ADCC

Experiment-03 (QPSK)

Manas Kumar Mishra (ESD18I011)

07-NOVEMBER-2021

Lab Incharge: Dr. Premkumar Karumbu

1 Aim

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

2 Theory

QPSK is the abbreviation for Quadrature Phase Shift Keying. Quadrature implies four possible values at a time. Phase shift implies change in phase for carrier signal to represent the information. Keying implies the way to represent the information in another form.

In simple terms, system for QPSK consider two bits at a time from information bit stream. Hence, it has four different possibilities. To modulate each possibility in transmitted signal, it changes the phase of the carrier signal in any four discrete values. For a fixed time interval, carrier signal contains fixed phase i.e. phase of the modulated signal. This signal propagate through the channel/medium and captured by receiver or receivers. Based on the phase of received signals, a receiver is supposed to decode the information. That implies, phase difference between any two consecutive phase i.e resolution, should be as maximum as possible.

In this scheme, phases can be any value out of four values with phase difference between two consecutive phase as $\pi/2$ *(Maximum possible phase difference between two consecutive phases set of four phases)*. For this configuration, there are many possibilities to choose four phases out of 360 degree angle. For example, 0, 90, 180, and 270. For this experiment consider 45, 135, 225, and 315.

It is very obvious to say that, two dimensional constellation diagram is required for this scheme. Constellation diagram is given in figure 1.

