/4;

%to start at where sampling interval starts
t_enc = 1/(2*fs):1/fs:encoded_bits/dataRate;
carrier_enc = cos(2*pi*fc*t_enc);
numSample = fs*N/dataRate*7/4;
```

Fig 4.2: Parameter initialisation

4.2.1 Code Explanation

- For convenience of our usage in phase 3, we converted phase 2 to be a function such that the output of phase 2 will return the OOK BERs and BPSK BERs as arrays
- Carrier frequency f_c = 10000 (10kH_z)
- dataRate = 1000 (1kbps)
- Sampling frequency $f_s = f_c * 16$ (Carrier frequency is 16 times oversampled)
- Number of encoded bits = total data sent *7/4 (7 is the code word length and 4 is the message length, this would mean that 3 bits are used for parity checks)
- Like previously, we start when the sampling starts, from $1/(2*f_s)$, at intervals of $1/f_s$ until encoded_bits/dataRate, which means we run through the whole sampling
- Carrier signal calculation is slightly different from previous phases as the time value is adjusted to be based off the encoded bits
- We also alter the numSample as the amount of data sent is affected by the number of encoded bits

4.3 Encoding

```
% encoding
poly = cyclpoly(7,4);
parmat=cyclgen(7,poly);
genmat=gen2par(parmat);
encoded_data = encode(data, 7, 4, 'linear/binary', genmat);
extension_vector = ones(1, fs/dataRate);
encoded_data = kron(encoded_data, extension_vector);

OOK_mod_signal = OOK(encoded_data, carrier_enc);
BPSK_mod_signal = BPSK(encoded_data, carrier_enc);

dBSNR = zeros(1,10,'double');
linear_OOK_error = zeros(1,10,'double');
linear_BPSK_error = zeros(1,10,'double');
```

Fig 4.3: Data Encoding

4.3.1 Code Explanation

- The first 3 lines of code is to create the parity check and generator matrix for error checking. Parmat (parity check matrix) is in the form $[-P^T\ I_{n-k}]$ or $[I_{n-k}\ -P^T]$ where we run gen2par to convert it to $[I_k\ P]$ or $[P\ I_k]$ to get the generator matrix
- Encode function will encode the generator matrix with linear block codes
- extension_vector is an array filled with ones, for length of (f_s/dataRate), to extend the array for array operations
- Kron will return the kronecker tensor product of the matrices, in this case it returns the encoded data that we use for BPSK and OOK modulation.
- Then, we feed the encoded data into the OOK and BPSK functions with the new carrier frequency.
- dBSNR, linear_OOK_error and linear_BPSK_error are ‘double’ type arrays initialised with 10 zeros

4.4 Phase 3 error code modulation and demodulation

```
for i = 1:11
    dBSNR(i) = (i-1)*5;
    noiseData = noise(numSample, dBSNR(i));
    OOK_rx = signalAdd(OOK_mod_signal, noiseData);
    BPSK_rx = signalAdd(BPSK_mod_signal, noiseData);
    OOK_demod_data = demod_phase3(OOK_rx, encoded_bits, fs, dataRate, carrier_enc);
    BPSK_demod_data = demod_phase3(BPSK_rx, encoded_bits, fs, dataRate, carrier_enc);

    OOK_demod_data = decode(OOK_demod_data,7,4,'linear/binary',genmat);
    BPSK_demod_data = decode(BPSK_demod_data,7,4,'linear/binary',genmat);
    linear_OOK_error(i) = checkBitErrorRate(OOK_demod_data, data);
    linear_BPSK_error(i) = checkBitErrorRate(BPSK_demod_data, data);
end
```

Fig 4.4: Error Code Modulation and Demodulation

For every iteration, we demodulate the OOK and BPSK signals and decode the data to find out the total number of errors present. That data is then stored into the linear_error arrays for BPSK and OOK separately. This will run until completion and generate the BERs based on error detection methods. Similar to encode(), decode() is a predefined MATLAB function.

4.5 demod_phase3(receivedSig, encoded_bits, fs, dataRate, carrier)

