WIRELESS COMMUNICATION PRACTISE

EXPERIMENT - 02 (Selective Gain Diversity)

Manas Kumar Mishra (ESD18I011)

25-JANUARY-2022

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

1 Aim:

To analyze the fadding channel with selective gain diversity. To evaluate performance of BPSK from BER (probability of error) vs SNR ratio in fadding channel with selective gain diversity.

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.
Same data can be transmitted in three different ways.

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

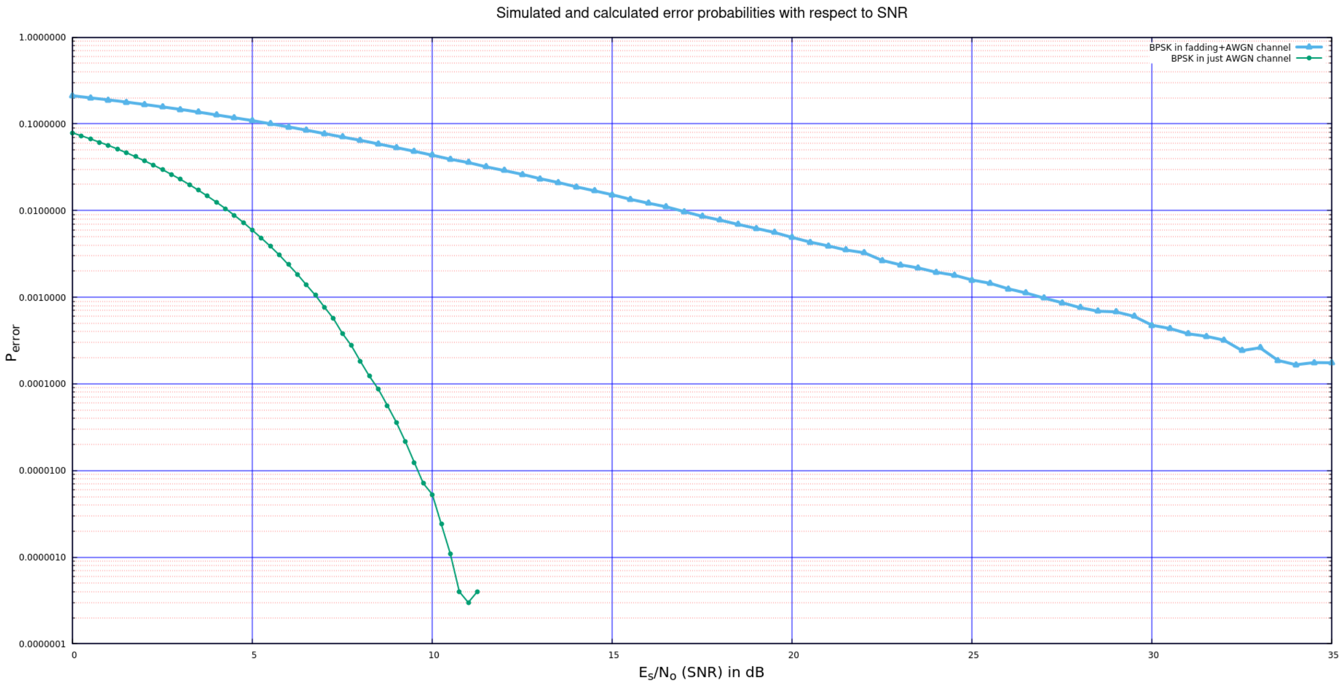

Figure 1: BPSK performance overview (Source:- Experiment-01)

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 Selection gain diversity

This is the receiver based decision technique to improve the performance. In this technique, receiver take decision based on the SNR value of received signal.

3.4 What is it?

Let x is transmitted signal in fadding channel with L diversity. Therefore, receiver receives multiple signals $y_1, y_2 ... y_L$.

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

In selection gain diversity final accepted received signal is

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

$$\theta_i = \begin{cases} 1 & \text{if } h_i^2 \; > \; h_j^2 \;\; \forall \; i \neq j \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Basically, receiver estimate the SNR values of each y_i then choose y_i which has maximum SNR value. The device that perform this task is known as *combiner*. At combiner all received signal assumed to be independent from each other.

3.4.1 SNR

Let x is transmitted symbol of energy E and y is the received signal in fadding channel, then

$$\begin{aligned} y &= hx + n \\ n &\sim N(0, \sigma^2) && \text{(AWGN component)} \\ SNR \;\; \gamma &= \frac{h^2 E}{\sigma^2} && \text{(Signal energy is } h^2E \text{ and Noise energy is } \sigma^2\text{)} \\ Let, \;\; g &= h^2 \\ \gamma &= g \frac{E}{\sigma^2} \\ Let, \;\; \gamma_o &= \frac{E}{\sigma^2} && \text{(SNR in normal AWGN case)} \\ \gamma &= g \gamma_o \end{aligned}$$

Here, it is clear that g is the gain in SNR term. If g turns out to be more than one, then it would be good for receiving end, otherwise, it will impact severely.
In selective gain, g would be maximum among all receive signal case g_i.

$$g = max\{g_1, g_2, ...g_L\}$$

From experiment one, if H is Rayleigh then H^2 would be exponential.

3.4.2 ML rule

Since, in selection gain diversity, there is choice made on received signal by combiner, but does not change decoder. Hence, ML decoder should be same as

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

where, B^o is the decoded bit.

3.4.3 Distribution of G

Let G is a random variable that has g or h^2 as realization. In selective gain case, it is important to know the distribution of the G, because of this combiner suppose to improve the performance.

$$
\begin{aligned}
G &= max\{G_1, G_2, ...G_L\} \\
F_G(g) &= P(G < g) \\
&= P\left(max\{G_1, G_2, ...G_L\} < g\right) \\
&= P\left(G_1 < g,\ G_2 < g,\ ...\ G_L < g\right) \\
&= \prod_{i=1}^{L} P(G_i < g) && \text{(All received signals are independent)} \\
&= \left[1 - \exp(-g)\right]^L && \text{(CDF of exponential distribution)} \\
hence\ \ f_G(g) &= L\ \exp(-g)\left[1 - \exp(-g)\right]^{L-1}
\end{aligned}
$$

