ADCC

Experiment-05 (QAM)

Manas Kumar Mishra (ESD18I011)

13-NOVEMBER-2021

Lab Incharge: Dr. Premkumar Karumbu

1 Aim

To design discrete time QAM scheme and analyse the performance.

2 Theory

QAM is a special type of digital modulation scheme, in that phase and amplitude of carrier signal varies to represent the symbol. Hence, it is a hybrid modulation scheme.
In practical implementation of QAM, finite amount of bit considered at a time and associate amplitude and phase for carrier signals based on the fixed constellation diagram. Due to this, it can arrange large number of symbols in given energy constrains. That carrier signal propagates through the channel/medium and captured by receiver or receivers. Based on the fixed constellation mapping of amplitude and phase, a receiver is supposed to decode the information symbols out of received signals.
There are many possible QAM. For this experiment consider 2, 3, and 4 bits at a time cases i.e. 4-QAM, 8-QAM, and 16-QAM. Actually, 4-QAM is similar as 4-PSK (QPSK). But 8-QAM and 16-QAM is totally different than MPSK.
Constellation diagrams of 4-QAM, 8-QAM and 16-QAM are given below. Since, every symbols has different projections on the axis of constellation diagram (Basis functions), one has to define average energy for a scheme. In simple terms, average energy is the average of the magnitude of symbol vector from origin in a constellation diagram. Normally, a is defined as $\sqrt{E}$ to simplify the computation in QAM.

