# ADCC

Experiment-02 (BFSK)

Manas Kumar Mishra (ESD18I011)

04-NOVEMBER-2021

Lab Incharge: Dr. Premkumar Karumbu

## 1 Aim

To design discrete time BFSK scheme and analyse the bit error rate with signal to noise energy ratio.

## 2 Theory

BFSK is the abbreviation for binary frequency shift keying. Binary implies two values at a time, either 0 or 1. Frequency shift implies the change in frequency of carrier signal based on the information signal. Keying implies the way of representing one signal to the another form.
In very simple terms, carrier signal has two frequency options. Based on the values of information signal, system sets a particular frequency for carrier signal in a fixed time interval i.e. width of information signal. That carrier signal propagates through the channel/medium and captured by receiver or receivers. Based on these different signal frequencies in a particular time interval, a receiver is supposed to decode the information. This is the core idea of frequency shift keying.
In the practical implementation, the gap between the different frequency components should be large enough, such that any receiver can easily recognize the change. But, actually, this requirement is disadvantage of FSK, becuase of inherently large bandwidth.
Mathematically, different frequency components are orthogonal to each other. That means, it is orthogonal modulation scheme. Hence, two dimensional constellation diagram is require for BFSK. Let say 0 is map to $f_o$ and 1 is map to $f_1$ corresponding basis function is $\phi_1(t)$ and $\phi_2(t)$. And each signal has same amplitude $\sqrt{E}$.

$$0 \mapsto (\sqrt{E}, 0)$$
$$1 \mapsto (0, \sqrt{E})$$

Constellation is given below.

