ADCC

Experiment-01 (BPSK)

Manas Kumar Mishra (ESD18I011)

04-NOVEMBER-2021

Lab Incharge: Dr. Premkumar Karumbu

1 Aim

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

2 Theory

BPSK is the abbreviation of binary phase shift keying. Binary implies only two possible values (0 or 1) at a time. Phase shift implies the changing the phase (only phase) of the carrier signal based on the binary values. Keying implies the modulation way to represent some information in another way. This is a very simple modulation scheme.
Due to binary values, one can select two extreme phases for mapping i.e. 0 and π combination or $\frac{\pi}{2}$ and $\frac{3\pi}{2}$. Consider 0 and π phase combination for this experiment.
Consider a constellation diagram with one axis as $\phi(t)$ (one dimensional case).Amplitude on the constellation diagram is $\sqrt{E}$. Phase is the angle between positive axis to mapped point in counter clock wise direction.

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

That constellation diagram is given below.

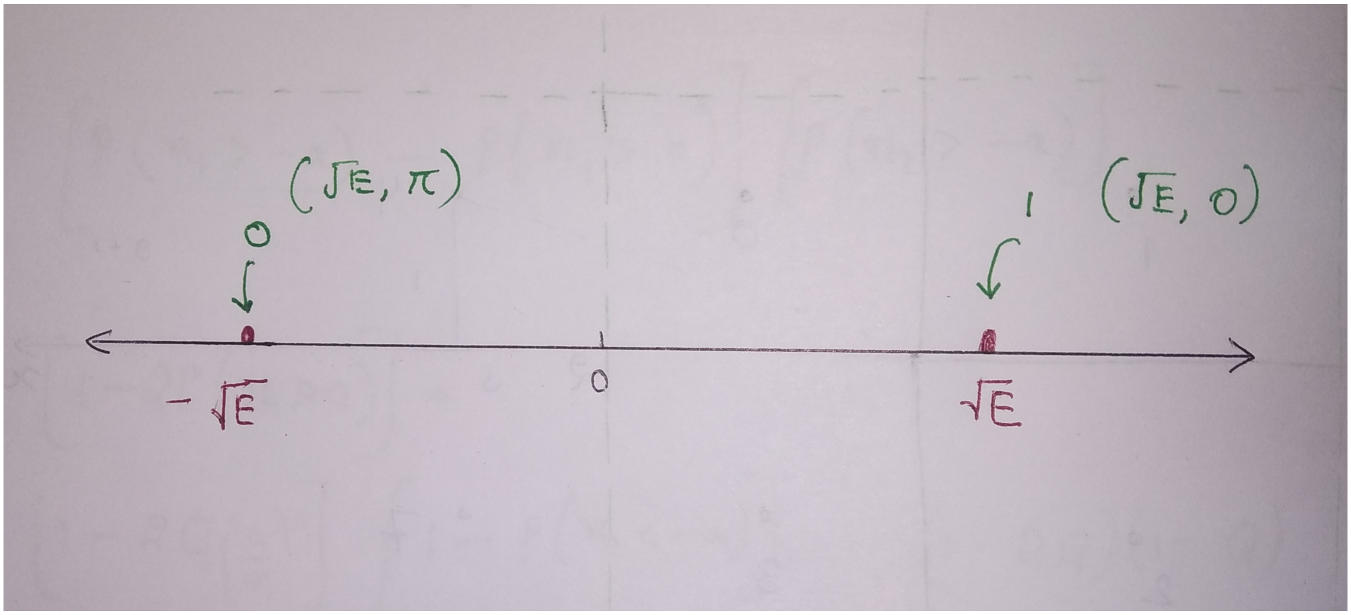

Figure 1: BPSK constellation diagram

3 Design

3.1 Block diagram

Block diagram represent the design and working of BPSK scheme.