```
function data = demod_phase3(receivedSig, encoded_bits, fs, dataRate, carrier)
% Demodulation - for phase 3
    % sigA is the demodulated signal
    sigA = receivedSig .* (2 * carrier);
    % pass through low pass filter - 6th order, 0.2 cutoff freq
    [b,a] = butter(6, 0.2);
    filteredSig = filtfilt(b, a, sigA);
    % generate output data to be compared, using threshold
    data = zeros(1, encoded_bits);
    for i = 1:encoded_bits
        thresholdOut = filteredSig(1 /2 * fs/dataRate + (i - 1) * fs/dataRate);
        if thresholdOut > 0.5
            data(i) = 1;
        else
            data(i) = 0;
        end
    end
end
```

Fig 4.5: Demodulation procedure for phase 3

4.5.1 Code Explanation

- We derive the demodulated signal by array multiplying the received signal with 2*carrier signal. We then pass it through the 6th order Butterworth filter with a cut off frequency of $0.2H_z$ and generate the output via a threshold logic. The threshold logic in Fig 4.5 is different from the previous one in Fig 2.7.
- From this, we can get the demodulated signal that will be decoded.

4.6 Graph Plotting

```
%Graph generation
hold on
title("SNR vs ErrorRate")
xlabel("dBSNR");
ylabel("errorRate");
semilogy(dBSNR, OOK_error, '--', 'LineWidth', 2, 'DisplayName', 'OOK w/o encoding')
semilogy(dBSNR, BPSK_error, '--', 'LineWidth', 2, 'DisplayName', 'BPSK w/o encoding')
semilogy(dBSNR, linear_OOK_error, 'DisplayName', 'Linear OSK')