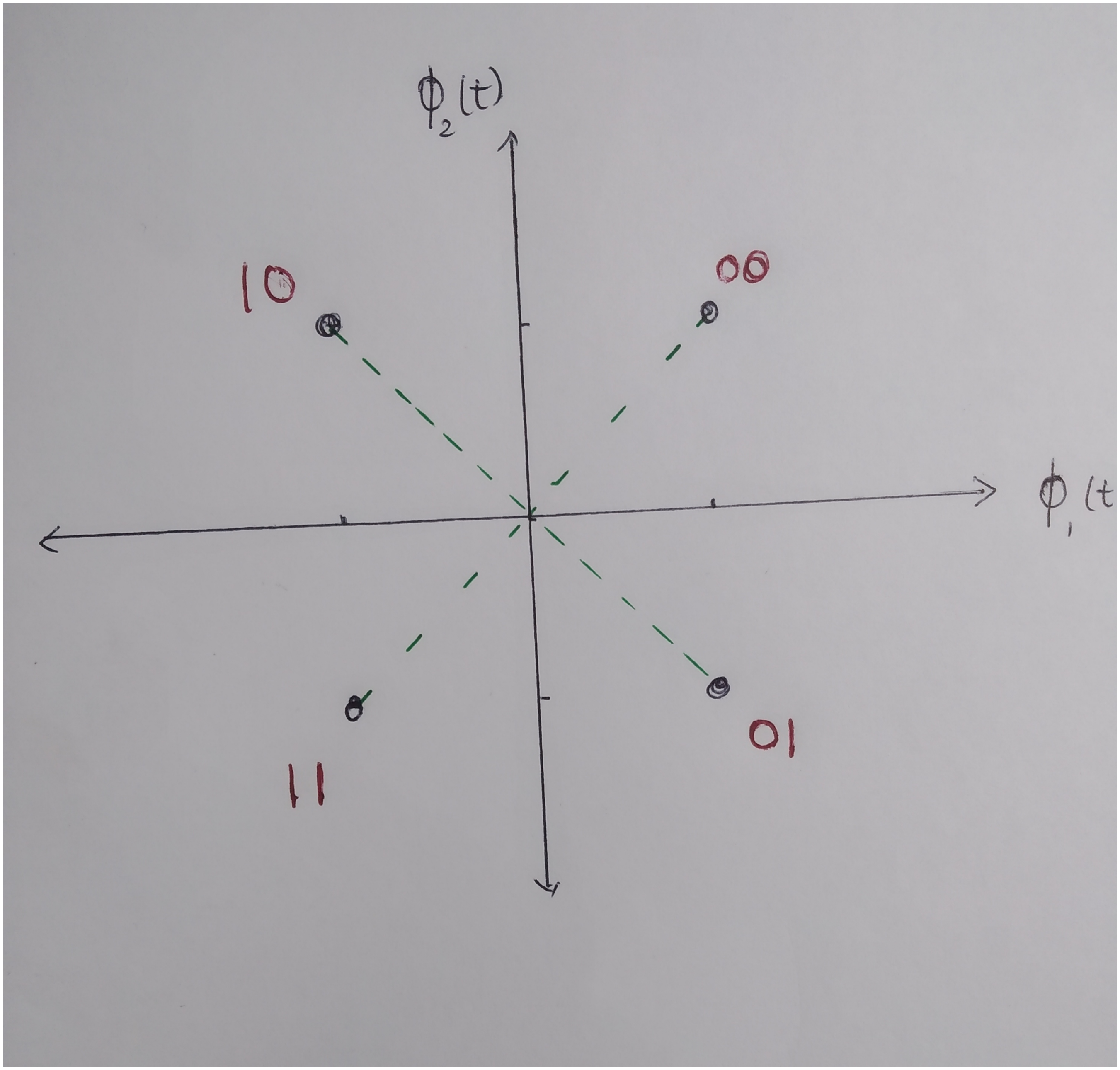


Figure 1: QPSK constellation diagram

symbol	bit	Angle (degree)
0	00	45
1	01	315
