ADCC

Experiment-04 (MPSK)

Manas Kumar Mishra (ESD18I011)

11-NOVEMBER-2021

Lab Incharge: Dr. Premkumar Karumbu

1 Aim

To design discrete time MPSK scheme and analyse the performance.

2 Theory

MPSK is the abbreviation for M-ary Phase Shift Keying. M-ary implies M possible values at a time. Phase shift implies change in phase for carrier signal to represent the information. Keying implies the way to represent the information in another form.
Usually for practical implementations, M would be some exponent of 2 such that $log_2 M$ bits can be consider at a time i.e. base of scheme. In this particular scheme, system has M different symbols to modulate. For this task to achieve, it chooses M different phase values for the carrier signal. Chosen phase value would be constant for a particular time interval i.e. width of information symbol. After that system changes phase of carrier signal based on the current symbol. In this way, M-ary system continuously modulate the symbols in carrier signal. That modulated carrier signal propagates through channel/medium and captured by receiver or receivers. Based on phase of the received signal, a receiver is supposed to decode the information symbol. Hence, difference between any two phase angle should be maximum as much as possible.
Generally, phase starts with 0 and increase by $\frac{2\pi}{M}$ and M > 2. Consider a constellation diagram as given below.

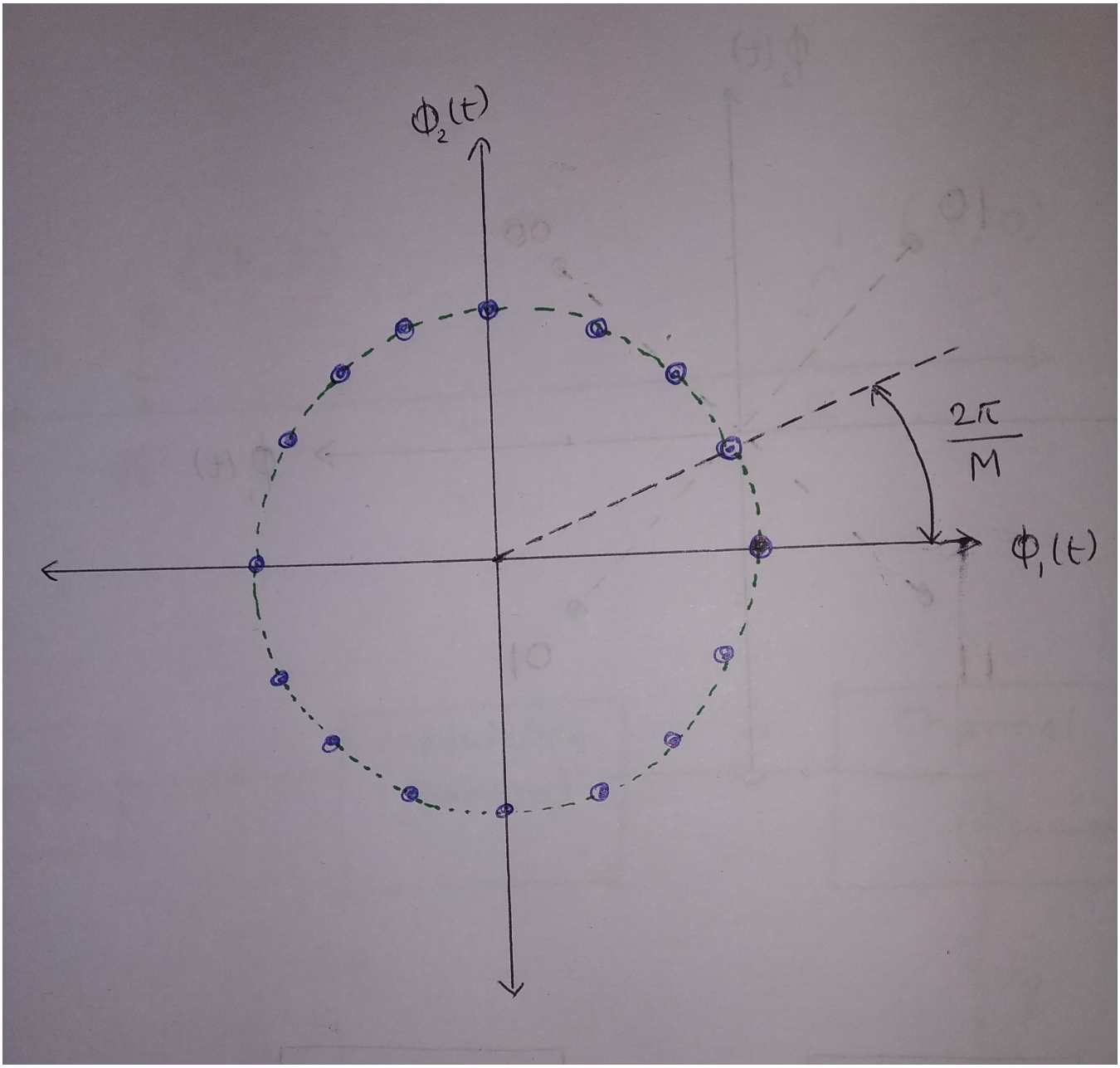

Figure 1: MPSK constellation diagram

3 Design

3.1 Block diagram

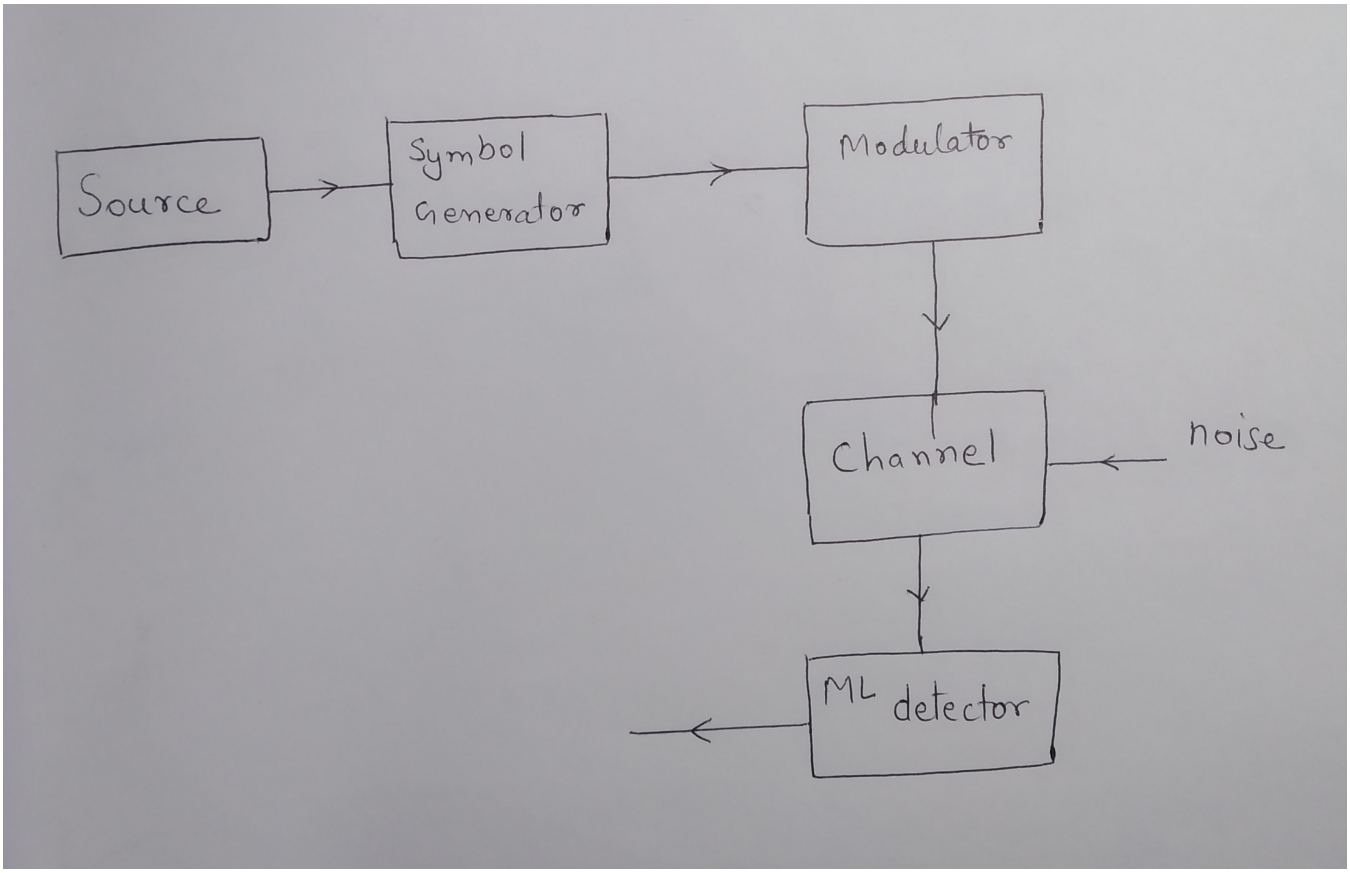

Figure 2: Block diagram

Where source block generates binary bit stream. Symbol generator is the block that takes $log_2 M$ bits at a time and convert into symbols. Modulation block does mapping based on constellation diagram figure 1. Channel block is the representation of AWGN (Additive White Gaussian Noise), in that gaussian noise adds up to the signal values.

3.2 ML detector

For the ML detector, angular bisector region would be the best choice for decision. For visualization of angular bisector a constellation diagram is given below.