3.4.4 Probability of error

$$
\begin{aligned}
P(Error) &= \sum_{i=0}^{1} P(Error, B = i) \\
&= \sum_{i=0}^{1} P(Error|B = i)P(B = i) \\
consider\ \ P(Error|B = 1) &= P(B^o = 0|B = 1) \\
&= \int_0^\infty P(B^o = 0|B = 1, G = g) f_G(g)dg \\
&= \int_0^\infty P(y > 0|B = 0, G = g) f_G(g)dg \\
&= \int_0^\infty P(\eta > \sqrt{gE}) f_G(g)dg \\
&= \int_{g=0}^\infty \int_{x=\sqrt{gE}}^\infty \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{x^2}{2\sigma^2}\right) dx\ f_G(g)dg
\end{aligned}
$$

After plotting the x and g for visualization of limit see figure 2. As of limits of integration like horizontal bars, one can change that into vertical bars. In that, x moves from 0 to ∞ and g moves from 0 to $\frac{x^2}{E}$.

$$
\begin{aligned}
P(B^o = 0|B = 1) &= \int_{x=0}^\infty \int_{g=0}^{\frac{x^2}{E}} \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{x^2}{2\sigma^2}\right) dx\ f_G(g)dg \\
&= \int_{x=0}^\infty \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{x^2}{2\sigma^2}\right) \int_{g=0}^{\frac{x^2}{E}} f_G(g)dg\ dx \\
&= \int_{x=0}^\infty \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{x^2}{2\sigma^2}\right) F_G\left(\frac{x^2}{E}\right)\ dx
\end{aligned}
$$

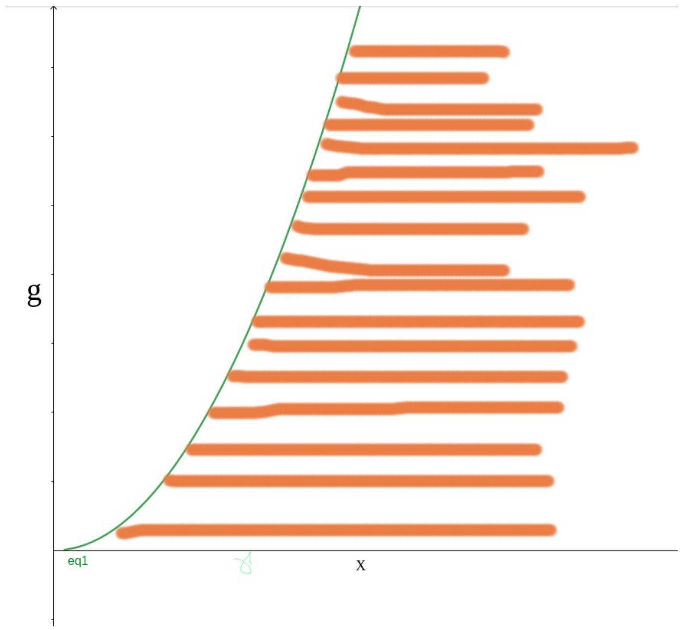

Figure 2: Limit visualization

$$
\begin{aligned}
P(B^o = 0|B = 1) &= \int_{x=0}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{x^2}{2\sigma^2}\right)\ F_G\left(\frac{x^2}{E}\right)\ dx \\
&= \int_{x=0}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{x^2}{2\sigma^2}\right)\ \left[1-\exp(-x^2/E)\right]^L\ dx \\
&= \int_{x=0}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{x^2}{2\sigma^2}\right)\ \sum_{k=0}^{L}\binom{L}{k}(-1)^k \exp\left(-\frac{kx^2}{E}\right)\ dx \quad \text{(Binomial expansion)} \\
&= \sum_{k=0}^{L}\binom{L}{k}(-1)^k \frac{1}{\sqrt{2\pi}\sigma}\int_{x=0}^{\infty} exp\left[-\left[\frac{k}{E}+\frac{1}{2\sigma^2}\right]x^2\right]\ dx \\
Let,\quad \frac{s}{\sqrt{2}} &= x\sqrt{\frac{1}{2\sigma^2}+\frac{k}{E}} \\
dx &= \frac{ds}{\sqrt{2}\sqrt{\frac{1}{2\sigma^2}+\frac{k}{E}}} \\
&= \sum_{k=0}^{L}\binom{L}{k}(-1)^k \frac{1}{\sqrt{2\pi}\sigma}\int_{x=0}^{\infty} exp\left[-\frac{s^2}{2}\right]\ \frac{ds}{\sqrt{2}\sqrt{\frac{1}{2\sigma^2}+\frac{k}{E}}} \\
&= \sum_{k=0}^{L}\binom{L}{k}(-1)^k\ \frac{1}{2}\ \frac{1}{\sigma\sqrt{2}\sqrt{\frac{1}{2\sigma^2}+\frac{k}{E}}} \\
&= \sum_{k=0}^{L}\binom{L}{k}(-1)^k\ \frac{1}{2}\ \frac{1}{\sqrt{1+\frac{2k}{\gamma}}}
\end{aligned}
$$

Above expression has been used to validate the result.

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. Multiply rayleigh coff matrix with itself (element by element) to compute h_i^2
s10. Check every column of h_i^2 matrix and find the index that contain maximum value. (To compute received signal corresponding to this maximum h_i^2).
s11. Find received signal element from received matrix, one element in each column that corresponds to the maximum value index.
s12. Apply ML rule and decode the received signal.
s13. Count errors
S14. Repeat S1 to S13 for many SNR values.
S15. Store data (SNR, Error count) in .dat file.
S16. Compute theoretical probability of error. Store into .dat file
S17. 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 Selection Gain Diversity code

```
////////////////////////////////////////////////////////////////
//////////////////////////////////////
// Author:- MANAS KUMAR MISHRA
// Organisation:- IIITDM KANCHEEPURAM
// Topic:- Performance of selective gain combiner using BPSK in Rayleigh fadding channel
////////////////////////////////////////////////////////////////
//////////////////////////////////////

#include <iostream>
#include <cmath>
```

```
#include <iterator>
#include <random>
#include <chrono>
#include <time.h>
#include <fstream>
#include "BPSK.h"
#include "MyMatrixOperation.h"

#define one_million 1000000

using namespace std;

// Function for printing the vector on the console output.
void PrintVectorDouble(vector<double> vectr)
{
    std::copy(begin(vectr), end(vectr),  std::ostream_iterator<double>(std::cout, "
"));
    cout<<endl;
}


// 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 Rayleigh fadding cofficients
// Inputs are two vectors one is the gaussion noise as real
//part and second as Gaussian noise as imazinary part
// Output is a vector that contain rayliegh noise coff, sqrt(real_part^2 + imz_part^2)
vector<double> RayleighFaddingCoff(vector<double> realGaussian,
                                    vector<double> ImziGaussian)
{
    vector<double> rayleighNoise;
    double temp;

    for(int times=0; times<realGaussian.size(); times++){

        temp = sqrt(pow(realGaussian[times], 2)+pow(ImziGaussian[times], 2));
        rayleighNoise.insert(rayleighNoise.end(), temp);
    }