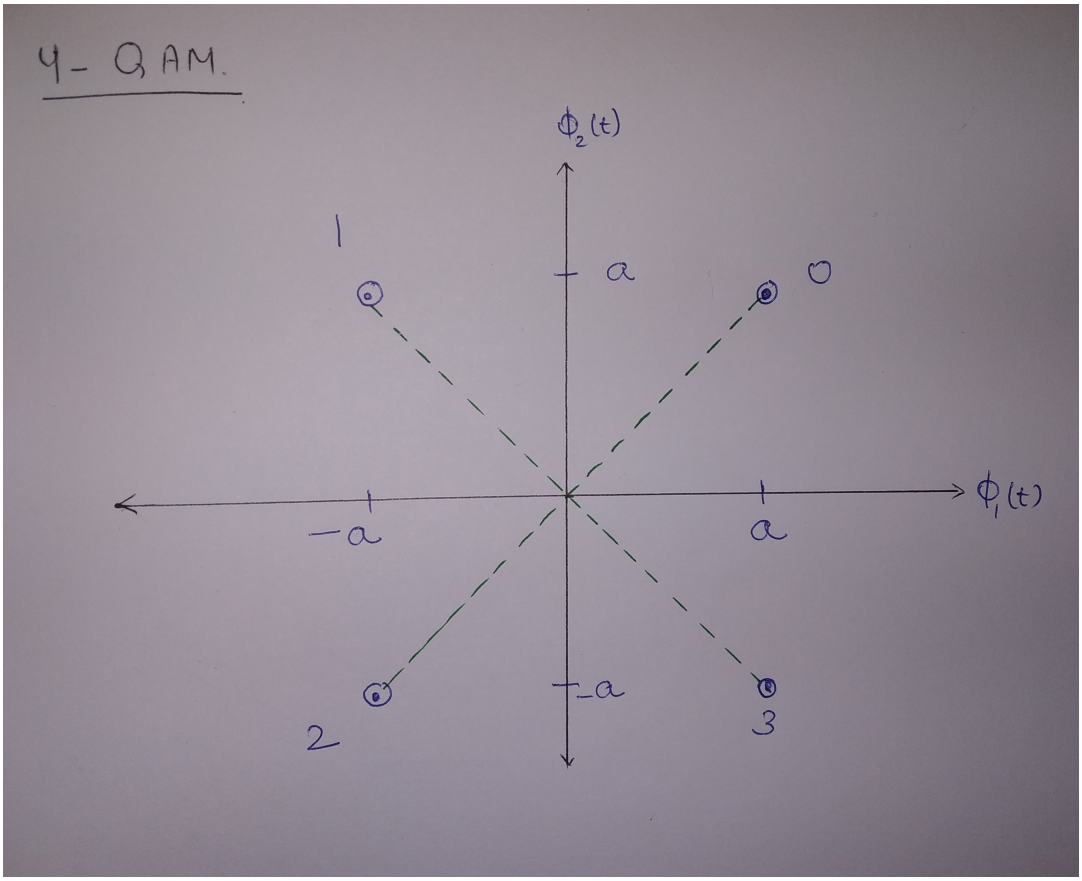

Figure 1: 4-QAM constellation

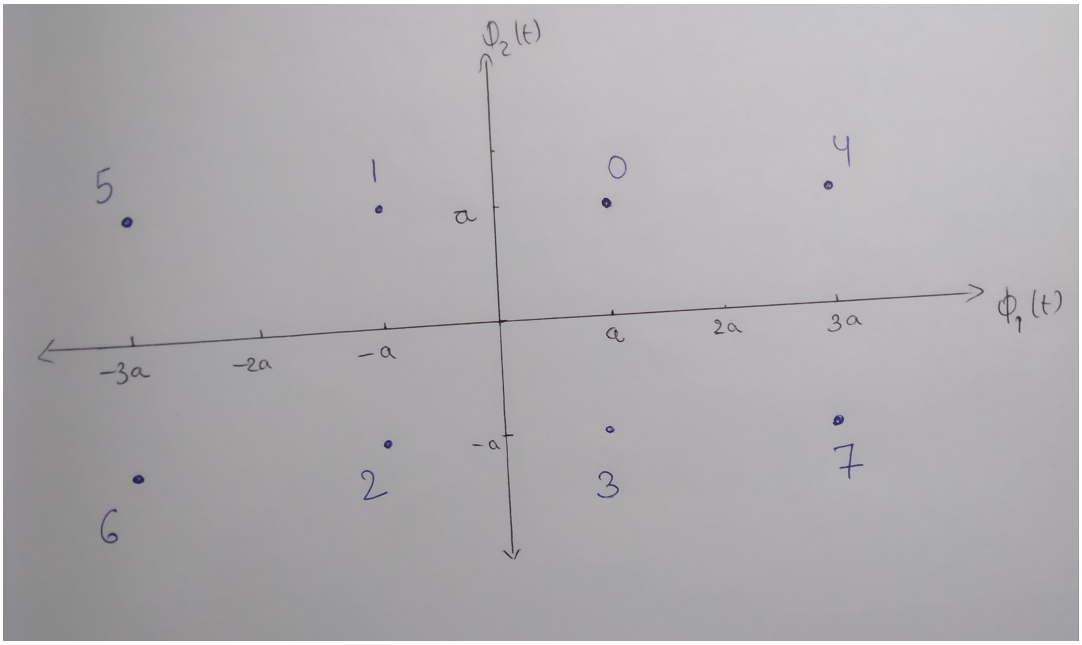

Figure 2: 8-QAM constellation

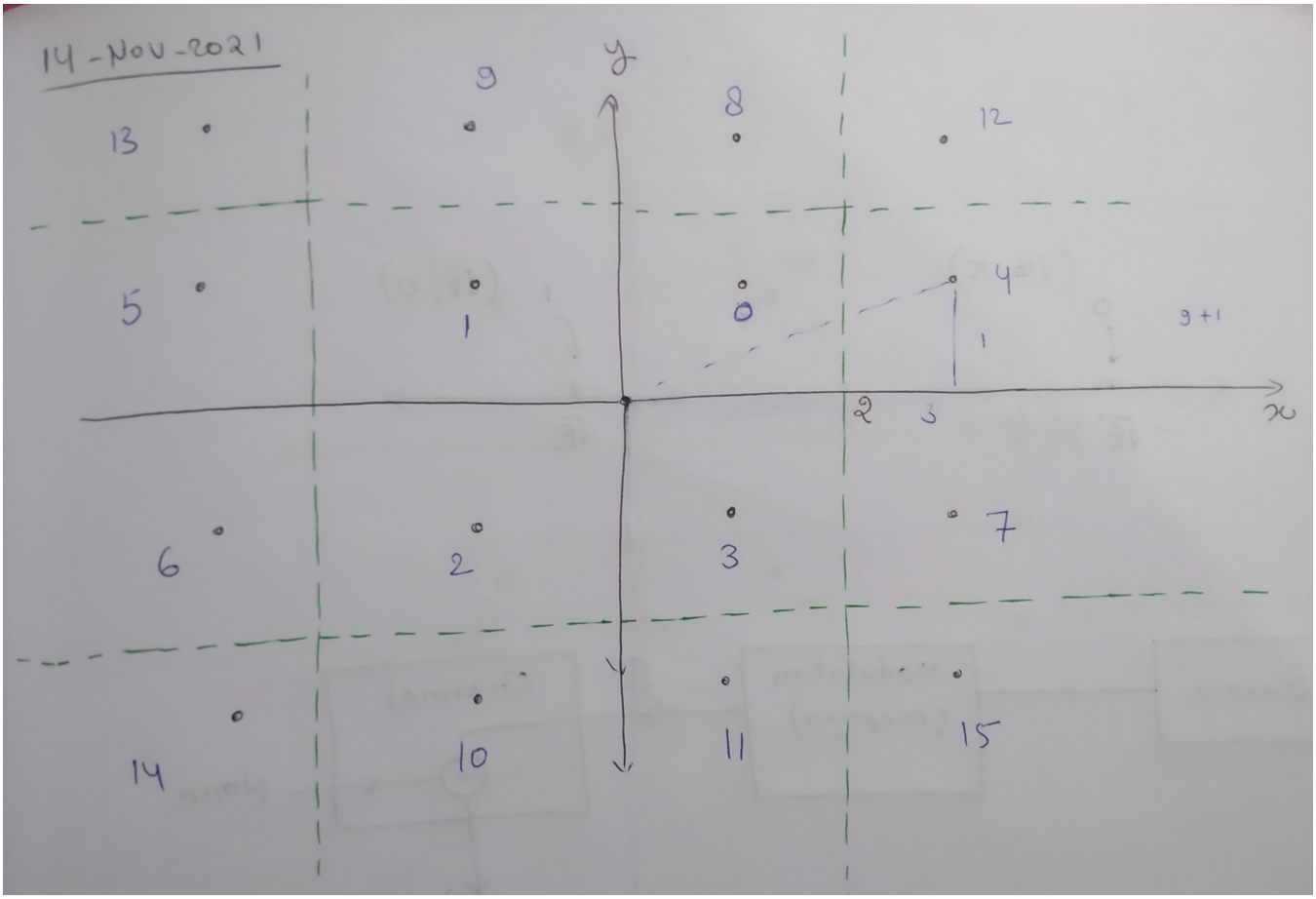

Figure 3: 16-QAM constellation

3 Design

3.1 Block diagram

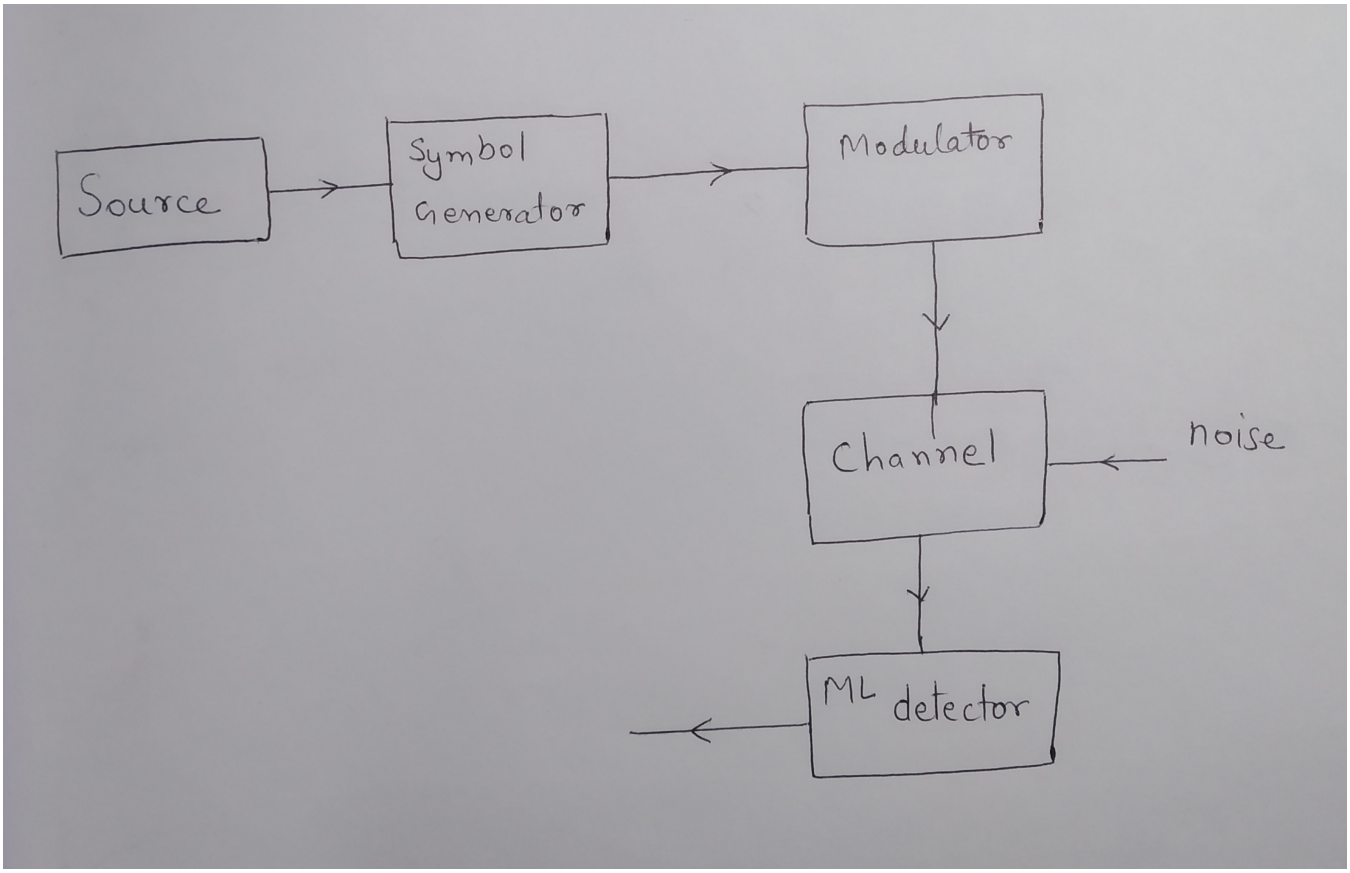

Figure 4: Block diagram

Where source block generates binary bit stream. Symbol generator is the block that takes $log_2 M$ bits at a time and convert into symbols. Modulation block does mapping based on constellation diagram figure 1, 2, and 3 (4-QAM, 8-QAM, and 16-QAM). Channel block is the representation of AWGN (Additive White Gaussian Noise), in that gaussian noise adds up to the signal values. ML detector is a block that receive the signal and take decision about symbols. This a general block for any QAM.

3.2 4-QAM

3.2.1 ML detector

Based on the perpendicular bisector logic, ml rule for 4-QAM is simply quadrant divisions for particular symbol.

