WIRELESS COMMUNICATION PRACTISE

EXPERIMENT - 04 (Maximum Ratio Combiner)

Manas Kumar Mishra (ESD18I011)

08-FEBRUARY-2022

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

1 Aim:

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

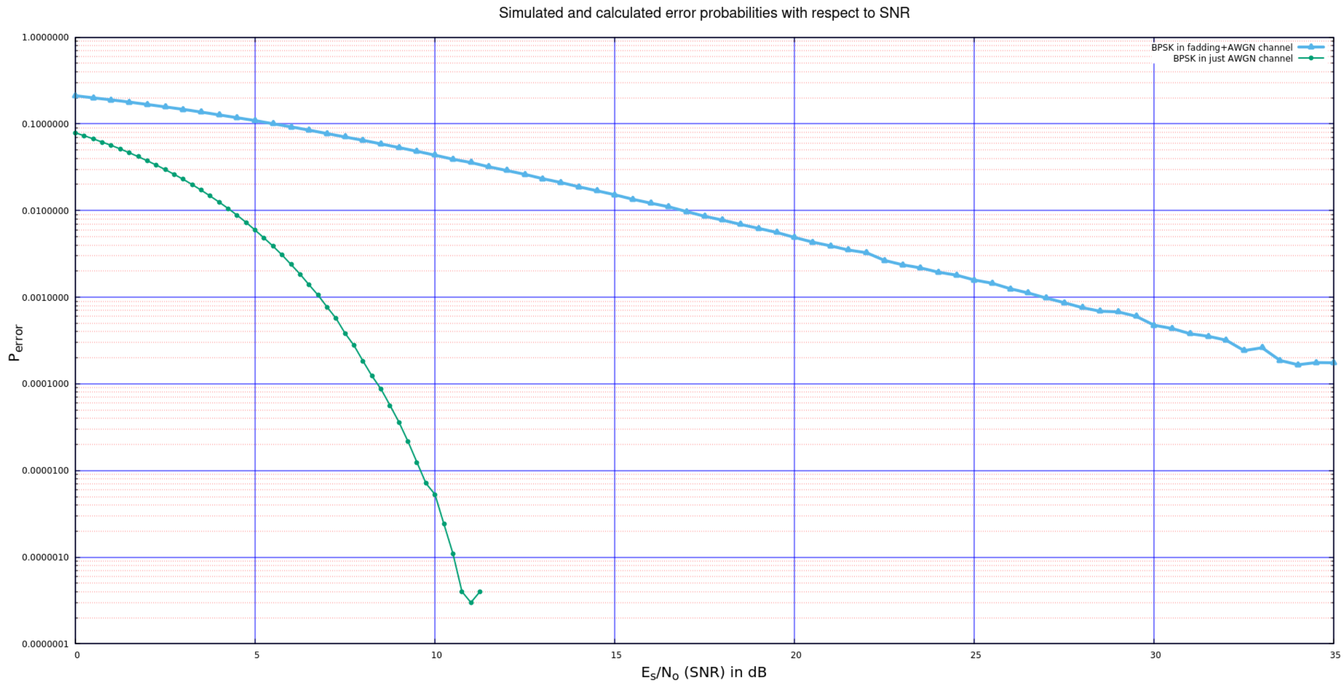

Figure 1: BPSK performance overview (Source:- Experiment-01)

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 Maximum ratio combiner

This is the receiver based decision technique to improve the performance. In that, receiver receives the signals from each antenna branch and give appropriate weights after co-phasing. Then receiver combines all weighted co-phased signal algebraically. After that combined signal passes through decoder block.

3.4 What is it?

Let x is transmitted signal in slow fadding channel or flat fadding channel with L diversity (L different receiver antennas). 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 maximum ratio combining accepted received signal are co-phased and multiplied with weights to maximize the overall SNR. Now the question is how to chooses weights to maximize the SNR.

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

$$where \;\; \theta_i = \alpha_i^* \;\; \forall \;\; 1 \le i \le L \qquad\qquad (\alpha_i^* \text{ weights of } i^{th} \text{ branch})$$

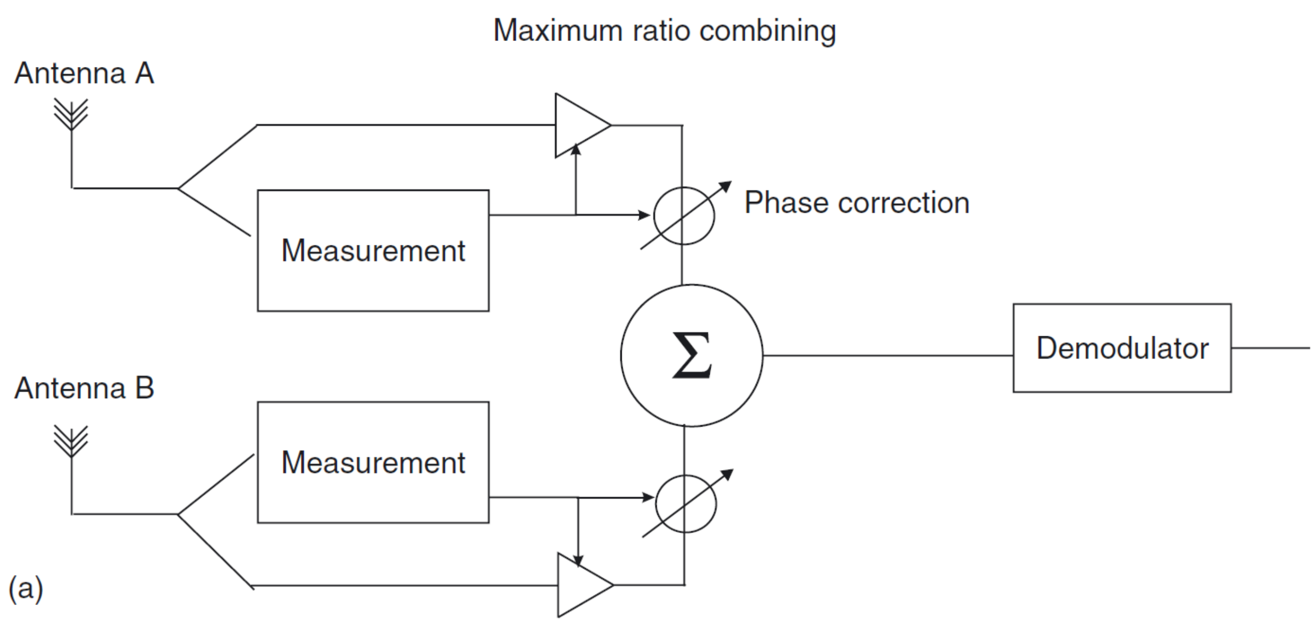

Figure 2: MRC for two antenna diversity (A simplified model)

This is a very simplified block diagram of MRC only for understanding purpose.