semilogy(dBSNR, linear_BPSK_error, 'DisplayName', 'Linear BPSK')
semilogy(dBSNR, hamming_OOK_error,'DisplayName', 'Hamming OSK')
semilogy(dBSNR, hamming_BPSK_error, 'DisplayName', 'Hamming BPSK')
semilogy(dBSNR, cyclic_OOK_error, 'DisplayName', 'Cyclic OOK')
semilogy(dBSNR, cyclic_BPSK_error, 'DisplayName', 'Cyclic BPSK')
hold off
clear all;
```

Fig 4.6: Graph plotting code

For ease of access and for better comparison between results, we have coded it in a convenient manner where we only need to comment one corresponding line out to remove a certain error. If we want to highlight a specific result such as comparing the effects of non-encoded BPSK and OOK with linear BPSK and OOK it is possible simply by commenting the rest out.

4.7 Comparison of Linear block codes

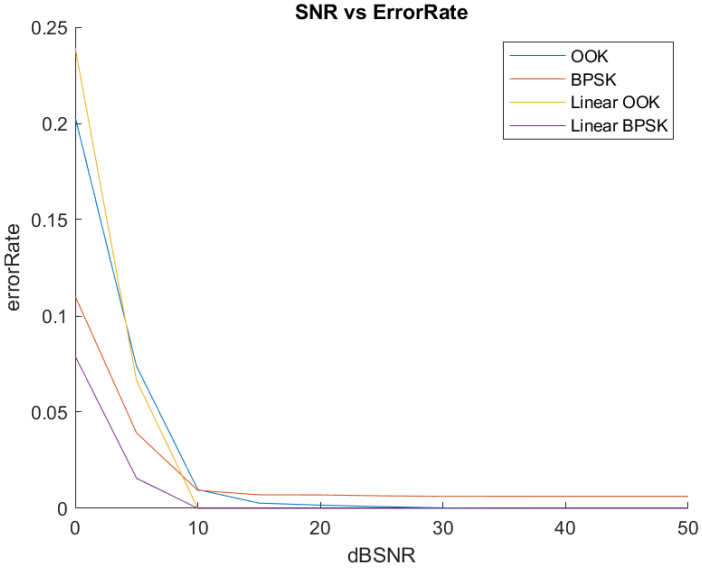

Fig 4.7: Linear block code graph error rate comparison

Initially linear OOK has the highest error rate of around 0.25, but as dBSNR increases its error rate dips significantly and when compared to the error rate for OOK without encoding, it performs better as it reaches zero error earlier at 10 dBSNR. Likewise, Linear BPSK performed much better compared to its counterpart without encoding. It starts off having the lowest error rate and is able to reach a 0 error rate which BPSK without encoding is not able to achieve.

4.8 Comparison of Hamming error correction

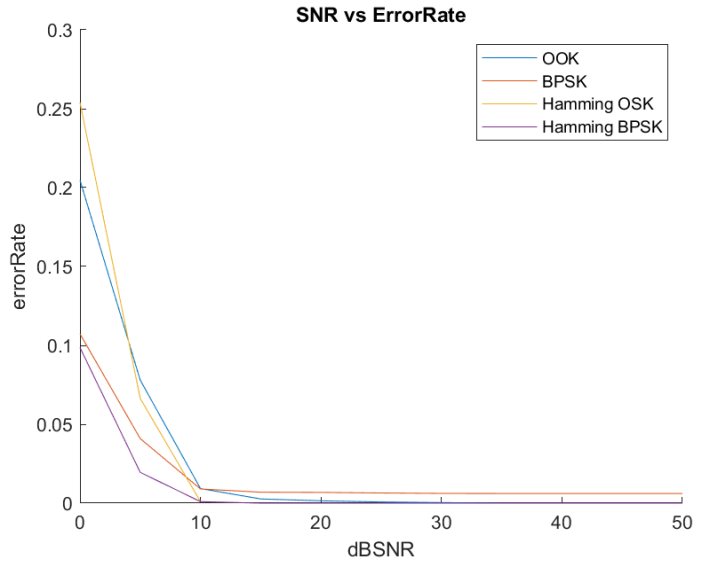

