WIRELESS COMMUNICATION PRACTISE

EXPERIMENT - 03 (Equal Gain Diversity)

Manas Kumar Mishra (ESD18I011)

01-FEBRUARY-2022

Lab In-charge: Dr. Premkumar Karumbu
Organization: IIITDM Kancheepuram

1 Aim:

To analyze the fadding channel with equal gain diversity. To evaluate performance of BPSK from BER (probability of error) vs SNR ratio in fadding channel with equal gain diversity. Compare with selective gain diversity. Check Array gain and Diversity gain. Repeat the analysis with different modulation schemes (Experimentally).

2 Software:

To perform this experiment, c++ language and gnuplot (open source) have been used. Data have been generated by the program in c++ language and plots are made by gnuplot.

3 Theory

3.1 Why diversity?

Basic result of BPSK in fadding channel conspicuously reflects the large difference between AWGN channel and fadding AWGN (fadding and AWGN) channel see figure 1. For achieving probability of error as 10^{-4}, AWGN channel requires SNR less than 10dB, but fadding channel requires more than 35dB. There is huge loss of power in fadding channel.
Now, question is, how can one improve the fadding channel? More ambitiously, it is possible for fadding channel to perform better than AWGN channel. It turns out, the answer of first question is yes, but for second ambitious question, answer is *It Depends*.
For improving the fadding channel, Diversity is a technique to be used.

3.2 What is diversity?

Sending same data (signals with same data) from different multipaths and receive signals with improved SNR, is known as diversity. In this technique, transmitter transmit replica of data and after receiving the signals receiver apply some kind of decision rule to the signal.

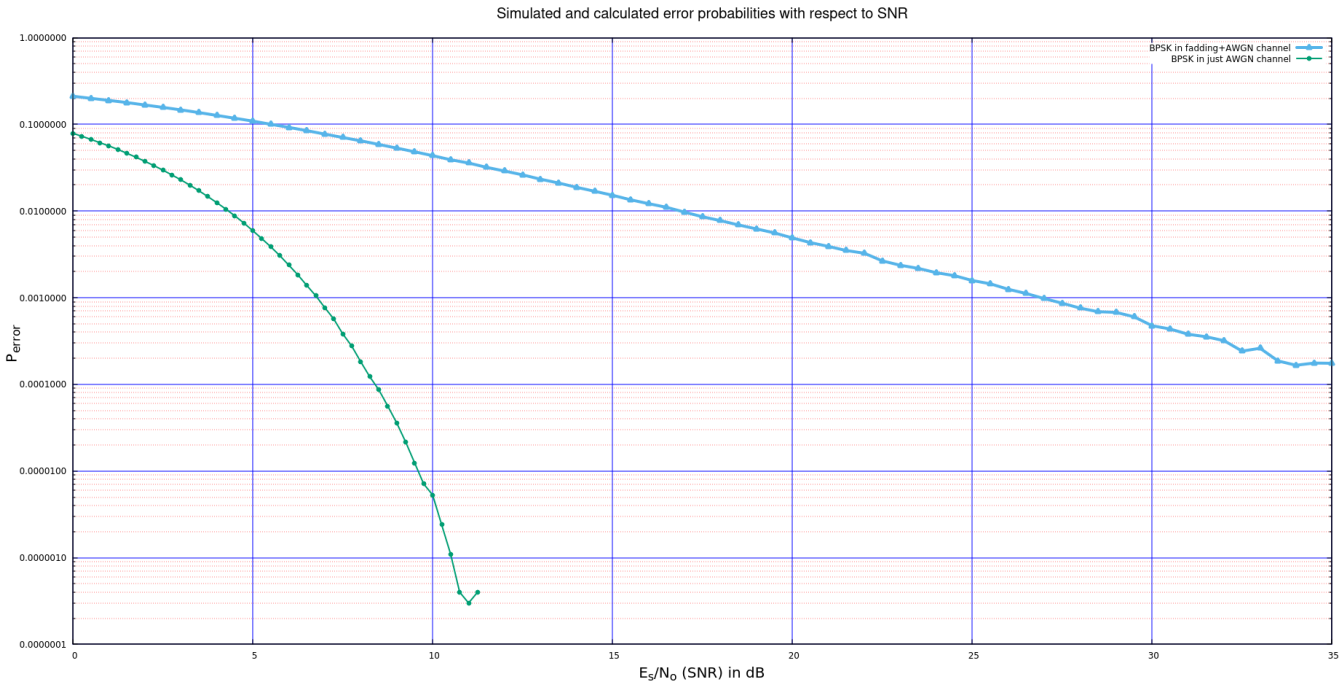

Figure 1: BPSK performance overview (Source:- Experiment-01)

Same data can be transmitted in three different ways.

1. Space Diversity
2. Frequency Diversity
3. Time Diversity

In space diversity, different transmitter antenna is used. Particular number of antennas are placed with certain gap such that probability of deep fade in every multipath is very small.
In frequency diversity, same narrowband signal has been transmitted over different frequency bands. There has to be sufficient gap between any two bands for avoiding interference.
In time diversity, same data has been transmitted in certain consecutive time slots. That is also known as repetition technique.
All of these are transmitter based diversity techniques. But actual intelligence lies in the receiver side, where receiver receives the diversity signal and take decision to improve performance.

3.3 Equal gain diversity

This is the receiver based decision technique to improve the performance. It is more computationally complex than selective gain technique. In this techniques received signal from each antenna branch is multiplied by $\alpha_i \, e^{-j\phi_i}$. That is known as co-phasing factor. For simplicity, α_i as one for all antenna, it need not be one, any constant number would suffice this. Due to this equal constant multiplication at each antenna element with co-phasing, it named as $equal \;\; gain \;\; combiner/diversity$ (EGC).

3.4 What is it?

Let x is transmitted signal in slow fadding channel with L diversity (L different receiver antenna). Therefore, receiver receives multiple signals y_1, y_2...y_L.

$$\begin{aligned} y_1 &= h_1 x_1 + n_1 \\ y_2 &= h_2 x_2 + n_2 \\ &\cdot \quad \ldots \\ &\cdot \quad \ldots \\ &\cdot \quad \ldots \\ y_L &= h_L x_L + n_L \end{aligned}$$

In equal gain diversity final accepted received signal is

$$y = \sum_{i=1}^{L} \theta_i \, y_i$$

$$where \;\; \theta_i = 1 \;\; \forall \;\; 1 \leq i \leq L \qquad (\theta_i = 1 \text{ just for simplicity})$$

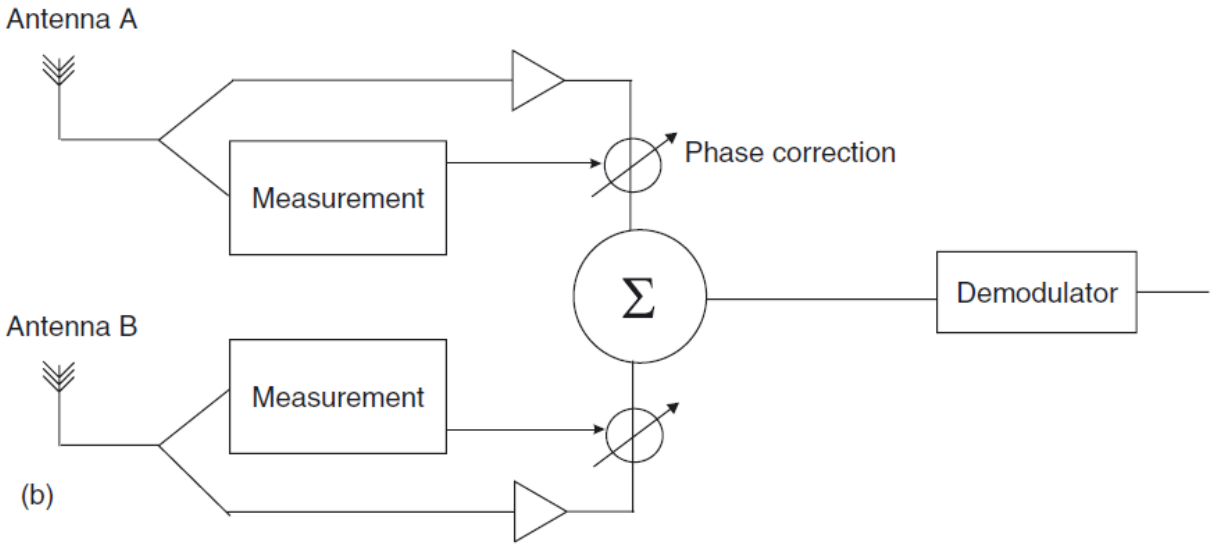

Figure 2: EGC for two antenna diversity

3.4.1 SNR

Let x is transmitted symbol of energy E and y_i is the received signal in fadding channel at i^{th} antenna element, then in equal gain combining

$$y = \left(\sum_{i=1}^{L} h_i\right) x + \sum_{i=1}^{L} n_i$$

$$n_i \sim N(0,\sigma^2) \qquad \text{(AWGN component)}$$

$$\sum_{i=1}^{L} n_i \sim N(0, L\sigma^2)$$

$$SNR\ \ \gamma = \frac{\left(\sum_{i=1}^{L} h_i\right)^2 E}{L\sigma^2}$$

$$Let,\ \ g = \frac{\left(\sum_{i=1}^{L} h_i\right)^2}{L}$$

$$\gamma = g\frac{E}{\sigma^2}$$

It turns out that distribution of g does not have close form. One can think to find moment generating function of Rayleigh distribution and try to multiply L times, and then take square of that. But after that it is unsolvable for general value of L. In literature, L as 2 and 3 has some approximate expression, but for L more than 3 does not have form as such.