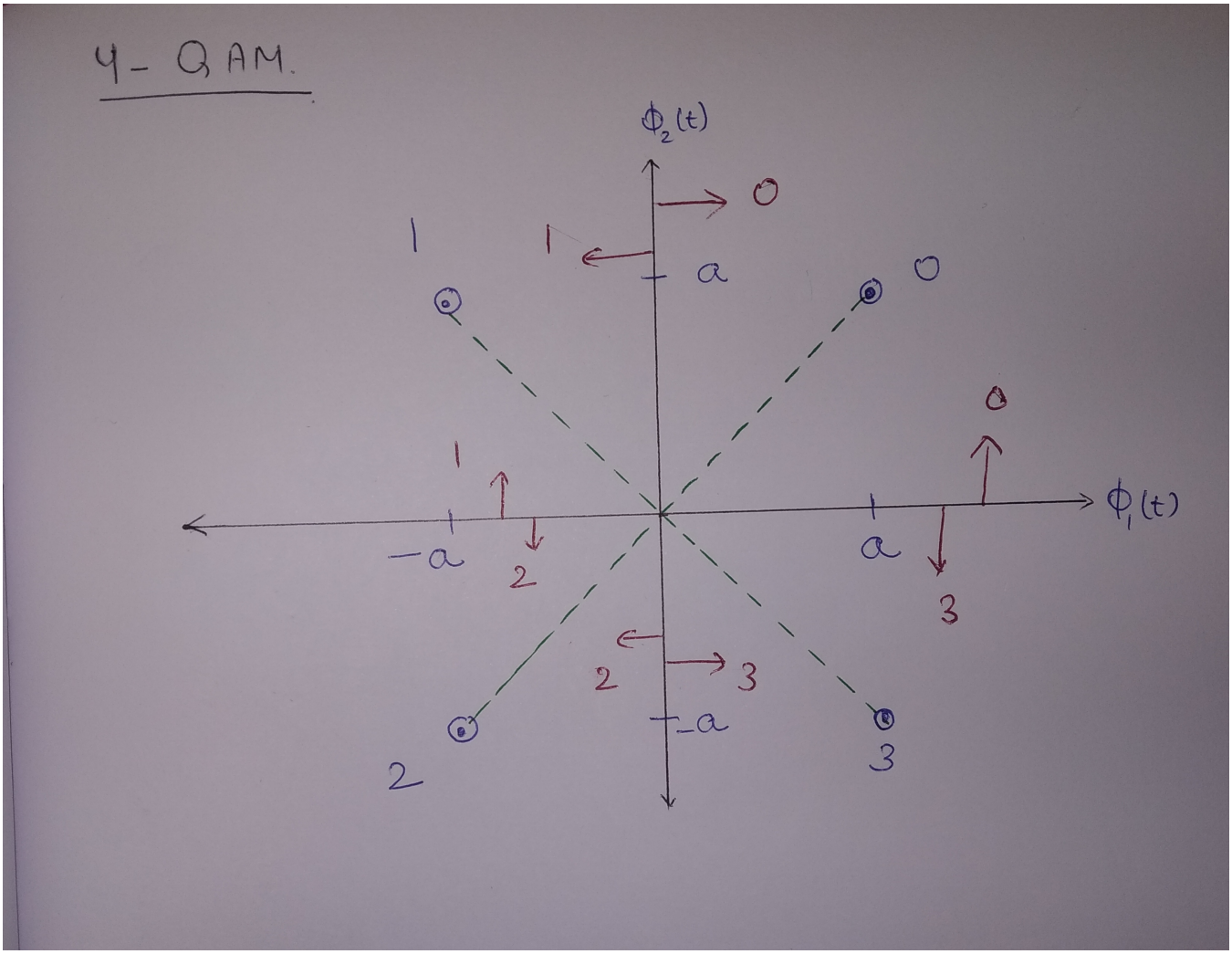

Figure 5: 4-QAM ML detection

3.2.2 Probability of error

In the case,

$$P(Error) = 1 - P(Correct)$$
$$P(Correct) = P(S_0)P(Correct|S_0) + P(S_1)P(Correct|S_1) + P(S_2)P(Correct|S_2) + P(S_3)P(Correct|S_3)$$

Due to high order of symmetry figure 1

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

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

3.2.3 Average energy

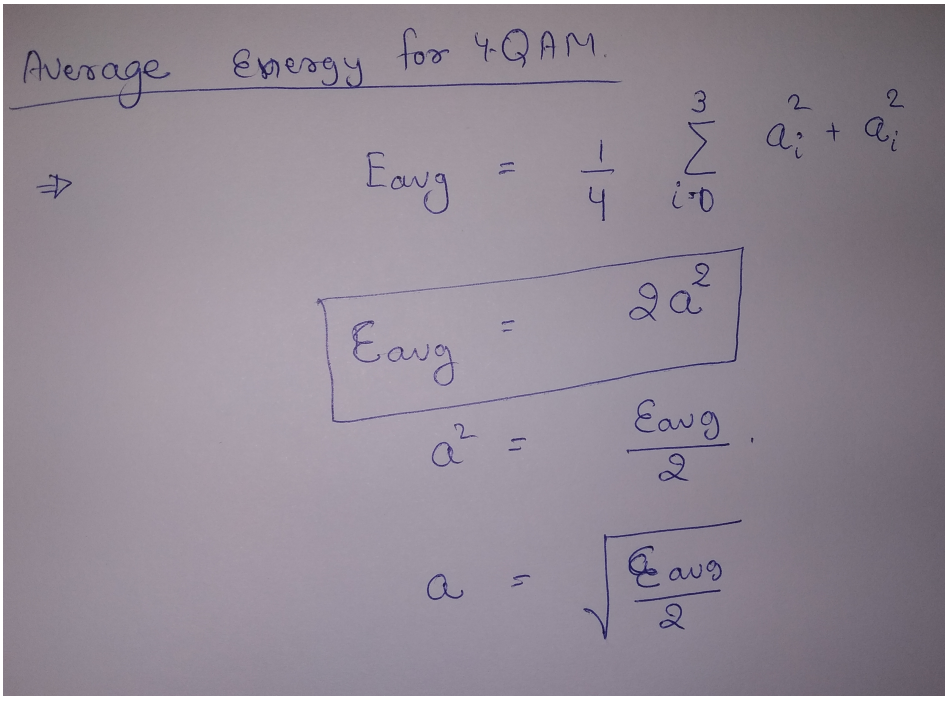

Figure 6: 4-QAM average energy

3.3 8-QAM

3.3.1 ML detector

Based on the perpendicular bisector logic, ml rule for 8-QAM is simply given in image below.

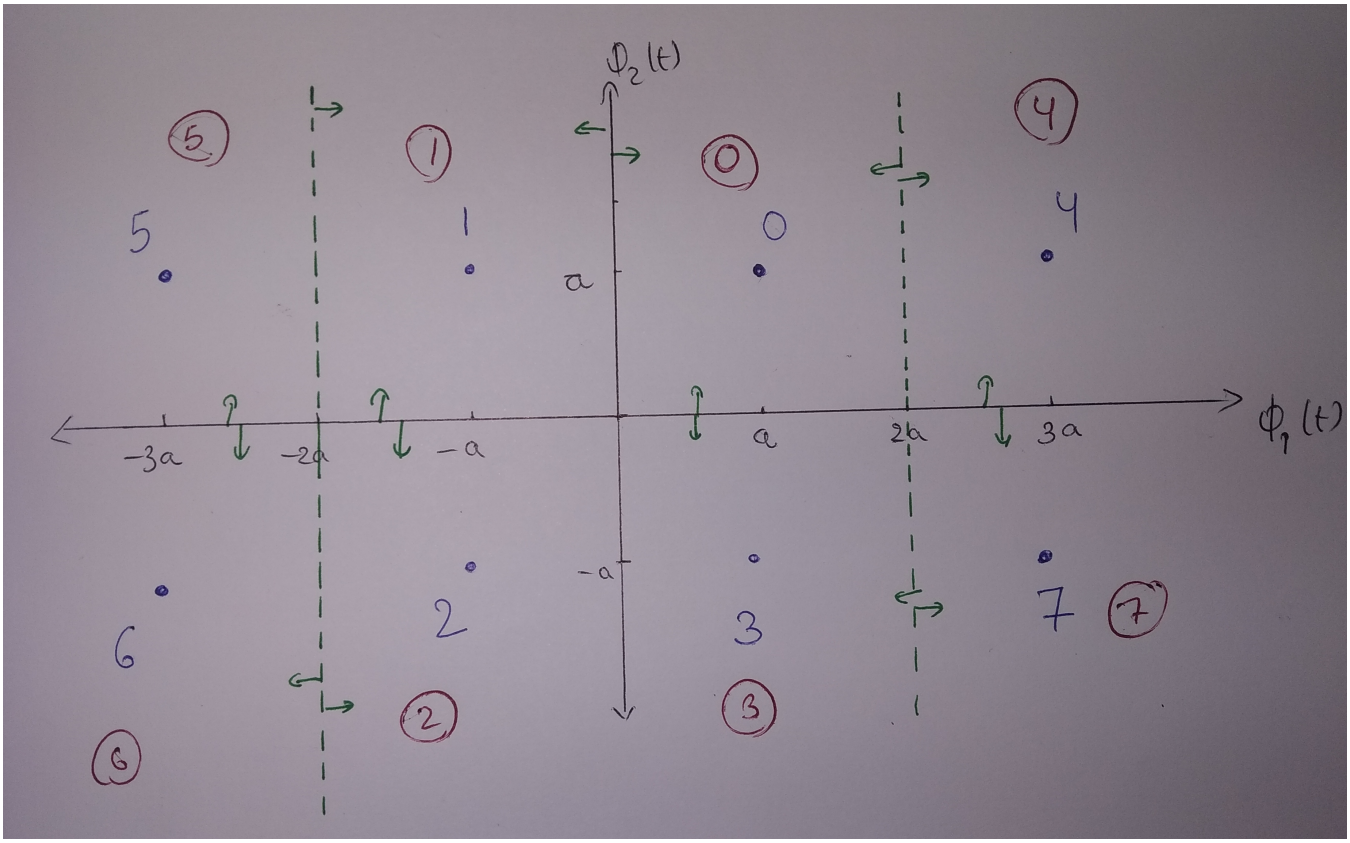

Figure 7: 8-QAM ML detection

3.3.2 Probability of error

In the case,

$$P(Error) = 1 - P(Correct)$$
$$P(Correct) = \sum_{i=0}^{7} P(S_i)P(Correct|S_i)$$