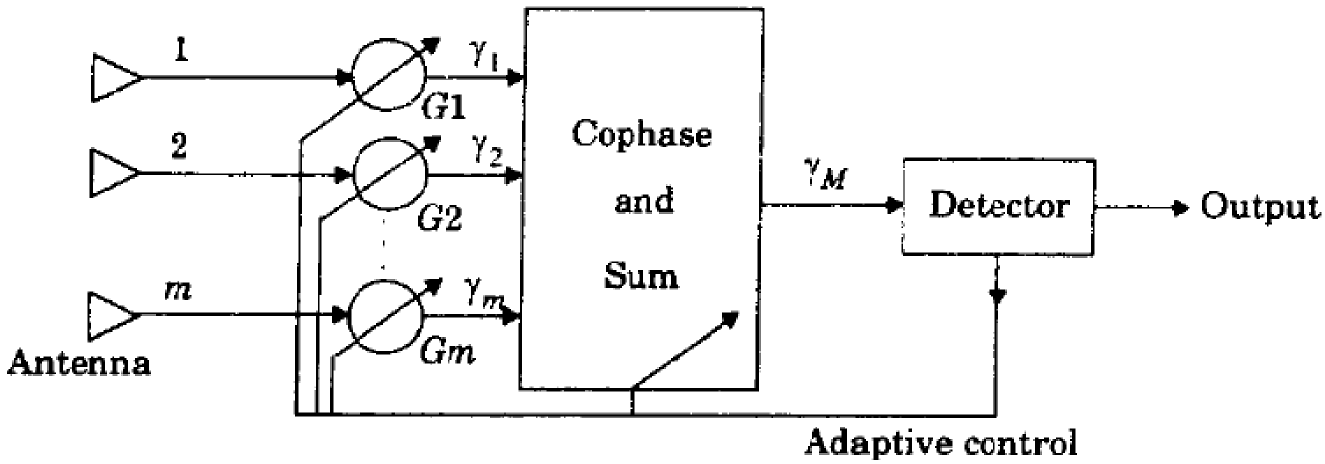

Figure 3: MRC block diagram with adaptive control

3.4.1 How to choose weights?

Final signal, after combiner

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

$$SNR = \frac{\left(\sum_{i=1}^{L}(h_i\alpha_i^*)\right)^2 E_s}{\sigma^2\sum_{i=1}^{L}(\alpha_i^*)^2}$$

$$\left(\sum_{i=1}^{L}(h_i\alpha_i^*)\right)^2 \leq \sum_{i=1}^{L}|h_i|^2\sum_{i=1}^{L}|\alpha_i^*|^2 \qquad \text{(Cauchy–Schwartz inequality)}$$

$$Maximum\ attains\ at\ equality\ h_i = \alpha_i$$

Therefore, fadding coefficients are itself a optimum weights for the MRC. But now question is, how to estimate this optimum weight through received signal. These details are beyond the scope of this experiment.
Since this technique attains maximum SNR through weights that's why it is known as maximum ratio combining.

3.4.2 SNR

Let x is transmitted symbol of energy E and y_i is the received signal in fadding channel at i^{th} antenna element, and consider weights of i^{th} branch is exactly equal to the h_i. Then SNR is

$$
\begin{aligned}
y &= \left(\sum_{i=1}^{L} h_i^2\right) x + \sum_{i=1}^{L} h_i n_i \\
n_i &\sim N(0, \sigma^2) && \text{(AWGN component)} \\
\sum_{i=1}^{L} h_i n_i &\sim N(0, \sum_{i}^{L} h_i^2 \sigma^2) \\
SNR \;\; \gamma &= \frac{\left(\sum_{i=1}^{L} h_i^2\right)^2 E}{\left(\sum_{i=1}^{L} h_i^2\right) \sigma^2} \\
\gamma &= \frac{\left(\sum_{i=1}^{L} h_i^2\right) E}{\sigma^2} \\
Let, \;\; g &= \left(\sum_{i=1}^{L} h_i^2\right) \\
\gamma &= g \frac{E}{\sigma^2}
\end{aligned}
$$

Here, one can observe that g is a random variable (function of a random variable h_i). That makes, difficult to infer anything from single gain values. Because, at some instant gain should be very high or very low. In other form, one can represent the γ as,

$$
\begin{aligned}
\gamma &= \sum_{i}^{L} \gamma_i && (\gamma_i \text{ is SNR in } i^{th} \text{ branch}) \\
\gamma_i &= h_i^2 \frac{E}{\sigma^2}
\end{aligned}
$$

3.4.3 Average SNR

From the previous experiments, recall that distribution of the h_i^2 is exponential with parameter λ. Where λ is statically property of the exponential distribution known as inverse of the mean of the exponential distribution. One more important point is to observe, distribution of γ_i is same as distribution of h_i^2. Therefore, mean of γ_i will be mean of h_i^2 multiples with E/σ^2. Now consider Γ as mean of γ_i, mean SNR in i^{th} branch.

$$
\begin{aligned}
\mathbb{E}[\gamma_i] &= \Gamma \\
\mathbb{E}[\gamma_{MRC}] &= \mathbb{E}\left(\sum_{i=1}^{L} \gamma_i\right) \\
\mathbb{E}[\gamma_{MRC}] &= L\,\Gamma
\end{aligned}
$$

Hence, in MRC, average SNR is the just sum of all SNRs in each antenna branch.

3.4.4 Comparison of gain and average SNR in SG, EGC and MRC

In SG

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

In EGC

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

In MRC

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

That is straight forward

$$g_{MRC} \geq g_{EGC} > g_{SG}$$

Similar, trend can be observed through average SNR. One should rely on the average SNR results, because, in diversity, gain is a random variable (function of h_i that is random variable).

$$\mathbb{E}[\gamma_{MRC}] \;\; \geq \;\; \mathbb{E}[\gamma_{EGC}] \;\; > \;\; \mathbb{E}[\gamma_{SG}]$$

3.4.5 ML rule

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

$$\begin{aligned} if,\;\; 0 &\mapsto \sqrt{E} \\ 1 &\mapsto -\sqrt{E} \\ then\;\; y &\gtrless_{B^o=1}^{B^o=0} 0 \\ where\;\; y &= \sum_{i=1}^{L} y_i h_i \end{aligned}$$

where, B^o is the decoded bit.

3.4.6 Distribution of SNR

$$\begin{aligned} g_i &\sim exp(1) \\ f_g(g_i) &= e^{-g_i} && (g_i > 0) \\ G &= \sum_{i=1}^{L} g_i \\ f_G(x) &= \frac{x^{L-1}}{(L-1)!}\, e^{-x} && (\chi \text{ square distribution}) \end{aligned}$$

In the form of moment generating function

$$\begin{aligned} \mathbb{M}_{g_i}(s) &= \frac{1}{1-s} \\ \mathbb{M}_{G}(s) &= \frac{1}{(1-s)^L} \end{aligned}$$

3.4.7 Probability of error

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

Alternative form of Q-function

$$
Q(x) = \frac{1}{\pi}\int_{\phi=0}^{\frac{\pi}{2}} exp\left(-\frac{x^2}{2sin^2(\phi)}\right) d\phi
$$

For alternative form of Q-function, please refer Goldsmith chapter 6 section 6.2.

$$
\begin{aligned}
P(Error|B=1) &= \frac{1}{\pi}\int_{\phi=0}^{\frac{\pi}{2}}\int_{g=0}^\infty f_G(g) e^{-\left(\frac{gE}{\sigma^2\, 2\, sin^2(\phi)}\right)} dg\, d\phi \\
&= \frac{1}{\pi}\int_{\phi=0}^{\frac{\pi}{2}} \mathbb{M}\left(-\frac{SNR}{2sin^2(\phi)}\right) d\phi \qquad (\text{SNR} = \tfrac{E}{\sigma^2}) \\
&= \frac{1}{\pi}\int_{\phi=0}^{\frac{\pi}{2}} \frac{1}{\left(1+\frac{SNR}{2sin^2(\phi)}\right)^L} d\phi
\end{aligned}
$$

This expression has been used for computing probability of error at any SNR value and for any number of antenna element L. To calculate integration of the function, a header file has been designed just for calculating integration.