Fig 4.8: Hamming error correction code graph error rate comparison

Fig 4.8 is almost identical to Fig 4.7, with the only difference being that hamming BPSK starts out with an error rate closer to 0.1.

4.9 Comparison of Cyclic error correction

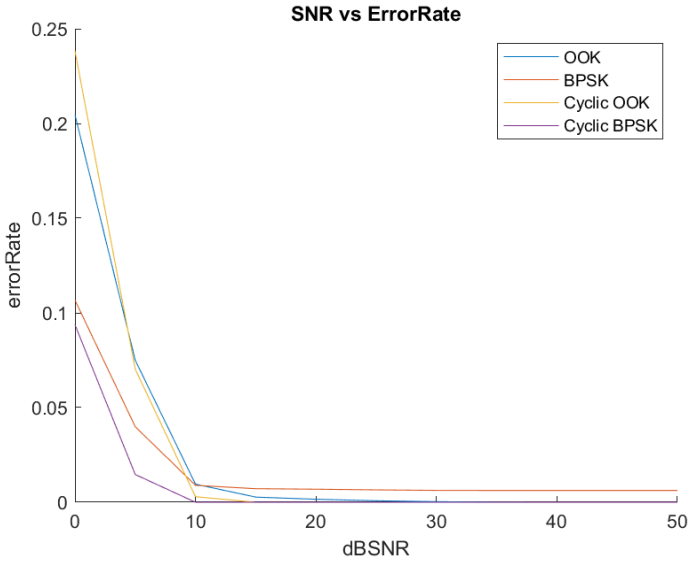

Fig 4.9.1: Cyclic error correction code graph error rate comparison

From Fig 4.9.1, we can see that cyclic and hamming error corrections function very similarly, as there is close to no difference between the 2 figs. This aligns with the theoretical explanation, as both techniques are convolutional in nature and should perform slightly better when compared to linear block coding. This comparison is clearly highlighted in Fig 4.9.2.

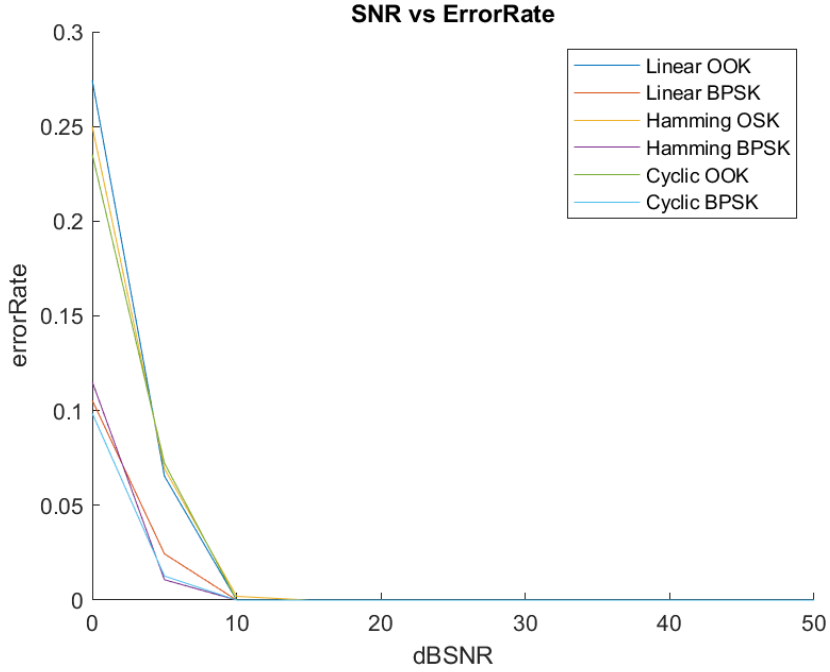

Fig 4.9.2: Comparison of all Error Correction techniques

5 Additional Analysis - BFSK Exploration

5.1 Non-coherent Detection

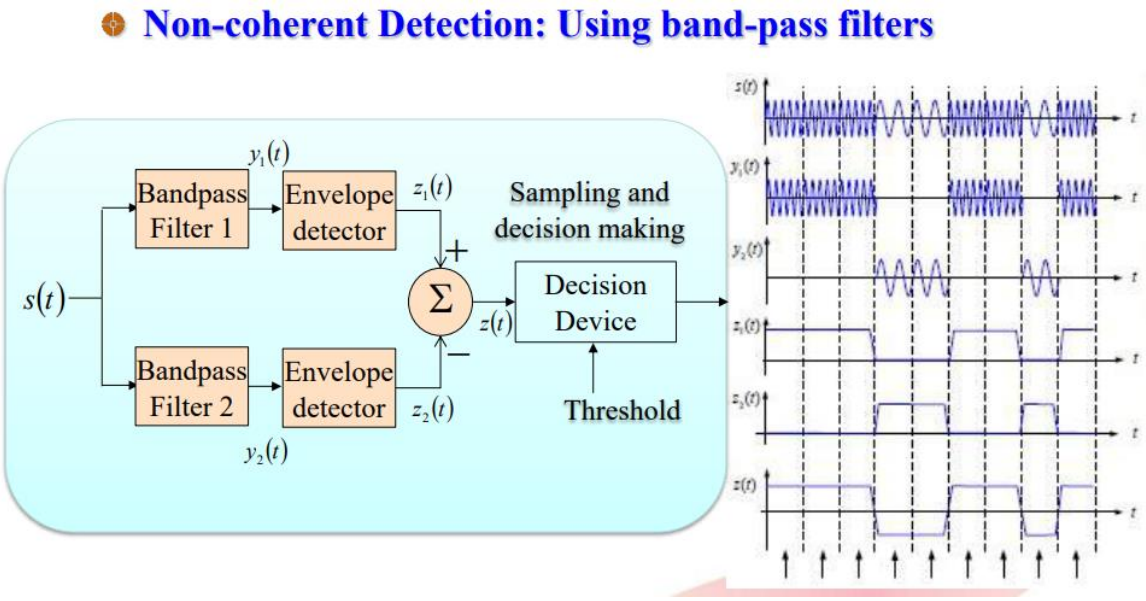

Fig 5.1: Non-coherent Detection of FSK

Binary Frequency Shift Keying (BFSK) is the simplest form of FSK, which utilises a change in frequency to signify a change in levels. With reference to Fig 5.1, we will be using non-coherent detection. Therefore, there will not be any reference signal used in the calculation. In this exploration, the main goal is to find if a bandpass filter vs a low and high pass filter setup will generate different results.

5.2 Bandpass filter specific to frequency