```

```
    return rayleighNoise;
}


// Function for finding index that contain maximum value in every column
// Input is a matrix
// Output is the vector with each element as index
vector<double> MaximumSNRIndex(vector<vector<double>> SNRMat)
{
    vector<double> MaxIndexes;
    double index;

    for(int i =0; i<SNRMat[0].size(); i++){
        index =0;

        for(int j =0; j<SNRMat.size(); j++){

            if(SNRMat[index][i]<SNRMat[j][i]){
                index = j;
            }
        }

        MaxIndexes.insert(MaxIndexes.end(), index);
    }

    return MaxIndexes;
}


// 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 for factorial of n
// Input is a integer number greater than 0
// Output is the factorial result
double factorial(double Num)
{
    if(Num==1){
        return 1;
    }
    else if(Num ==0){
```

```
        return 1;
    }
    else if(Num<0){
        cout<< "Worng Output"<<endl;
        return 0;
    }
    else{
        return (Num*factorial(Num-1));
    }
}


// Function for the combination l_C_k
// Inputs are the two integer numbers l and k
// Output is the answer of the l choose K.
double l_Choose_K(double l, double k)
{
    double ans;
    if(l<k){
        cout<<"L should not be less than k!!!"<<endl;
        return 0;
    }
    else{
        ans = factorial(l)/(factorial(k)*factorial(l-k));
        return ans;
    }
}

double errorCalculation(double l, double snr)
{
    vector<double> calerror;
    double fac, val, error;

    for(int i =0; i<=l; i++){
        val = 1/pow((1+(2*i/snr)),0.5);
        fac = l_Choose_K(l,i);
        val = pow(-1, i)*(val/2);
        calerror.insert(calerror.end(), fac*val);
    }
    error =0;
    for(int i =0; i<calerror.size(); i++){
        error = error + calerror[i];
    }

    calerror.clear();

    return error;
}



vector<double> calculatedError(vector <double> SNR_dB, double l)
{

    vector <double> Qvalue;

```

```
    double po, normalValue, inter;
    for (int k =0; k<SNR_dB.size(); k++){
        normalValue = pow(10, (SNR_dB[k]/10));
        po = errorCalculation(l, 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 ErrorValueInFile(vector <double> SNR, vector <double> Qvalue)
{
    ofstream outfile;

    outfile.open("SGcalL3.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();
}


// 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("SelectiveGainL3.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();
}


int main(){
```

```

    // source defination
    vector<double> sourceBits;

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

    // Noise definition
    vector<double> gnoise;

    vector<double> realGaussian;
    vector<double> imziGaussian;
    vector<double> RayleighNoise;

    vector<double> MaxiSNRdetection;

    vector<double> combinerOutput;
    vector<double> decodedBits;

    double sigmaSquare = 0.5;
    double stddevRayleigh = sqrt(sigmaSquare);
    double N_o =4;
    double p, stdnoise;
    stdnoise = sqrt(N_o);
    double counterror, P_error;

    double L =3;

    vector<vector<double>> MultipleSignals;
    vector<vector<double>> RaleighMat;
    vector<vector<double>> SNRMat;
    vector<vector<double>> GnoiseMat;
    vector<vector<double>> NewMat;
    vector<vector<double>> ReceiveMat;

    vector<double> SNR_dB;
    for(float i =0; i<=25; i=i+0.5)
    {
        SNR_dB.insert(SNR_dB.end(), i);
    }

    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], one_million);

        for(int i =0; i<L; i++){
            MultipleSignals.insert(MultipleSignals.end(), transmittedSymbol);
        }

        for(int i =0; i<L; i++){
            realGaussian = GnoiseVector(0.0, stddevRayleigh, one_million);
            imziGaussian = GnoiseVector(0.0, stddevRayleigh, one_million);

            RayleighNoise = RayleighFaddingCoff(realGaussian, imziGaussian);

            RaleighMat.insert(RaleighMat.end(), RayleighNoise);

            // realGaussian.clear();
            // imziGaussian.clear();
            // RayleighNoise.clear();
        }

        for(int i=0; i<L; i++){
            gnoise = GnoiseVector(0.0, stdnoise, one_million);
            GnoiseMat.insert(GnoiseMat.end(), gnoise);
            gnoise.clear();
        }

        NewMat = ElementWiseMultiplication(MultipleSignals, RaleighMat);

        ReceiveMat = ElementWiseAddition(NewMat, GnoiseMat);

        SNRMat = ElementWiseMultiplication(RaleighMat, RaleighMat);

        MaxiSNRdetection = MaximumSNRIndex(SNRMat);

        for(int j =0; j<MaxiSNRdetection.size(); j++){
            combinerOutput.insert(combinerOutput.end(),
            ReceiveMat[MaxiSNRdetection[j]][j]);
        }

        decodedBits = decisionBlock(combinerOutput);

        counterror = errorCalculation(sourceBits, decodedBits);

        P_error = counterror/one_million;
        Prob_error.insert(Prob_error.end(), P_error);

        cout<<"Error count: "<< counterror<<endl;
        cout<<"P_e:         "<< P_error<<endl;
        cout<<endl;

        MultipleSignals.clear();
        NewMat.clear();
        RaleighMat.clear();
        GnoiseMat.clear();
        ReceiveMat.clear();
        SNRMat.clear();
```

```
        combinerOutput.clear();
    }

    datafile(SNR_dB, Prob_error);
    vector<double> qvalue = calculatedError(SNR_dB, L);
    ErrorValueInFile(SNR_dB, qvalue);

    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>
```

```cpp
#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)
{
    vector<vector<double>> Mat;
    Mat.insert(Mat.end(), Vec);
    return Mat;
}


// Function for Multiplying two matrixs element by elements
// Input are two matrix of same dimension
// Output is another matrix that contain each element
```

```
//  as multiplication of each element.
vector<vector<double>> ElementWiseMultiplication(vector<vector<double>> Mat1,
                                                  vector<vector<double>> Mat2)
{
    vector<vector<double>> MatResult;

    vector<double> multi;

    for(int i =0; i<Mat1.size();i++){
        for(int j=0; j<Mat1[0].size(); j++){
            multi.insert(multi.end(), Mat1[i][j]*Mat2[i][j]);
        }

        MatResult.insert(MatResult.end(), multi);
        multi.clear();
    }

    return MatResult;
}


// Function for Adding two matrixs element by elements
// Input are two matrix of same dimension
// Output is another matrix that contain each element
//  as Addition of both element.
vector<vector<double>> ElementWiseAddition(vector<vector<double>> Mat1,
                                            vector<vector<double>> Mat2)
{

    vector<vector<double>> MatResult;

    vector<double> Add;

    for(int i =0; i<Mat1.size();i++){
        for(int j=0; j<Mat1[0].size(); j++){
            Add.insert(Add.end(), Mat1[i][j]+Mat2[i][j]);
        }

        MatResult.insert(MatResult.end(), Add);
        Add.clear();
    }

    return MatResult;
}
```