3.4.2 Average SNR

$$\mathbb{E}[\gamma_{EGC}] = \frac{E}{L\sigma^2}\mathbb{E}\left((\sum_{i=1}^{L} h_i)^2\right)$$

$$(h_1+h_2+...+h_L)(h_1+h_2+...+h_L) = (L-\ times)\ h_i^2\ +\ ((L^2-L)\ -times)\ h_i\,h_j(i\neq j)$$

$$= \frac{E}{L\sigma^2}\left[L\mathbb{E}h^2 + L(L-1)\mathbb{E}(h_1\ h_2)\right]$$

$$= \frac{E}{L\sigma^2}\left[L2\sigma_R^2 + L(L-1)\sigma_R^2\frac{\pi}{2}\right] \quad \text{(Rayleigh distribution property)}$$

$$= \frac{E}{\sigma^2}\left[2\sigma_R^2 + (L-1)\sigma_R^2\frac{\pi}{2}\right]$$

Since, for wireless (fadding channel) σ_R^2 as $1/2$.

$$\mathbb{E}[\gamma_{EGC}] = \frac{E}{\sigma^2}\left[2\sigma_R^2 + (L-1)\sigma_R^2\frac{\pi}{2}\right]$$

$$= \Gamma\ \left[1 + (L-1)\frac{\pi}{4}\right] \qquad (\Gamma = \tfrac{E}{\sigma^2})$$

3.4.3 Comparison of gain in SG and EGC

In SG

$$g_{SG} = |h_i|^2$$

But in EGC

$$g_{EGC} = \left(\sum_{i=1}^{L} |h_i| \right)^2$$

That is straight forward

$$g_{EGC} > g_{SG}$$

3.4.4 ML rule

Since, in equal gain diversity, there is operation going on received signal by combiner, but does not change decoder. Hence, ML decoder should be same for BPSK as

$$\begin{aligned} if,\ 0 &\mapsto \sqrt{E} \\ 1 &\mapsto -\sqrt{E} \\ then\ y &\gtrless_{B^o=1}^{B^o=0} 0 \end{aligned}$$

where, B^o is the decoded bit.

3.4.5 Probability of error

Since, distribution of SNR does not have closed form in EGC, it is difficult to come up a general expression for Probability of error. But one can use approximate results to define error probability.
For, one antenna in fadding channel has

$$P_e \approx \frac{K}{SNR} \qquad \text{(For large value of SNR and K is a real constant)}$$

Similarly, one can get a intuitive expression for EGC, where particular number of signals (say L) are independently adding with each other. Therefore

$$Pe_{EGC} \approx \frac{K}{(SNR)^L} \qquad \text{(For large value of SNR)}$$

It is a very lousy approximation, but it works well for large SNR values. Experimentally, K is set to be $(L-1)$. There is no prof for this, just by doing experiment, i realized that K as $(L-1)$ works pretty well with (SNR+1) value
Hence, in this experiment consider

$$Pe_{EGC} \approx \frac{(L-1)}{(SNR+1)^L} \qquad (L > 1)$$

4 Pseudo code

s1. Generate random bit stream of length equal to million.
s2. Map 0 to $-\sqrt{E}$ and 1 to $\sqrt{E}$ and store in a dynamic vector.
s3. Repeat s1 and s2 for L (number of diversity) times and store each resultant vector as Matrix. (i^{th} Row of matrix as i^{th} transmission signal).
s4. Generate two gaussian noise vectors of zero mean and variance as half, treat as real part and imaginary part of the complex random variable.
s5. Compute Rayleigh coefficient by using above random variables.
s6. Repeat s3 and s4, and store all rayleigh coff into matrix.(i^{th} Row of matrix as i^{th} transmission signal rayleigh).
s7. Generate gaussian noise components L times and store into matrix.
s8. Apply channel model, multiply transmission matrix element by element with rayleigh coff and add with gaussian noise matrix. Store into received matrix of dimension (L, one-million)
s9. Take sum of over all rows. Final resultant vector would be received vector.
s10. Apply ML rule and decode the received signal.
s11. Count errors
S12. Repeat S1 to S13 for many SNR values.
S13. Store data (SNR, Error count) in .dat file.
S14. Compute theoretical probability of error. Store into .dat file
S15. Plot the result using gnuplot in semilog scale.

5 Code and result

To support matrix operation a new header file has been designed and add to the code. Similar to the experiment-01, for BPSK channel a header file has been used.