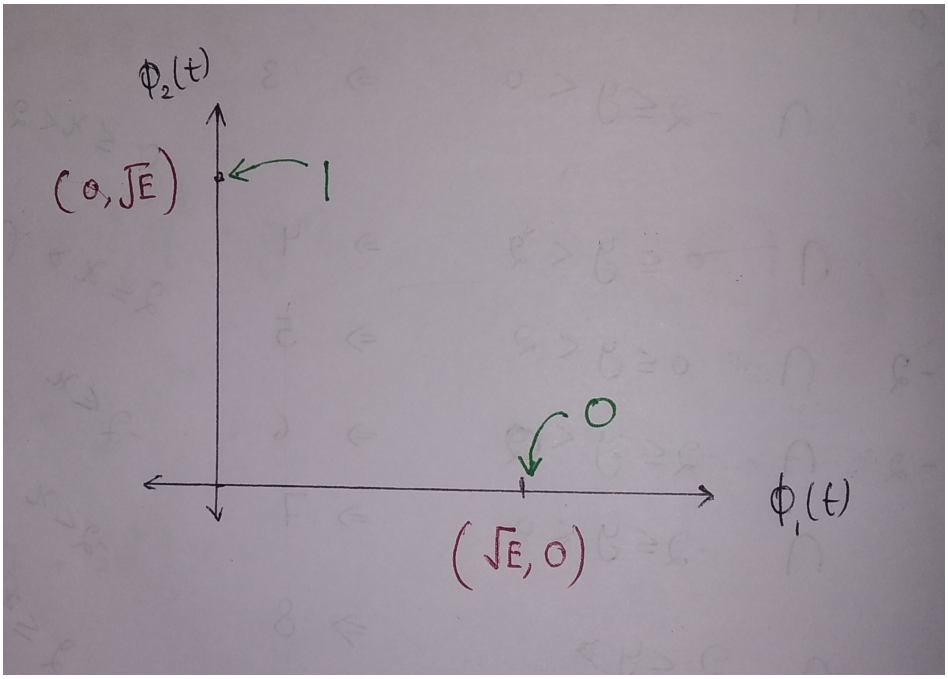


Figure 1: BFSK constellation

# 3 Design

## 3.1 Block diagram

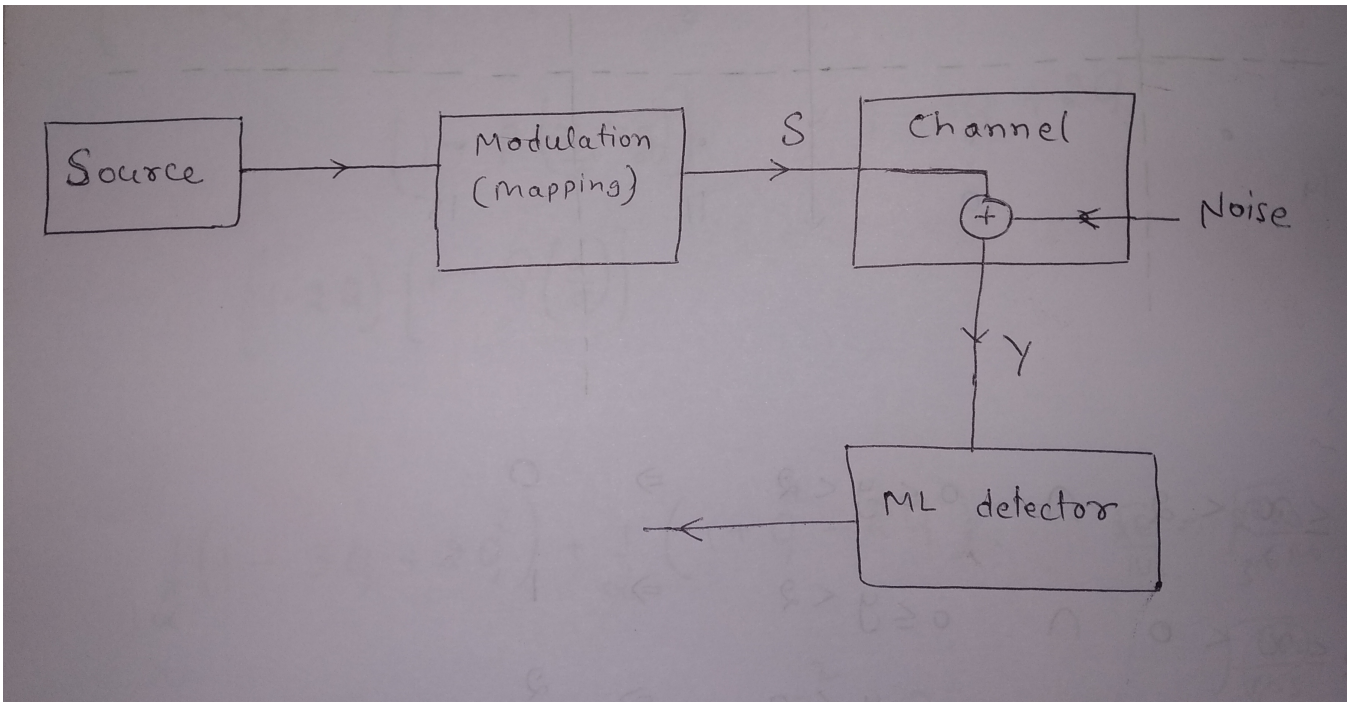


Figure 2: Ideal block diagram

Where source block generates binary bit stream. Modulation block does mapping based on constellation diagram figure 1. Channel block is the representation of AWGN (Additive White Gaussian channel), in that white gaussian noise adds up to the signal values.