2	11	225
3	10	135

3 Design

3.1 Block diagram

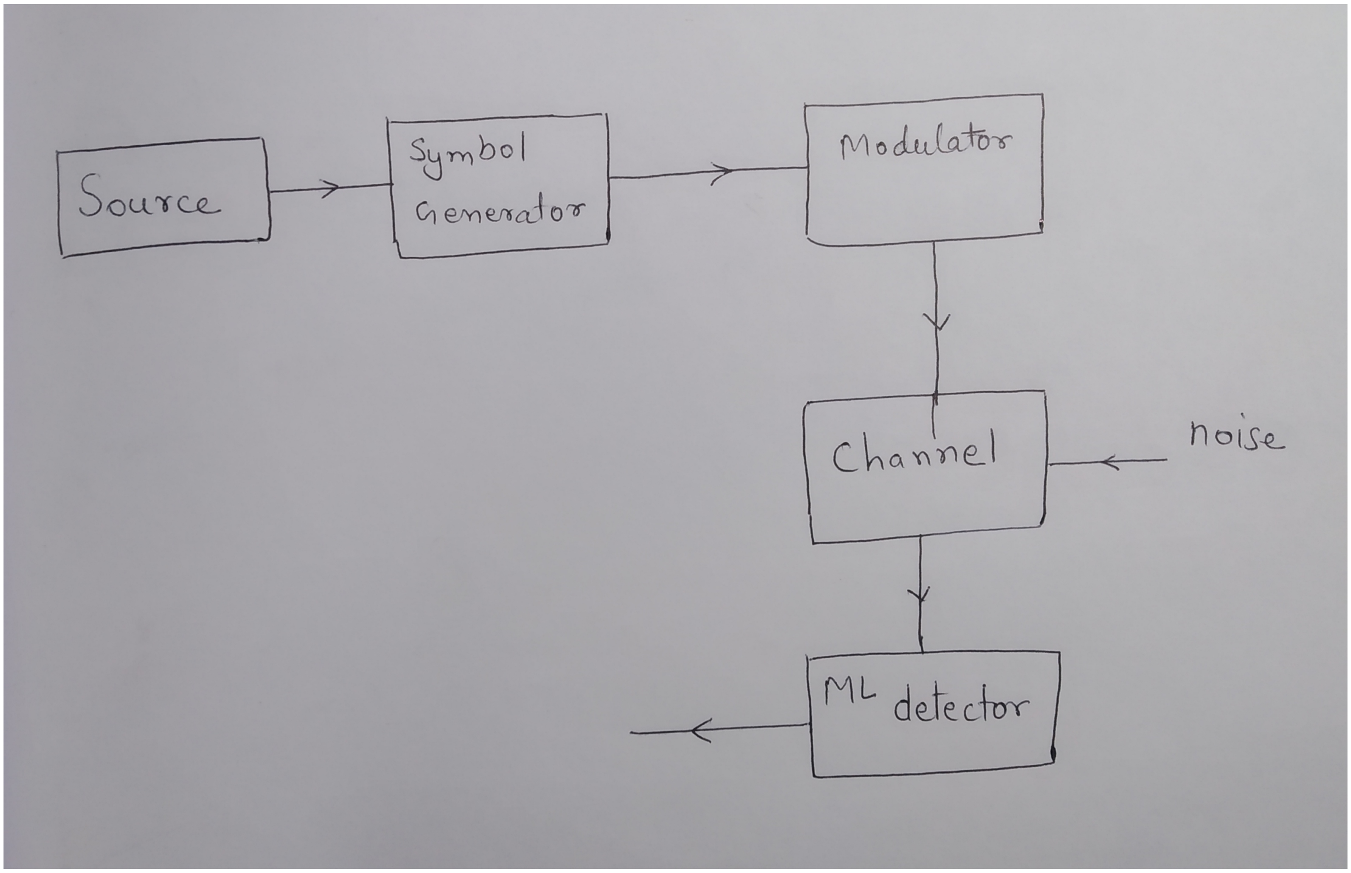
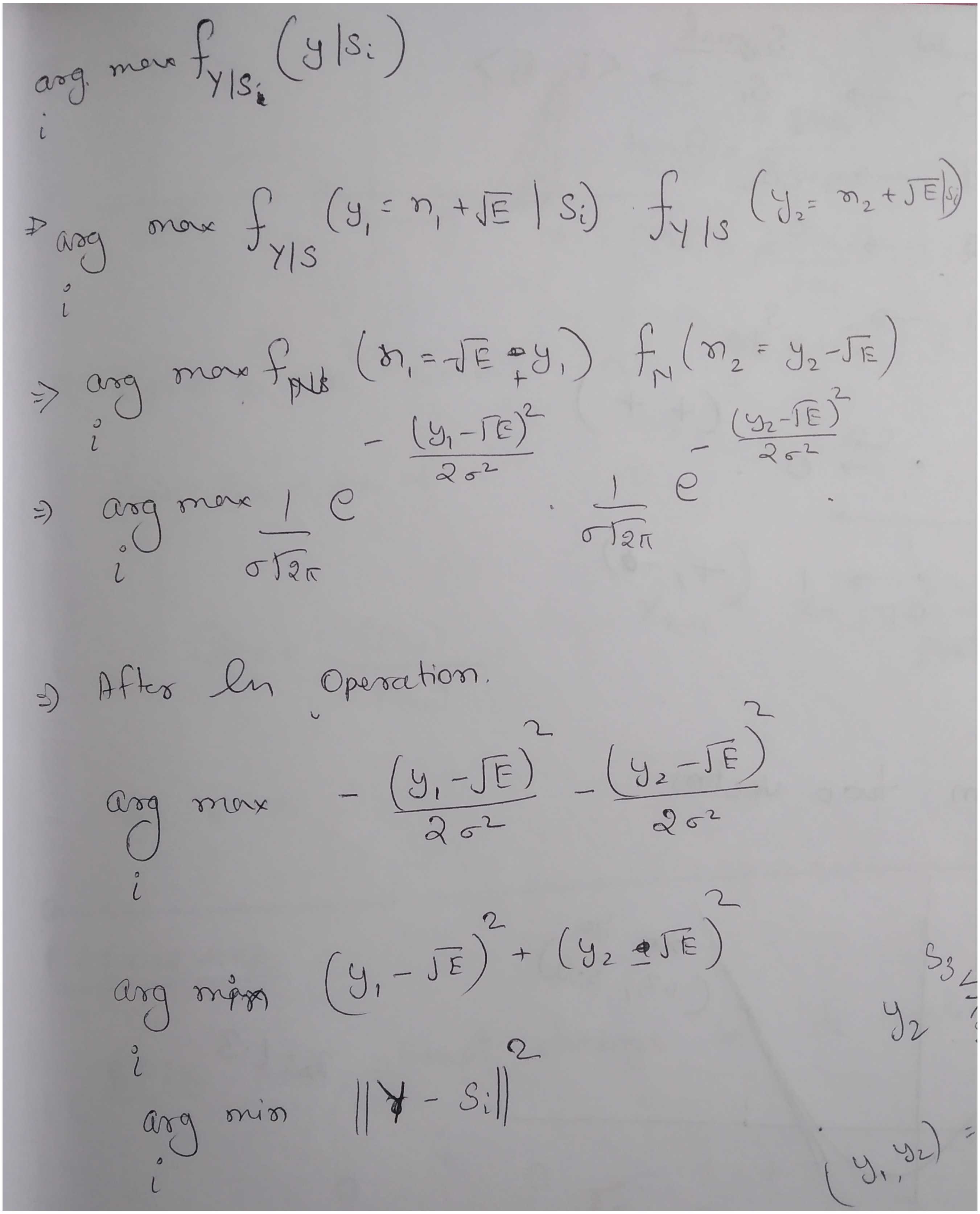