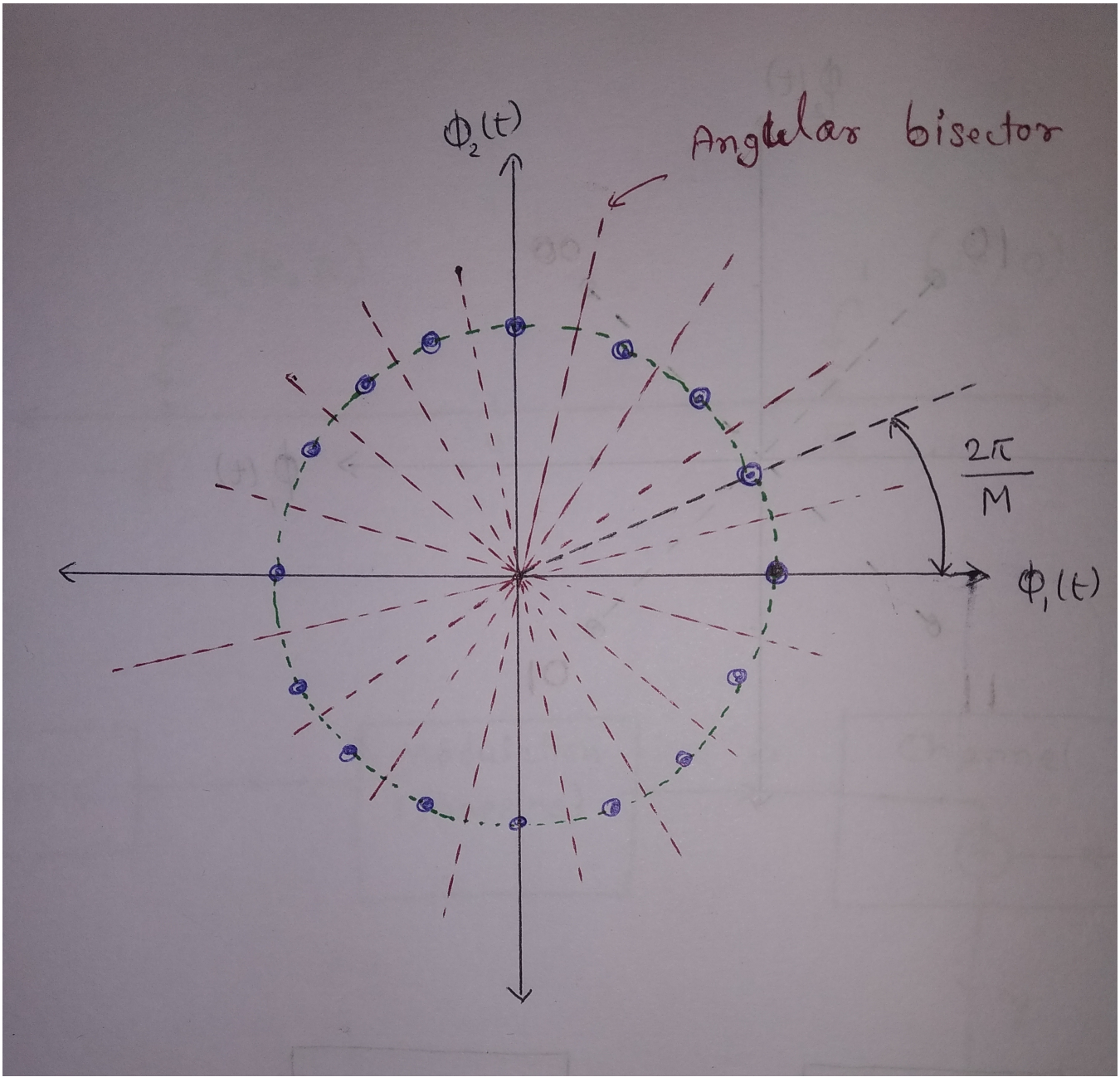

Figure 3: Angular bisector

3.3 Probability of error

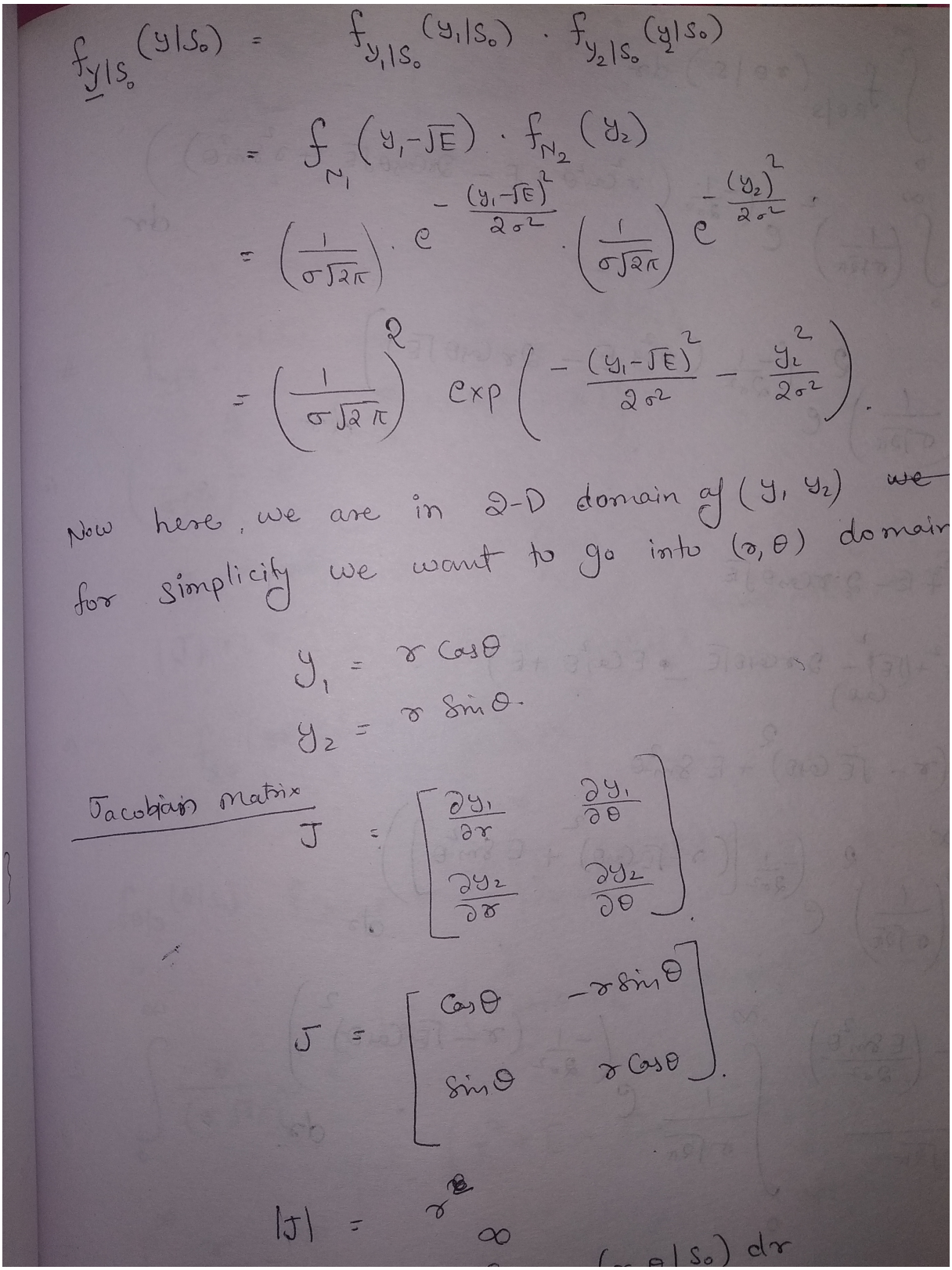

Figure 4: Probability of error calculation

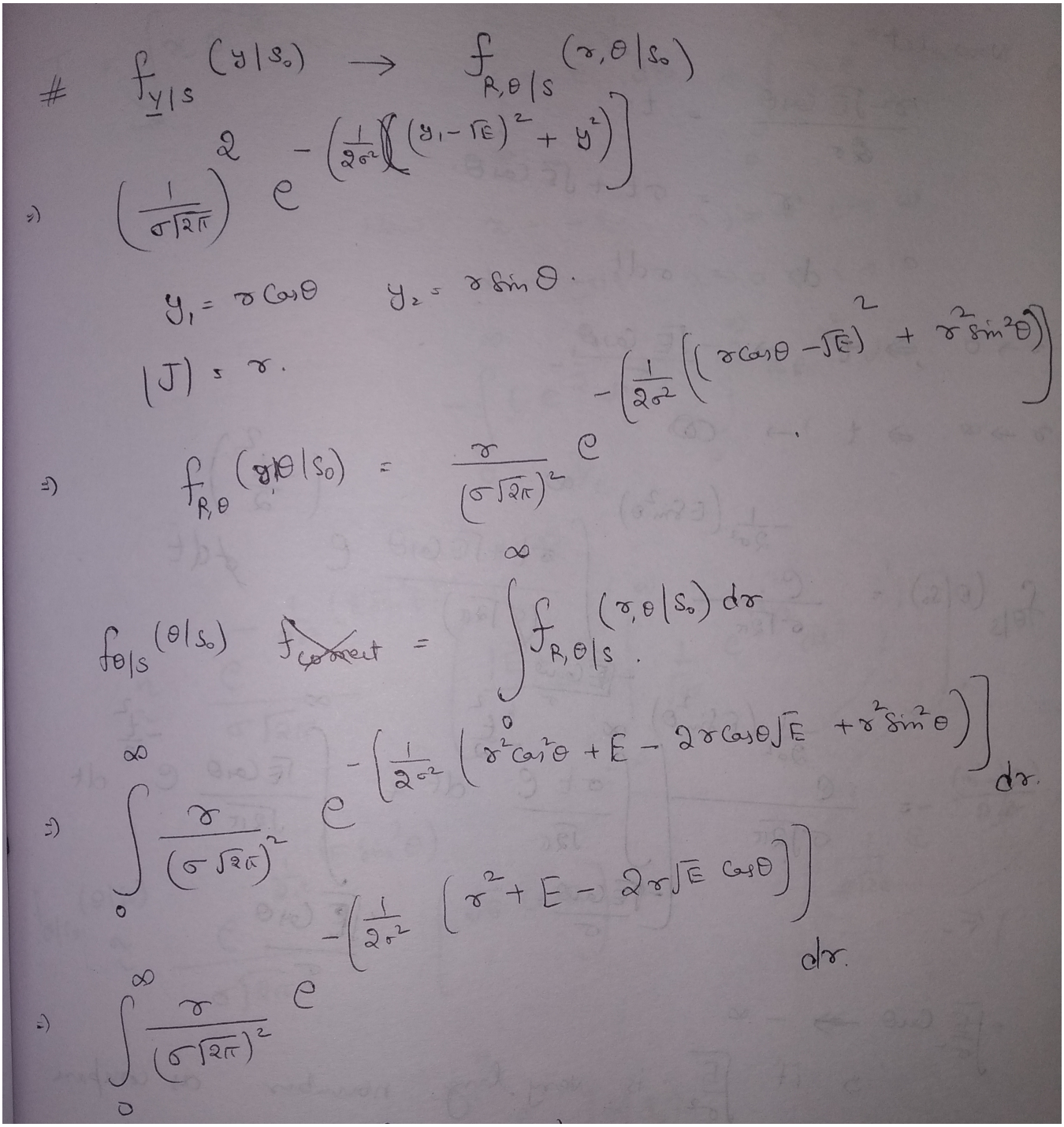

Figure 5: Probability of error calculation

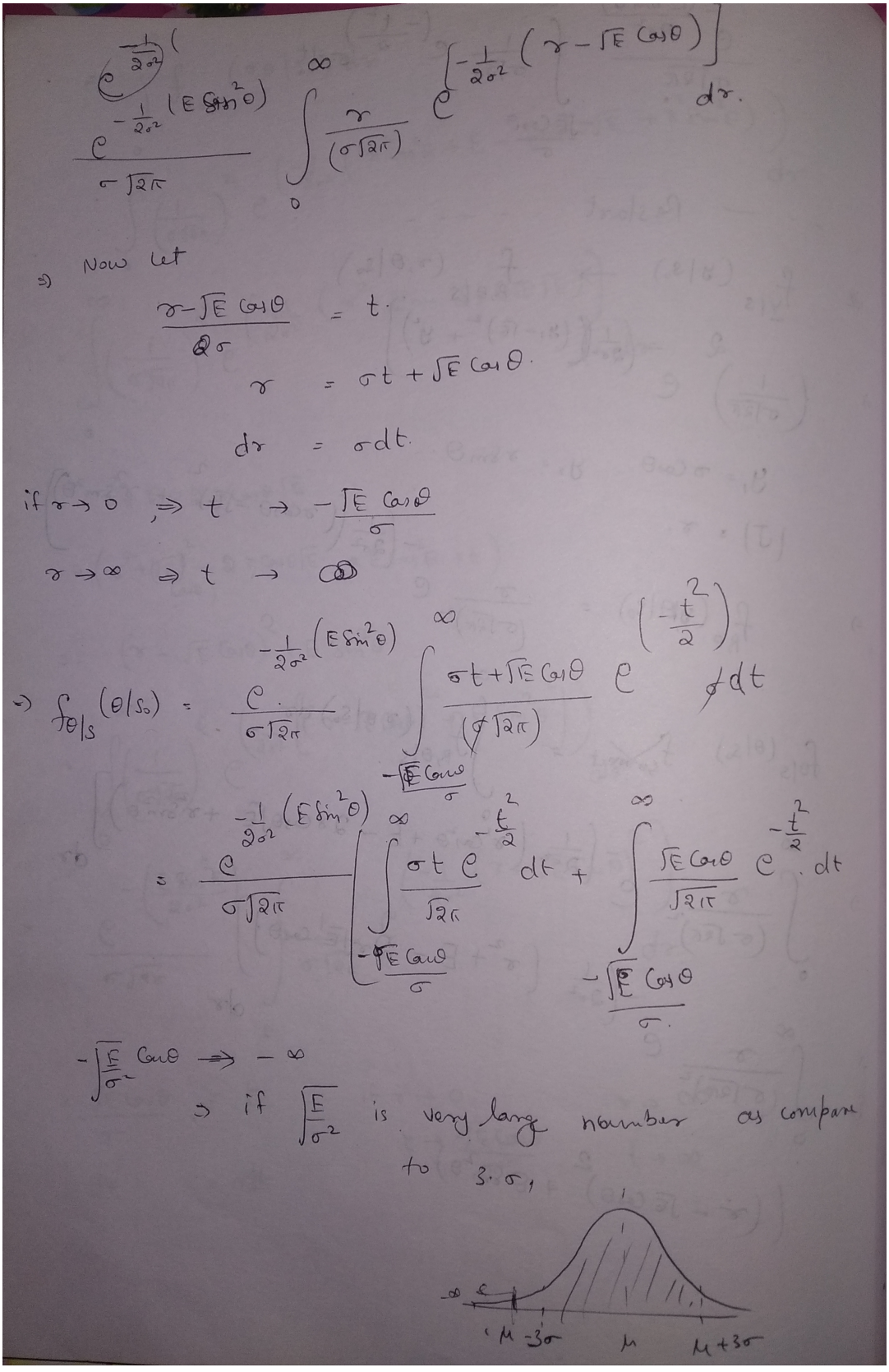

Figure 6: Probability of error calculation

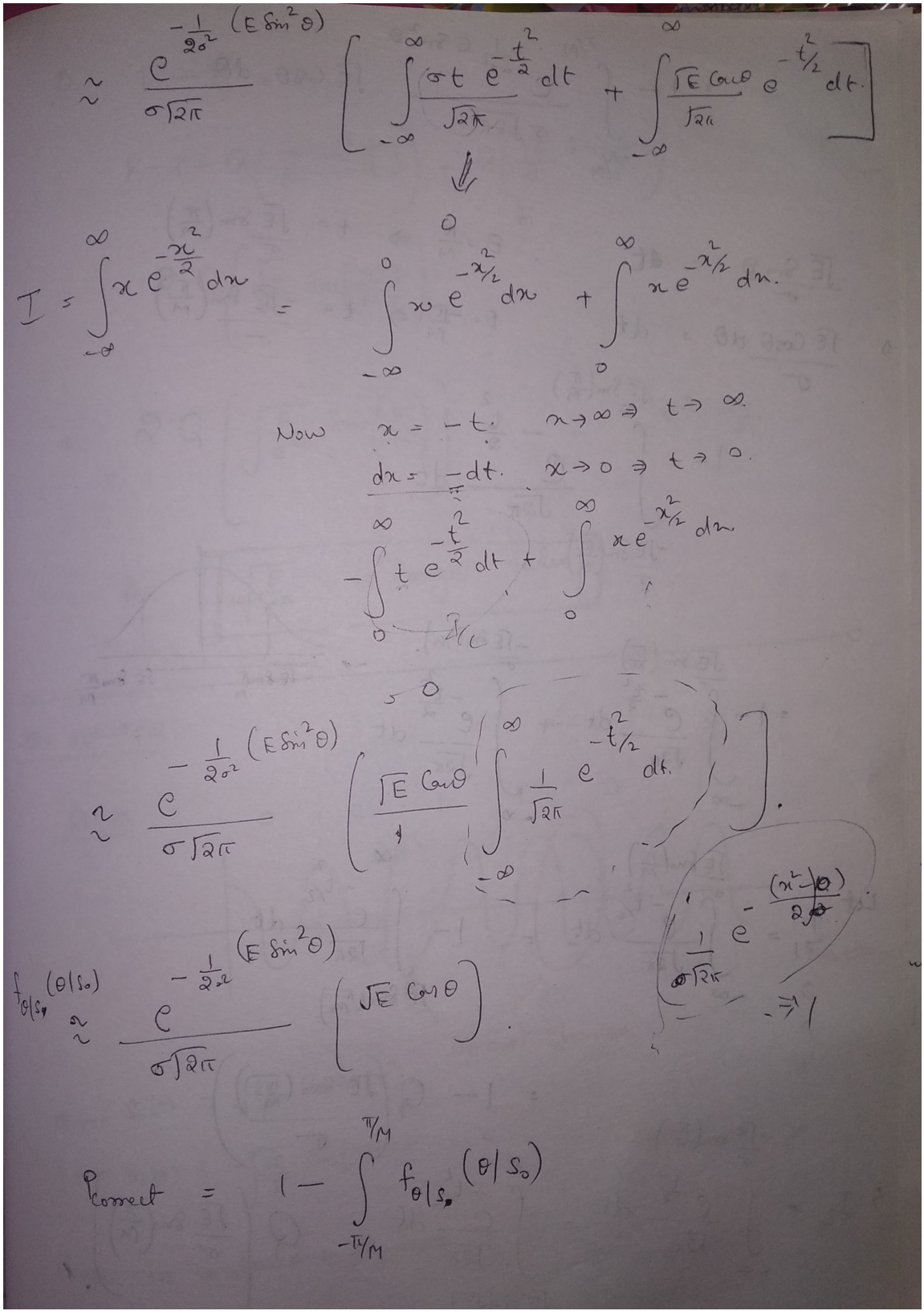

Figure 7: Probability of error calculation

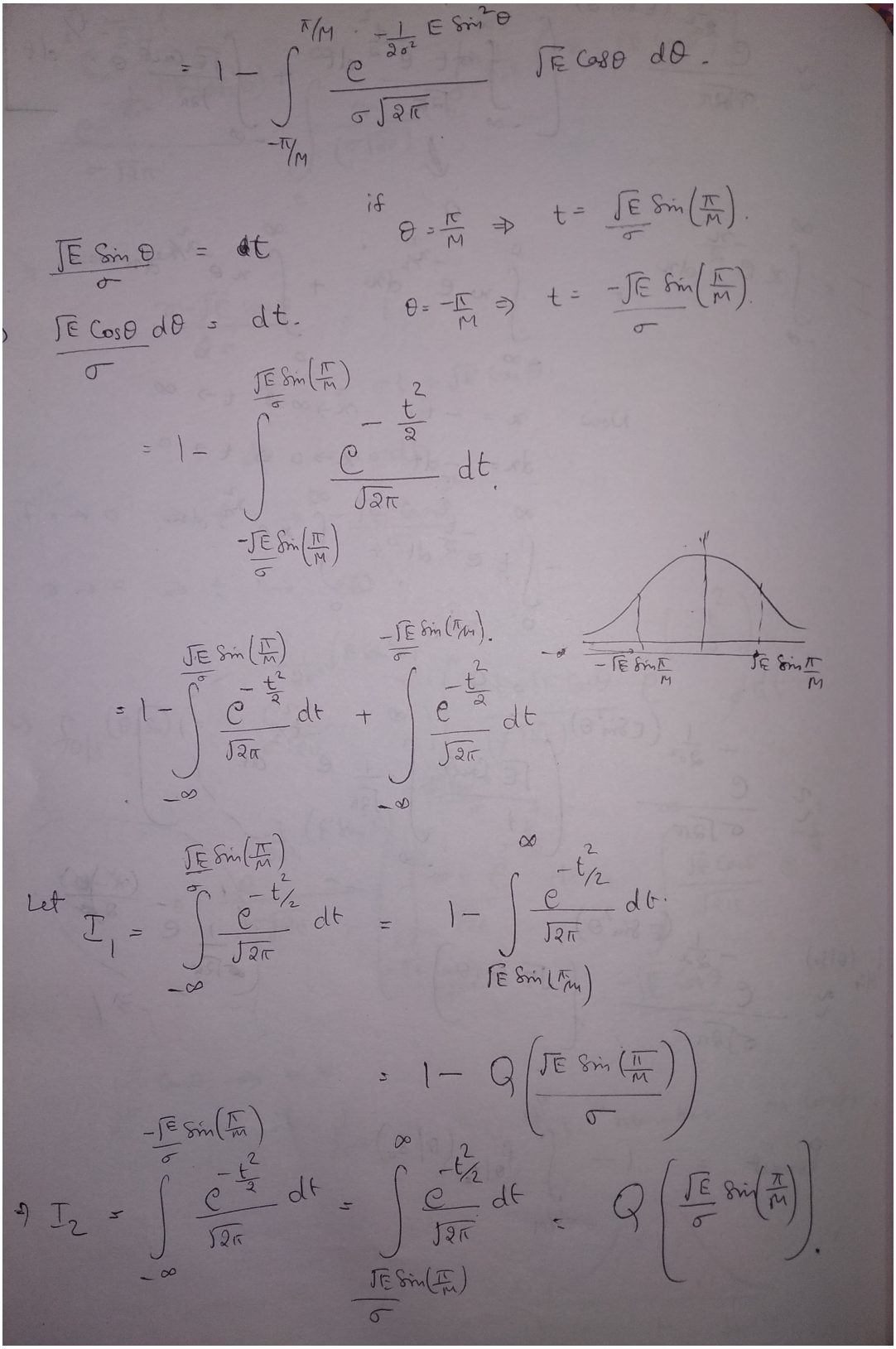

Figure 8: Probability of error calculation

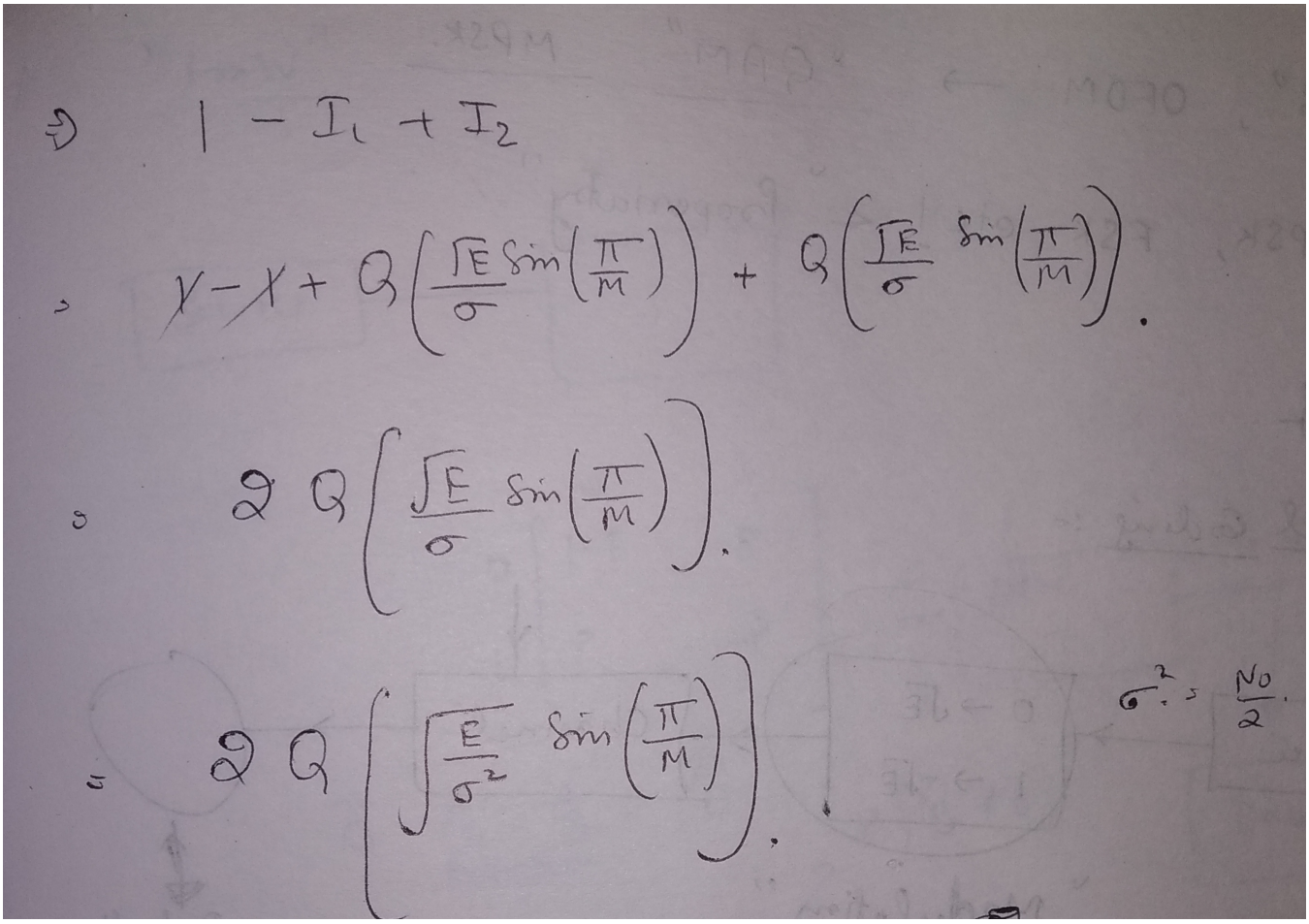

Figure 9: Probability of error calculation

Final result:
For MPSK probability of error