5.2 Results

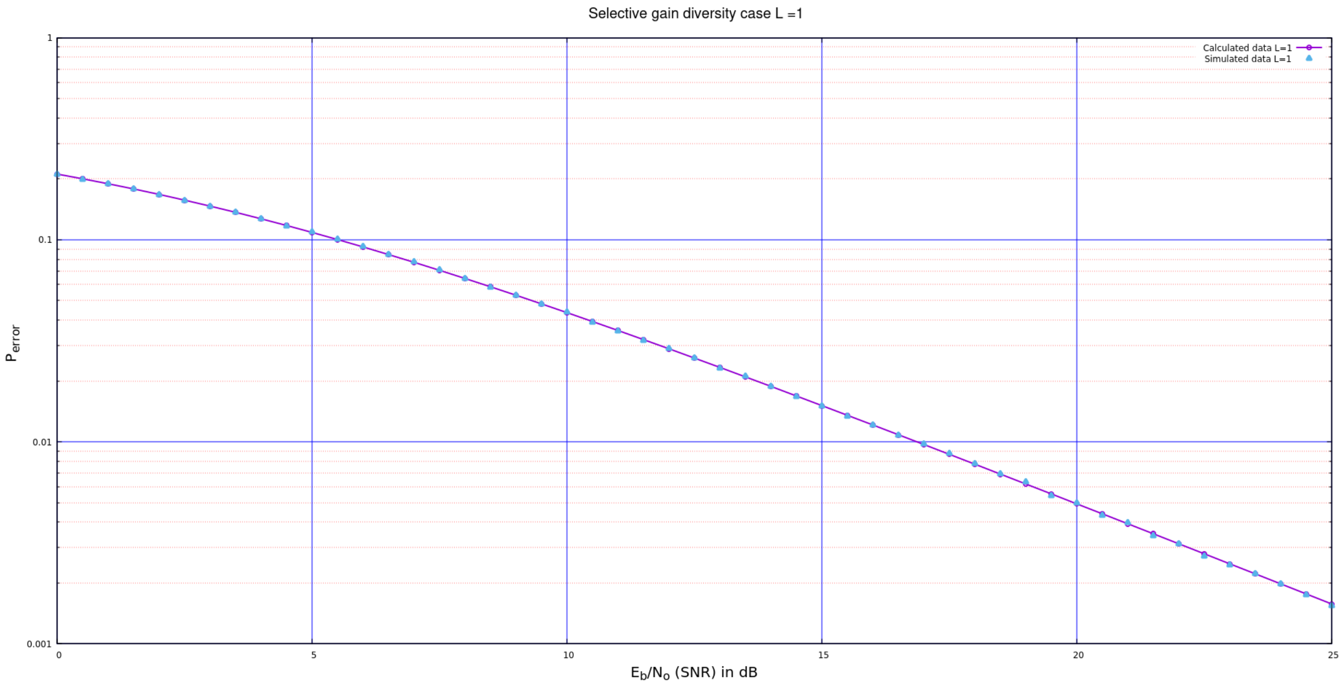

Figure 3: L=1 case simulated and calculated

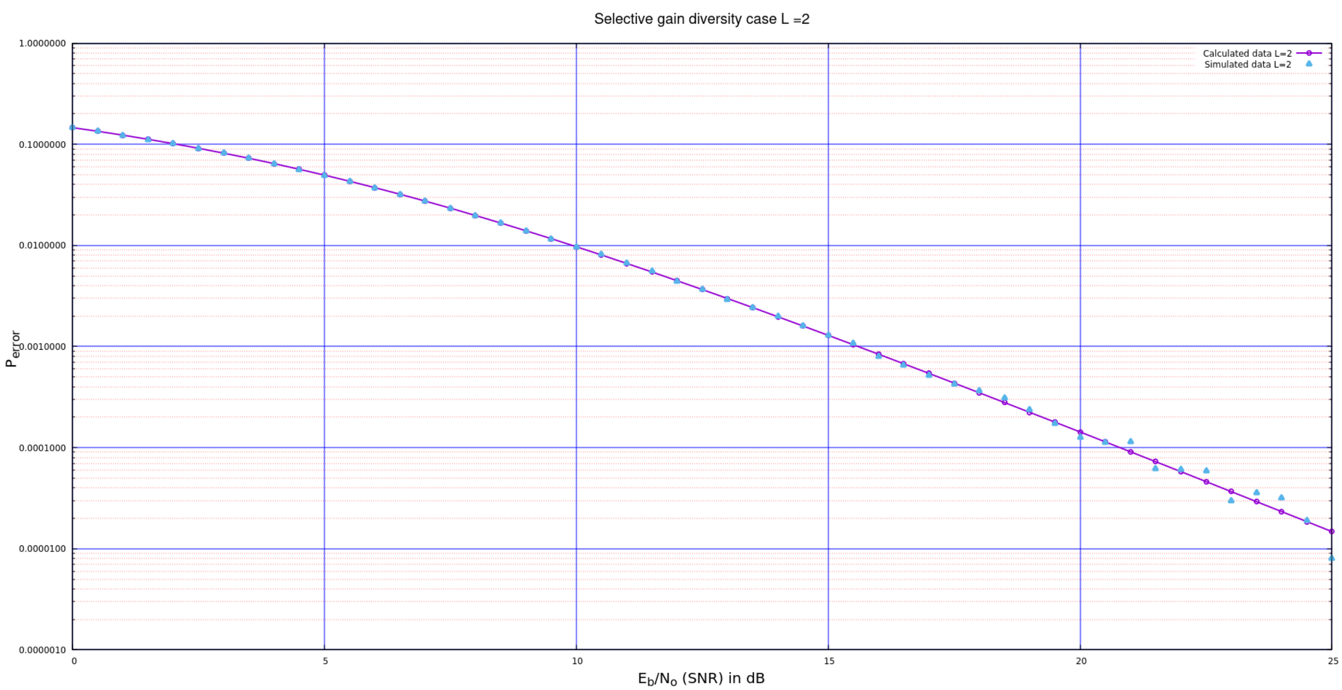

Figure 4: L=2 case simulated and calculated

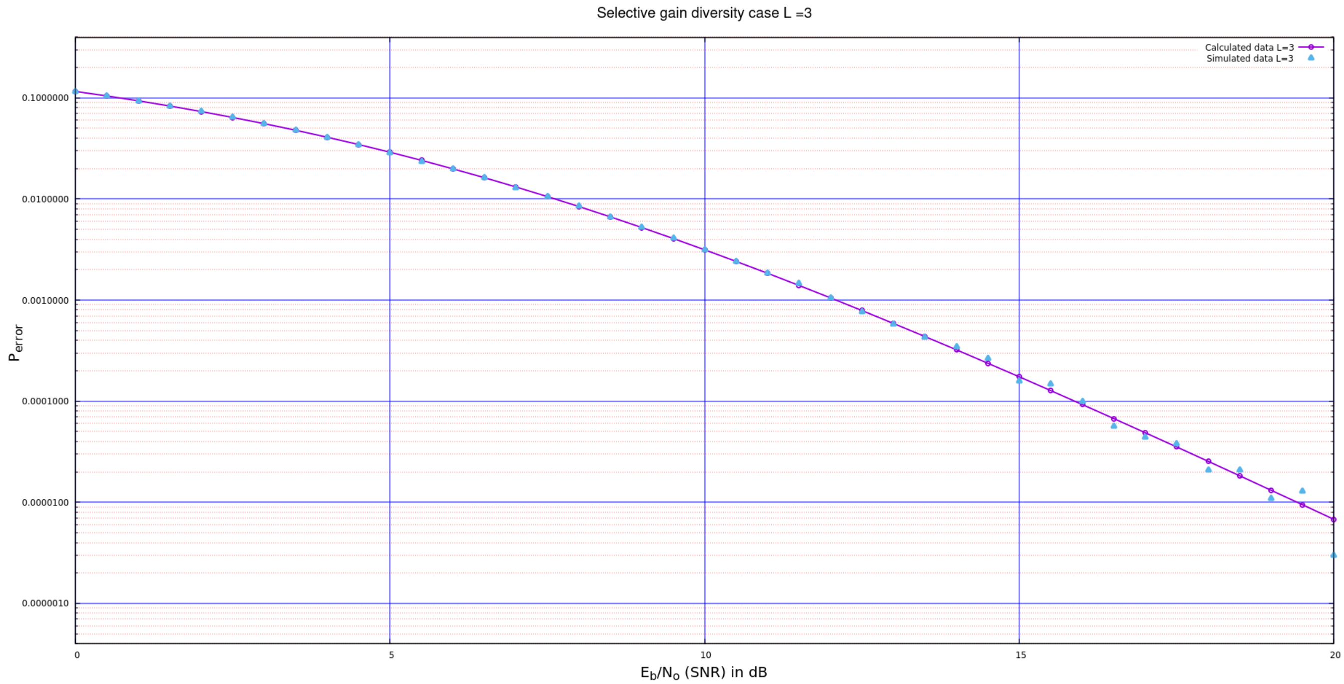

Figure 5: L=3 case simulated and calculated

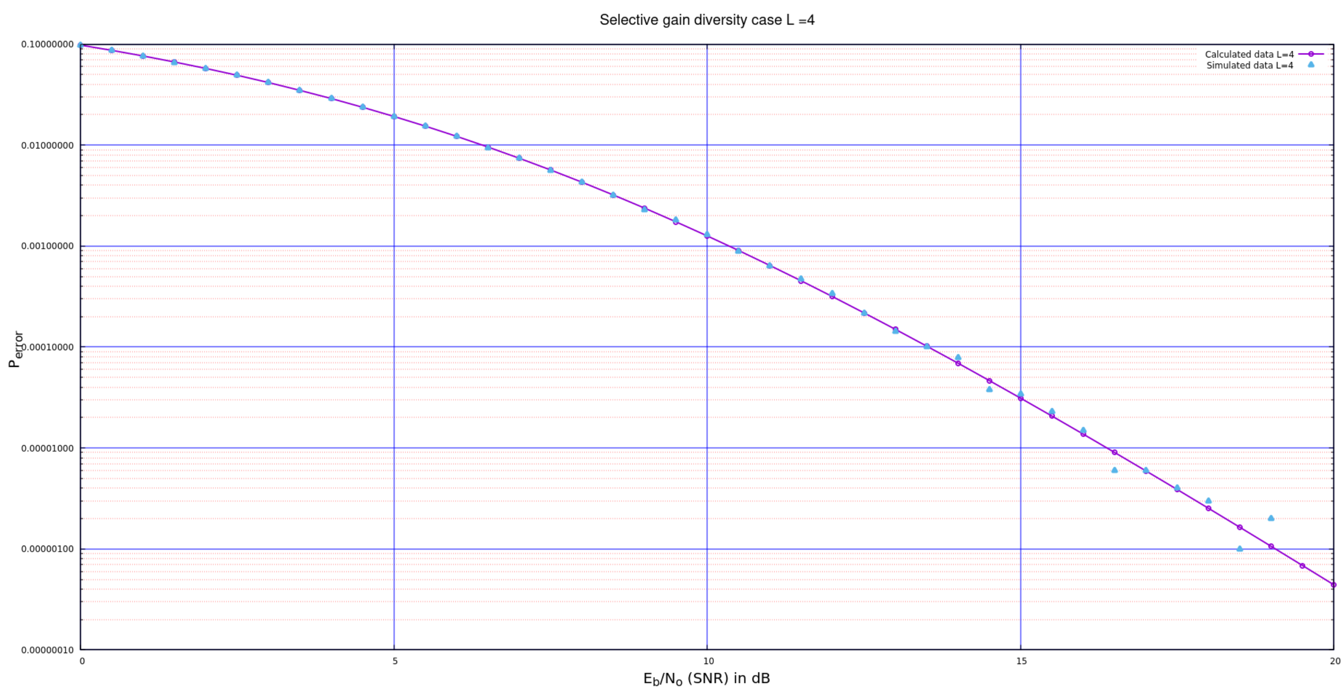

Figure 6: L=4 case simulated and calculated

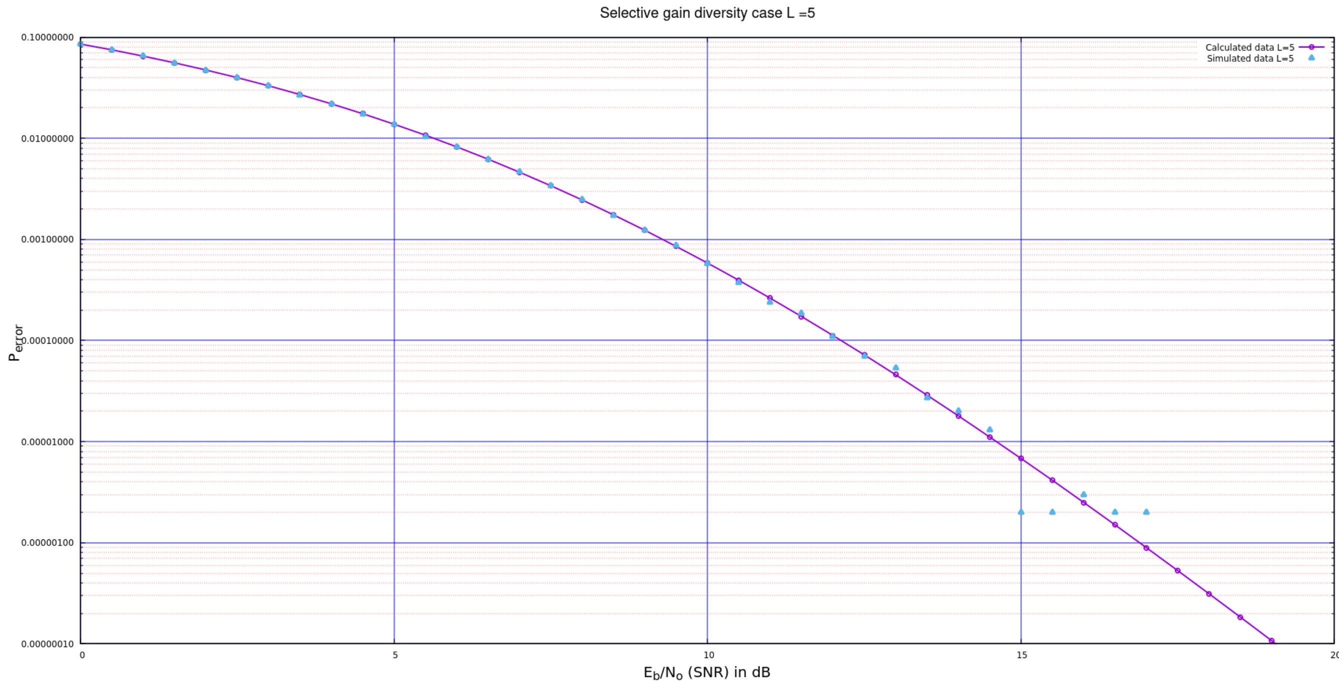

Figure 7: L=5 case simulated and calculated

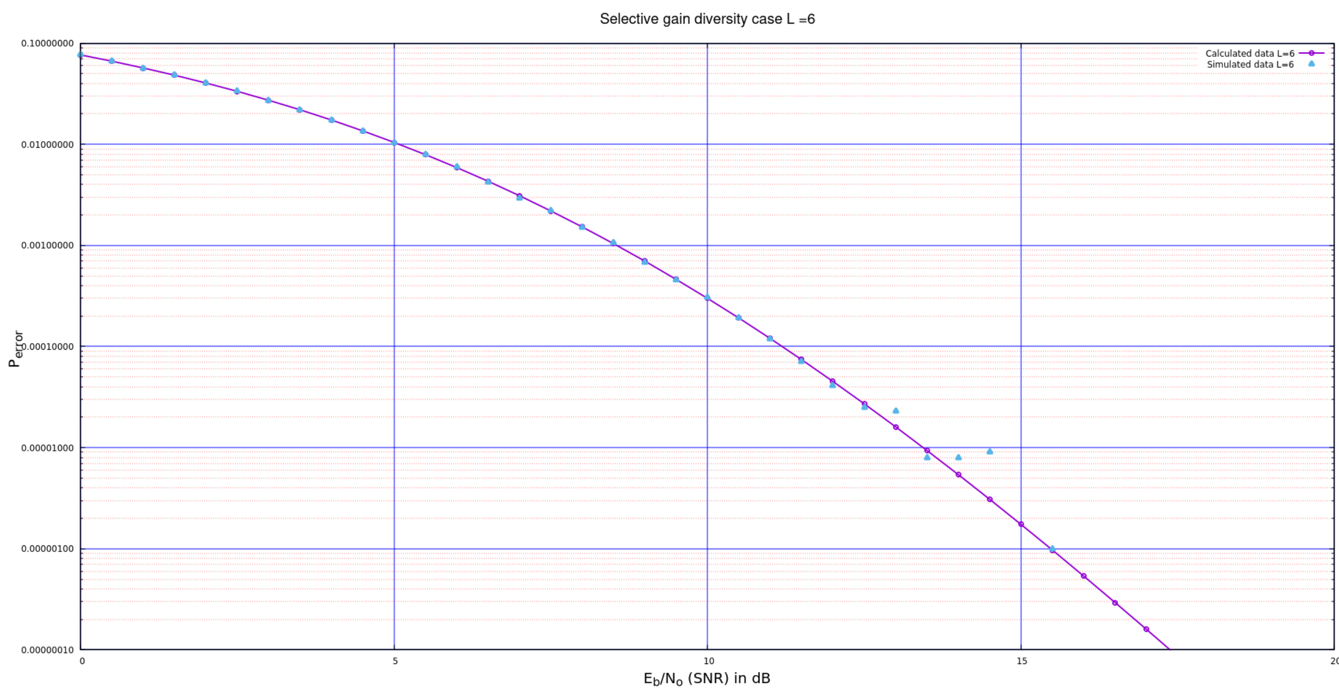