Figure 2: QPSK block diagram

Where source block generates binary bit stream. Symbol generator is the block that takes two bits at a time and convert into symbols. 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.

3.2 ML detector



$$\arg\max_i f_{Y|S_i}(y|s_i)$$

$$\Rightarrow \arg\max_i f_{Y|S}(y_1 = n_1 + \sqrt{E} \mid S_i) \; f_{Y|S}(y_2 = n_2 + \sqrt{E} \mid S_i)$$

$$\Rightarrow \arg\max_i f_{N}(n_1 = -\sqrt{E} + y_1) \; f_N(n_2 = y_2 - \sqrt{E})$$

$$\Rightarrow \arg\max_i \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y_1-\sqrt{E})^2}{2\sigma^2}} \cdot \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y_2-\sqrt{E})^2}{2\sigma^2}}$$

$\Rightarrow$ After ln operation.

$$\arg\max_i -\frac{(y_1-\sqrt{E})^2}{2\sigma^2} - \frac{(y_2-\sqrt{E})^2}{2\sigma^2}$$

$$\arg\min_i (y_1-\sqrt{E})^2 + (y_2 - \sqrt{E})^2$$

$$\arg\min_i \|y - S_i\|^2$$

S_3 y_2 $(y_1, y_2) =$

Figure 3: QPSK ML rule

3.3 Probability of error

In the case,

$$
\begin{aligned}
P(Error) &= 1 - P(Correct) \\
P(Correct) &= P(S_0)P(Correct|S_0) + P(S_1)P(Correct|S_1) + P(S_2)P(Correct|S_2) + P(S_3)P(Correct|S_3)
\end{aligned}
$$