Due to high order of symmetry figure 2

$$P(Correct|S_0) = P(Correct|S_1) = P(Correct|S_2) = P(Correct|S_3)$$
$$P(Correct|S_4) = P(Correct|S_5) = P(Correct|S_6) = P(Correct|S_7)$$

$$
\begin{aligned}
P(Correct|S_0) &= P(0 < y_1 < 2a, y_2 > 0|S_0) \\
&= P(0 < a+\eta_1 < 2a, a+\eta_2 > 0) \\
&= P(-a < \eta_1 < a)\, P(\eta_2 > -a) \\
&= [P(\eta_1 > -a) - P(\eta_1 > a)]\, P(\eta_2 > -a) \\
&= [1 - 2P(\eta_1 > a)]\, [1 - P(\eta_1 > a)] \\
&= \left[1 - 2Q\left(\frac{a}{\sigma}\right)\right] \left[1 - Q\left(\frac{a}{\sigma}\right)\right] \\
&= 1 - 3Q\left(\frac{a}{\sigma}\right) + 2\left(Q\left(\frac{a}{\sigma}\right)\right)^2 \\
P(Correct|S_4) &= P(y_1 > 2a, y_2 > 0|S_0) \\
&= P(3a+\eta_1 > 2a, a+\eta_2 > 0) \\
&= P(\eta_1 > -a)\, P(\eta_2 > -a) \\
&= [1 - P(\eta_1 > a)]\, [1 - P(\eta_2 > a)] \\
&= \left[1 - Q\left(\frac{a}{\sigma}\right)\right] \left[1 - Q\left(\frac{a}{\sigma}\right)\right] \\
&= 1 - 2Q\left(\frac{a}{\sigma}\right) + \left(Q\left(\frac{a}{\sigma}\right)\right)^2 \\
P(Correct) &= P(Correct|S_0) \left(\sum_{i=0}^{3} P(S_i)\right) + P(Correct|S_4) \left(\sum_{i=4}^{7} P(S_i)\right) \\
P(Correct) &= \frac{1}{2}\left[P(Correct|S_0) + P(Correct|S_4)\right] \\
P(Correct) &= \frac{1}{2}\left[1 - 3Q\left(\frac{a}{\sigma}\right) + 2\left(Q\left(\frac{a}{\sigma}\right)\right)^2 + 1 - 2Q\left(\frac{a}{\sigma}\right) + \left(Q\left(\frac{a}{\sigma}\right)\right)^2\right] \\
P(Correct) &= \frac{1}{2}\left[2 - 5Q\left(\frac{a}{\sigma}\right) + 3\left(Q\left(\frac{a}{\sigma}\right)\right)^2\right] \\
P(Error) &= 1 - P(Correct) \\
&= \frac{5}{2}Q\left(\frac{a}{\sigma}\right) - \frac{3}{2}\left(Q\left(\frac{a}{\sigma}\right)\right)^2 \\
P(Error) &= \frac{5}{2}Q\left(\frac{a}{\sigma}\right) - \frac{3}{2}\left(Q\left(\frac{a}{\sigma}\right)\right)^2
\end{aligned}
$$

3.3.3 Average energy

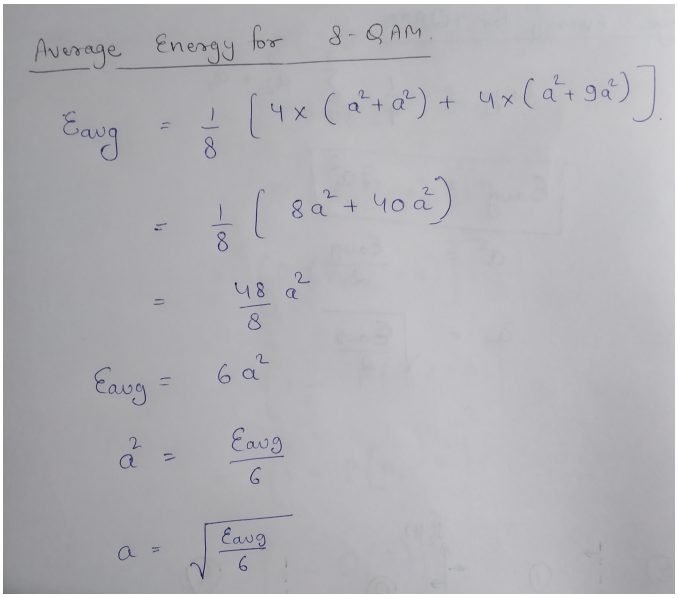

Figure 8: 8-QAM average energy

3.4 16-QAM

3.4.1 ML detector

Based on the perpendicular bisector logic, ml rule for 16-QAM is simply given in image below.

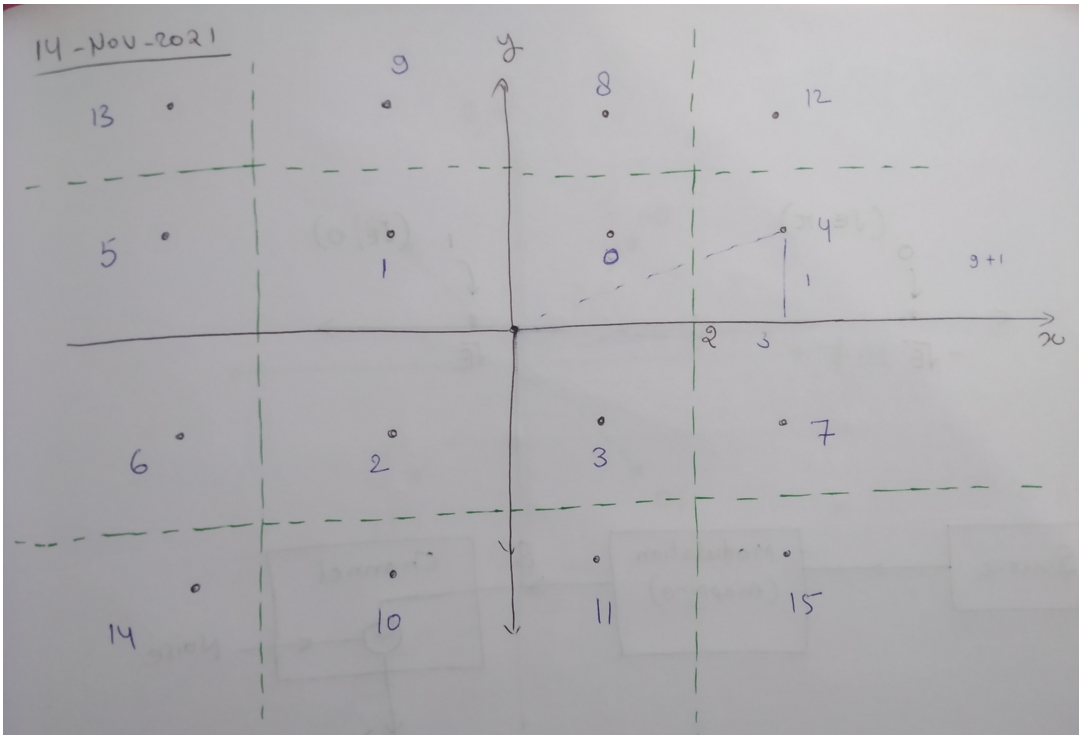

Figure 9: 8-QAM ML detection

3.4.2 Probability of error

In the case,

$$P(Error) = 1 - P(Correct)$$
$$P(Correct) = \sum_{i=0}^{15} P(S_i)P(Correct|S_i)$$

Due to high order of symmetry figure 3

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

$$P(Correct|S_4) = P(Correct|S_5) = P(Correct|S_6) = P(Correct|S_7) = P(Correct|S_8) = P(Correct|S_9) = P(Correct|S_{10}) = P(Correct|S_{11})$$

$$P(Correct|S_{12}) = P(Correct|S_{13}) = P(Correct|S_{14}) = P(Correct|S_{15})$$

$$\begin{aligned} P(Correct|S_0) &= P(0 < y_1 < 2a, 0 < y_2 < 2a|S_0) \\ &= P(0 < a + \eta_1 < 2a, 0 < a + \eta_2 < 2a) \\ &= P(-a < \eta_1 < a) P(-a < \eta_2 < a) \\ &= [P(\eta > -a) - P(\eta > a)]^2 \\ &= [1 - 2P(\eta > a)]^2 \\ &= \left[1 - 2Q\left(\frac{a}{\sigma}\right)\right]^2 \\ &= 1 - 4Q\left(\frac{a}{\sigma}\right) + 4\left(Q\left(\frac{a}{\sigma}\right)\right)^2 \end{aligned}$$

$$\begin{aligned}
P(Correct|S_4) &= P(2a < y_1, 0 < y_2 < 2a|S_4) \\
&= P(2a < 3a + \eta_1, 0 < a + \eta_2 < 2a) \\
&= P(-a < \eta_1) P(-a < \eta_2 < a) \\
&= [1 - P(\eta > a)] [P(\eta > -a) - P(\eta > a)] \\
&= \left[1 - Q\left(\frac{a}{\sigma}\right)\right] \left[1 - 2Q\left(\frac{a}{\sigma}\right)\right] \\
&= 1 - 3Q\left(\frac{a}{\sigma}\right) + 2\left(Q\left(\frac{a}{\sigma}\right)\right)^2
\end{aligned}$$

$$\begin{aligned}
P(Correct|S_{12}) &= P(y_1 > 2a, y_2 > 2a|S_{12}) \\
&= P(3a + \eta_1 > 2a, 3a + \eta_2 > 2a) \\
&= P(\eta_1 > -a) P(\eta_2 > -a) \\
&= [1 - P(\eta > a)]^2 \\
&= \left[1 - Q\left(\frac{a}{\sigma}\right)\right]^2 \\
&= 1 - 2Q\left(\frac{a}{\sigma}\right) + \left(Q\left(\frac{a}{\sigma}\right)\right)^2
\end{aligned}$$

$$\begin{aligned}
P(Correct) &= \sum_{i=0}^{15} P(S_i) P(Correct|S_i) \\
&= \frac{4}{16} P(Correct|S_0) + \frac{8}{16} P(Correct|S_4) + \frac{4}{16} P(Correct|S_{12}) \\
&= \frac{1}{4}\left[1 - 4Q\left(\frac{a}{\sigma}\right) + 4\left(Q\left(\frac{a}{\sigma}\right)\right)^2\right] + \frac{1}{2}\left[1 - 3Q\left(\frac{a}{\sigma}\right) + 2\left(Q\left(\frac{a}{\sigma}\right)\right)^2\right] \\
&+ \frac{1}{4}\left[1 - 2Q\left(\frac{a}{\sigma}\right) + \left(Q\left(\frac{a}{\sigma}\right)\right)^2\right] \\
&= 1 - 3Q\left(\frac{a}{\sigma}\right) + \frac{9}{4}\left(Q\left(\frac{a}{\sigma}\right)\right)^2 \\
P(Error) &= 3Q\left(\frac{a}{\sigma}\right) - \frac{9}{4}\left(Q\left(\frac{a}{\sigma}\right)\right)^2
\end{aligned}$$