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. After adding with gaussian noise, multiply with rayleigh coff.
s10. Take sum of over all rows. Final resultant vector would be received vector.
s11. Apply ML rule and decode the received signal.
s12. Count errors
S13. Repeat S1 to S13 for many SNR values.
S14. Store data (SNR, Error count) in .dat file.
S15. Compute theoretical probability of error. Store into .dat file
S16. Plot the result using gnuplot in semilog scale.

5 Code and result

To support matrix operation a new header file has been designed and added to the code. Similarly BPSK channel a header file has been used.

5.1 Maximum ratio combiner code

```
////////////////////////////////////////////////////////////
//////////////////////////////////////
// Author:- MANAS KUMAR MISHRA
// Organisation:- IIITDM KANCHEEPURAM
// Topic:- Performance of maximum ratio combiner using BPSK in
//Rayleigh fading channel
////////////////////////////////////////////////////////////
//////////////////////////////////////

/*
Flow of the program
SourceBIts --> Modulation --> Multiply with Rayleigh (L different branch)
--> Add gaussian noise in each branch --> Multiply with Rayleigh in each branch
--> Combine all values -->Put into BPSK decoder --> Count errors.
*/

#include <iostream>
```

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

#define one_million 1000000
#define pi 3.14179

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 multiplying fadding coff to particular antenna
// Inputs are the Transmitted energy and Rayleigh fadding coff
// Output is the multiplication of element by element fadding and energy
vector<double> RayleighOperation(vector<double> TxEnergy,
                                 vector<double> RayleighFaddingCoff)
{
    vector<double> Resultant;

    for(int j =0; j<TxEnergy.size(); j++){
        Resultant.insert(Resultant.end(), TxEnergy[j]*RayleighFaddingCoff[j]);
    }

    return Resultant;
}


// Function for gaussian noise for each tx bit to particular antenna
// Inputs are the Transmitted energy and Gnoise
// Output is the addition of element by element Gnoise and Tx
vector<double> GaussianNoiseAdd(vector<double> TxEnergy,
                                 vector<double> Gnoise)
{
    vector<double> Resultant;

    for(int j =0; j<TxEnergy.size(); j++){
        Resultant.insert(Resultant.end(), TxEnergy[j]+Gnoise[j]);
    }

    return Resultant;
}


// Function for sum of all raws in a matrix
// Input is a Matrix
// Output is a vector result as sum of all raws
vector<double> RawWiseSum(vector<vector<double>> ReceiveMat)
{
    int numberOfRaws = ReceiveMat.size();

    vector<double> SumofRaws= ReceiveMat[0];

    for(int t=0; t<numberOfRaws-1; t++){
        SumofRaws = VectorSum(SumofRaws, ReceiveMat[t+1]);
    }

    return SumofRaws;
}
```

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

    return countError;
}


// Function to store the data in the file (.dat)
// Input is the SNR per bit in dB and calculated probability of error
// Output is the nothing but in processing it is creating a file and writing data into it.
void datafile(vector<double> xindB, vector<double> Prob_error, 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 pe;
    double a =0;
    double b = pi/2;
    double num =200;
    double width = (b-a)/num;

    double currentValue =a;
    double fun, interg, sinF;
    vector<double> y_axis;
    for(int i =0; i<num; i++){
```

```
        sinF = pow(sin(currentValue), 2);
        fun = 1/(1+(SNR/(2*sinF)));

        y_axis.insert(y_axis.end(), pow(fun, L));

        currentValue = currentValue + width;
    }
    interg = DefiniteIntegration(y_axis, b, a, num);
    pe = interg/pi;

    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;
}


int main(){

    // source defination
    vector<double> sourceBits;

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

    // Noise definition
    vector<double> gnoise;

    //Rayleigh noise (Real guass and Img Guass)
    vector<double> realGaussian;
    vector<double> imziGaussian;
    vector<double> RayleighNoise;

    //Combiner output
    vector<double> AfterCombining;

    //Output of ML detector
    vector<double> decodedBits;

    double L =8;
    double sigmaSquare = 0.5;
    double stddevRayleigh = sqrt(sigmaSquare);
    double N_o =4;
    double p, stdnoise;
```

```
    stdnoise = sqrt(N_o);
    double counterror, P_error;


    vector<vector<double>> RaleighMat;
    vector<vector<double>> GnoiseMat;
    vector<vector<double>> faddingOperation;
    vector<vector<double>> ReceiveMat;
    vector<vector<double>> AfterCophasing;


    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();    //Source bit streame

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

        //Rayleigh distributed fadding coffcients
        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();
        }

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

```

```
        //Fadding operation
        for(int k =0; k<L; k++){
            faddingOperation.insert(faddingOperation.end(),
                                    RayleighOperation(transmittedSymbol, RaleighMat[k]));
        }

        //Additive gaussian noise
        for(int k =0; k<L; k++){
            ReceiveMat.insert(ReceiveMat.end(),
                              GaussianNoiseAdd(faddingOperation[k], GnoiseMat[k]));
        }

        // After co-phaseing multiply with H_i
        for(int k =0; k<L; k++){
            AfterCophasing.insert(AfterCophasing.end(),
                                  RayleighOperation(ReceiveMat[k], RaleighMat[k]));
        }

        //combining operation
        AfterCombining = RawWiseSum(AfterCophasing);

        //ML decoder
        decodedBits = decisionBlock(AfterCombining);

        //Count error
        counterror = errorCalculation(sourceBits, decodedBits);

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

        cout<<"Error value : "<<counterror<<endl;
        cout<<"Probability of error: "<<P_error<<endl;
        cout<<endl;

        RaleighMat.clear();
        GnoiseMat.clear();
        faddingOperation.clear();
        ReceiveMat.clear();
        AfterCophasing.clear();
    }

    char NameofFile1[30] = "MRCL8";         //Name of file

    char NameofFile2[30] = "MRCL8Error";    //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)
{
    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;
}

// function for sum of two vectors
// Inputs are two vectors
// Output is the sum of two vectors.
vector<double> VectorSum(vector<double> Vec1, vector<double> Vec2)
{
    vector<double> SumResult;

    if(Vec1.size()==Vec2.size()){
        for(int k=0; k<Vec1.size(); k++){
            SumResult.insert(SumResult.end(), Vec1[k]+Vec2[k]);
        }

        return SumResult;
    }else{
        cout<<"Error !!!! Vector size should be same!!!"<<endl;
        return SumResult;
    }
}
```

Code for definite integration header file

```
//////////////////////////////////////////////////////////////
//////////////////////////////////////
// Author:- MANAS KUMAR MISHRA
// Organisation:- IIITDM KANCHEEPURAM
// Topic:- Definite integration [a, b]
//////////////////////////////////////////////////////////////
//////////////////////////////////////

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

double DefiniteIntegration(vector<double> y_axis,
                           double upperLimit,
                           double lowerLimit,
                           int numOfDiv)
{

    double width = (upperLimit-lowerLimit)/numOfDiv;


    double area, integration;
    integration=0;

    for(int k =0; k<numOfDiv; k++){
        area = y_axis[k]*width;

        integration = integration +area;
    }
    return integration;

}
```