5.1 Equal Gain Diversity code

```
1  ////////////////////////////////////////////////////////////
2  ////////////////////////////////////
3  // Author:- MANAS KUMAR MISHRA
4  // Organisation:- IIITDM KANCHEEPURAM
5  // Topic:- Performance of equal gain combiner using BPSK in Rayleigh fadding channel
6  ////////////////////////////////////////////////////////////
7  ////////////////////////////////////
8
9  #include <iostream>
10 #include <cmath>
11 #include <iterator>
12 #include <random>
13 #include <chrono>
14 #include <time.h>
15 #include <fstream>
16 #include "BPSK.h"
17 #include "MyMatrixOperation.h"
18
```

```
19 #define one_million 1000000
20
21 using namespace std;
22
23
24 // Function for printing the vector on the console output.
25 void PrintVectorDouble(vector<double> vectr)
26 {
27     std::copy(begin(vectr), end(vectr),  std::ostream_iterator<double>(std::cout, "
   "));
28     cout<<endl;
29 }
30
31
32
33 // Function for generating binary bits at source side. Each bit is equiprobable.
34 // Input is nothing
35 // Output is a vector that contains the binary bits of length one_million*1.
36 vector<double> sourceVector()
37 {
38     vector<double> sourceBits;
39
40     // Use current time as seed for random generator
41     srand(time(0));
42
43     for(int i = 0; i<one_million; i++){
44         sourceBits.insert(sourceBits.end(), rand()%2);
45     }
46
47     return sourceBits;
48 }
49
50
51
52 // Function for Rayleigh fadding cofficients
53 // Inputs are two vectors one is the gaussion noise as real part and second as
54 //Gaussian noise as imazinary part
55 // Output is a vector that contain rayliegh noise coff, sqrt(real_part^2 + imz_part^2)
56 vector<double> RayleighFaddingCoff(vector<double> realGaussian,
57                                    vector<double> ImziGaussian)
58 {
59     vector<double> rayleighNoise;
60     double temp;
61
62     for(int times=0; times<realGaussian.size(); times++){
63
64         temp = sqrt(pow(realGaussian[times], 2)+pow(ImziGaussian[times], 2));
65         rayleighNoise.insert(rayleighNoise.end(), temp);
66     }
67
68     return rayleighNoise;
69 }
70
71
72 // Function for multiplying fadding coff to particular antenna
73 // Inputs are the Transmitted energy and Rayleigh fadding coff
```

```
74  // Output is the multiplication of element by element fadding and energy
75  vector<double> RayleighOperation(vector<double> TxEnergy,
76                                   vector<double> RayleighFaddingCoff)
77  {
78      vector<double> Resultant;
79
80      for(int j =0; j<TxEnergy.size(); j++){
81          Resultant.insert(Resultant.end(), TxEnergy[j]*RayleighFaddingCoff[j]);
82      }
83
84      return Resultant;
85  }
86
87
88  // Function for gaussian noise for each tx bit to particular antenna
89  // Inputs are the Transmitted energy and Gnoise
90  // Output is the addition of element by element Gnoise and Tx
91  vector<double> GaussianNoiseAdd(vector<double> TxEnergy,
92                                  vector<double> Gnoise)
93  {
94      vector<double> Resultant;
95
96      for(int j =0; j<TxEnergy.size(); j++){
97          Resultant.insert(Resultant.end(), TxEnergy[j]+Gnoise[j]);
98      }
99
100     return Resultant;
101 }
102
103
104 // Function for sum of all raws in a matrix
105 // Input is a Matrix
106 // Output is a vector result as sum of all raws
107 vector<double> RawWiseSum(vector<vector<double>> ReceiveMat)
108 {
109     int numberOfRaws = ReceiveMat.size();
110
111     vector<double> SumofRaws= ReceiveMat[0];
112
113     for(int t=0; t<numberOfRaws-1; t++){
114         SumofRaws = VectorSum(SumofRaws, ReceiveMat[t+1]);
115     }
116
117     return SumofRaws;
118 }
119
120
121 // Function to count number of errors in the received bits.
122 // Inputs are the sourcebits and decodedbits
123 // OUtput is the number of error in received bits.
124 // error: if sourcebit  != receivebit
125 double errorCalculation (vector<double> sourceBits, vector<double> decodedBits)
126 {
127     double countError =0;
128     for(int i =0; i<sourceBits.size();i++){
129         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, char strName[])
{
    ofstream outfile;

    string filename = strName;

    outfile.open(filename +"."+"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();
}


double ProbabilityOferror(double SNR, double L)
{

    double inter = pow(SNR+1, L);
    double pe = (L-1)/inter;
    return pe;
}


vector<double> CalculatedError(vector<double> SNR_dB, double L)
{
    vector<double> ProbError;

    double po, normalValue, inter;
    for (int k =0; k<SNR_dB.size(); k++){
        normalValue = pow(10, (SNR_dB[k]/10));
        po = ProbabilityOferror(normalValue, L);
        ProbError.insert(ProbError.end(), po);
    }

    return ProbError;
}
```

```
186 int main(){
187
188     // source defination
189     vector<double> sourceBits;
190
191     // Mapping of bits to symbols;
192     vector<double> transmittedSymbol;
193
194     // Noise definition
195     vector<double> gnoise;
196
197     //Rayleigh noise (Real guass and Img Guass)
198     vector<double> realGaussian;
199     vector<double> imziGaussian;
200     vector<double> RayleighNoise;
201
202     //Combiner output
203     vector<double> AfterCombining;
204
205     //Output of ML detector
206     vector<double> decodedBits;
207
208     double L =8;
209     double sigmaSquare = 0.5;
210     double stddevRayleigh = sqrt(sigmaSquare);
211     double N_o =4;
212     double p, stdnoise;
213     stdnoise = sqrt(N_o);
214     double counterror, P_error;
215
216
217     vector<vector<double>> RaleighMat;
218     vector<vector<double>> GnoiseMat;
219     vector<vector<double>> faddingOperation;
220     vector<vector<double>> ReceiveMat;
221
222
223     vector<double> SNR_dB;
224     for(float i =0; i<=25; i=i+0.5)
225     {
226         SNR_dB.insert(SNR_dB.end(), i);
227     }
228
229     vector<double> energyOfSymbol;
230     vector<double> Prob_error;
231     double normalValue;
232
233     for(int i =0; i<SNR_dB.size(); i++){
234
235         normalValue = pow(10, (SNR_dB[i]/10));
236         energyOfSymbol.insert(energyOfSymbol.end(), N_o*normalValue);
237     }
238
239     for(int step = 0; step<energyOfSymbol.size(); step++){
240
241         sourceBits = sourceVector();    //Source bit streame
```

```
242
243         transmittedSymbol = bit_maps_to_symbol_of_energy_E(sourceBits,
244                                                            energyOfSymbol[step],
245                                                            one_million);