3.4.3 Average energy

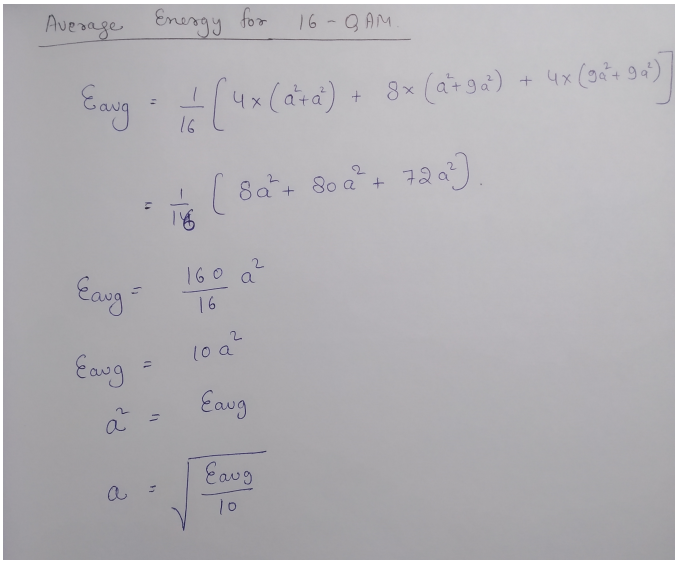

Figure 10: 16-QAM average energy

3.4.4 Summary

Final result:
For 4-QAM

$$P(Error) = 2Q\left(\frac{a}{\sigma}\right) - \left(Q\left(\frac{a}{\sigma}\right)\right)^2$$
$$E_{avg} = 2a^2$$

For 8-QAM

$$P(Error) = \frac{5}{2}Q\left(\frac{a}{\sigma}\right) - \frac{3}{2}\left(Q\left(\frac{a}{\sigma}\right)\right)^2$$
$$E_{avg} = 6a^2$$

For 16-QAM

$$P(Error) = 3Q\left(\frac{a}{\sigma}\right) - \frac{9}{4}\left(Q\left(\frac{a}{\sigma}\right)\right)^2$$
$$E_{avg} = 10a^2$$

4 Code and result

4.1 Code for QAM in cpp

```
////////////////////////////////////////////////////////////////////
//////////////////////////////
// Author:- MANAS KUMAR MISHRA
// Organisation:- IIITDM KANCHEEPURAM
// Topic:- QAM MODEL
////////////////////////////////////////////////////////////////////
//////////////////////////////

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

#define one_million 1000000
#define pi 3.141592

using namespace std;



// This class contains all functions/methods for QAM
class QAM
{
    public:
        vector<double> Source();
        vector<double> TransSymbol_colVect1(vector<double> symbols,
             vector <double> energy, vector <double> angle);
        vector<double> TransSymbol_colVect2(vector<double> symbols,
              vector <double> energy, vector <double> angle);
        vector<double> GnoiseVector(double& mean, double& stddev);
        vector<double> ChannelModel(vector<double> sourceSymbols,
                vector<double> AWGnoise);
        vector <double> DecisionBlock_M4(vector<double> receive1,
                 vector<double> receive2);
        vector <double> DecisionBlock_M8(vector<double> receive1,
                vector<double> receive2, double energy);
        vector <double> DecisionBlock_M16(vector<double> receive1,
                vector<double> receive2, double energy);
        int ErrorCount(vector <double> sourceSymbols, vector<double>
                decisionSymbols);

};



// Function for generating binary bits at source side. Each bit is equiprobable.
// Input is nothing
```

```
51 // Output is a vector that contains the binary bits of length one_million*1.
52 vector<double>QAM::Source()
53 {
54     vector<double> sourceBits;
55 
56     // Use current time as seed for random generator
57     srand(time(0));
58 
59     for(int i = 0; i<one_million; i++){
60         sourceBits.insert(sourceBits.end(), rand()%2);
61     }
62 
63     return sourceBits;
64 }
65 
66 
67 
68 vector <double> EnergyVector(const double& M, double energy)
69 {
70     vector <double> EnergyComponent;
71     double magnitude;
72     if(M==4)
73     {
74         magnitude = sqrt(2)*energy;
75         for(int i =0; i<M; i++){
76             EnergyComponent.insert(EnergyComponent.end(), magnitude);
77         }
78         return EnergyComponent;
79     }
80     if(M==8){
81         magnitude = sqrt(2)*energy;
82         for(int i =0; i<4; i++){
83             EnergyComponent.insert(EnergyComponent.end(), magnitude);
84         }
85         magnitude = sqrt(10)*energy;
86         for(int i =4; i<M; i++){
87             EnergyComponent.insert(EnergyComponent.end(), magnitude);
88         }
89         return EnergyComponent;
90     }
91     if(M==16){
92         magnitude = sqrt(2)*energy;
93         for(int i =0; i<4; i++){
94             EnergyComponent.insert(EnergyComponent.end(), magnitude);
95         }
96         magnitude = sqrt(10)*energy;
97         for(int i =4; i<12; i++){
98             EnergyComponent.insert(EnergyComponent.end(), magnitude);
99         }
100        magnitude = sqrt(18)*energy;
101        for(int i =12; i<M; i++){
102            EnergyComponent.insert(EnergyComponent.end(), magnitude);
103        }
104        return EnergyComponent;
105    }
106    else{
```

```
        EnergyComponent.insert(EnergyComponent.end(), 0);
        return EnergyComponent;
    }
}



vector<double> AngleVector(const double& M )
{
    vector<double> angles;
    double angle;
    if(M == 4){
        angle = atan2(1,1);
        for(int i =0; i<M; i++){
            angles.insert(angles.end(), angle*180/pi);
        }
        return angles;
    }
    if(M == 8){
        angle = atan2(1,1);
        for(int i =0; i<4; i++){
            angles.insert(angles.end(), angle*180/pi);
        }
        angle = atan2(1,3);
        for(int i =4; i<M; i++){
            angles.insert(angles.end(), angle*180/pi);
        }
        return angles;
    }

    if(M == 16){
        angle = atan2(1,1);
        for(int i =0; i<4; i++){
            angles.insert(angles.end(), angle*180/pi);
        }
        angle = atan2(1,3);
        for(int i =4; i<8; i++){
            angles.insert(angles.end(), angle*180/pi);
        }
        angle = atan2(3,1);
        for(int i =8; i<12; i++){
            angles.insert(angles.end(), angle*180/pi);
        }
        angle = atan2(1,1);
        for(int i= 12; i<M; i++){
            angles.insert(angles.end(), angle*180/pi);
        }

        return angles;
    }
    else{
        angles.insert(angles.end(), 0);
        return angles;
    }
}