5.2 Results

5.2.1 MRC for different antenna elements

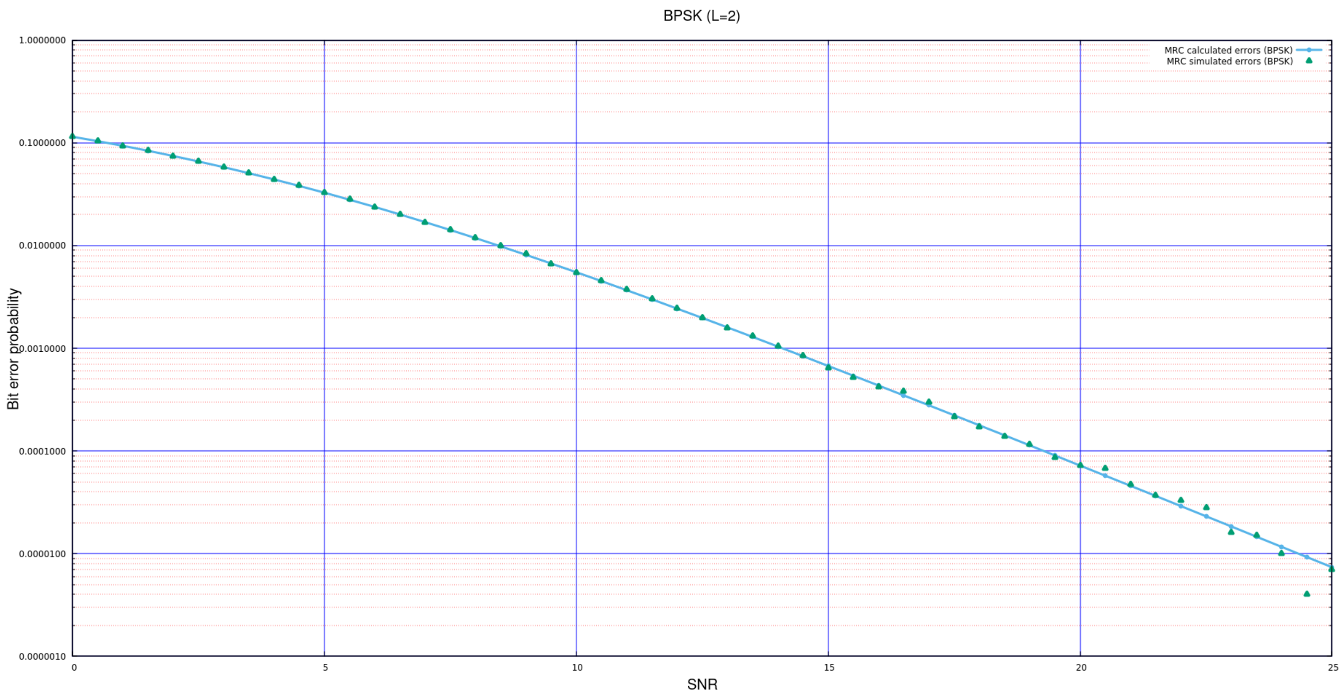

Figure 4: L=2 case simulated and calculated

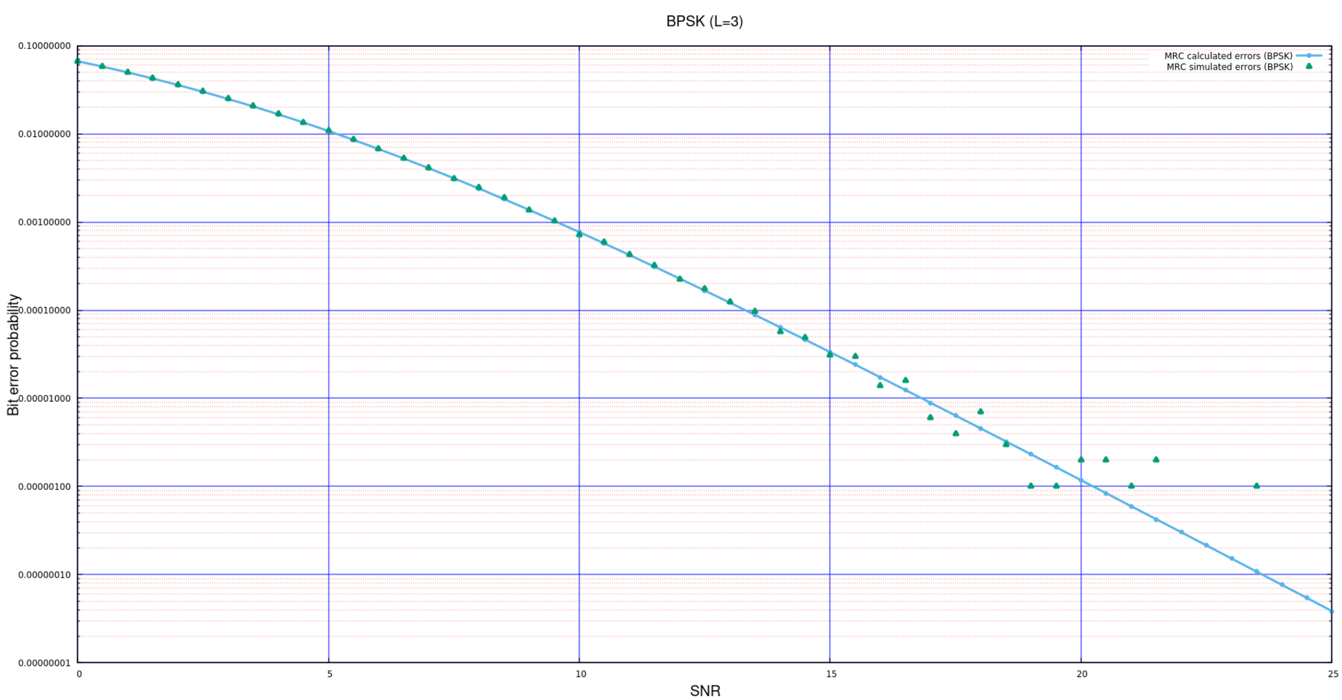

Figure 5: L=3 case simulated and calculated