246
247         //Rayleigh distributed fadding coffcients
248         for(int i =0; i<L; i++){
249             realGaussian = GnoiseVector(0.0, stddevRayleigh, one_million);
250             imziGaussian = GnoiseVector(0.0, stddevRayleigh, one_million);
251
252             RayleighNoise = RayleighFaddingCoff(realGaussian, imziGaussian);
253
254             RaleighMat.insert(RaleighMat.end(), RayleighNoise);
255
256             // realGaussian.clear();
257             // imziGaussian.clear();
258             // RayleighNoise.clear();
259         }
260
261         //Gaussian noise (white noise)
262         for(int i=0; i<L; i++){
263             gnoise = GnoiseVector(0.0, stdnoise, one_million);
264             GnoiseMat.insert(GnoiseMat.end(), gnoise);
265             gnoise.clear();
266         }
267
268         //Fadding operation
269         for(int k =0; k<L; k++){
270             faddingOperation.insert(faddingOperation.end(),
271                                     RayleighOperation(transmittedSymbol, RaleighMat[k]));
272         }
273
274         //Additive gaussian noise
275         for(int k =0; k<L; k++){
276             ReceiveMat.insert(ReceiveMat.end(),
277                               GaussianNoiseAdd(faddingOperation[k], GnoiseMat[k]));
278         }
279
280         //combining operation
281         AfterCombining = RawWiseSum(ReceiveMat);
282
283         //ML decoder
284         decodedBits = decisionBlock(AfterCombining);
285
286         //Count error
287         counterror = errorCalculation(sourceBits, decodedBits);
288
289         //Probability of error
290         P_error = counterror/one_million;
291         Prob_error.insert(Prob_error.end(), P_error);
292
293         cout<<"Error value : "<<counterror<<endl;
294         cout<<"Probability of error: "<<P_error<<endl;
295         cout<<endl;
296
297         RaleighMat.clear();
```

```
        GnoiseMat.clear();
        faddingOperation.clear();
        ReceiveMat.clear();
    }

    char NameofFile1[30] = "EGCL8";          //Name of file

    char NameofFile2[30] = "EGCL8Error";     //Name of file


    datafile(SNR_dB, Prob_error, NameofFile1);

    // vector<double> Error = CalculatedError(SNR_dB, L);

    // datafile(SNR_dB, Error, NameofFile2);

    return 0;
}
```

Code for BPSK header file

```
////////////////////////////////////////////////////////////
////////////////////////////////////
// Author:- MANAS KUMAR MISHRA
// Organisation:- IIITDM KANCHEEPURAM
// Topic:- header file BPSK scheme
////////////////////////////////////////////////////////////
////////////////////////////////////
#include <cmath>
#include <iterator>
#include <random>
#include <chrono>
#include <time.h>

using namespace std;

// 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,
                                              const int one_million)
{
    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, const int one_million)
{
    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;
}
```

Matrix operation header file

```
#include <iostream>
#include <vector>
#include <iterator>

using namespace std;

// Function of error message in matrices multiplications.
void errorMSG(){
    cout<<endl;
    cout<<"! Matrices size are not proper for multiplication !!! :-("<<endl;
    cout<<endl;
}

// Function for printing the matrix on console.
void PrintMat(vector<vector<double> > & MAT)
{
    for(int j =0; j<MAT.size();j++){
        for(int k =0; k<MAT[j].size(); k++){
            cout<<MAT[j][k]<< "   ";
        }
        cout<<endl;
    }
}


// Function for taking transpose of the given matrix.
// Input is the matrix for some m*n dimension.
// Output is the matrix with n*m dimension having transpose of actual matrix
vector<vector <double> > Transpose_MAT(vector<vector <double> > MAT)
{
    vector<vector<double> > TransMAT;
    vector <double> interMAT;

    for(int i =0; i<MAT[0].size(); i++){
        for(int j =0; j<MAT.size(); j++){
            interMAT.insert(interMAT.end(), MAT[j][i]);
        }
        TransMAT.push_back(interMAT);
        interMAT.clear();
    }

    return TransMAT;
}

// Function to fix the issue of vector and matrixes
// Input is the vector signal
// Output is the Matrix that contain vector as it's first raw
vector<vector<double>> convertVectorToMatrix(vector<double> Vec)
{
```

```
50      vector<vector<double>> Mat;
51      Mat.insert(Mat.end(), Vec);
52      return Mat;
53  }
54
55
56  // Function for Multiplying two matrixs element by elements
57  // Input are two matrix of same dimension
58  // Output is another matrix that contain each element
59  //  as multiplication of each element.
60  vector<vector<double>> ElementWiseMultiplication(vector<vector<double>> Mat1,
61                                                   vector<vector<double>> Mat2)
62  {
63      vector<vector<double>> MatResult;
64
65      vector<double> multi;
66
67      for(int i =0; i<Mat1.size();i++){
68          for(int j=0; j<Mat1[0].size(); j++){
69              multi.insert(multi.end(), Mat1[i][j]*Mat2[i][j]);
70          }
71
72          MatResult.insert(MatResult.end(), multi);
73          multi.clear();
74      }
75
76      return MatResult;
77  }
78
79
80  // Function for Adding two matrixs element by elements
81  // Input are two matrix of same dimension
82  // Output is another matrix that contain each element
83  //  as Addition of both element.
84  vector<vector<double>> ElementWiseAddition(vector<vector<double>> Mat1,
85                                             vector<vector<double>> Mat2)
86  {
87
88      vector<vector<double>> MatResult;
89
90      vector<double> Add;
91
92      for(int i =0; i<Mat1.size();i++){
93          for(int j=0; j<Mat1[0].size(); j++){
94              Add.insert(Add.end(), Mat1[i][j]+Mat2[i][j]);
95          }
96
97          MatResult.insert(MatResult.end(), Add);
98          Add.clear();
99      }
100
101     return MatResult;
102 }
103
104 // function for sum of two vectors
105 // Inputs are two vectors
```

```
106 // Output is the sum of two vectors.
107 vector<double> VectorSum(vector<double> Vec1, vector<double> Vec2)
108 {
109     vector<double> SumResult;
110
111     if(Vec1.size()==Vec2.size()){
112         for(int k=0; k<Vec1.size(); k++){
113             SumResult.insert(SumResult.end(), Vec1[k]+Vec2[k]);
114         }
115
116         return SumResult;
117     }else{
118         cout<<"Error !!!! Vector size should be same!!!"<<endl;
119         return SumResult;
120     }
121 }
```