```

```
163
164
165 // Function to generate symbols from the binary bit stream.
166 // Input is the binary bit vector and base for symbol conversion
167 // Output is the symbol vector containing values from (0, 2^(base-1))
168 vector <double> binaryToDecimalConversion(vector<double> sourceBits, const int& base)
169 {
170     vector <double> convertedBits;
171     int start =0;
172
173     if((sourceBits.size()% base) == 0 ){
174
175         int finalSize = sourceBits.size()/base;
176         start = 0;
177         int conversion;
178
179
180         for(int i=0; i<finalSize; i++){
181             conversion =0;
182             for(int j =base-1; j>-1; j--){
183                 conversion = conversion + (sourceBits[start])*(pow(2, j));
184                 start++;
185             }
186             convertedBits.insert(convertedBits.end(), conversion);
187         }
188
189     }
190     else{
191         int addedBitsNO =  base - (sourceBits.size()%base);
192
193         for(int q=0;q<addedBitsNO;q++){
194             sourceBits.insert(sourceBits.end(), 0);
195         }
196
197         int finalSize = sourceBits.size()/base;
198         start = 0;
199         int conversion;
200
201
202         for(int i=0; i<finalSize; i++){
203             conversion =0;
204             for(int j =0; j<base; j++){
205                 conversion = conversion + (sourceBits[start])*(pow(2, j));
206                 start++;
207             }
208             convertedBits.insert(convertedBits.end(), conversion);
209         }
210
211     }
212
213     return convertedBits;
214 }
215
216
217
218 double radianConv(double degree){
```

```
219     double rad = degree*pi/180;
220
221     return rad;
222 }
223
224
225
226 vector<double>QAM::TransSymbol_colVect1(vector<double> symbols,
227             vector <double> energy, vector <double> angle)
228 {
229     vector<double> y1;
230     double egy, ang;
231     int index;
232     for(int step = 0; step<symbols.size(); step++){
233         index = symbols[step];
234         egy = energy[index];
235
236         if((index)%4 == 0){
237             ang = angle[index];
238         }
239         else if((index)%4 == 1){
240             ang = 180- angle[index] ;
241         }
242         else if((index)%4 == 2){
243             ang = angle[index] + 180;
244         }
245         else{
246             ang = -1*angle[index] + 360;
247         }
248
249         ang = radianConv(ang);
250
251         y1.insert(y1.end(), egy*cos(ang));
252
253     }
254
255     return y1;
256
257 }
258
259
260
261 vector<double>QAM::TransSymbol_colVect2(vector<double> symbols,
262             vector <double> energy, vector <double> angle)
263 {
264     vector<double> y2;
265     double egy, ang;
266     int index;
267     for(int step = 0; step<symbols.size(); step++){
268         index = symbols[step];
269         egy = energy[index];
270
271         if((index)%4 == 0){
272             ang = angle[index];
273         }
274         else if((index)%4 == 1){
```

```
            ang = 180-angle[index];
        }
        else if((index)%4 == 2){
            ang = angle[index] + 180;
        }
        else{
            ang = -1*angle[index] +360;
        }

        ang = radianConv(ang);

        y2.insert(y2.end(), egy*sin(ang));

    }

    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>QAM::GnoiseVector(double& mean, double& stddev)
{
    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<=one_million; i++) {
        noise.insert(noise.end(),dist(generator));
    }

    return noise;
}




// Function for model the channel
// Input sourcesymbols and AWGN
// Output is a vector that represent the addition of two vectors
vector<double>QAM::ChannelModel(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]);
    }
```

```
331
332     return received;
333 }
334
335
336
337
338 // Function for take decision after receiving the vector
339 // Input is the 2D receiver vector
340 // Output is the decision {0, 1}
341 vector <double>QAM::DecisionBlock_M4(vector<double> receive1, vector<double> receive2)
342 {
343     vector<double> decision;
344
345     for(int j =0; j<receive1.size(); j++){
346
347         if(receive1[j]>=0 && receive2[j]>=0){
348             decision.insert(decision.end(), 0);
349         }else if(receive1[j]< 0 && receive2[j]>=0){
350             decision.insert(decision.end(), 1);
351         }else if(receive1[j]< 0 && receive2[j]<0){
352             decision.insert(decision.end(), 2);
353         }else{
354             decision.insert(decision.end(), 3);
355         }
356     }
357
358     return decision;
359 }
360
361
362
363 // Function for take decision after receiving the vector
364 // Input is the 2D receiver vector
365 // Output is the decision {0, 1}
366 vector <double>QAM::DecisionBlock_M8(vector<double> receive1,
367                 vector<double> receive2, double energy)
368 {
369     vector<double> decision;
370
371     for(int j =0; j<receive1.size(); j++){
372
373         if(receive1[j]>=0 && receive1[j]<2*energy && receive2[j]>=0){
374             decision.insert(decision.end(), 0);
375         }else if(receive1[j]<0 && receive1[j]>-2*energy && receive2[j]>=0){
376             decision.insert(decision.end(), 1);
377         }else if(receive1[j]<0 && receive1[j]>-2*energy && receive2[j]<0){
378             decision.insert(decision.end(), 2);
379         }else if(receive1[j]>=0 && receive1[j]<2*energy && receive2[j]<0){
380             decision.insert(decision.end(), 3);
381         }else if(receive1[j]>=2*energy && receive2[j]>=0){
382             decision.insert(decision.end(), 4);
383         }else if(receive1[j]<= -2*energy && receive2[j]>=0){
384             decision.insert(decision.end(), 5);
385         }else if(receive1[j]<= -2*energy && receive2[j]<0){
386             decision.insert(decision.end(), 6);
```

```
387         }else{
388             decision.insert(decision.end(), 7);
389         }
390     }
391
392     return decision;
393 }
394
395
396
397 // Function for take decision after receiving the vector
398 // Input is the 2D receiver vector
399 // Output is the decision {0, 1}
400 vector <double>QAM::DecisionBlock_M16(vector<double> receive1,
401             vector<double> receive2, double energy)
402 {
403     vector<double> decision;
404
405     for(int j =0; j<receive1.size(); j++){
406
407         if(receive1[j]>=0 && receive1[j]<2*energy && receive2[j]>=0
408             && receive2[j]<2*energy){
409             decision.insert(decision.end(), 0);
410         }else if(receive1[j]<0 && receive1[j]>-2*energy
411             && receive2[j]>=0 && receive2[j]<2*energy){
412             decision.insert(decision.end(), 1);
413         }else if(receive1[j]<0 && receive1[j]>=-2*energy
414             && receive2[j]<0 && receive2[j]>=-2*energy){
415             decision.insert(decision.end(), 2);
416         }else if(receive1[j]>=0 && receive1[j]<2*energy
417             && receive2[j]<0 && receive2[j]>=-2*energy){
418             decision.insert(decision.end(), 3);
419         }else if(receive1[j]>=2*energy && receive2[j]>=0 && receive2[j]<2*energy){
420             decision.insert(decision.end(), 4);
421         }else if(receive1[j]>=0 &&receive1[j]<2*energy &&  receive2[j]>=2*energy){
422             decision.insert(decision.end(), 8);
423         }else if(receive1[j]<0 &&receive1[j]>=-2*energy &&  receive2[j]>=2*energy){