Due to high order of symmetry figure 1

$$P(Correct|S_0) = P(Correct|S_1) = P(Correct|S_2) = P(Correct|S_3)$$

$$
\begin{aligned}
P(Correct|S_0) &= P(y_1 > 0, y_2 > 0|S_0) \\
&= P\left(\sqrt{E} + \eta_1 > 0, \sqrt{E} + \eta_2 > 0\right) \\
&= P\left(\eta_1 > -\sqrt{E}\right) P\left(\eta_2 > -\sqrt{E}\right) \\
&= \left(P(\eta > -\sqrt{E})\right)^2 \\
&= \left(1 - P(\eta > \sqrt{E})\right)^2 \\
&= \left(1 - Q\left(\frac{\sqrt{E}}{\sigma}\right)\right)^2 \\
&= 1 - 2Q\left(\frac{\sqrt{E}}{\sigma}\right) + \left(Q\left(\frac{\sqrt{E}}{\sigma}\right)\right)^2 \\
P(correct) &= P(Correct|S_0)(P(S_0) + P(S_1) + P(S_2) + P(S_3)) \\
P(Correct) &= P(Correct|S_0) \\
P(error) &= 1 - P(Correct) \\
&= 2Q\left(\frac{\sqrt{E}}{\sigma}\right) - \left(Q\left(\frac{\sqrt{E}}{\sigma}\right)\right)^2 \\
P(Error) &= 2Q\left(\frac{\sqrt{E}}{\sigma}\right) - \left(Q\left(\frac{\sqrt{E}}{\sigma}\right)\right)^2
\end{aligned}
$$

4 Code and result

4.1 Code for QPSK in cpp

```cpp
/////////////////////////////////////////////////
//////////////////////////////////////////////////
//
// Author:- MANAS KUMAR MISHRA
// Organisation:- IIITDM KANCHEEPURAM
// Topic:-QPSK (Quadrature phase shift keying)
/////////////////////////////////////////////////
//////////////////////////////////////////////////
//

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

using namespace std;

#define oneMillion 1000000

// Function to generate message symbols (0, 1, 2, 3)
// Input is nothing
// Output is the vector containing symbols
vector<double> source_bits()
{
    vector<double> messageInSymbols;

    srand(time(0));

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

    return messageInSymbols;

}

vector <double> binaryToDecimalConversion(vector<double> sourceBits, int base)
{
    vector <double> convertedBits;
    int start =0;

    if((sourceBits.size()% base) == 0 ){

        int finalSize = sourceBits.size()/base;
        start = 0;
        int conversion;