5.2 Results

5.2.1 EGC

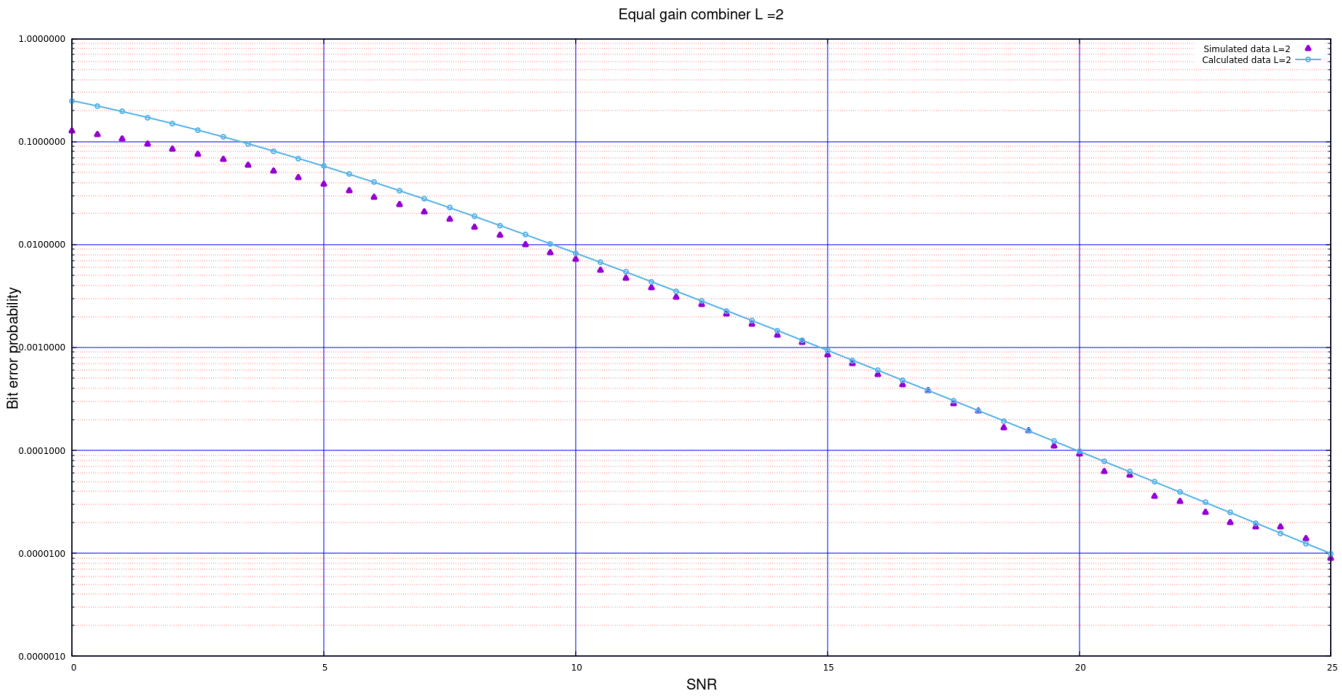

Figure 3: L=2 case simulated and calculated

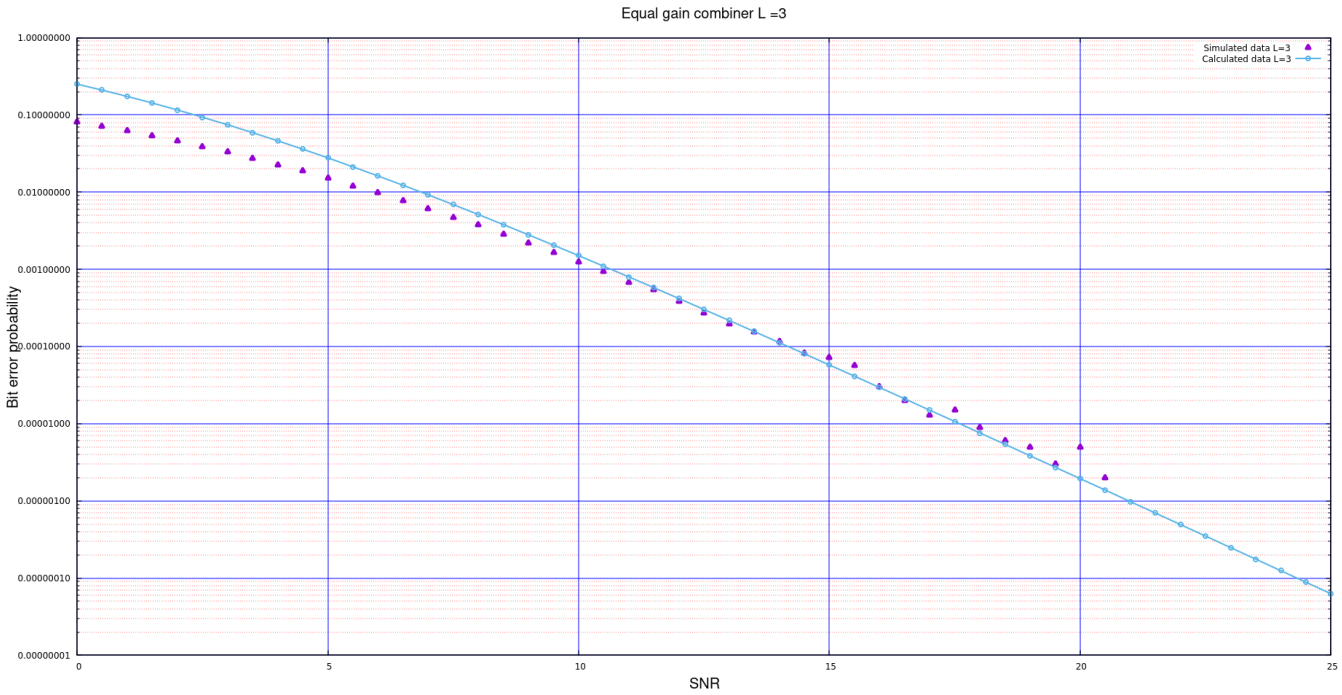

Figure 4: L=3 case simulated and calculated

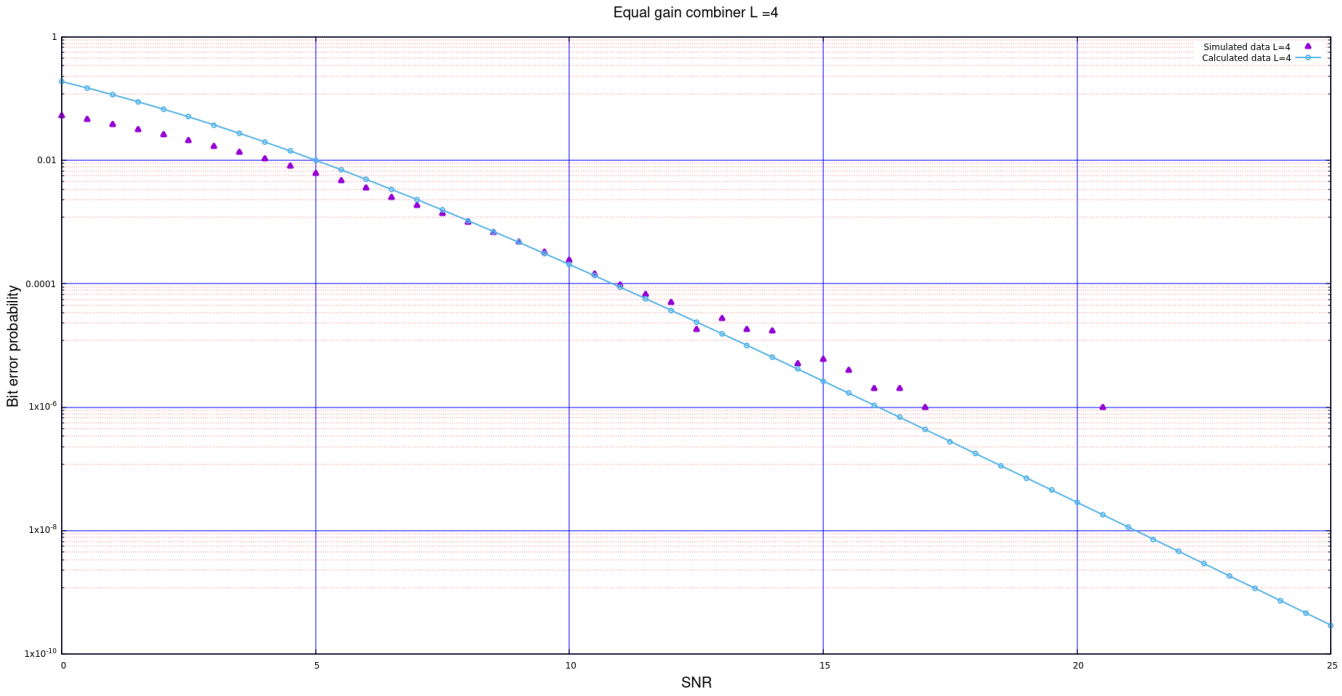

Figure 5: L=4 case simulated and calculated

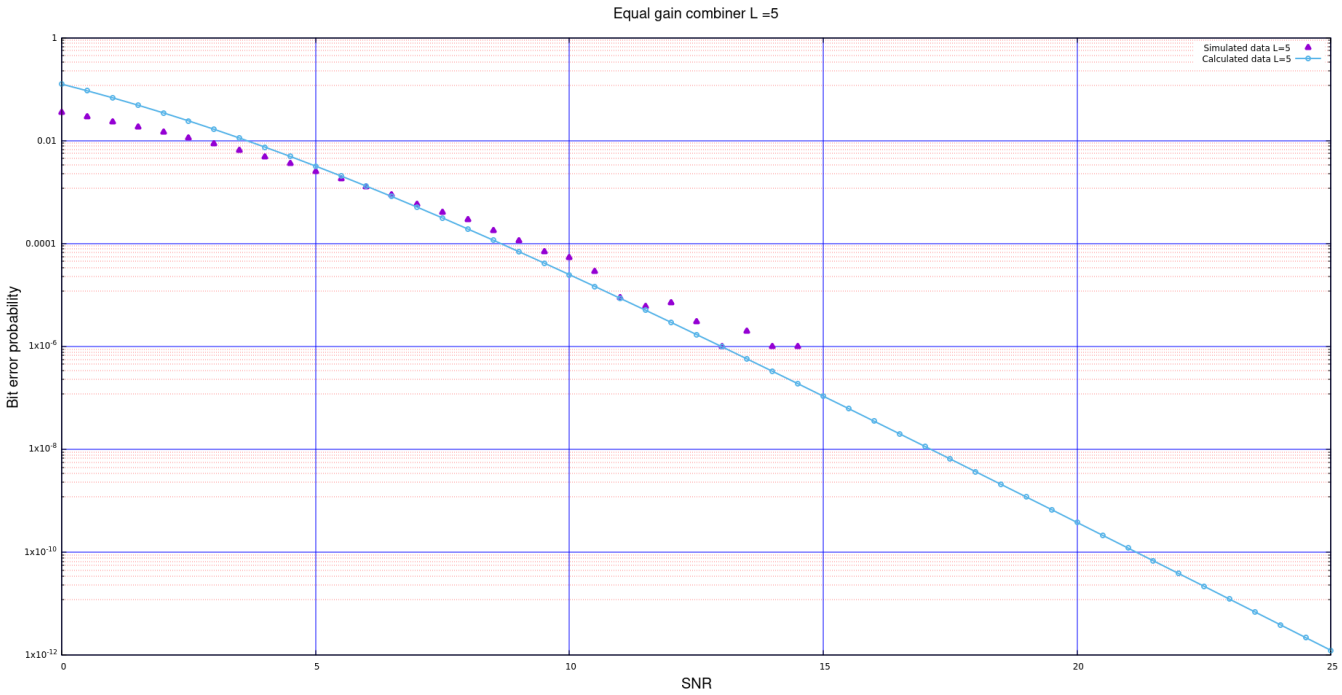

Figure 6: L=5 case simulated result