424             decision.insert(decision.end(), 9);
425         }else if(receive1[j]<-2*energy &&  receive2[j]>=0 &&receive2[j]<2*energy){
426             decision.insert(decision.end(), 5);
427         }else if(receive1[j]<-2*energy &&  receive2[j]<0 &&receive2[j]>=-2*energy){
428             decision.insert(decision.end(), 6);
429         }else if(receive1[j]>=-2*energy &&  receive1[j]<0 &&receive2[j]<-2*energy){
430             decision.insert(decision.end(), 10);
431         }else if(receive1[j]<2*energy &&  receive1[j]>=0 &&receive2[j]<-2*energy){
432             decision.insert(decision.end(), 11);
433         }else if(receive1[j]>=2*energy &&  receive2[j]<0 &&receive2[j]>=-2*energy){
434             decision.insert(decision.end(), 7);
435         }else if(receive1[j]>=2*energy &&  receive2[j]>=2*energy){
436             decision.insert(decision.end(), 12);
437         }else if(receive1[j]<-2*energy &&  receive2[j]>=2*energy){
438             decision.insert(decision.end(), 13);
439         }else if(receive1[j]<-2*energy &&  receive2[j]<-2*energy){
440             decision.insert(decision.end(), 14);
441         }else if(receive1[j]>=2*energy &&  receive2[j]<-2*energy){
442             decision.insert(decision.end(), 15);
```

```
443         }
444     }
445 
446     return decision;
447 }
448 
449 
450 
451 
452 // Function for counting errors in the symbols
453 // Input are the source symbols and decided bits
454 // Output is the error count
455 int QAM::ErrorCount(vector <double> sourceSymbols, vector<double> decisionSymbols)
456 {
457     int count =0;
458 
459     for(int i=0; i<sourceSymbols.size(); i++){
460         if(sourceSymbols[i] != decisionSymbols[i]){
461             count++;
462         }
463     }
464 
465     return count;
466 }
467 
468 
469 
470 
471 // Function to store the data in the file (.dat)
472 // Input is the SNR per bit in dB and calculated probability of error
473 // Output is the nothing but in processing it is creating a file and writing data into it.
474 void datafile(vector<double> xindB, vector<double> Prob_error)
475 {
476     ofstream outfile;
477 
478     outfile.open("QAM_M16_final.dat");
479 
480     if(!outfile.is_open()){
481         cout<<"File opening error !!!"<<endl;
482         return;
483     }
484 
485     for(int i =0; i<xindB.size(); i++){
486         outfile<< xindB[i] << " "<<"\t"<<" "<< Prob_error[i]<< endl;
487     }
488 
489     outfile.close();
490 }
491 
492 
493 
494 // Function for calculate the Q function values.
495 // Input is any positive real number.
496 // Output is the result of erfc function (equal form of Q function).
497 double Qfunc(double x)
498 {
```

```
    double Qvalue = erfc(x/sqrt(2))/2;
    return Qvalue;
}



vector<double> Qfunction_M4(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(normalValue))-pow(Qfunc(sqrt(1*normalValue)),2);
        Qvalue.insert(Qvalue.end(), po);
    }

    return Qvalue;
}


vector<double> Qfunction_M8(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 = ((5*(Qfunc(sqrt(normalValue/3))))-(3*pow(Qfunc(sqrt(normalValue/3)), 2)))/2;
        Qvalue.insert(Qvalue.end(), po);
    }

    return Qvalue;
}


vector<double> Qfunction_M16(vector <double> SNR_dB)
{
    vector <double> Qvalue;
    double po, normalValue, value;
    for (int k =0; k<SNR_dB.size(); k++){
        normalValue = pow(10, (SNR_dB[k]/10));
        value = Qfunc(sqrt(normalValue/5));
        po = ((12*value)-(9*pow(value, 2)))/4;
        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;
```

```
555
556     outfile.open("QAM_Qvalue_M16.dat");
557
558     if(!outfile.is_open()){
559         cout<<"File opening error !!!"<<endl;
560         return;
561     }
562
563     for(int i =0; i<SNR.size(); i++){
564         outfile<< SNR[i] << " "<<"\t"<<" "<< Qvalue[i]<< endl;
565     }
566
567     outfile.close();
568 }
569
570
571
572 int main(){
573     // Object of class
574     QAM qam1;
575
576     int base = 4;
577     double M = pow(2, base);
578     double mean =0.0;
579     double N_o =8;
580     double stddev = sqrt(N_o/2);
581     double errors=0;
582     double sizeOfTrans, pe, a;
583
584     // source defination
585     vector<double> sourceBits;
586
587     vector<double> EnergyComponent;
588     vector<double> AngleComponent;
589
590     vector<double> Symbols;
591
592     // Transmiited vectors
593     vector<double> trans1;
594     vector<double> trans2;
595
596     // Noise vector
597     vector<double> gnoise1;
598     vector<double> gnoise2;
599
600     // Receive bits
601     vector<double> receivedBits1;
602     vector<double> receivedBits2;
603
604     // Decision block
605     vector<double> decodedBits;
606
607
608     // SNR in dB
609     vector<double> SNR_dB;
610     for(float i =0; i<=18; i=i+0.125)
```

```
611     {
612         SNR_dB.insert(SNR_dB.end(), i);
613     }
614 
615 
616     vector<double> energyOfSymbols;
617     vector<double> Prob_error;
618 
619     double normalValue;
620 
621     for(int i =0; i<SNR_dB.size(); i++){
622 
623         normalValue = pow(10, (SNR_dB[i]/10));
624         energyOfSymbols.insert(energyOfSymbols.end(), N_o*normalValue);
625     }
626 
627 
628     for(int step =0; step<SNR_dB.size(); step++){
629         sourceBits = qam1.Source();
630 
631         if(base==2){
632             a = sqrt(energyOfSymbols[step]/2);
633         }else if(base ==3){
634             a = sqrt(energyOfSymbols[step]/6);
635         }else if(base == 4){
636             a = sqrt(energyOfSymbols[step]/10);
637         }else{
638             a = 1;
639         }
640 
641         EnergyComponent=EnergyVector(M, a);
642         AngleComponent = AngleVector(M);
643 
644         Symbols = binaryToDecimalConversion(sourceBits, base);
645 
646 
647         trans1 = qam1.TransSymbol_colVect1(Symbols, EnergyComponent, AngleComponent);
648         trans2 = qam1.TransSymbol_colVect2(Symbols, EnergyComponent, AngleComponent);
649 
650 
651         gnoise1 = qam1.GnoiseVector(mean, stddev);
652         gnoise2 = qam1.GnoiseVector(mean, stddev);
653 
654 
655         receivedBits1 = qam1.ChannelModel(trans1, gnoise1);
656         receivedBits2 = qam1.ChannelModel(trans2, gnoise2);
657 
658 
659         if(base==2){
660             decodedBits = qam1.DecisionBlock_M4(receivedBits1, receivedBits2);
661         }else if(base ==3){
662             decodedBits = qam1.DecisionBlock_M8(receivedBits1, receivedBits2, a);
663         }else if(base == 4){
664             decodedBits = qam1.DecisionBlock_M16(receivedBits1, receivedBits2, a);
665         }else{
666             cout<<"NO decoder define "<<endl;
```

```
667         }
668
669
670         errors = qam1.ErrorCount(Symbols, decodedBits);
671         sizeOfTrans = Symbols.size();
672         pe = errors/sizeOfTrans;
673
674         Prob_error.insert(Prob_error.end(), pe);
675
676         cout<< " Error count : "<<errors<< endl;
677         cout<< " Pe : "<<pe << endl;
678
679     }
680
681
682     datafile(SNR_dB, Prob_error);
683
684     vector<double> qvalue;
685
686     if(base ==2){
687         qvalue = Qfunction_M4(SNR_dB);
688         qvalueInFile(SNR_dB, qvalue);
689     }else if(base ==3){
690         qvalue = Qfunction_M8(SNR_dB);
691         qvalueInFile(SNR_dB, qvalue);
692     }else if(base == 4){
693         qvalue = Qfunction_M16(SNR_dB);
694         qvalueInFile(SNR_dB, qvalue);
695     }
696
697     return 0;
698
699 }
```