```

```

        for(int i=0; i<finalSize; i++){
            conversion =0;
            for(int j =base-1; j>-1; j--){
                conversion = conversion + (sourceBits[start])*(pow(2, j));
                start++;
            }
            convertedBits.insert(convertedBits.end(), conversion);
        }

    }
    else{
        int addedBitsNO =  base - (sourceBits.size()%base);

        for(int q=0;q<addedBitsNO;q++){
            sourceBits.insert(sourceBits.end(), 0);
        }

        int finalSize = sourceBits.size()/base;
        start = 0;
        int conversion;


        for(int i=0; i<finalSize; i++){
            conversion =0;
            for(int j =0; j<base; j++){
                conversion = conversion + (sourceBits[start])*(pow(2, j));
                start++;
            }
            convertedBits.insert(convertedBits.end(), conversion);
        }

    }

    return convertedBits;
}



// Functions SignalVector1 and SignalVector2 for convert sourceSymbols to 2D-vector
// Input are the source Symbols and energy of symbols for mapping
// Output is a vector represent signal component
vector<double> SignalVectors1(vector<double> sourceSymbols, double eneryOfSymbols)
{
    vector<double> y1;

    for(int i=0; i<sourceSymbols.size(); i++){
        if(sourceSymbols[i]==0)
        {
            y1.insert(y1.end(), sqrt(eneryOfSymbols));
        }
        else if (sourceSymbols[i]==1)
        {
            y1.insert(y1.end(), sqrt(eneryOfSymbols));
        }
        else if (sourceSymbols[i]==2)
```

```
        {
            y1.insert(y1.end(), -1*sqrt(eneryOfSymbols));
        }
        else
        {
            y1.insert(y1.end(), -1*sqrt(eneryOfSymbols));
        }

    }

    return y1;
}
vector<double> SignalVectors2(vector<double> sourceSymbols, double eneryOfSymbols)
{
    vector<double> y2;

    for(int i=0; i<sourceSymbols.size(); i++){
        if(sourceSymbols[i]==0)
        {
            y2.insert(y2.end(), sqrt(eneryOfSymbols));
        }
        if (sourceSymbols[i]==1)
        {
            y2.insert(y2.end(), -1*sqrt(eneryOfSymbols));
        }
        if (sourceSymbols[i]==2)
        {
            y2.insert(y2.end(), -1*sqrt(eneryOfSymbols));
        }
        if(sourceSymbols[i]==3)
        {
            y2.insert(y2.end(), sqrt(eneryOfSymbols));
        }

    }

    return y2;
}


// 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> noise;

    // 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<oneMillion; i++) {
        noise.insert(noise.end(),dist(generator));
```

```
    }

    return noise;
}


// Function for model the received symbols
// Input sourcesymbols and AWGN
// Output is a vector that represent the addition of two vectors
vector<double> receiveBits(vector<double> sourceSymbols, vector<double> AWGnoise)
{
    vector<double> received;

    for(int i=0; i<sourceSymbols.size(); i++){
        received.insert(received.end(), sourceSymbols[i]+AWGnoise[i]);
    }

    return received;
}


// Function for take decision after receiving the vector
// Input is the 2D receiver vector
// Output is the decision {0, 1, 2, 3}
vector <double> decisionBlock(vector<double> receive1, vector<double> receive2)
{
    vector<double> decision;

    for(int i=0; i<receive1.size(); i++){
        if(receive1[i]>= 0 && receive2[i]>= 0){
            decision.insert(decision.end(), 0);

        }
        else if (receive1[i]>= 0 && receive2[i]<0)
        {
            decision.insert(decision.end(), 1);
        }
        else if (receive1[i] <0 && receive2[i]<0)
        {
            decision.insert(decision.end(), 2);
        }
        else{
            decision.insert(decision.end(), 3);
        }

    }

    return decision;
}


// Function for counting errors in the symbols
// Input are the source symbols and decided bits
// Output is the error count
int errorCount(vector <double> sourceSymbols, vector<double> decisionSymbols)
{
```

```
    int count =0;

    for(int i=0; i<sourceSymbols.size(); i++){
        if(sourceSymbols[i] != decisionSymbols[i]){
            count++;
        }
    }

    return count;
}



// 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> SNR, vector<double> Prob_error)
{
    ofstream outfile;

    outfile.open("QPSK.dat");

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

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

    outfile.close();
}



// Function to compute Q function value.
// Input is the x.
// Output is Q function output.
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 = (2*Qfunc(sqrt(2*normalValue)))-pow(Qfunc(sqrt(2*normalValue)), 2);
        // Defination of Pe from calculations.
```

```
        Qvalue.insert(Qvalue.end(), po);
    }

    return Qvalue;
}


void qvalueInFile(vector <double> SNR, vector <double> Qvalue)
{
    ofstream outfile;

    outfile.open("QPSK_Qvalue.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()
{

    vector<double> SourceBits;


    vector<double> Symbols;

    vector<double> Y1;
    vector<double> Y2;

    vector<double> gnoise1;
    vector<double> gnoise2;

    vector<double> receivedBits1;
    vector<double> receivedBits2;

    vector<double> decision;
    vector<double> EnergyVector;

    vector<double> p_error;

    int Base =2;
    double errors;

   // SNR in dB
    vector<double> SNR_dB;
```

```
    for(float i =0; i<=14; i=i+0.125)
    {
        SNR_dB.insert(SNR_dB.end(), i);
    }



    // copy(begin(SNR), end(SNR), std::ostream_iterator<double>(std::cout, "   "));
    double N_o = 8;
    double stddev = sqrt(N_o/2);
    double pe;


    double normalValue;

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

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



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

        // Source Bits
        SourceBits = source_bits();

        Symbols = binaryToDecimalConversion(SourceBits, Base);

        Y1=SignalVectors1(Symbols, EnergyVector[step]);
        Y2=SignalVectors2(Symbols, EnergyVector[step]);


        // Noise definition
        gnoise1 = GnoiseVector(0.0, stddev);
        gnoise2 = GnoiseVector(0.0, stddev);

        receivedBits1 = receiveBits(Y1, gnoise1);
        receivedBits2 = receiveBits(Y2, gnoise2);

        decision = decisionBlock(receivedBits1, receivedBits2);

        errors = errorCount(Symbols, decision);

        std::cout << "\n";

        cout<< "Error : "<<errors;
        cout<<" \n";

        pe = errors/Symbols.size();

        cout<<"Pe : "<<pe;

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