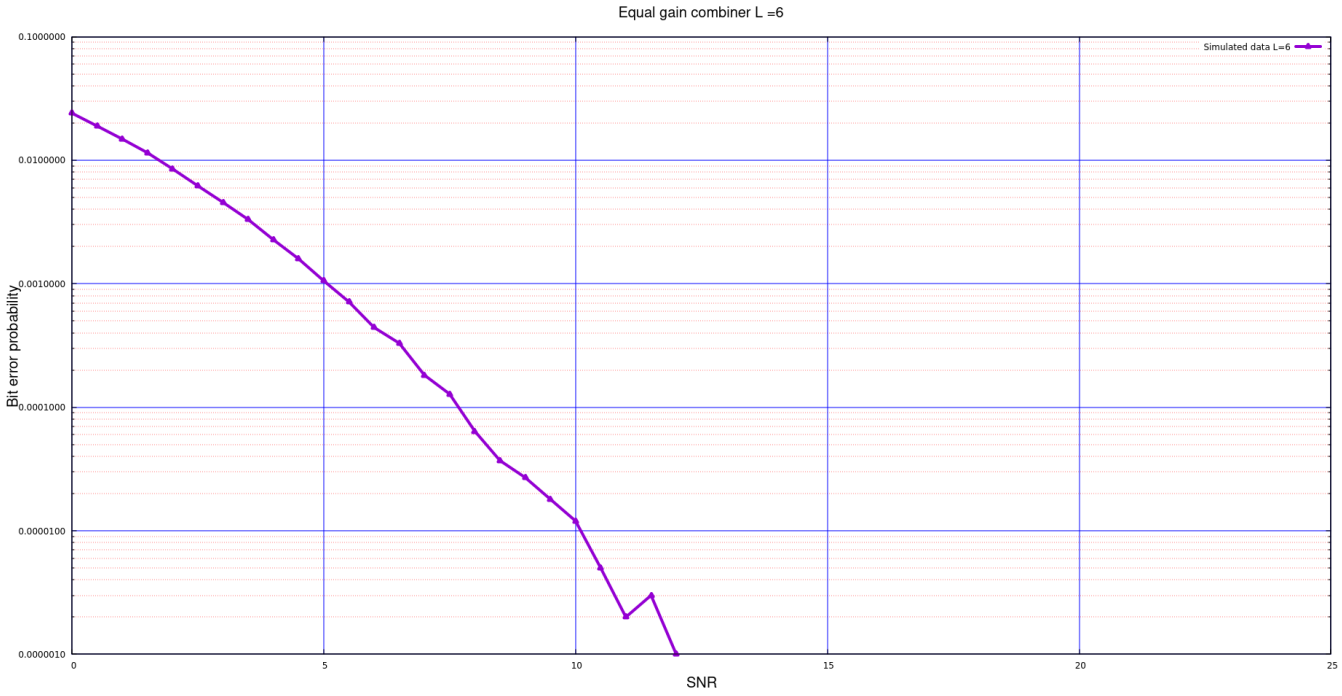

Figure 7: L=5 case simulated result

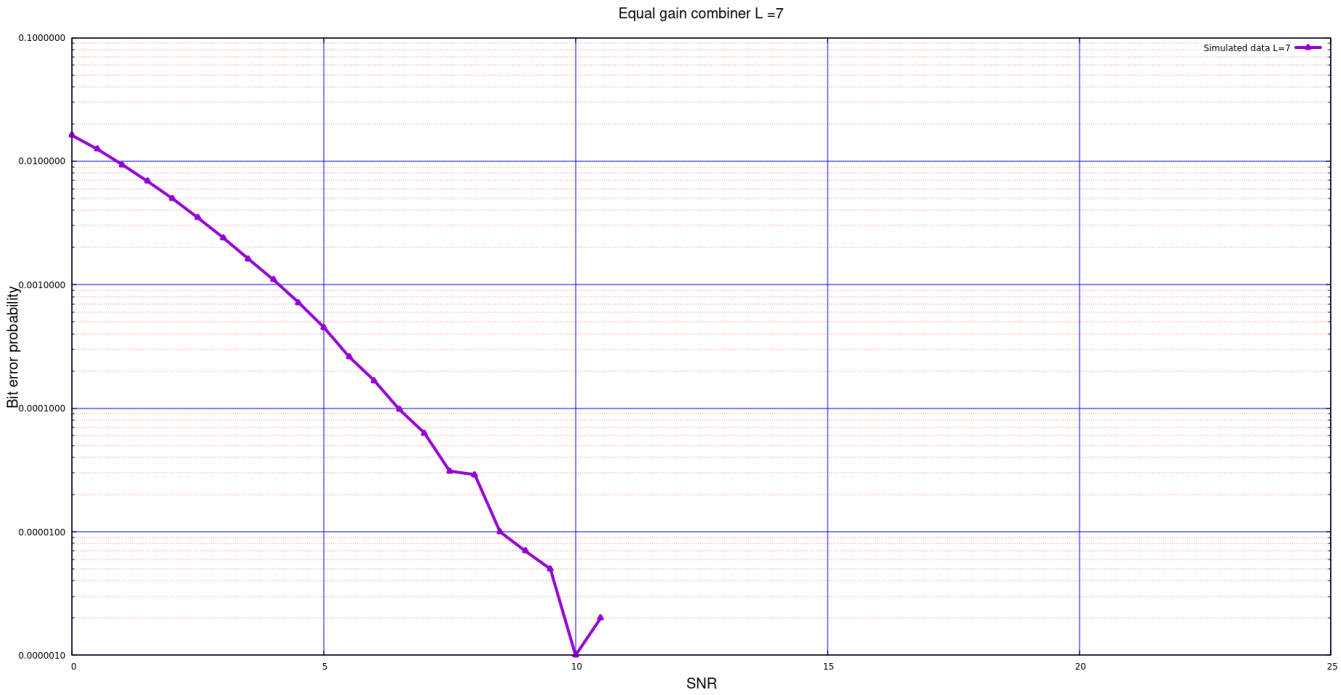

Figure 8: L=6 case simulated result

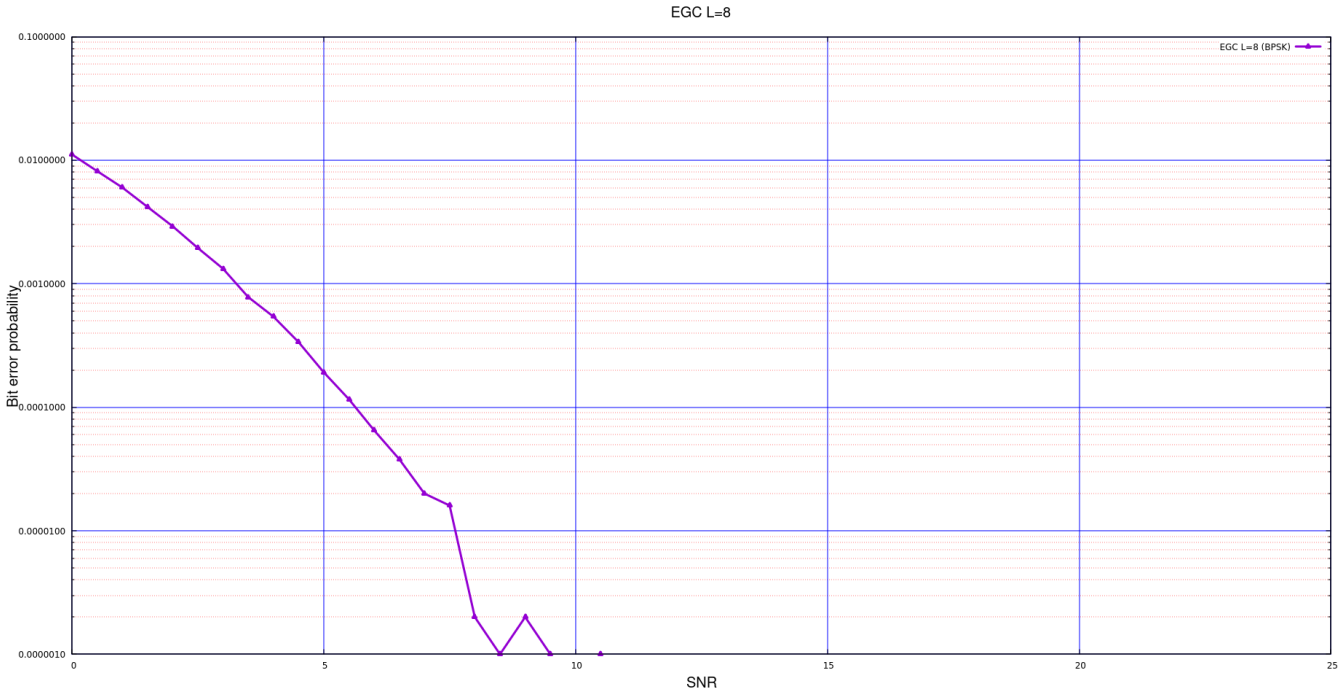

Figure 9: L=7 case simulated result

In all cases, simulated results are same as calculated results, that means that Calculated probability of error is correct.