4.2 Results

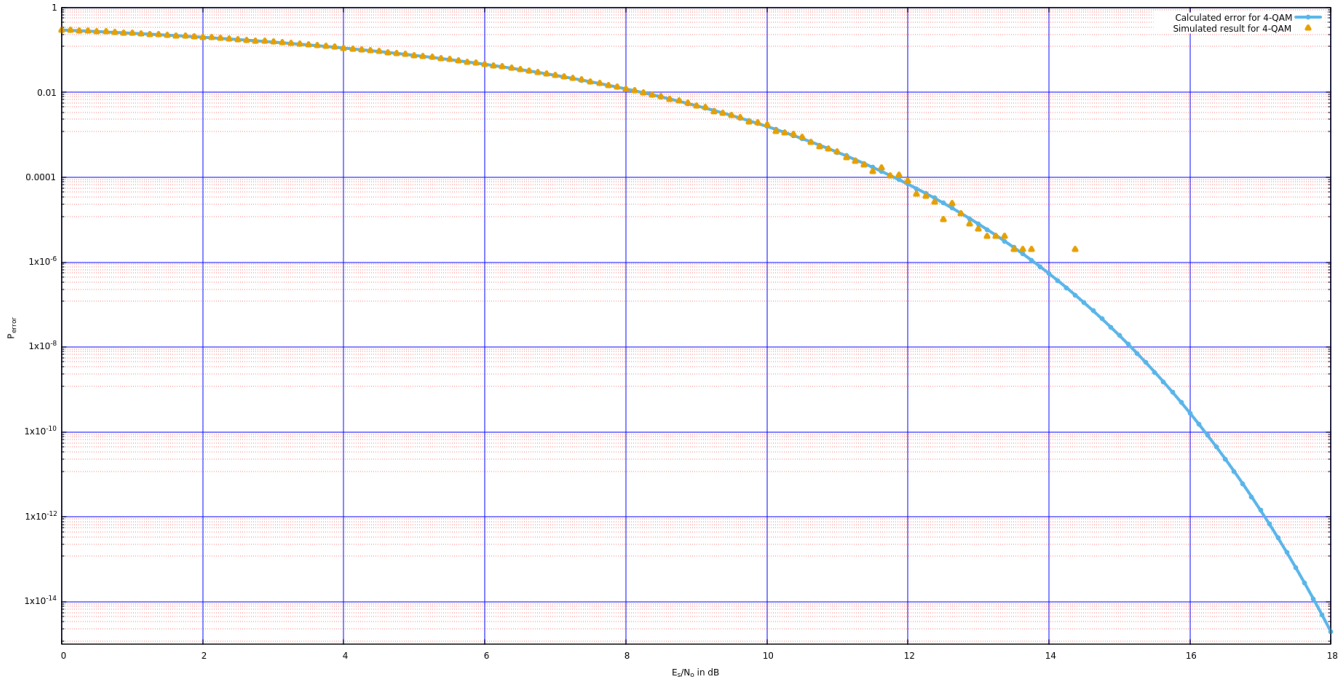

Figure 11: 4-QAM result

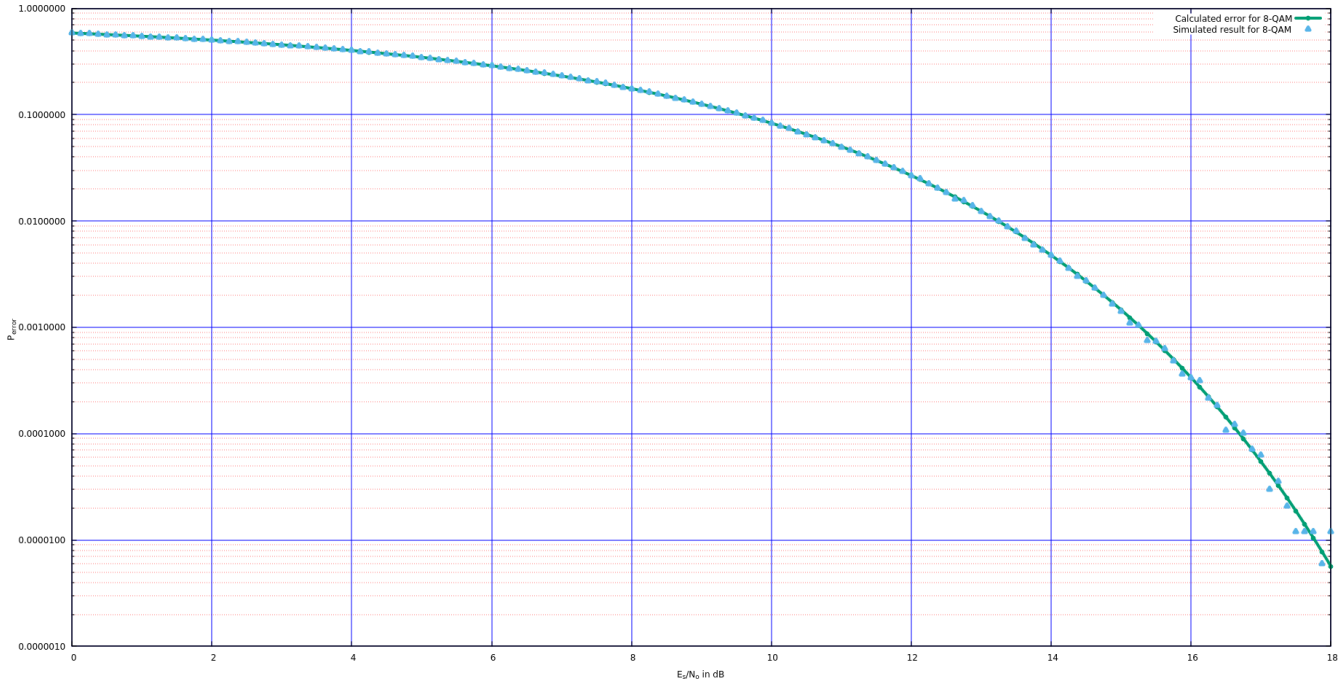

Figure 12: 8-QAM result

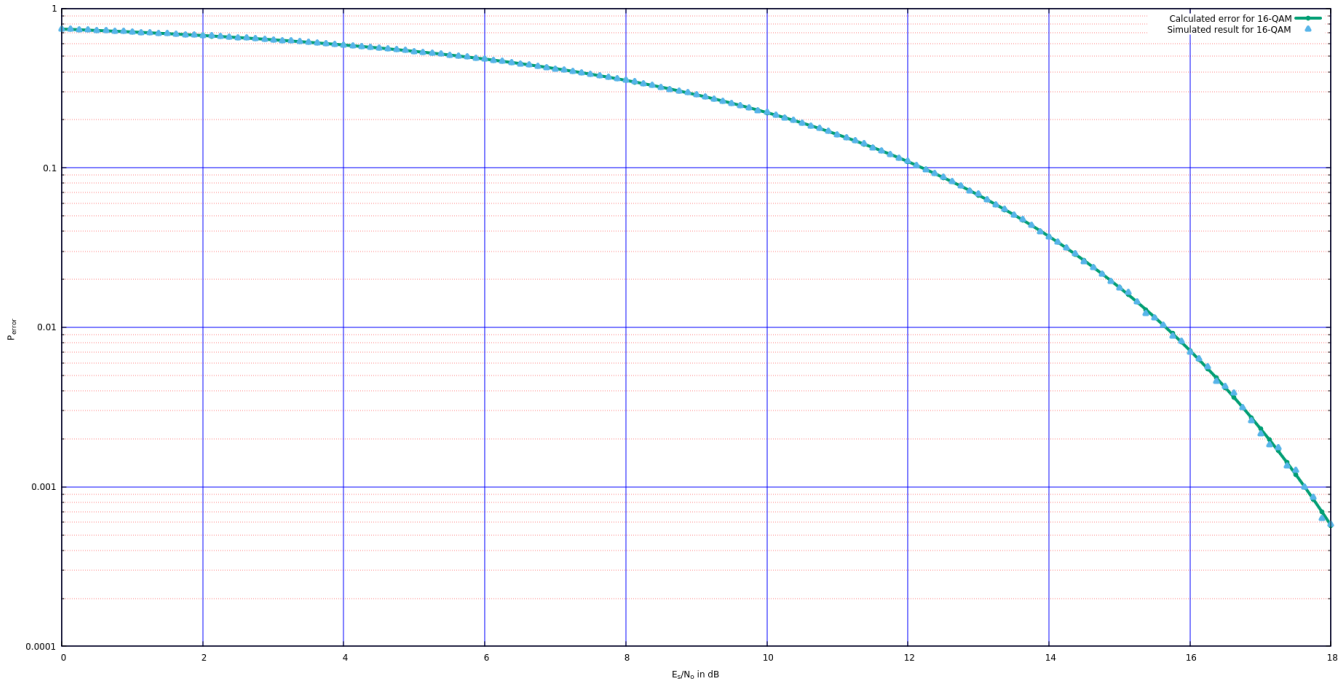

Figure 13: 16-QAM result

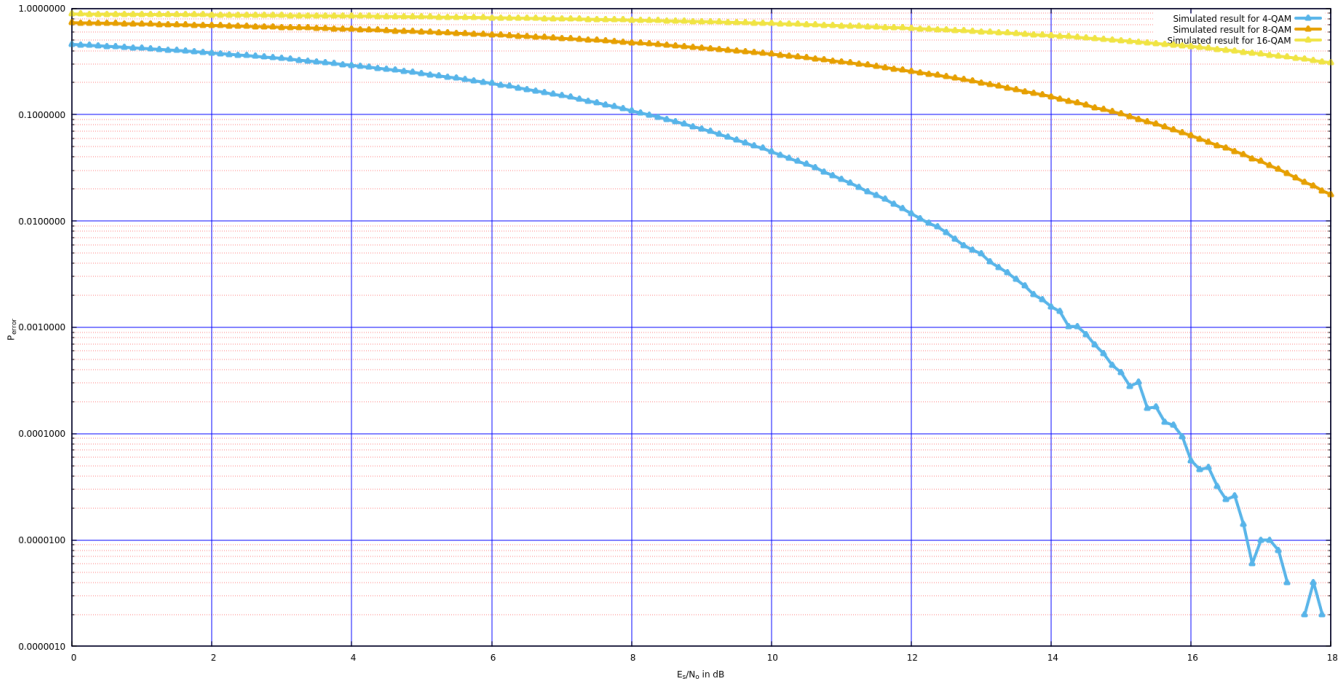

Figure 14: 4-QAM, 8-QAM, and 16-QAM

5 Inferences

1. AS $\frac{E_s}{N_o}$ increases, Probability of error decreases for all three cases. 16-QAM has slow decrease.
2. For particular value of a 16-QAM requires more average energy as compare to 8-QAM and 4-QAM. But 16-QAM has high date rate as compare to others.
3. At lower SNR, in 16-QAM symbols would be closer to each other that makes, difficult to distinguish symbols without error. But 4-QAM symbols are in different quadrants, therefore, even in lower SNR, it is easy to distinguish different symbols. 8-QAM is in the mid of 4-QAM and 16-QAM.
4. Finally, without any doubt, for good data rate with the cost of energy, one should choose 16-QAM. For good data rate with constrain energy applications, one should choose 8-QAM. But for energy saving modes in communication, one should choose 4-QAM with the cost of slow data rate (relatively).