## 3.2 ML detector

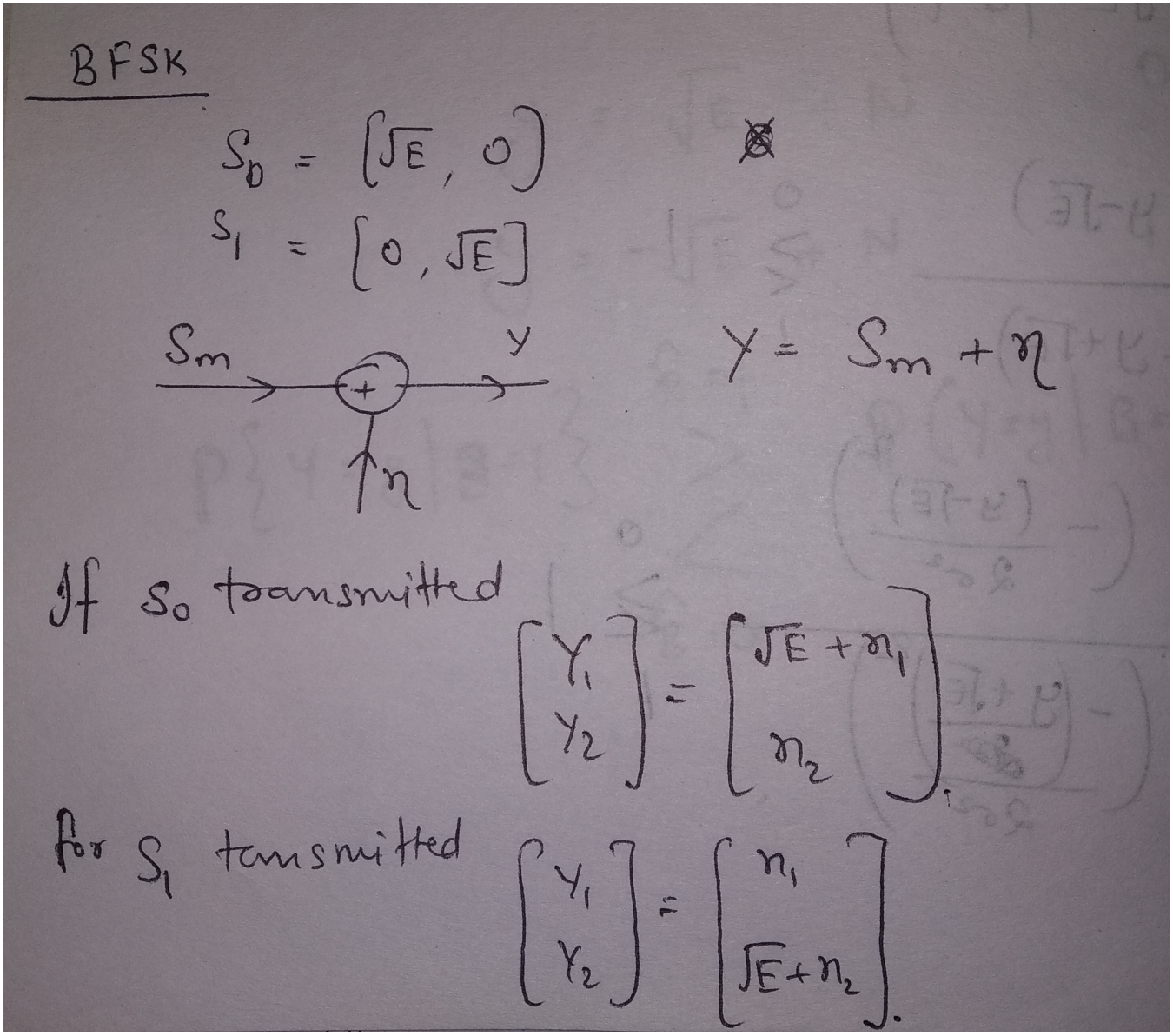


Figure 3: Basic representation for ML calculation

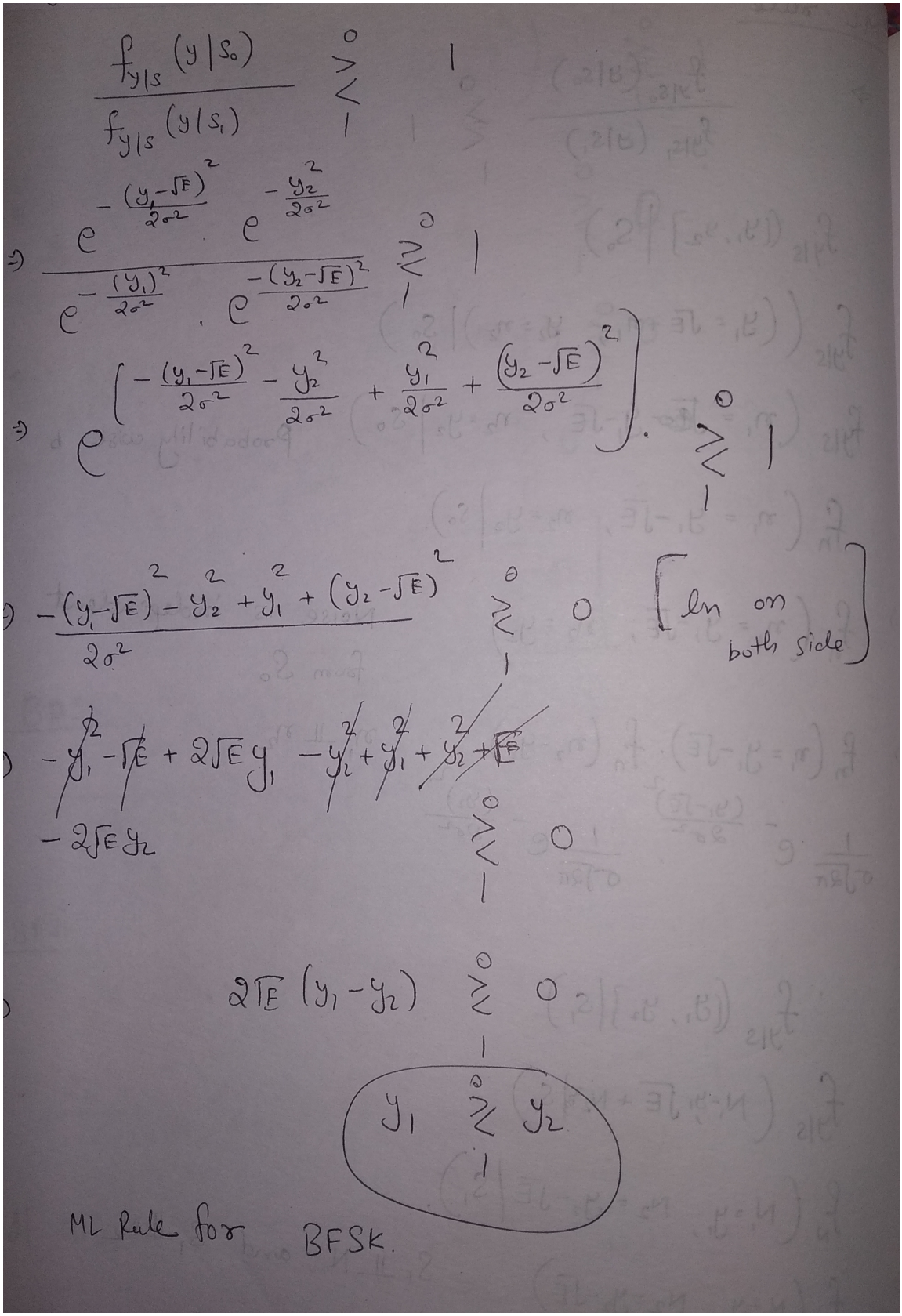


Figure 4: ML rule derivation

This rule can be view as perpendicular bi-sector rule. In that, one side of bisector is one bit region, similarly other side of bisector is for another bit.

## 3.3 Probability of error

$$
\begin{aligned}
P(error) &= P(B=1)P(Error|B=1) + P(B=0)P(Error|B=0) \\
P(Error|B=0) &= P(\hat{B}=1|B=0) \\
&= P(y_2 > y_1|B=0) && \text{(Decision rule)} \\
&= P(\eta_2 > \eta_1 + \sqrt{E}) \\
&= P(\eta_2 - \eta_1 > \sqrt{E}) \\
&= P\left(\frac{\eta_2 - \eta_1}{\sqrt{2}\sigma} > \frac{\sqrt{E}}{\sqrt{2}\sigma}\right) \\
&\text{(Variance of sum of two iid random variables is equal to 2 times their variance)} \\
&= Q\left(\sqrt{\frac{E}{2\sigma^2}}\right) \\
&\Longrightarrow Q\left(\sqrt{\frac{E}{N_o}}\right) \\
similarly\ P(Error|B=1) &= P(\hat{B}=0|B=1) \\
&= P(y_1 > y_2|B=0) && \text{(Decision rule)} \\
&= P(\eta_1 > \eta_2 + \sqrt{E}) \\
&= P(\eta_1 - \eta_2 > \sqrt{E}) \\
&= P\left(\frac{\eta_1 - \eta_2}{\sqrt{2}\sigma} > \frac{\sqrt{E}}{\sqrt{2}\sigma}\right) \\
finally\ , P(Error) &= Q\left(\sqrt{\frac{E}{N_o}}\right)(P(B=0) + P(B=1)) \\
P(Error) &= Q\left(\sqrt{\frac{E}{N_o}}\right)
\end{aligned}
$$