$$P(Error) = 2Q\left(\sqrt{\frac{2E}{N_o}}\sin\left(\frac{\pi}{M}\right)\right)$$

That means, probability of error for MPSK depends on two quantities, one is $\frac{E}{N_o}$ and second is M.

4 Code and result

4.1 Code for MPSK in cpp

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


#include <iostream>
#include <cmath>
#include <iterator>
#include <random>
#include <chrono>
#include <time.h>
#include <fstream>
#include <bits/stdc++.h>
#include <vector>

#define one_million 1000000
#define pi 3.14159

using namespace std;


// 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> Source()
{
    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 to generate symbols from the binary bit stream.
// Input is the binary bit vector and base for symbol conversion
// Output is the symbol vector containing values from (0, 2^(base-1))
vector <double> binaryToDecimalConversion(vector<double> sourceBits, const int& base)
{
    vector <double> convertedBits;
```

```
51      int start =0;
52
53      if((sourceBits.size()% base) == 0 ){
54
55          int finalSize = sourceBits.size()/base;
56          start = 0;
57          int conversion;
58
59
60          for(int i=0; i<finalSize; i++){
61              conversion =0;
62              for(int j =base-1; j>-1; j--){
63                  conversion = conversion + (sourceBits[start])*(pow(2, j));
64                  start++;
65              }
66              convertedBits.insert(convertedBits.end(), conversion);
67          }
68
69      }
70      else{
71          int addedBitsNO =  base - (sourceBits.size()%base);
72
73          for(int q=0;q<addedBitsNO;q++){
74              sourceBits.insert(sourceBits.end(), 0);
75          }
76
77          int finalSize = sourceBits.size()/base;
78          start = 0;
79          int conversion;
80
81
82          for(int i=0; i<finalSize; i++){
83              conversion =0;
84              for(int j =0; j<base; j++){
85                  conversion = conversion + (sourceBits[start])*(pow(2, j));
86                  start++;
87              }
88              convertedBits.insert(convertedBits.end(), conversion);
89          }
90
91      }
92
93      return convertedBits;
94  }
95
96
97
98
99  // Functions SignalVector1 and SignalVector2 for convert sourceSymbols to 2D-vector
100 // Input are the source Symbols and energy of symbols for mapping
101 // Output is a vector represent signal component
102 vector<double> SignalVectors1(vector<double> sourceSymbols, double eneryOfSymbols,
103 const double& M)
104 {
105     vector<double> y1;
106     double theta;
```