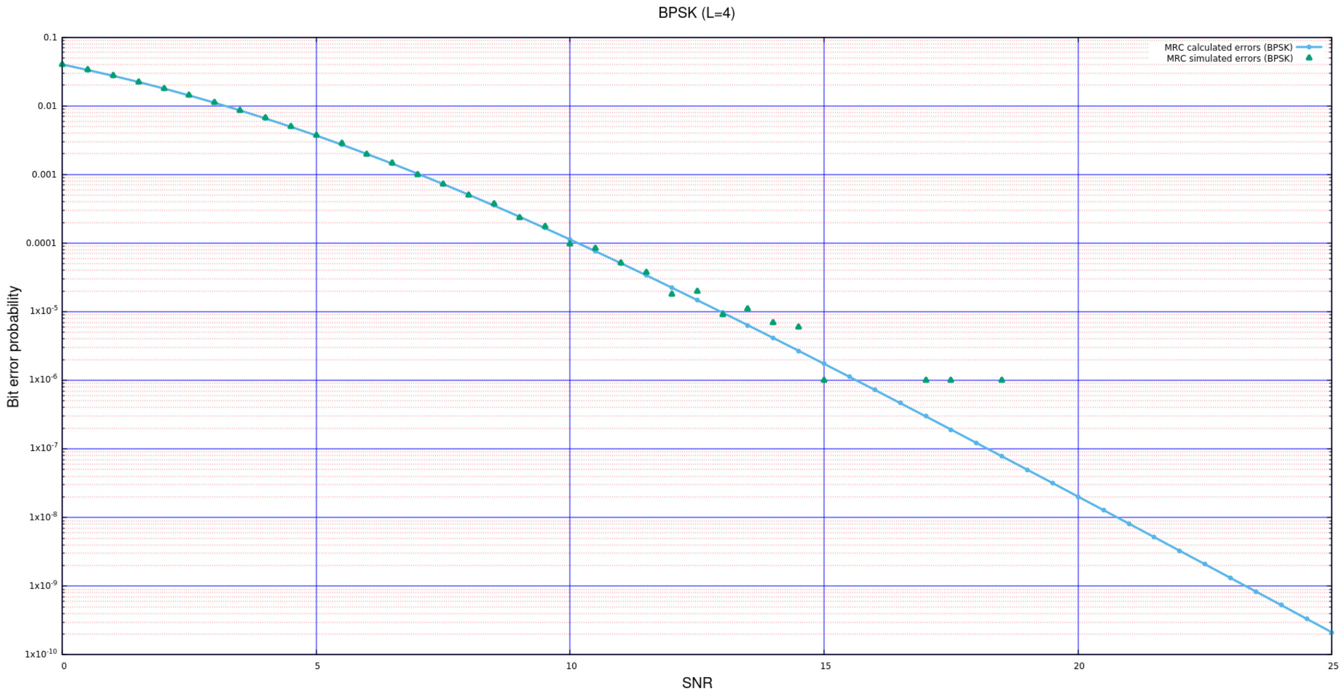

Figure 6: L=4 case simulated and calculated

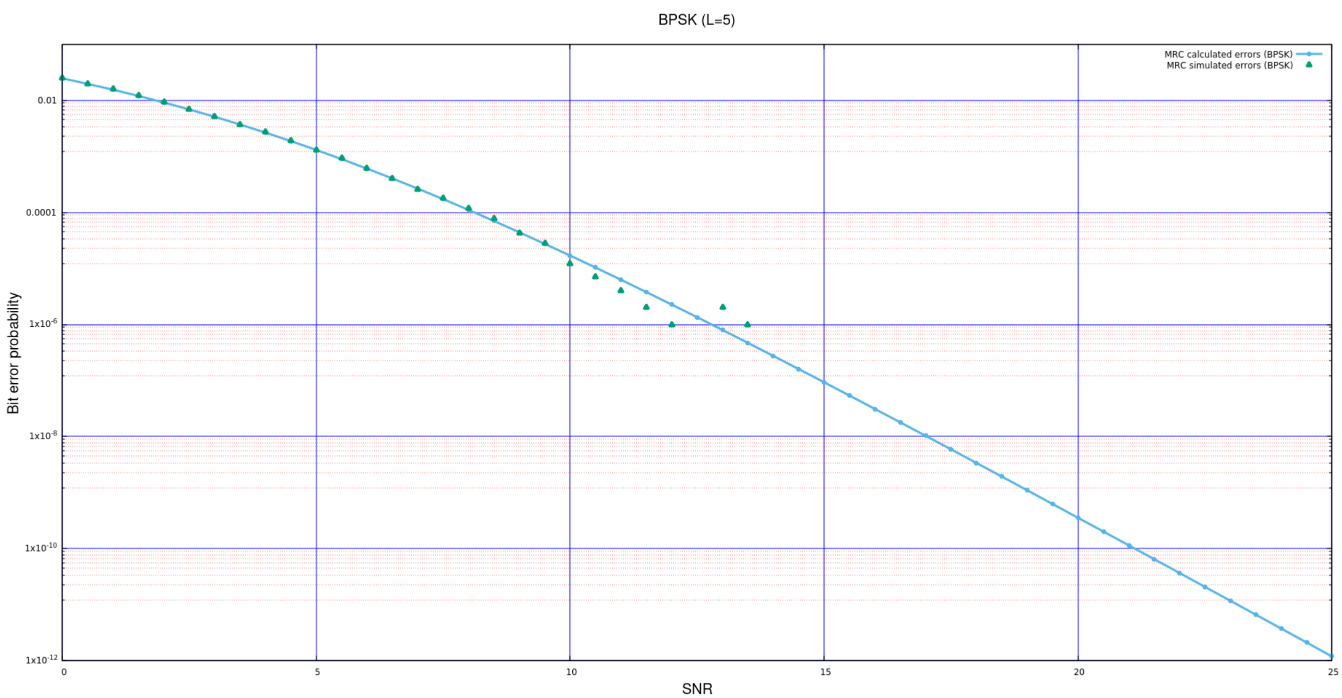

Figure 7: L=5 case simulated result

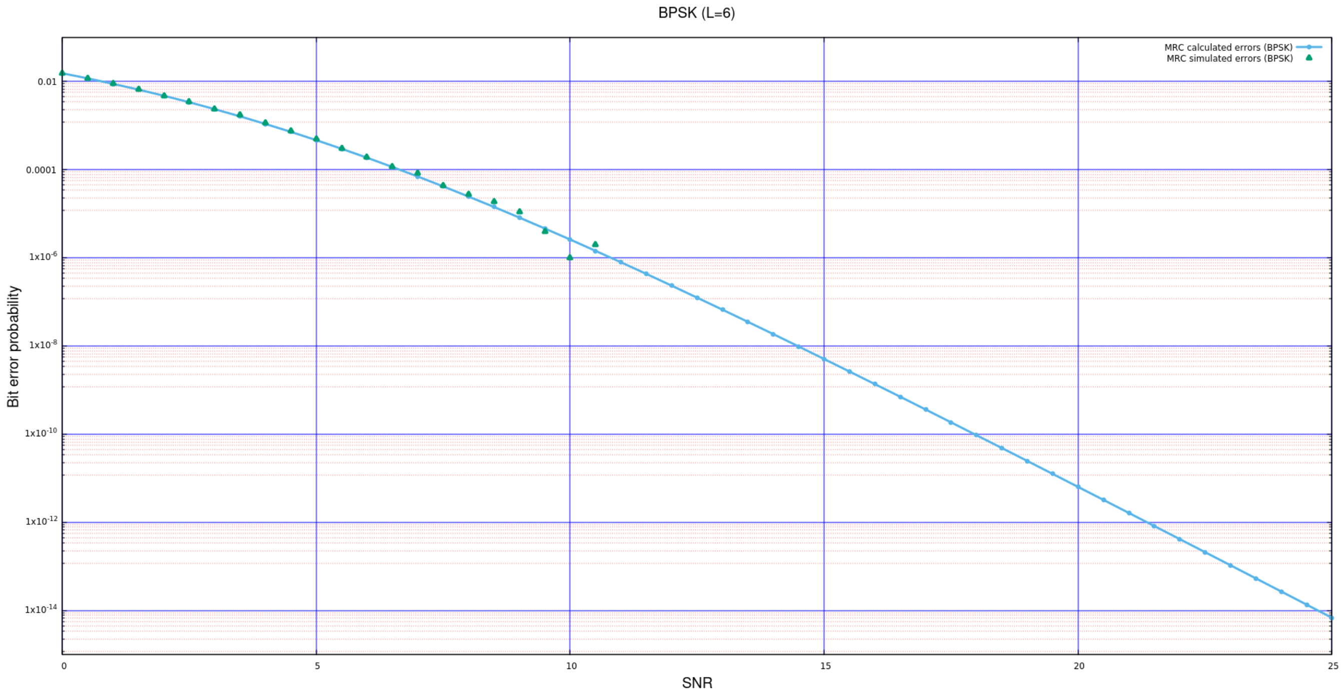

Figure 8: L=6 case simulated result