```

```
    }


    datafile(SNR_dB,p_error);

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

    return 0;
}
```

4.2 Result

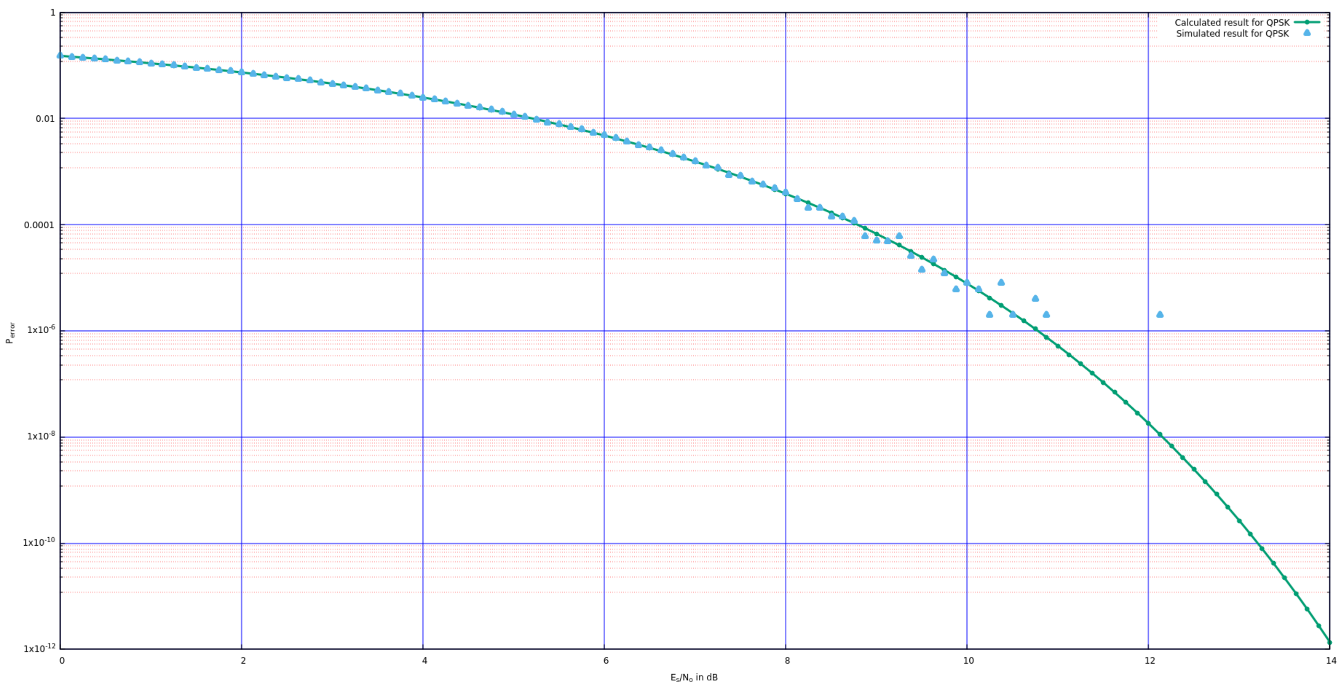


Figure 4: Result for this experiment

Here, blue dots for simulated error in QPSK. Green solid lines is for calculated probability of error.