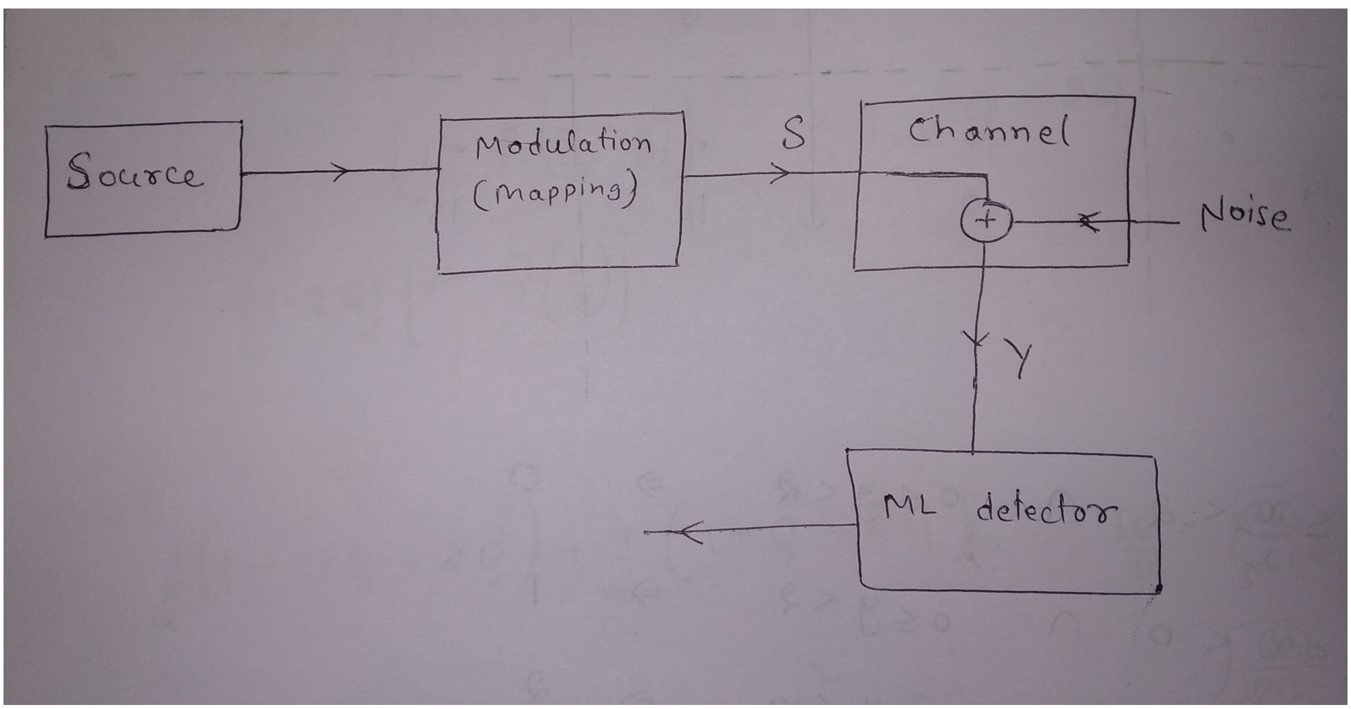

Figure 2: BPSK block diagram

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 Noise), in that gaussian noise adds up to the signal values.
Channel is the main issue that has to tackle carefully. For that, ML detector added. That maps the receive value to the binary value (0 or 1).

$$Y = S + \eta$$

Where S is transmitted value.