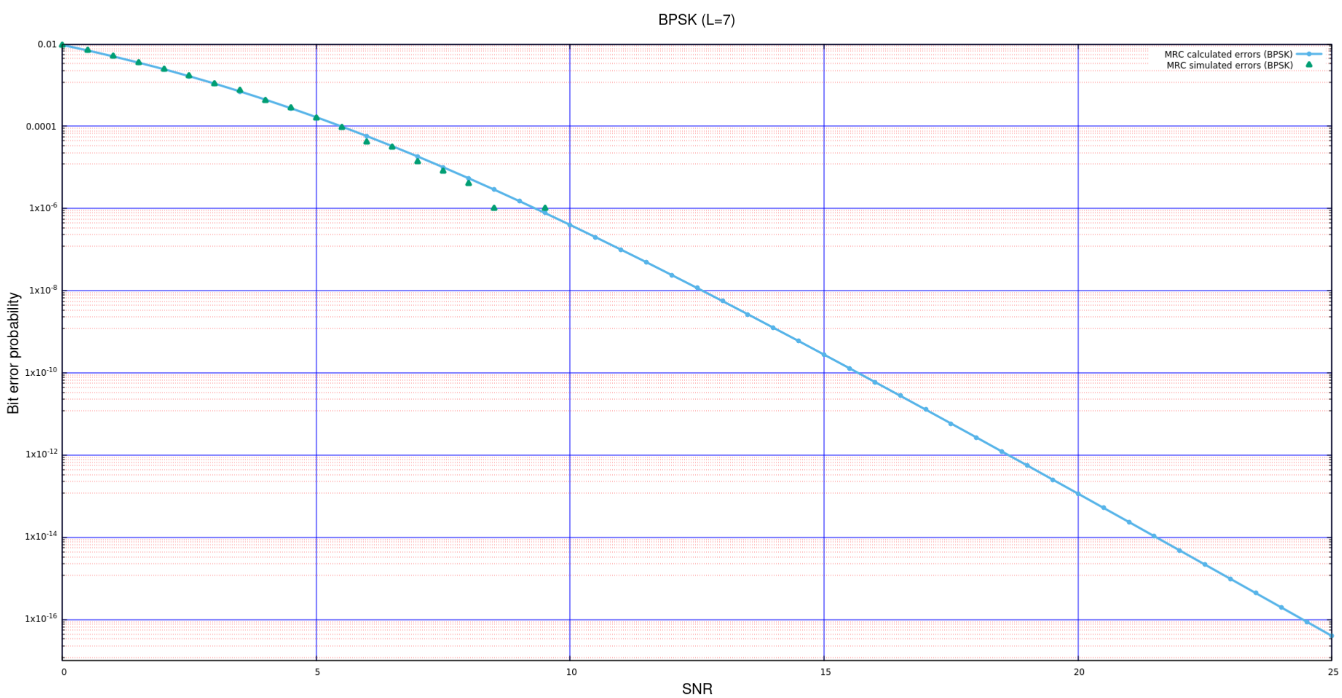

Figure 9: L=7 case simulated result

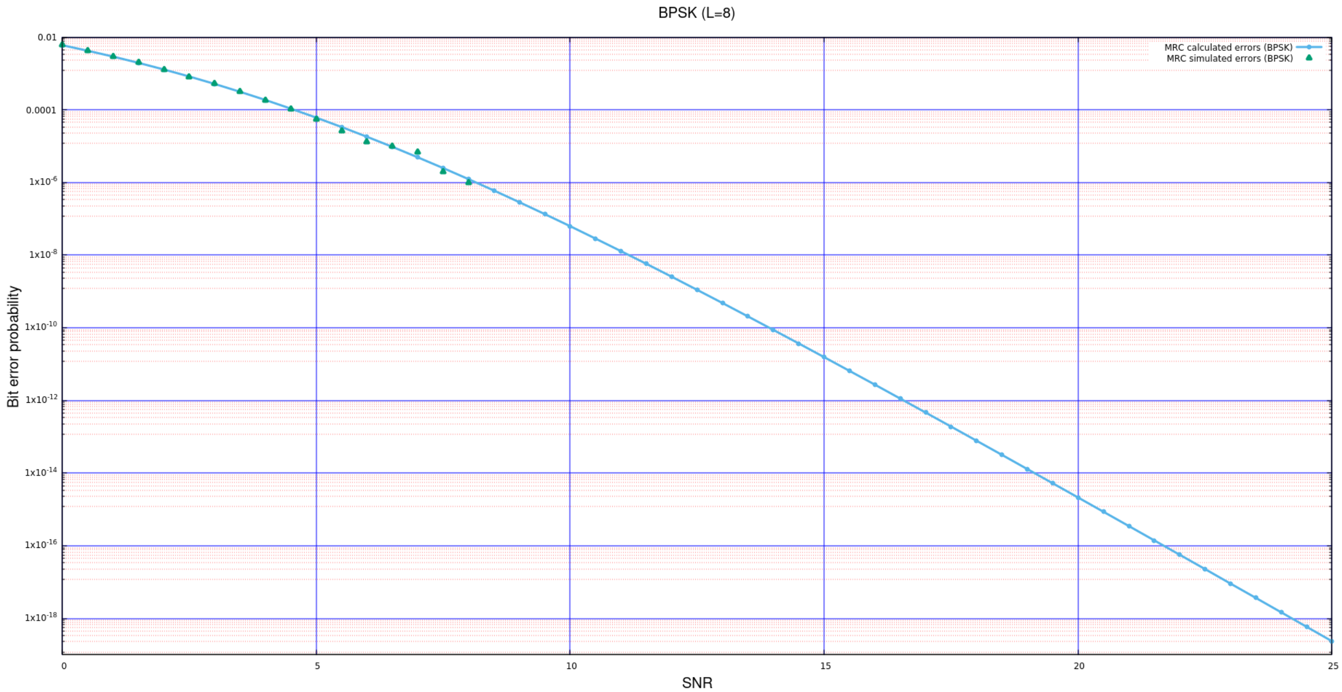

Figure 10: L=8 case simulated result

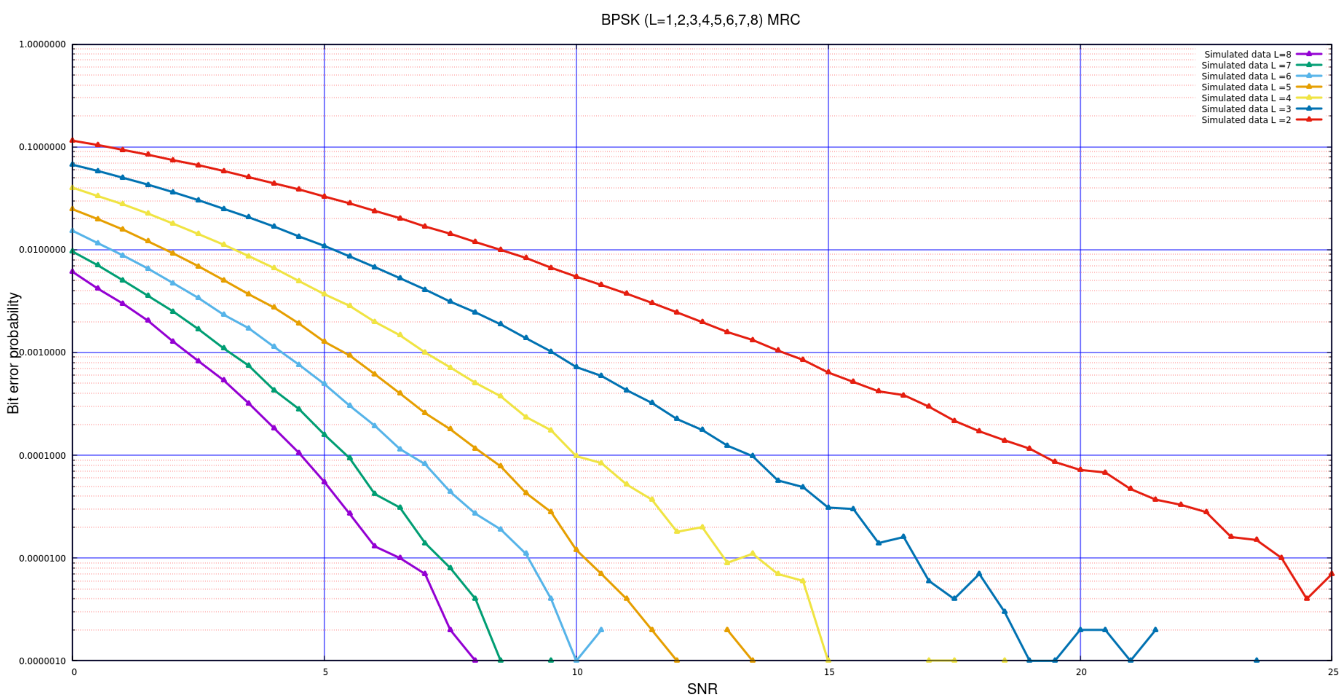

Figure 11: All simulated data