```
N = 1024;
dBSNR = 10;
data = randi([0,1], [1,N]);

fc1 = 10000; %carrier freq
fc2 = 2 * fc1;
dataRate = 1000;
fs = fc2 * 16; %sampling freq
%to start at where sampling interval starts
t = 1/(2*fs):1/fs:N/dataRate;
carrier1 = cos(2*pi*fc1*t);
carrier2 = cos(2*pi*fc2*t);
numSample = fs*N/dataRate;
```

Fig 5.2: Initialisation of BFSK

With reference to Fig 5.2, we initialise the BFSK parameters. We assume that the second carrier frequency (fc2) is double that of the first carrier frequency (fc1).

5.3 Datastream Plot

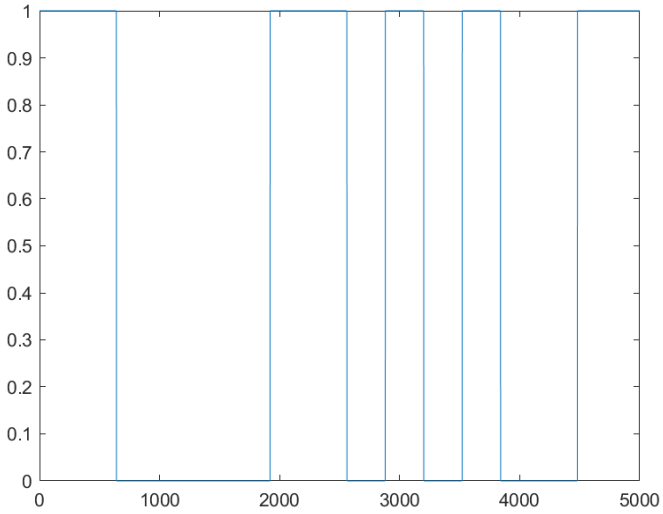

Fig 5.3: Datastream Plot

Fig 5.3 shows the current data before processing it through BFSK modulation.

5.4 BFSK Modulation

```
BFSK_mod_signal = BFSK_mod(dataStream, carrier1, carrier2);

noiseData = noise(numSample, dBSNR);
BFSK_rx = signalAdd(BFSK_mod_signal, noiseData);
% data = BFSK_demod(BFSK_rx, carrier1, carrier2, t);

[b1,a1] = butter(6, [0.10,0.15], 'bandpass');
[b0,a0] = butter(6, [0.05,0.08], 'bandpass');

filteredOnes = filtfilt(b1,a1,BFSK_rx);
filteredZeros = filtfilt(b0,a0,BFSK_rx);

[upperOnes, lowerOnes] = envelope(filteredOnes, 10, 'peak');
[upperZeros, lowerZeros] = envelope(filteredZeros, 10, 'peak');