$\eta \sim$ gaussian$(0, \frac{N_o}{2})$

Y is received value

3.2 ML rule

By the definition,

$$\begin{aligned}
\frac{f_{Y|S}(y|0)}{f_{Y|S}(y|1)} &\gtrless_{B=1}^{B=0} 1 \\
\implies \frac{f_\eta(\eta = y + \sqrt{E})}{f_\eta(\eta = y - \sqrt{E})} &\gtrless_{B=1}^{B=0} 1 \\
\implies \left(\frac{\frac{e^{-(y+\sqrt{E})^2/2\sigma^2}}{\sqrt{2\pi}\sigma}}{\frac{e^{-(y-\sqrt{E})^2/2\sigma^2}}{\sqrt{2\pi}\sigma}} \right) &\gtrless_{B=1}^{B=0} 1 \\
\implies \left(e^{\frac{(y-\sqrt{E})^2-(y+\sqrt{E})^2}{2\sigma^2}} \right) &\gtrless_{B=1}^{B=0} 1 \\
\implies \left(\frac{(y-\sqrt{E})^2-(y+\sqrt{E})^2}{2\sigma^2} \right) &\gtrless_{B=1}^{B=0} 0 \qquad \text{(Apply ln on both side)} \\
\implies \left(-2\sqrt{E}y \right) &\gtrless_{B=1}^{B=0} 0 \\
\implies y &\gtrless_{B=0}^{B=1} 0
\end{aligned}$$

Due to length of every step, few steps has been skipped from the ML rule derivation reader can fill the step on there own.

3.3 Probability of error

$$
\begin{aligned}
P(error) &= P(B=1)P(Error|B=1) + P(B=0)P(Error|B=0) \\
P(Error|B=0) &= P(\hat{B}=1|B=0) \\
&= P(y>0|B=0) \\
&= P(\eta - \sqrt{E} > 0) \\
&= P(\eta > \sqrt{E}) \\
&= P(\frac{\eta}{\sigma} > \frac{\sqrt{E}}{\sigma}) \\
&\implies Q(\sqrt{\frac{E}{\sigma^2}}) \\
&\implies Q(\sqrt{\frac{2E}{N_o}}) \\
similarly\ P(Error|B=1) &= Q(\sqrt{\frac{2E}{N_o}}) \qquad \text{(Symmetrical constellation diagram about origin)} \\
finally\ \ ,P(Error) &= Q(\sqrt{\frac{2E}{N_o}})(P(B=0)+P(B=1)) \\
P(Error) &= Q(\sqrt{\frac{2E}{N_o}})
\end{aligned}
$$