**RECALL:**
***For BPSK probability of error***

$$P(Error) = Q\left(\sqrt{\frac{2E}{N_o}}\right)$$

That means, BFSK has more error than BPSK for a fixed value of $\frac{E}{N_o}$

# 4 Code and result

## 4.1 Code for BFSK in cpp

```
1  ////////////////////////////////////////////////////////////////////////////////////////
2  // Author:- MANAS KUMAR MISHRA
3  // Organisation:- IIITDM KANCHEEPURAM
4  //Topic:-BFSK MODEL AND "PROBABILITY OF Bit ERROR" vs "RATIO OF SIGNAL TO NOISE ENERGY(dB)"
5  ////////////////////////////////////////////////////////////////////////////////////////
6
7  #include <iostream>
8  #include <cmath>
9  #include <iterator>
10 #include <random>
11 #include <chrono>
12 #include <time.h>
13 #include <fstream>
14
15 #define one_million 1000000
16
17 using namespace std;
18
19 // This class contains all functions/methods for BFSK
20 class BFSK
21 {
22     public:
23         vector<double> Source();
24         vector<double> TransmittedSymbol_columnVector1(vector<double> sourceBits,
25                     double& energyOfbit);
26         vector<double> TransmittedSymbol_columnVector2(vector<double> sourceBits,
27                      double& energyOfbit);
28         vector<double> GnoiseVector(double& mean, double& stddev);
29         vector<double> ChannelModel(vector<double>sourceSymbols,
30                     vector<double> AWGnoise);
31         vector <double> DecisionBlock(vector<double> receive1, vector<double> receive2);
32         int ErrorCount(vector <double> sourceSymbols, vector<double> decisionSymbols);
33
34 };
35
36
37
38 // Function for generating binary bits at source side. Each bit is equiprobable.
39 // Input is nothing
40 // Output is a vector that contains the binary bits of length one_million*1.
41 vector<double>BFSK::Source()
42 {
43     vector<double> sourceBits;
44
45     // Use current time as seed for random generator
46     srand(time(0));
47
48     for(int i = 0; i<one_million; i++){
49         sourceBits.insert(sourceBits.end(), rand()%2);
50     }
```

```
51
52      return sourceBits;
53  }
54
55
56
57  // Function of generate vector form of signal, this function generate the upper
58  //value of the vector.
59  // Input is the source bit stream and energy of bit.
60  // Output is the vector containing all upper values based on the source bits.
61  vector<double>BFSK::TransmittedSymbol_columnVector1(vector<double> sourceBits,
62  double& energyOfbit)
63  {
64      vector <double> transmittedSymbol;
65
66       for(int i=0; i<one_million; i++){
67          if(sourceBits[i]== 0){
68              transmittedSymbol.insert(transmittedSymbol.end(), sqrt(energyOfbit));
69          }
70          else{
71              transmittedSymbol.insert(transmittedSymbol.end(), 0);
72          }
73
74      }
75
76      return transmittedSymbol;
77  }
78
79
80
81  // Function of generate vector form of signal, this function generate the lower
82  //value of the vector.
83  // Input is the source bit stream and energy of bit.
84  // Output is the vector containing all lower values based on the source bits.
85  vector<double>BFSK::TransmittedSymbol_columnVector2(vector<double> sourceBits,
86  double& energyOfbit)
87  {
88      vector <double> transmittedSymbol;
89
90       for(int i=0; i<one_million; i++){
91          if(sourceBits[i]== 0){
92              transmittedSymbol.insert(transmittedSymbol.end(), 0);
93          }
94          else{
95              transmittedSymbol.insert(transmittedSymbol.end(), sqrt(energyOfbit));
96          }
97
98      }
99
100     return transmittedSymbol;
101 }
102
103
104
105 // Function for generating random noise based on gaussian distribution N(mean, variance).
106 // Input mean and standard deviation.
```

```
107 // Output is the vector that contain gaussian noise as an element.
108 vector<double>BFSK::GnoiseVector(double& mean, double& stddev)
109 {
110     vector<double> noise;
111 
112     // construct a trivial random generator engine from a time-based seed:
113     unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
114     std::default_random_engine generator (seed);
115 
116     std::normal_distribution<double> dist(mean, stddev);
117 
118     // Add Gaussian noise
119     for (int i =0; i<=one_million; i++) {
120         noise.insert(noise.end(),dist(generator));
121     }
122 
123     return noise;
124 }
125 
126 
127 
128 
129 // Function for model the channel
130 // Input sourcesymbols and AWGN
131 // Output is a vector that represent the addition of two vectors
132 vector<double>BFSK::ChannelModel(vector<double> sourceSymbols, vector<double> AWGnoise)
133 {
134     vector<double> received;
135 
136     for(int i=0; i<sourceSymbols.size(); i++){
137         received.insert(received.end(), sourceSymbols[i]+AWGnoise[i]);
138     }
139 