Figure 8: L=6 case simulated and calculated

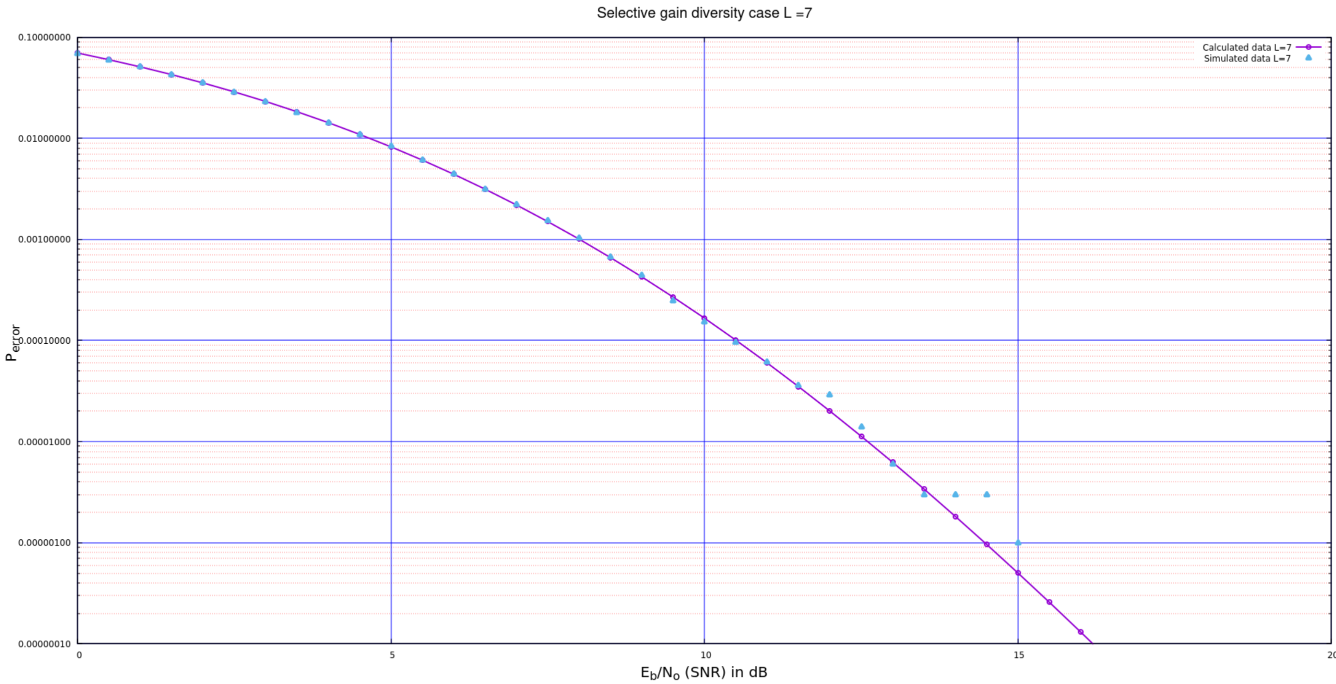

Figure 9: L=7 case simulated and calculated

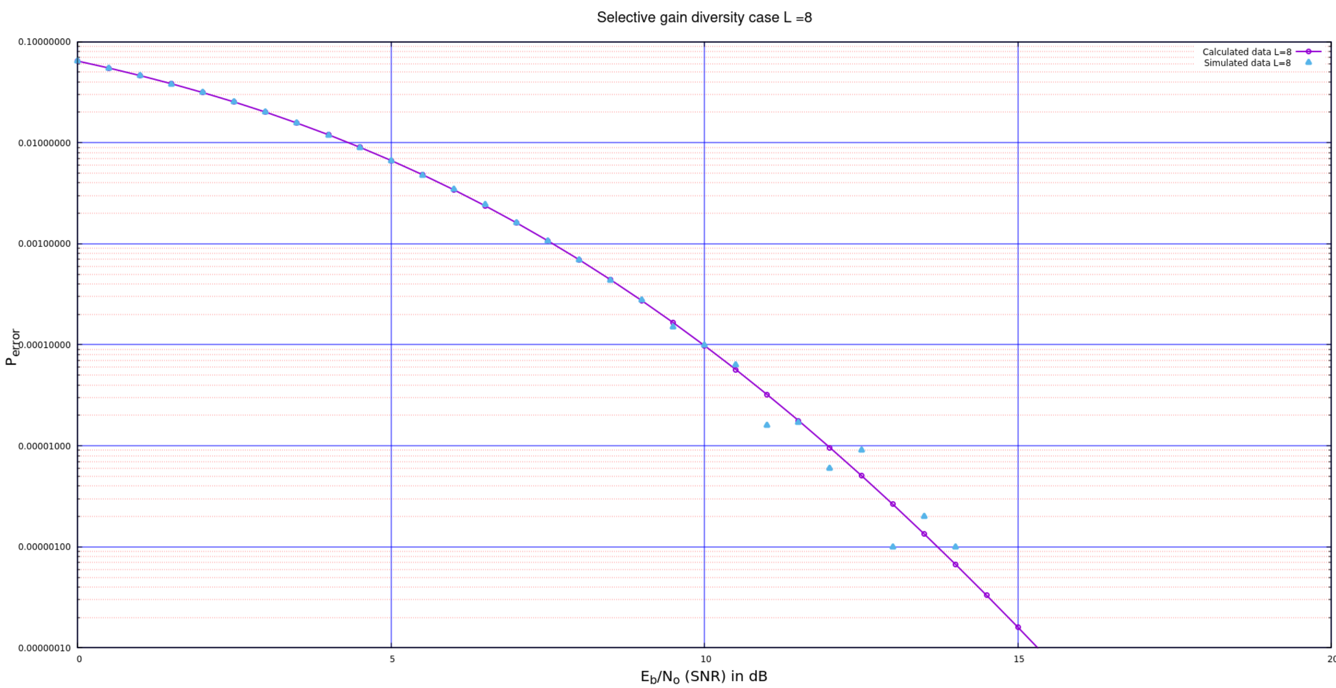

Figure 10: L=8 case simulated and calculated

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

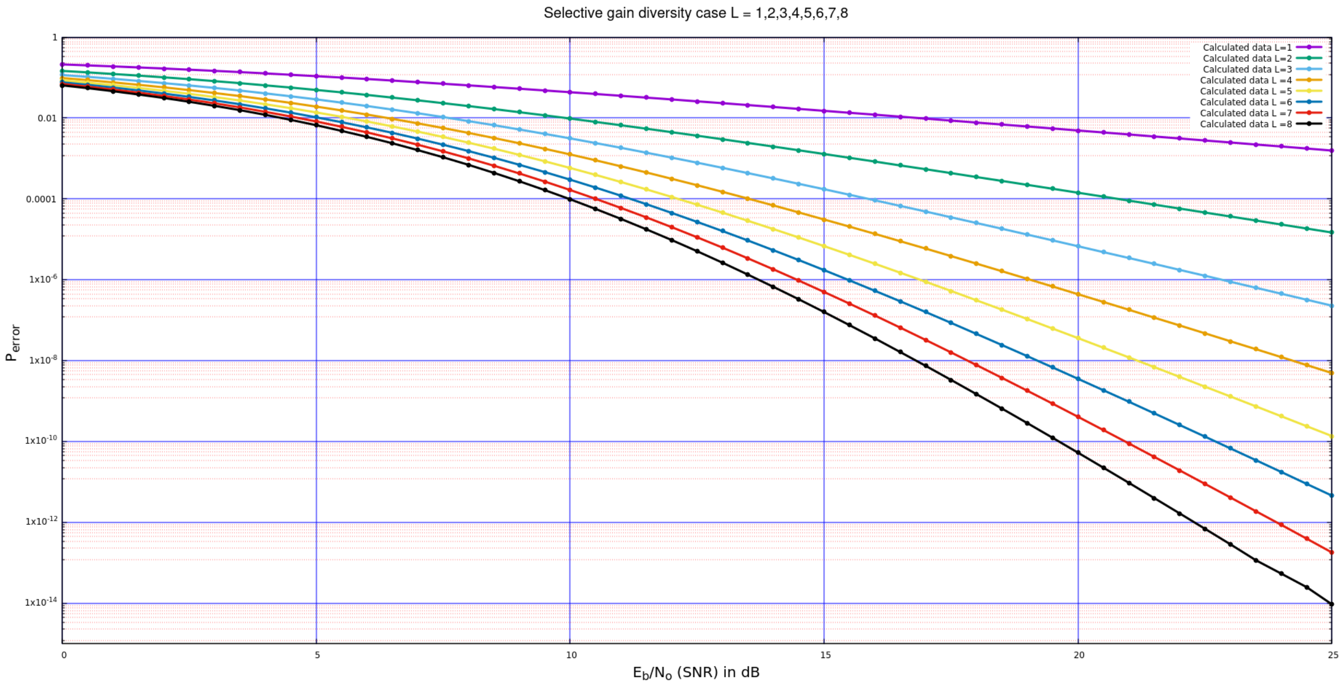

Figure 11: All case Calculated data

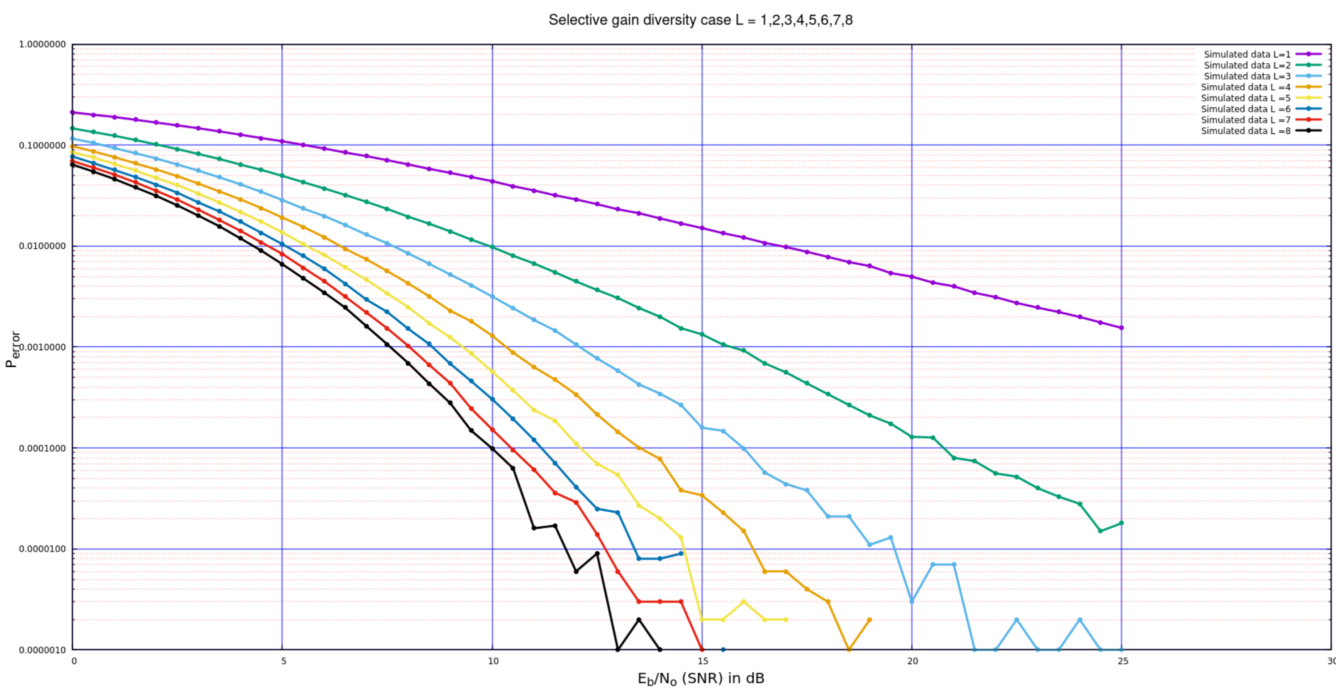

Figure 12: All case Simulated data

6 Inferences

Inferences list:
Based on the final graph
As we increase L, Probability of error (P_e) decreases for a given SNR, but decrements of values, in successive values of the L, decreases. In simple words, as we can see in figure 11, gap between L and L+1 values are decreasing as L increases. Basically, after certain L, decrements in P_e would be insignificantly small.
Based on the L=8 and AWGN
From figure 12 and figure 1, we can see that values of $L = 8$ case is very close to the AWGN values. That implies, if we increase L more, then performance may exceed the only AWGN channel performance. Certainly through diversity, one can design systems more better than AWGN.
Based on the L values
As we increase L performance of channel increases, but for that we need to pay some cost. In time diversity, we loss data rates and in frequency diversity, we have to supply same signal at multiple frequencies at same time, that will cause more power at transmitter end and more channel resource utilization for one communication link (not efficient). Based on the delay and error requirements of application, one need to decide value of L.

7 Result/Conclusion

7.1 What did I learn?

1. I derived the probability of error for selection gain diversity.
2. I understood how can we improve the performance of fadding channel.
3. Calculated and simulated P_e are same.
4. Understood the Selection gain diversity and use in improvement of fadding channel.