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

5.2.2 SG, EGC, and MRC

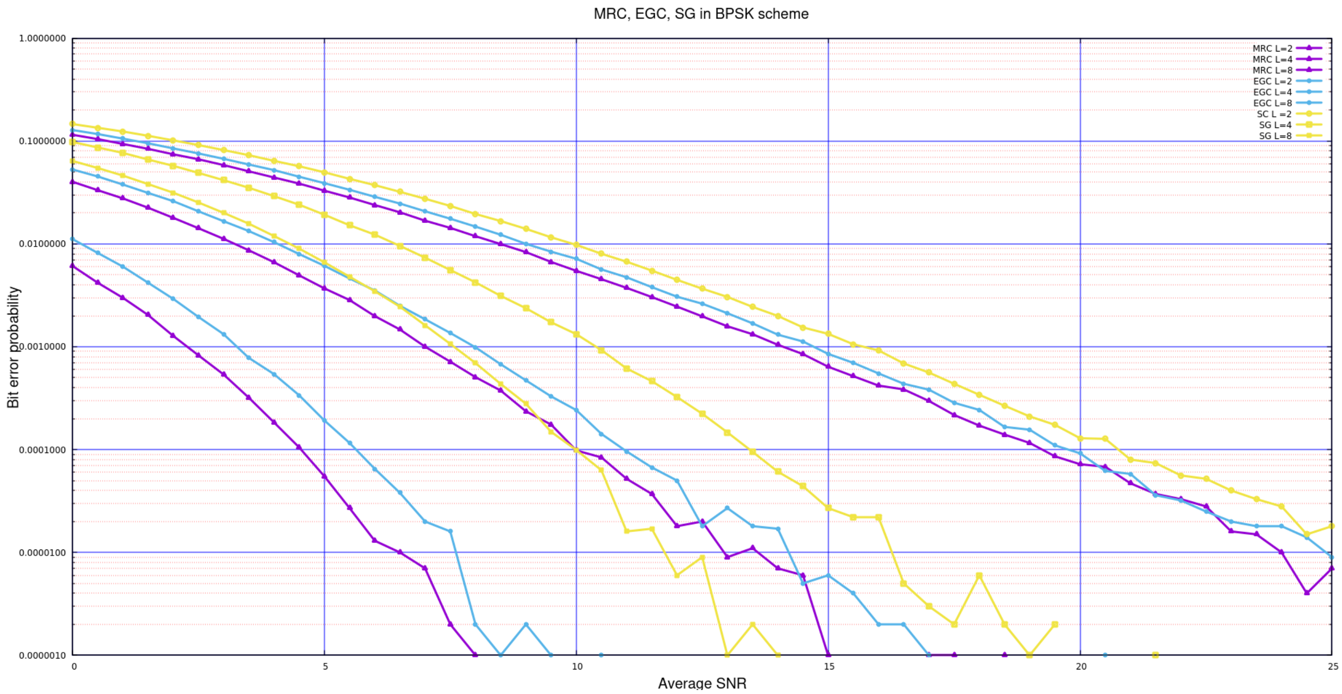

Figure 12: EGC, SG and MRC

5.2.3 Different modulation techniques

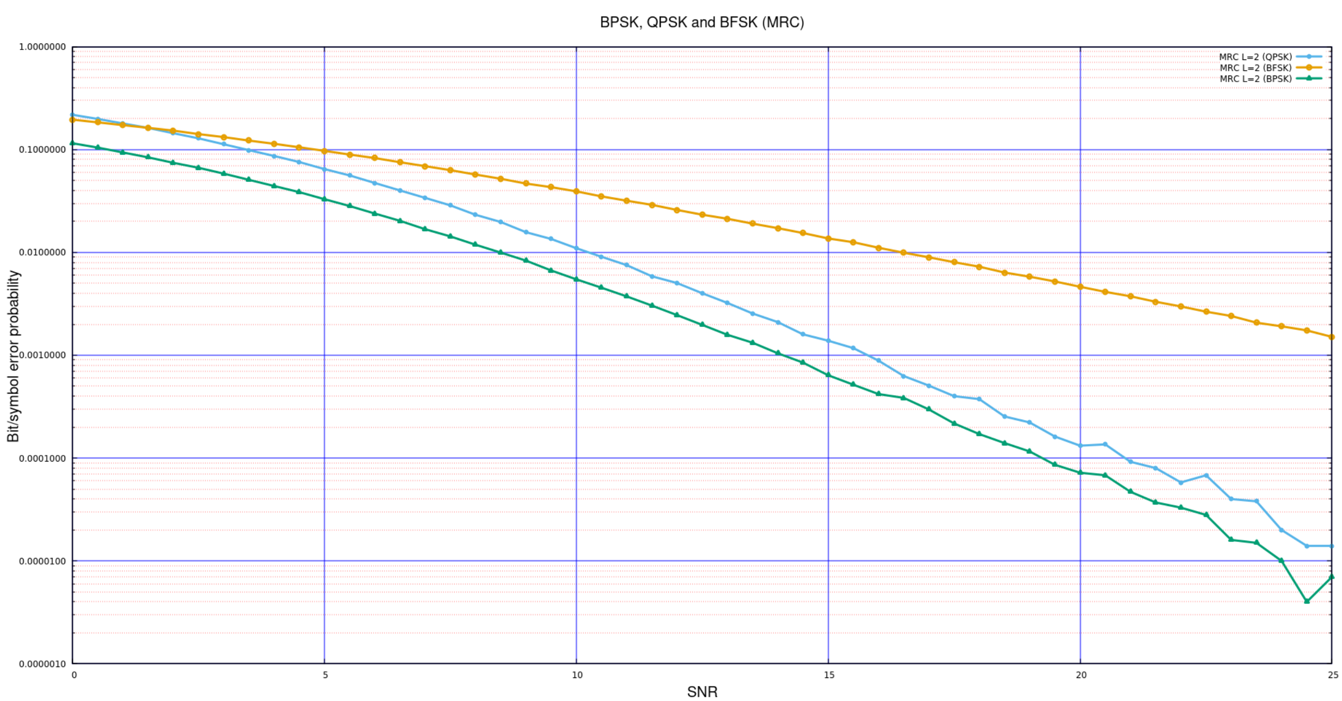

Figure 13: BPSK, QPSK, and BFSK (L=2)