filteredBFSK = upperOnes - upperZeros;
figure(2)
plot(filteredBFSK)
xlim([0 5000])
bitError = checkBitErrorRate(filteredBFSK, dataStream);
bitError
```

Fig 5.4: Code of BFSK modulation and demodulation

With reference to Fig 5.4, most of the code is identical to the other methods with the key highlight being the 2 Butterworth filters. Like shown in Fig 5.1, the filters are bandpass specific to their incoming signal and we feed the filtered outputs into an envelope detector function to get the x and y values for the 2 signals. The difference (upperOnes - upperZeros) will then be the actual filtered BFSK signal we use to check our error rate.

5.5 Filtered BFSK Signal

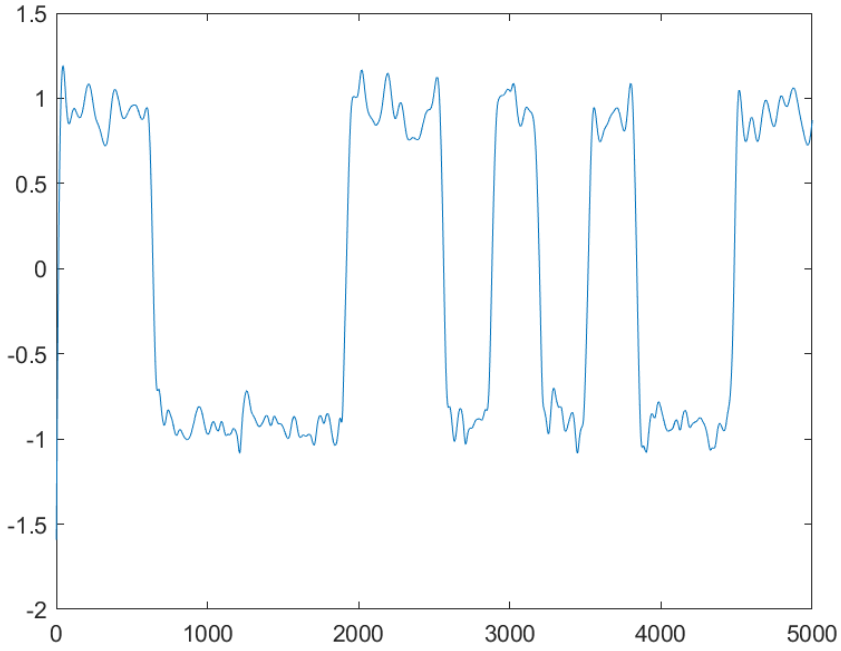

Fig 5.5: Filtered BFSK Signal

Fig 5.5 represents the final demodulated signal of BFSK and from this, we can get our bit error rate for comparison.

5.6 Bit Error Rate for Bandpass Filtration

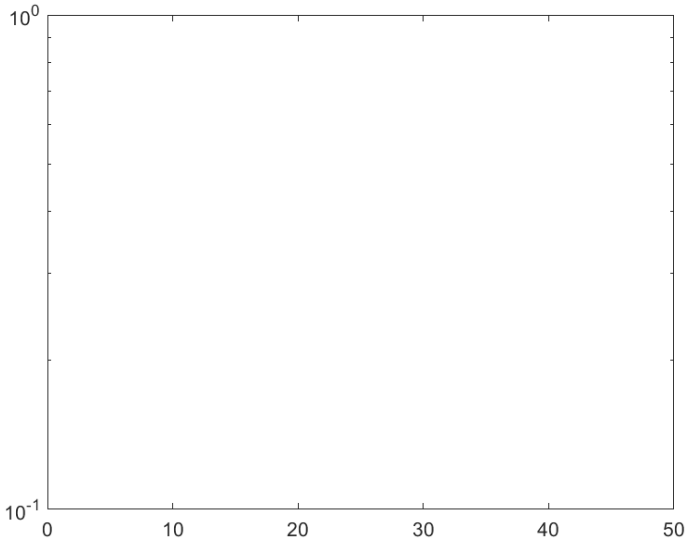

Fig 5.6: Bit Error Rate to dbSNR

Fig 5.6 shows the BER to dbSNR. The graph is empty because the error rate is 0, hence there will not be any points nor plotting done.

5.7 Low and High pass filter Exploration

For sections 5.1- 5.6, we have explored the use of a bandpass filter specific to the different frequencies. We will now be exploring the use of a low and high pass filter instead.