A comparison study among BPSK, BFSK, and QPSK has been done, result is given below.

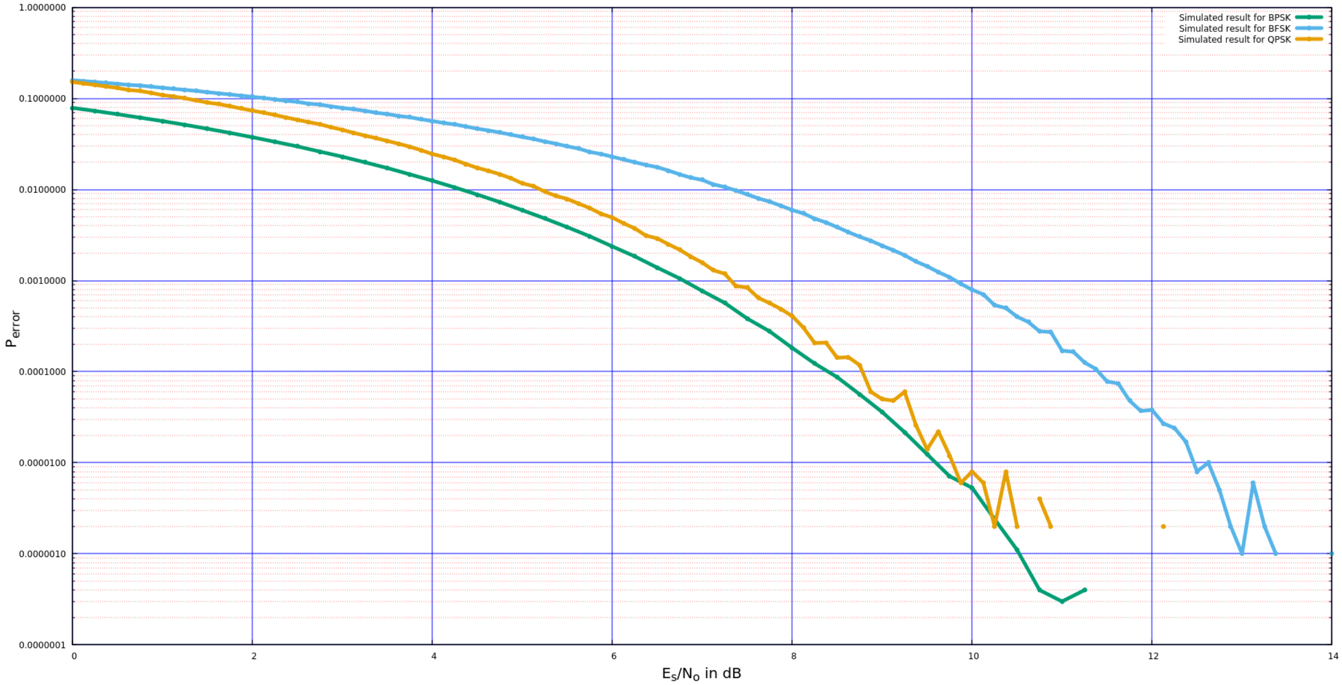


Figure 5: comparison result

5 Inferences

1. As $\frac{E_s}{N_o}$ increases probability of error decreases. Similar to BPSK and BFSK.
2. In this scheme, two bits at a same time can be transmitted for a given SNR. That implies better data rate as compare to BPSK and BFSK.
3. But it possess high error probability for a fixed SNR as compare to BPSK. That implies, higher data in QPSK comes with high error probability.
4. At low SNR values, all four symbols are close to the each other, that makes them vulnerable towards noise addition and go into wrong region. Hence, higher error in communication. But for large SNR, symbols moves away from each other. That implies, less effect of noise addition. Hence, less noise.
5. After SNR= 12dB, simulated results are exactly zero. That implies, SNR = 12dB ensure no error in QPSK.