140     return received;
141 }
142 
143 
144 
145 // Function for take decision after receiving the vector
146 // Input is the 2D receiver vector
147 // Output is the decision {0, 1}
148 vector <double>BFSK::DecisionBlock(vector<double> receive1, vector<double> receive2)
149 {
150     vector<double> decision;
151 
152     for(int j =0; j<receive1.size(); j++){
153 
154         if(receive1[j]< receive2[j]){
155             decision.insert(decision.end(), 1);
156         }else{
157             decision.insert(decision.end(), 0);
158         }
159     }
160 
161     return decision;
162 }
```

```
163
164
165
166 // Function for counting errors in the symbols
167 // Input are the source symbols and decided bits
168 // Output is the error count
169 int BFSK::ErrorCount(vector <double> sourceSymbols, vector<double> decisionSymbols)
170 {
171     int count =0;
172
173     for(int i=0; i<sourceSymbols.size(); i++){
174         if(sourceSymbols[i] != decisionSymbols[i]){
175             count++;
176         }
177     }
178
179     return count;
180 }
181
182
183
184 // Function to store the data in the file (.dat)
185 // Input is the SNR per bit in dB and calculated probability of error
186 // Output is the nothing but in processing it is creating a file and writing data into it.
187 void datafile(vector<double> xindB, vector<double> Prob_error)
188 {
189     ofstream outfile;
190
191     outfile.open("BFSK1.dat");
192
193     if(!outfile.is_open()){
194         cout<<"File opening error !!!"<<endl;
195         return;
196     }
197
198     for(int i =0; i<xindB.size(); i++){
199         outfile<< xindB[i] << " "<<"\t"<<" "<< Prob_error[i]<< endl;
200     }
201
202     outfile.close();
203 }
204
205
206 // Function for calculate the Q function values.
207 // Input is any positive real number.
208 // Output is the result of erfc function (equal form of Q function).
209 double Qfunc(double x)
210 {
211     double Qvalue = erfc(x/sqrt(2))/2;
212     return Qvalue;
213 }
214
215
216
217 vector<double> Qfunction(vector <double> SNR_dB)
218 {
```

```
    vector <double> Qvalue;
    double po, normalValue;
    for (int k =0; k<SNR_dB.size(); k++){
        normalValue = pow(10, (SNR_dB[k]/10));
        po = Qfunc(sqrt(1*normalValue));
        Qvalue.insert(Qvalue.end(), po);
    }

    return Qvalue;
}



// Function to store the data in the file (.dat)
// Input is the SNR per bit in dB and calculated Qfunction values
// Output is the nothing but in processing it is creating a file and writing data into it.
void qvalueInFile(vector <double> SNR, vector <double> Qvalue)
{
    ofstream outfile;

    outfile.open("BFSK_Qvalue1.dat");

    if(!outfile.is_open()){
        cout<<"File opening error !!!"<<endl;
        return;
    }

    for(int i =0; i<SNR.size(); i++){
        outfile<< SNR[i] << " "<<"\t"<<" "<< Qvalue[i]<< endl;
    }

    outfile.close();
}




int main(){


    // Object of class
    BFSK bfsk1;

    // source defination
    vector<double> sourceBits;

    // Transmiited vectors
    vector<double> trans1;
    vector<double> trans2;

    // Noise vector
    vector<double> gnoise1;
    vector<double> gnoise2;

    // Receive bits
    vector<double> receivedBits1;
```

```
    vector<double> receivedBits2;

    // Decision block
    vector<double> decodedBits;

    double N_o = 8;
    double stddev = sqrt(N_o/2);
    double mean = 0.0;
    double errors=0;
    double pe;


    // SNR in dB
    vector<double> SNR_dB;
    for(float i =0; i<=14; i=i+0.125)
    {
        SNR_dB.insert(SNR_dB.end(), i);
    }

    vector<double> energyOfBits;
    vector<double> Prob_error;
    double normalValue;

    for(int i =0; i<SNR_dB.size(); i++){

        normalValue = pow(10, (SNR_dB[i]/10));
        energyOfBits.insert(energyOfBits.end(), N_o*normalValue);
    }

    for(int step =0; step<energyOfBits.size(); step++){
        sourceBits = bfsk1.Source();

        trans1 = bfsk1.TransmittedSymbol_columnVector1(sourceBits, energyOfBits[step]);
        trans2 = bfsk1.TransmittedSymbol_columnVector2(sourceBits, energyOfBits[step]);

        gnoise1 = bfsk1.GnoiseVector(mean, stddev);
        gnoise2 = bfsk1.GnoiseVector(mean, stddev);

        receivedBits1 = bfsk1.ChannelModel(trans1, gnoise1);
        receivedBits2 = bfsk1.ChannelModel(trans2, gnoise2);

        decodedBits = bfsk1.DecisionBlock(receivedBits1, receivedBits2);

        errors = bfsk1.ErrorCount(sourceBits, decodedBits);

        pe = errors/one_million;

        Prob_error.insert(Prob_error.end(), pe);

        cout<< " Error count : "<<errors<< endl;
        cout<< " Pe : "<<pe << endl;

        cout<<endl;
    }

    datafile(SNR_dB, Prob_error);
```

```
331     vector<double> qvalue = Qfunction(SNR_dB);
332     qvalueInFile(SNR_dB, qvalue);
333
334     return 0;
335 }
```