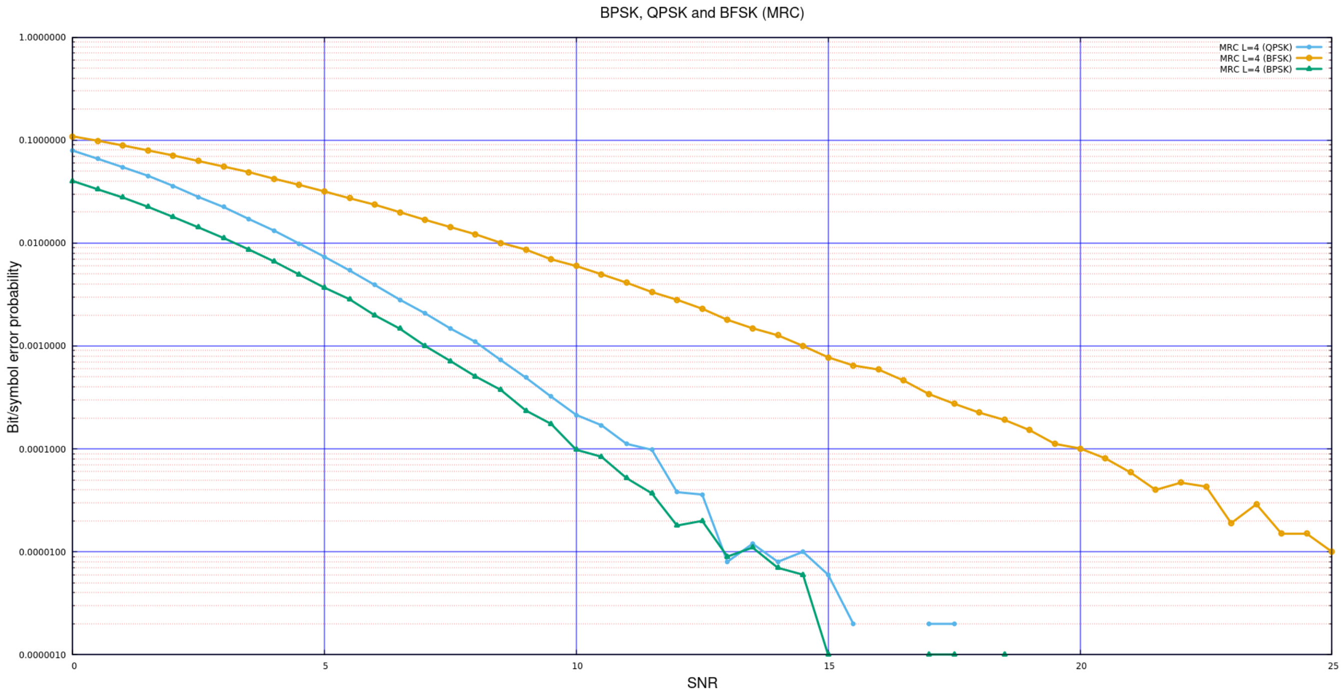

Figure 14: BPSK, QPSK, and BFSK (L=4)

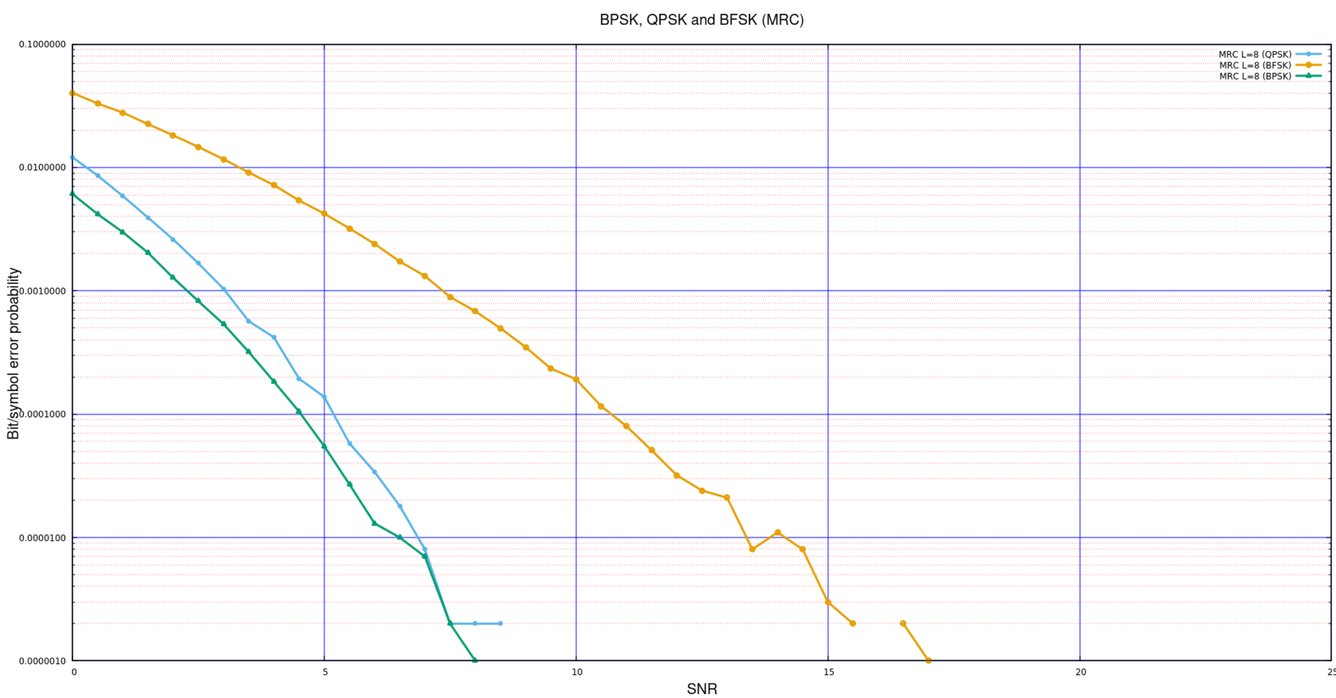

Figure 15: BPSK, QPSK, and BFSK (L=8)

6 Inferences

Inferences list:
Based on the final graph
Similar to Selective gain diversity and equal gain combining, figure 11, 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 12 selective gain, equal gain combining and maximum ratio combining are almost parallel to each other. That implies, array gains are nearly same for all three cases. That completely makes sense to me, because array gain reflects the gain due to multiple antenna. For L =2, 4, 8 number of antenna should reflect same array gains irrespective to the intelligent algorithm.
But as we increase value of L, gap between EGC or MRC values are higher than SG at same P_e, that implies diversity gain of EGC is higher than SG. Hence overall gain, in EGC and MRC should be larger than SG. When we compare MRC and EGC, one can see very close diversity gain, where MRC is winner. Hence, MRC has overall best gain as compare to the other two techniques.
Based on the Probability of error
During the derivation of probability of error in MRC, one can notices that distribution of SNR is the sum of SNR of all antenna branches. But in EGC and SC cases distribution does not consider SNR of all branches. That implies, MRC consider all antenna contributions as compare to other two techniques, that's why, probability of error is less relative to other two techniques.
Based on the Different modulations schemes
From figure 13 to figure 15, we can see that even in fadding channel BPSK is best performer among all modulation scheme. After that QPSK and BFSK. One can also notices that by diversity, we can achieve better performance than AWGN in any modulation scheme. That is something we desperately want to achieve but it has a significant cost to pay.

7 Result/Conclusion

7.1 What did I learn?

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