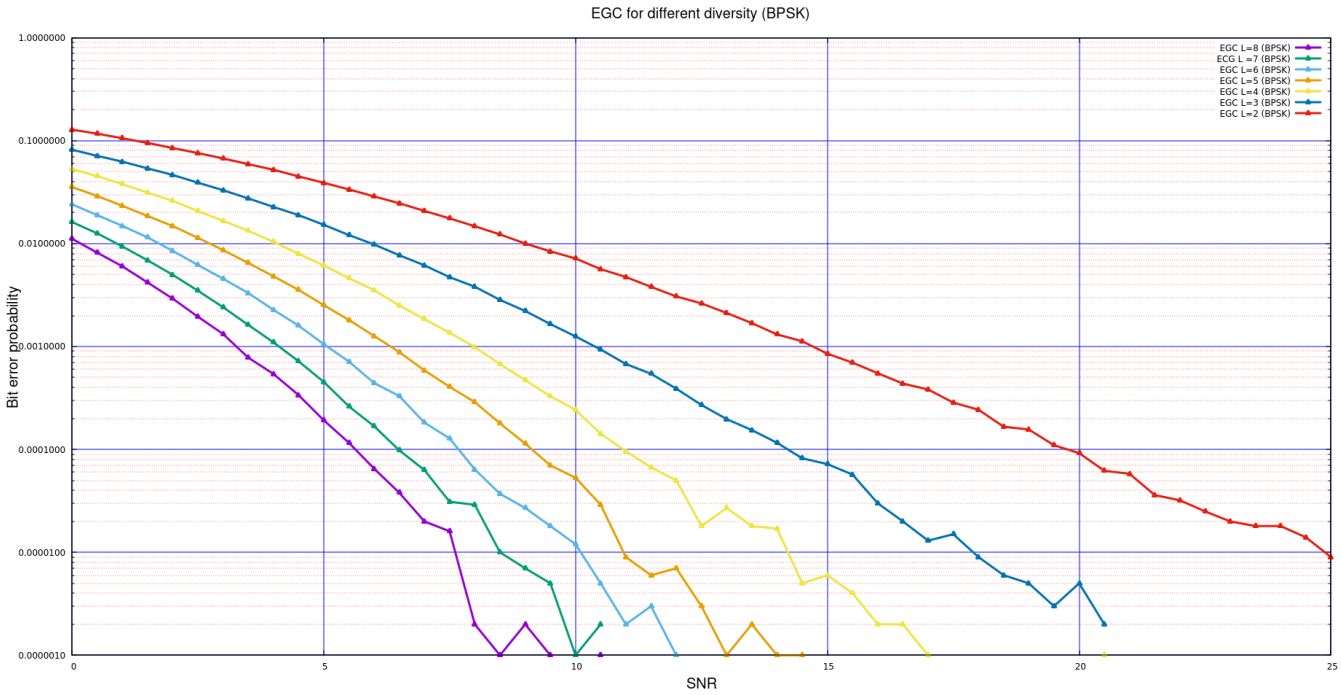

Figure 10: All simulated data

5.2.2 EGC and SG

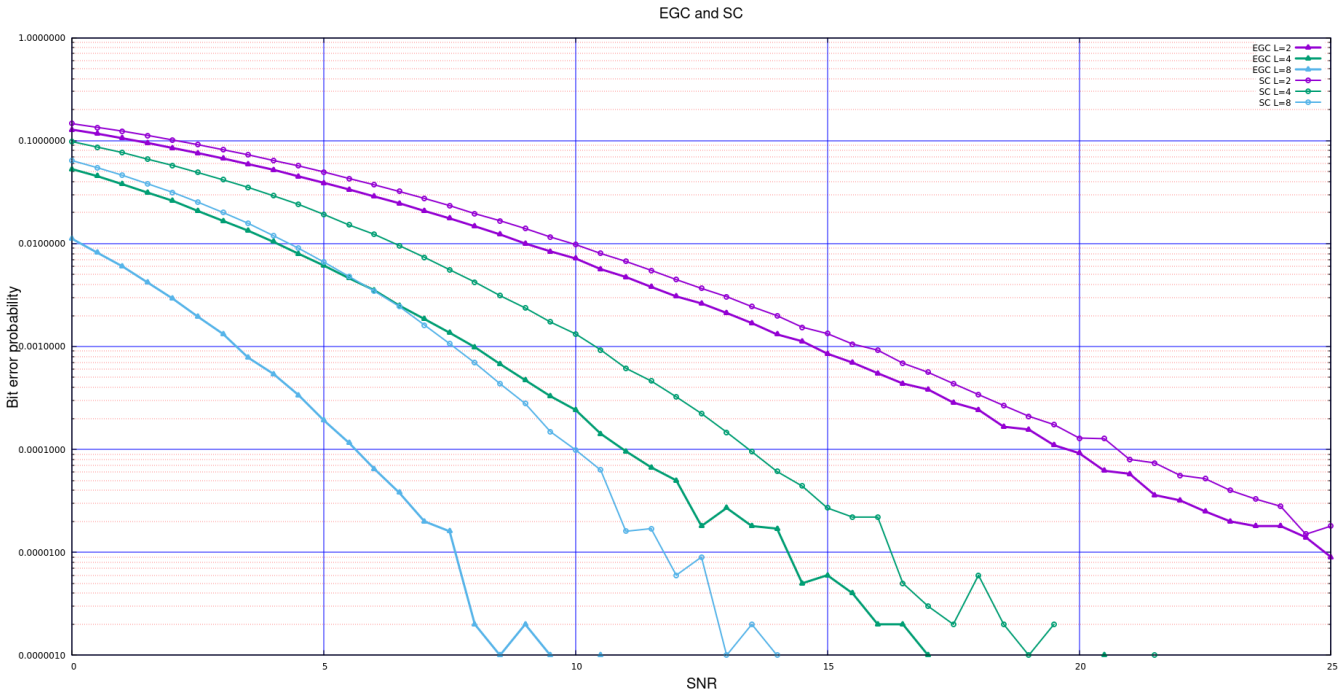

Figure 11: EGC vs SG

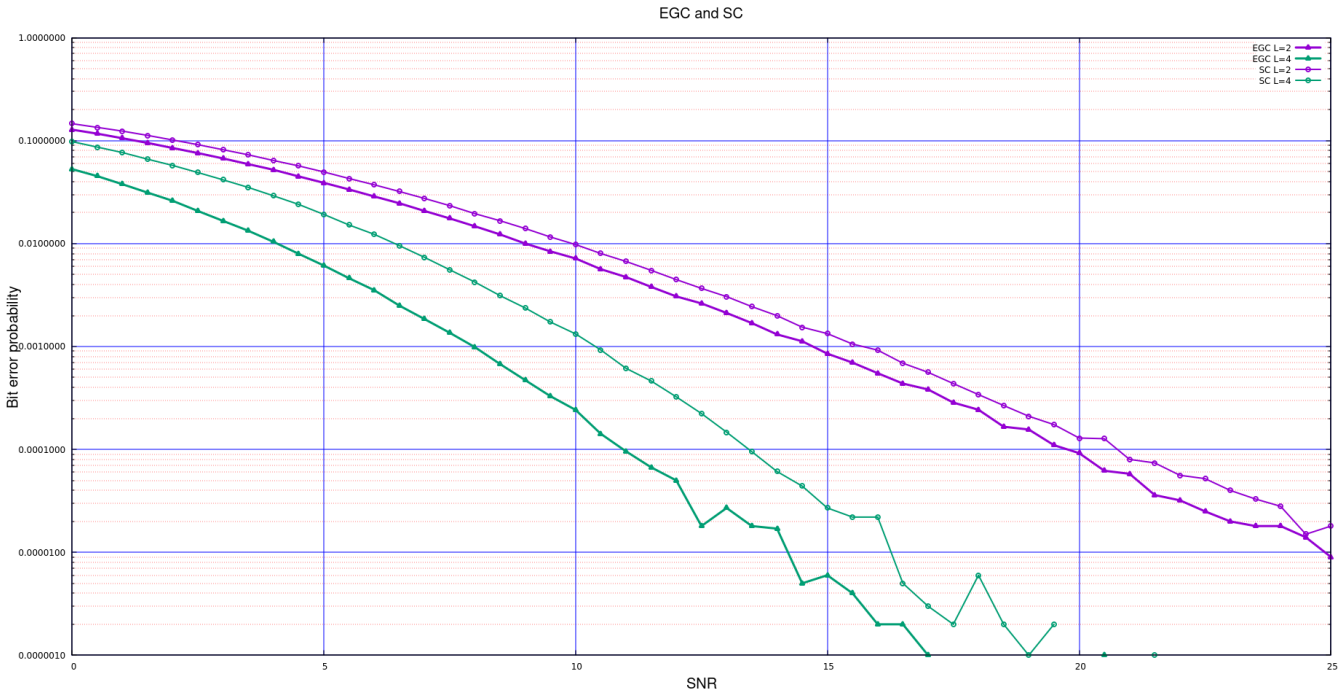

Figure 12: EGC vs SG

5.2.3 Different modulation techniques

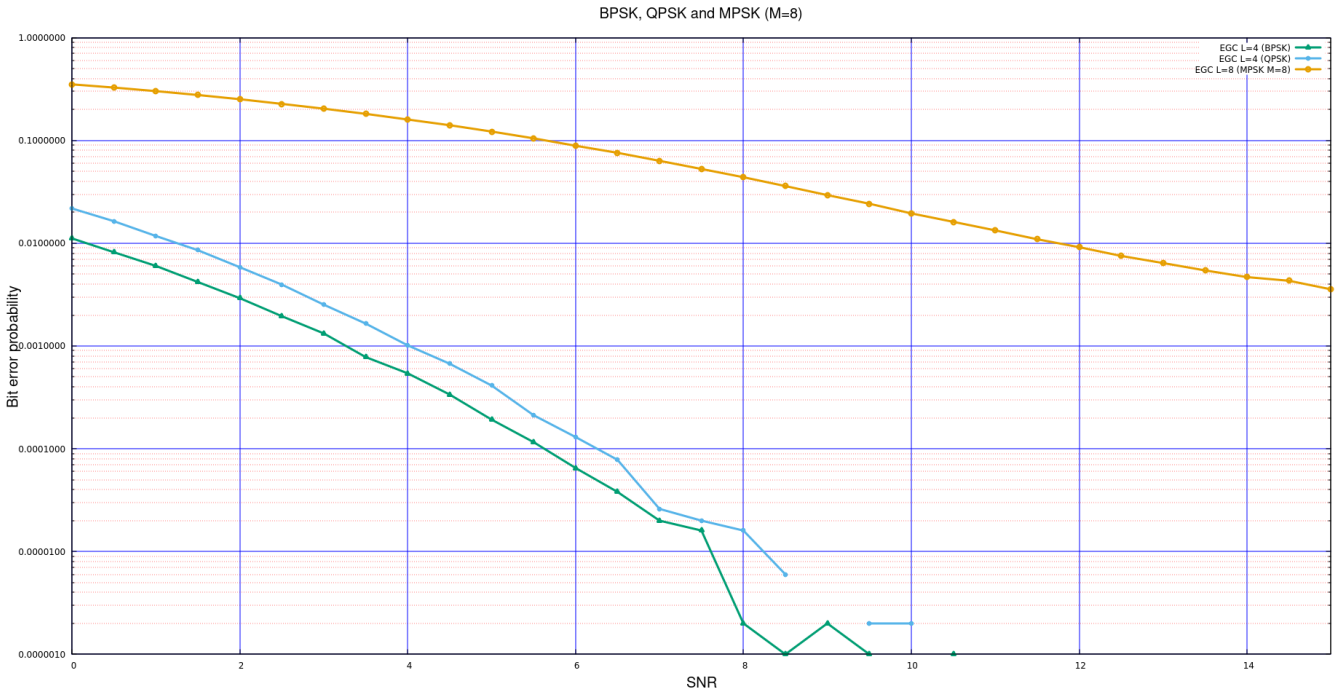

Figure 13: BPSK, QPSK, 8MPSK,

Since, 8MPSK require large SNR even for L=8, futher simulation has been done on QPSK and BPSK.