## 4.2 Result

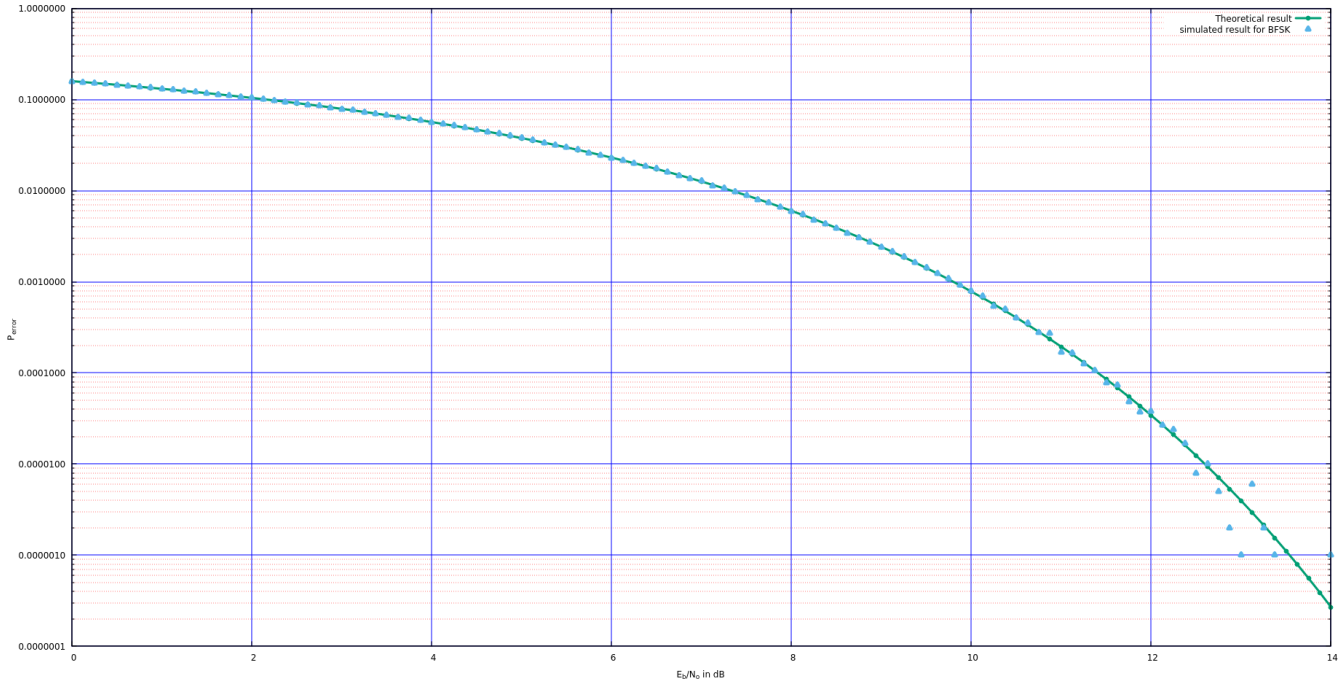


Figure 5: Result

After the simulation of BFSK, a comparison study has been done with BPSK experiment. That is given below.

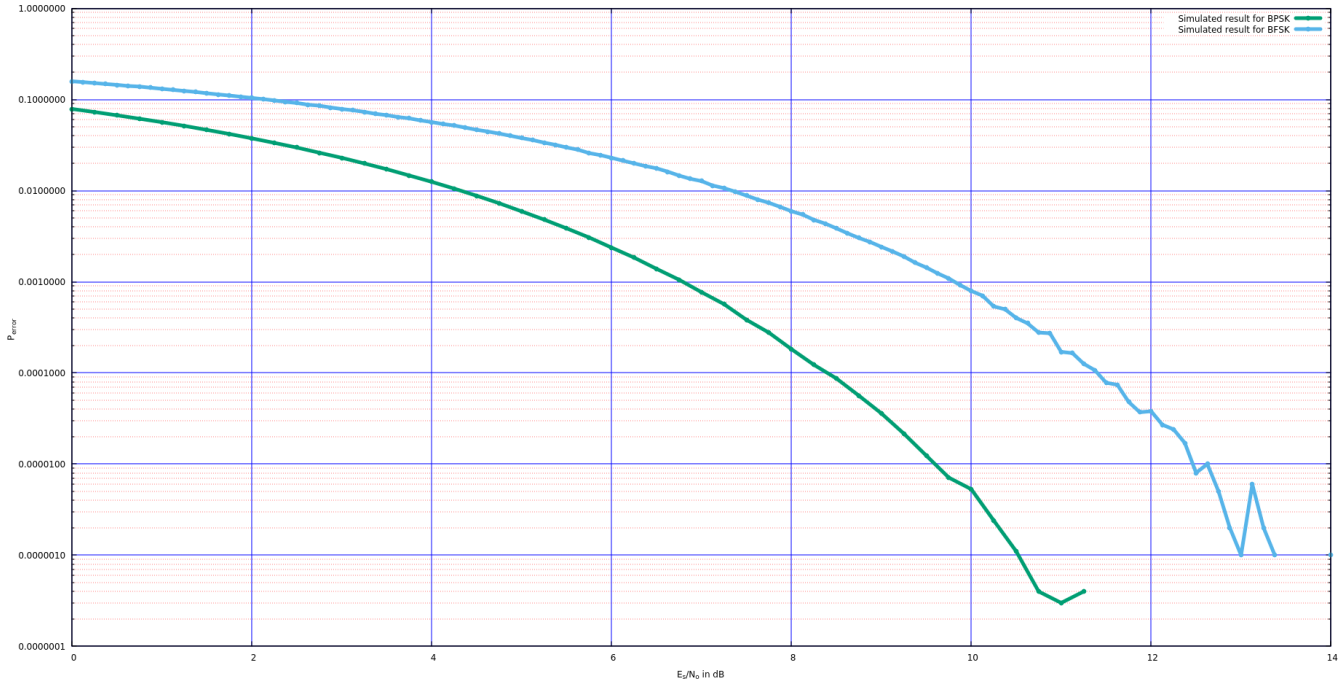


Figure 6: Result of comparison

# 5 Inferences

1. For large $\frac{E_b}{N_o}$ value probability of error decreases.
2. For large bit transmissions simulated result is same as calculated result.
3. After SNR =13dB, simulated results for error are exactly zero, that implies, 13dB SNR is sufficient to ensu no error in BFSK.
4. Since, BFSK has two frequency components in the transmitted signal, bandwidth is inherently higher than other schemes.
5. For data rate, BFSK require more bandwidth and having higher probability of errors as compare to BPSK.