4 Code and Results

4.1 Code for BPSK

```
////////////////////////////////////////////////////////
////////////////////////////////////////////
// Author:- MANAS KUMAR MISHRA
// Organisation:- IIITDM KANCHEEPURAM
// Topic:- BPSK MODEL AND "PROBABILITY OF BIT ERROR"
//vs "RATIO OF SIGNAL TO NOISE ENERGY (dB)"
///////////////////////////////////////////////////////
/////////////////////////////////////////////

#include <iostream>
#include <cmath>
#include <iterator>
#include <random>
#include <chrono>
#include <time.h>
#include <fstream>

#define one_million 1000000

using namespace std;
```

```
// Function for generating binary bits at source side.
//Each bit is equiprobable.
// Input is nothing
// Output is a vector that contains the binary bits of
//length one_million*1.
vector<double> sourceVector()
{
    vector<double> sourceBits;

    // Use current time as seed for random generator
    srand(time(0));

    for(int i = 0; i<one_million; i++){
        sourceBits.insert(sourceBits.end(), rand()%2);
    }

    return sourceBits;
}


// Function for mapping bits to symbol.
// Input is a binary bit vector. Here 0---> -(sqrt(Energy)) and 1---> (sqrt(Energy))
// Output is a vector that contains transmitted symbols.
vector<double> bit_maps_to_symbol_of_energy_E(vector<double> sourceBits,
double energyOfSymbol)
{
    vector<double> transmittedSymbol;

    for(int i=0; i<one_million; i++){
        if(sourceBits[i]== 0){
            transmittedSymbol.insert(transmittedSymbol.end(), -sqrt(energyOfSymbol));
        }
        else{
            transmittedSymbol.insert(transmittedSymbol.end(), sqrt(energyOfSymbol));
        }

    }

    return transmittedSymbol;
}


// Function for generating random noise based on gaussian distribution N(mean, variance).
// Input mean and standard deviation.
// Output is the vector that contain gaussian noise as an element.
vector<double> GnoiseVector(double mean, double stddev)
{
    std::vector<double> data;

    // construct a trivial random generator engine from a time-based seed:
    unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
    std::default_random_engine generator (seed);

    std::normal_distribution<double> dist(mean, stddev);
```

```

    // Add Gaussian noise
    for (int i =0; i<one_million; i++) {
        data.insert(data.end(),dist(generator));
    }

    return data;
}


// Function for modeling additive channel. Here gaussian
//noise adds to the transmitted bit.
// Inputs are the transmitted bit and gaussian noise with mean 0 and variance 1.
// Output is the receive bits.
vector<double> receiveBits(vector<double> transBit, vector<double> gnoise)
{
    vector<double> recievebits;

    for(int j =0; j<transBit.size(); j++){
        recievebits.insert(recievebits.end(), transBit[j]+gnoise[j]);
    }

    return recievebits;

}


// Function for deciding the bit value from the received bits
// Input is the received bits.
// Output is the decoded bits.
// Decision rule :- if receiveBit >0 then 1 otherwise 0 (simple Binary detection)
vector<double> decisionBlock(vector<double> receiveBits)
{
    vector<double> decodedBits;

    for(int i =0; i<receiveBits.size(); i++){
        if(receiveBits[i]>0){
            decodedBits.insert(decodedBits.end(), 1);
        }
        else{
            decodedBits.insert(decodedBits.end(), 0);
        }
    }

    return decodedBits;
}


// Function to count number of errors in the received bits.
// Inputs are the sourcebits and decodedbits
// OUtput is the number of error in received bits.
// error: if sourcebit  != receivebit
double errorCalculation (vector<double> sourceBits, vector<double> decodedBits)
{
    double countError =0;
    for(int i =0; i<sourceBits.size();i++){
```

```
        if(sourceBits[i]!= decodedBits[i]){
            countError++;
        }
    }

    return countError;
}


// Function to store the data in the file (.dat)
// Input is the SNR per bit in dB and calculated probability of error
// Output is the nothing but in processing it is creating
//a file and writing data into it.
void datafile(vector<double> xindB, vector<double> Prob_error)
{
    ofstream outfile;

    outfile.open("BPSK2.dat");

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

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

    outfile.close();
}


// Function for calculate the Q function values.
// Input is any positive real number.
// Output is the result of erfc function (equal form of Q function).
double Qfunc(double x)
{
    double Qvalue = erfc(x/sqrt(2))/2;
    return Qvalue;
}

vector<double> Qfunction(vector <double> SNR_dB)
{
    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(2*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("BPSK_Qvalue2.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(){

    // source defination
    vector<double> sourceBits;

    // Mapping of bits to symbols;
    vector<double> transmittedSymbol;

    // Noise definition
    vector<double> gnoise;

    // Receive bits
    vector<double> recevBIts;

    // Decision block
    vector<double> decodedBits;


    vector<double> SNR_dB;
    for(float i =0; i<=14; i=i+0.25)
    {
        SNR_dB.insert(SNR_dB.end(), i);
    }

    // N_o corresponds to the variance of noise
    double N_o =4; // can be any positive real number.
    // N_o= 3, 4, 5, 6, 7, 8, 9, 10 my experiment.

    double p, stdnoise;

    vector<double> energyOfSymbol;
    vector<double> Prob_error;
    double normalValue;