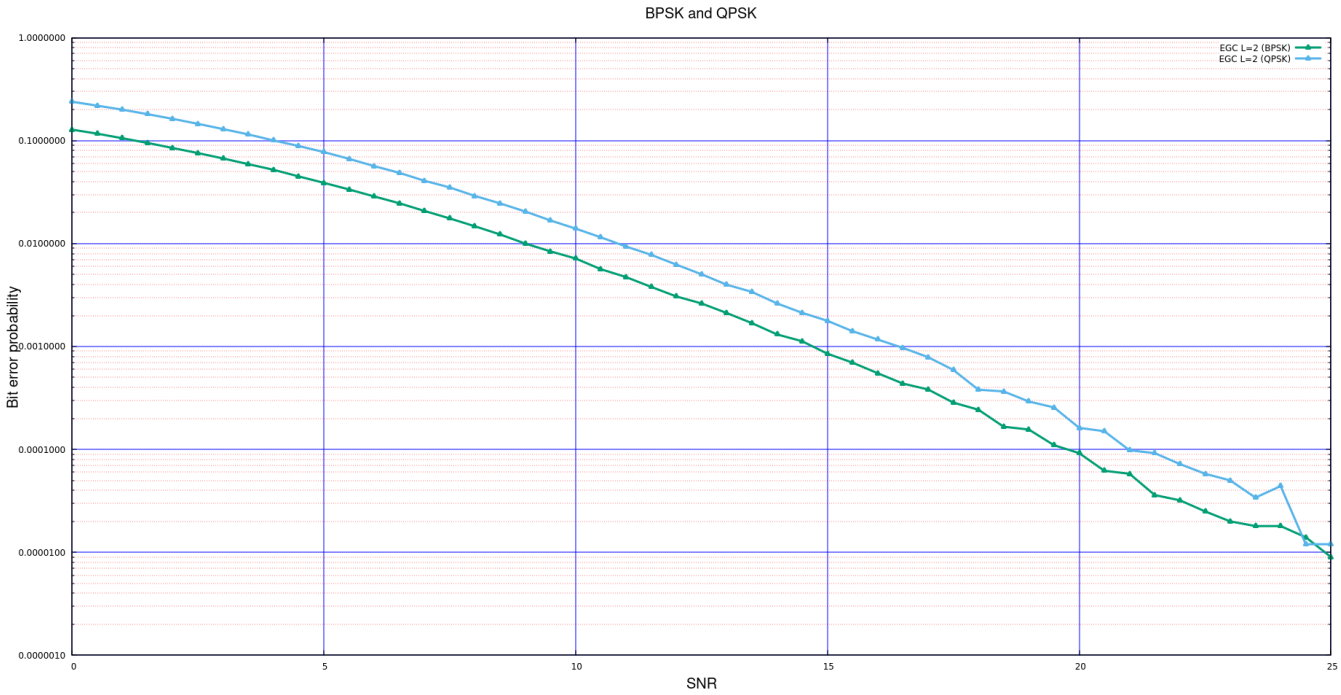

Figure 14: BPSK, and QPSK L=4

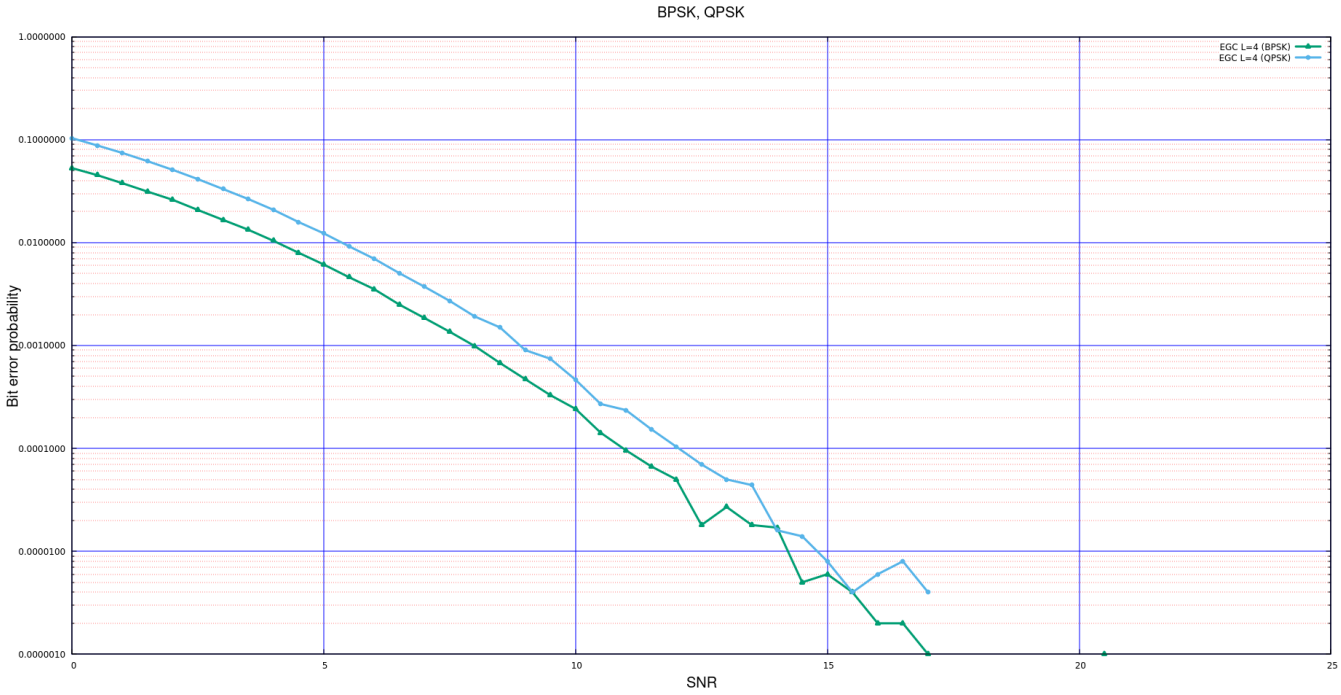

Figure 15: BPSK, and QPSK L=4

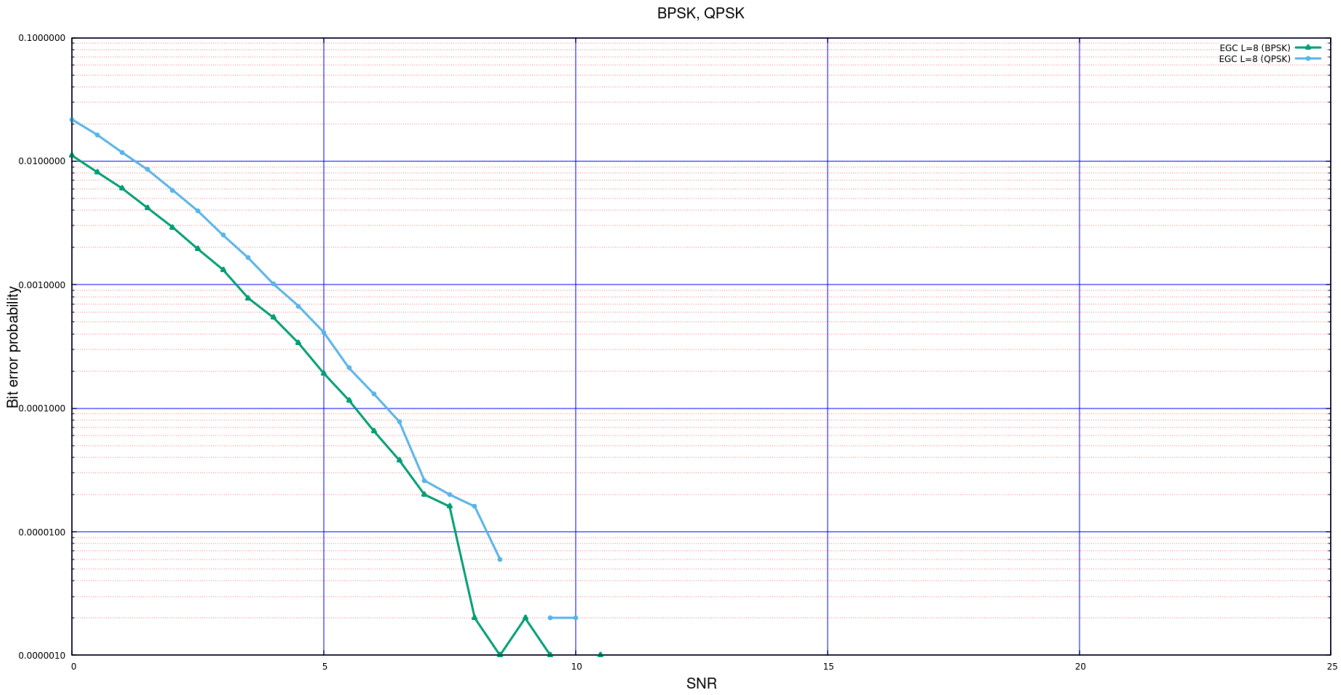

Figure 16: BPSK, and QPSK L=8

6 Inferences

Inferences list:

Based on the final graph

Similar to Selective gain diversity, figure 10, gap between L and L+1 diversity case values are decreasing as L increases. Basically, after certain L, decrements in P_e would be insignificantly small.

Based on Array gain and diversity gain

From figure 11 and figure 12, selective gain and equal gain combining cases are almost parallel to each other. That implies, array gain is same for both cases. But as we increase value of L, gap between EGC values are higher than SG, that implies diversity gain of EGC is higher than SG. Hence overall gain, in EGC should be larger than SG.

Based on the Probability of error

Since, close form of SNR distribution in EGC does not exist, but we can intuitively get the asymptotic approximation of P_e. From figure 1 to figure 4, shows that approximated values are very close to simulated values.

Based on the Different modulations schemes

From figure 13 to figure 16, we can see that even in fadding channel BPSK is best performer among all modulation scheme. And, we can say that for more complex modulation like 8, 16, 32, 64, 128-QAM, one could need very large SNR like 50dB.

7 Result/Conclusion

7.1 What did I learn?

1. I approximated the probability of error for equal gain diversity.
2. I understood how can we improve the performance of fadding channel.
3. Understood the different between EGC and SG based on array gain and diversity gain.
4. Understood the equal gain diversity technique and performance analysis.