```
% pass through low pass filter - 6th order, 0.2 cutoff freq
[b,a] = butter(6, 0.2);
[bhigh, ahigh] = butter(6, 0.2, "high");
```

Fig 5.7: Low pass and high pass filters

5.8 Demodulation using Square Law

```
for i = 1:11
    dBSNR(i) = (i-1)*5;
    noiseData = noise(numSample, dBSNR(i));
    BFSK_rx = signalAdd(BFSK_mod_signal, noiseData);

    % Demodulation
    % LPF
    BFSK_lowfilt = filtfilt(b, a, BFSK_rx);
    BFSK_highfilt = filtfilt(bhigh, ahigh, BFSK_rx);
    % Square-Law Device
    BFSK_lowsl = BFSK_lowfilt .^ 2;
    BFSK_highsl = BFSK_highfilt .^ 2;
    BFSK_sl = BFSK_highsl - BFSK_lowsl;
    BFSK_demod = threshold(BFSK_sl, 0);
    BFSK_error(i) = checkBitErrorRate(BFSK_demod, dataStream);
end
```

Fig 5.8: Square Law application

We also use a different method to derive the final error rate. We square the filtered output instead of running it through an envelope detector to explore the effect envelope detector has.

5.9 Plot Analysis

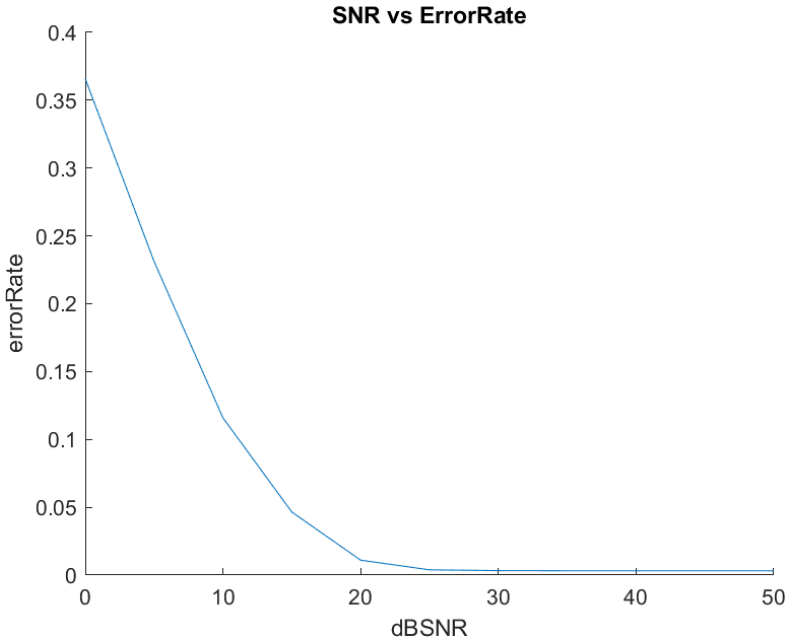

Fig 5.9: SNR vs Error Rate

We can deduce from Fig 5.9 that the error rate loss is rather low as it takes 20 dBSNR to reach saturation point. In comparison, the saturation point is observed at around 10 dBSNR for BPSK and OOK in Fig 3.9.2. Hence, we do not recommend the BFSK method in this experiment.

5.10 Overall Results comparison

Overall, it is evident that the Bandpass filters performed much better than the low pass and high pass filter, as the bandpass filters generated an error rate of 0 while the low pass and high pass filters resulted in a steady convergence to 0. This is likely due to the bandpass filters being calculated to fit an exact range, which is more effective in filtering the noise out. One extra thing to note is that during the calculations, the Bandpass filters contained much larger data and we also had more knowledge regarding the pre-processed signal.

6 Conclusion

Overall, after the experiments conducted, it is evident that convolutional techniques will reduce the error rate much better than linear error correction or no error encoding. When compared to linear error correction, the performance will always be better, but it is good to note that linear error correction initially performed better (as seen in Fig 4.5 when linear BPSK had a lower error rate). This suggests that for experiments where the dBSNR is very low, linear error correction might yield better results than convolutional techniques. We also found that generating bandpass filters yield better results as they are more specific in defining the range of frequencies in which we allow the signal to pass through. This allows the filter to be better fitted, hence generating lower error rates.