```

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

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


    for(int step =0; step<energyOfSymbol.size(); step++){
        sourceBits = sourceVector();

        transmittedSymbol = bit_maps_to_symbol_of_energy_E(sourceBits,
        energyOfSymbol[step]);

        stdnoise = sqrt(N_o/2); // std of noise.
        gnoise = GnoiseVector(0.0, stdnoise);

        recevBIts = receiveBits(transmittedSymbol, gnoise);

        decodedBits = decisionBlock(recevBIts);

        double countErrors = errorCalculation(sourceBits, decodedBits);
        cout<<"Energy of symbol "<< energyOfSymbol[step]<<endl;
        cout<< endl;

        cout<< "Error count "<< countErrors << endl;
        cout<< endl;

        double pe = countErrors/one_million;

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

    }

    datafile(SNR_dB, Prob_error);
    vector<double> qvalue = Qfunction(SNR_dB);
    qvalueInFile(SNR_dB, qvalue);

    return 0;
}
```

Simulated results and calculated results are given below.

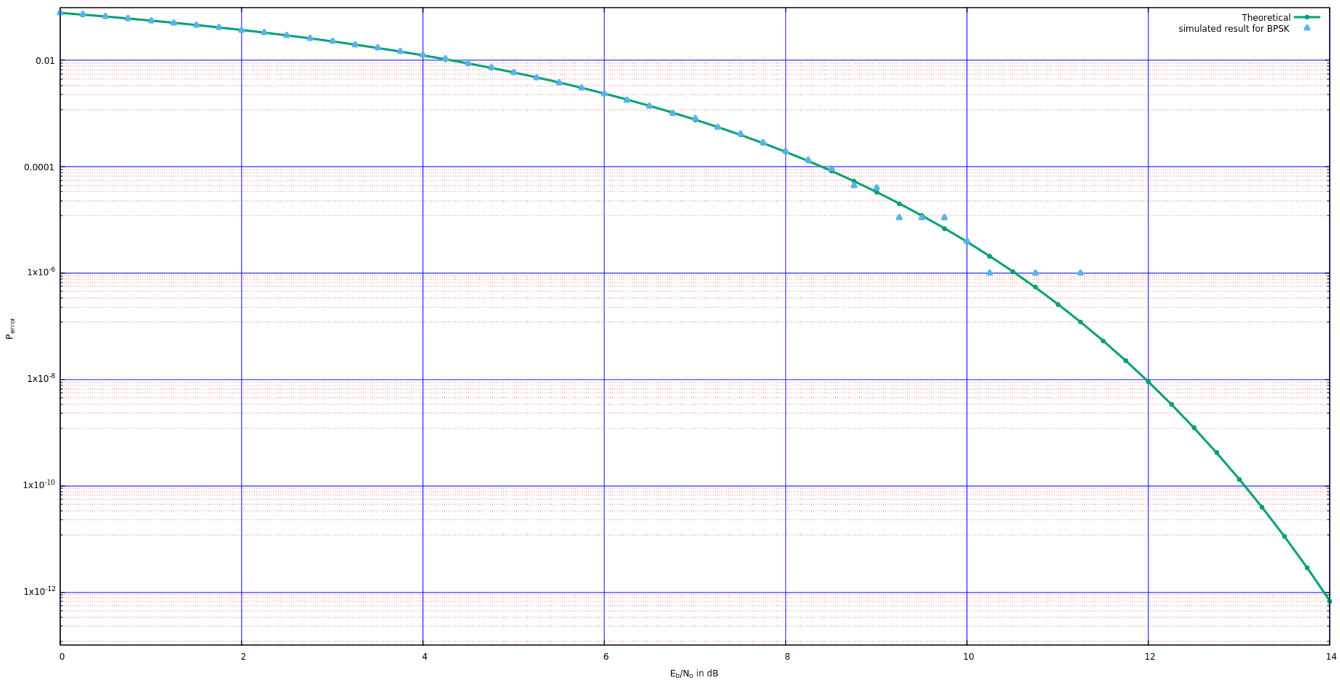

Figure 3: BPSK simulation for 1000000 bits

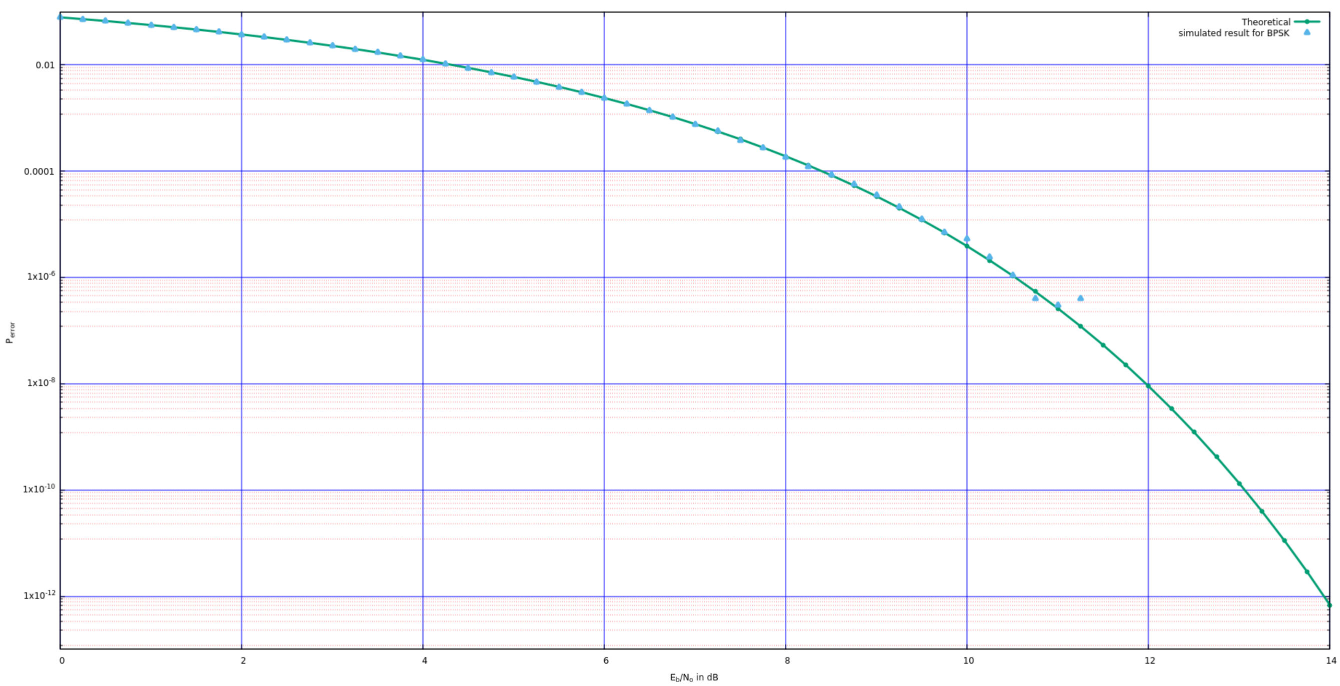

Figure 4: BPSK simulation for 10000000

Here solid line (green color) represent the Q function calculation based on probability of error calculation for BPSK. Blue triangular dots are result of simulation based on block diagram for BPSK.

5 Inference

1. As we increase the $\frac{E_b}{N_o}$ value, then probability of error decrease. That implies, if we transmit signal with large energy as compare to noise energy, one can get very less error in process of communication.
2. Theoretically probability of error in BPSK is a Q-function of SNR. Since, SNR always a positive quantity, that guarantees the output of Q-function would be less than half. Hence, maximum possible error in BPSK is half.
3. For large number of bit transmission at a particular SNR level, theoretical values exactly matches with simulated result. That may not be true for small number of transmission i.e. 100 bits transmission.
4. After SNR =10dB, simulated results for error are exactly zero, that implies, 10dB SNR is sufficient to ensure the no error in BPSK.