```
    double angDiv = (2*pi)/M;

    for(int i=0; i<sourceSymbols.size(); i++){
        theta = sourceSymbols[i]*angDiv;

        y1.insert(y1.end(), sqrt(eneryOfSymbols)*cos(theta));
    }

    return y1;
}


vector<double> SignalVectors2(vector<double> sourceSymbols, double eneryOfSymbols,
const double& M)
{
    vector<double> y2;
    double theta;
    double angDiv = (2*pi)/M;

    for(int i=0; i<sourceSymbols.size(); i++){
        theta = sourceSymbols[i]*angDiv;

        y2.insert(y2.end(), sqrt(eneryOfSymbols)*sin(theta));
    }


    return y2;
}



// Function for generating random noise based on gaussian distribution N(mean, variance).
// Input mean and standard deviation.
// Output is the vector that contain gaussian noise as an element.
vector<double> GnoiseVector(double mean, double stddev)
{
    vector<double> noise;

    // construct a trivial random generator engine from a time-based seed:
    unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
    std::default_random_engine generator (seed);

    std::normal_distribution<double> dist(mean, stddev);

    // Add Gaussian noise
    for (int i =0; i<=one_million; i++) {
        noise.insert(noise.end(),dist(generator));
    }

    return noise;
}



// Function for model the channel
// Input sourcesymbols and AWGN
```

```
163 // Output is a vector that represent the addition of two vectors
164 vector<double> channelModel(vector<double> sourceSymbols, vector<double> AWGnoise)
165 {
166     vector<double> received;
167 
168     for(int i=0; i<sourceSymbols.size(); i++){
169         received.insert(received.end(), sourceSymbols[i]+AWGnoise[i]);
170     }
171 
172     return received;
173 }
174 
175 
176 double arcTan(double yvalue, double xvalue)
177 {
178     double theta;
179 
180     if(yvalue==0 && xvalue ==0){
181         theta =0;
182     }else{
183         theta = atan2(yvalue, xvalue)*(180/pi);
184     }
185 
186     if(yvalue <0){
187         theta = theta +360;
188     }
189 
190     theta = theta*(pi/180);
191     return theta;
192 }
193 
194 
195 
196 // Function for take decision after receiving the vector
197 // Input is the 2D receiver vector
198 // Output is the decision {0, 1, 2, ..., M-1}
199 vector <double> decisionBlock(vector<double> receive1, vector<double> receive2,
200 double& M)
201 {
202     vector<double> decision;
203 
204     double theta;
205     double upperLimit;
206     double lowerLimit;
207     double increment;
208 
209     increment = 360/M;
210     lowerLimit = 180/M;
211 
212 
213     for( int index =0; index < receive1.size(); index++){
214         theta = arcTan(receive2[index], receive1[index]);
215 
216         theta = theta*(180/pi);
217 
218         increment = 360/M;
```

```
219         lowerLimit = 180/M;
220         upperLimit = 180/M;
221
222
223         double j = 1;
224         while(j<=M){
225             if(j==1 && (theta<=lowerLimit||theta-360>-lowerLimit)){
226                 decision.insert(decision.end(), j-1);
227                 j = j+1;
228                 upperLimit = lowerLimit + increment;
229                 continue;
230             }
231             if(theta<= upperLimit && theta> lowerLimit){
232                 decision.insert(decision.end(), j-1);
233             }
234             j = j+1;
235             lowerLimit = upperLimit;
236             upperLimit = lowerLimit+increment;
237
238
239         }
240
241     }
242     cout <<endl;
243
244     return decision;
245 }
246
247
248
249 // Function for counting errors in the symbols
250 // Input are the source symbols and decided bits
251 // Output is the error count
252 int errorCount(vector <double> sourceSymbols, vector<double> decisionSymbols)
253 {
254     int count =0;
255
256     for(int i=0; i<sourceSymbols.size(); i++){
257         if(sourceSymbols[i] != decisionSymbols[i]){
258             count++;
259         }
260     }
261
262     return count;
263 }
264
265
266
267
268 // Function to store the data in the file (.dat)
269 // Input is the SNR per bit in dB and calculated probability of error
270 // Output is the nothing but in processing it is creating a file and writing data into it.
271 void datafile(vector<double> SNR, vector<double> Prob_error)
272 {
273     ofstream outfile;
274
```

```
275     outfile.open("MPSK(m=8)Data.dat");
276
277     if(!outfile.is_open()){
278         cout<<"File opening error !!!"<<endl;
279         return;
280     }
281
282     for(int i =0; i<SNR.size(); i++){
283         outfile<< SNR[i] << " "<<"\t"<<" "<< Prob_error[i]<< endl;
284     }
285
286     outfile.close();
287 }
288
289
290
291
292 // Function to compute Q function value.
293 // Input is the x.
294 // Output is Q function output.
295 double Qfunc(double x)
296 {
297     double Qvalue = erfc(x/sqrt(2))/2;
298     return Qvalue;
299 }
300
301
302
303 vector<double> Qfunction(vector <double> SNR_dB, const double& M)
304 {
305     vector <double> Qvalue;
306     double po, normalValue, extra;
307     for (int k =0; k<SNR_dB.size(); k++){
308
309         normalValue = pow(10, (SNR_dB[k]/10));
310         extra = sin(pi/M);
311         po = (2*Qfunc(sqrt(2*normalValue)*extra)); // Defination of Pe from calculations.
312         Qvalue.insert(Qvalue.end(), po);
313     }
314
315     return Qvalue;
316 }
317
318
319 void qvalueInFile(vector <double> SNR, vector <double> Qvalue)
320 {
321     ofstream outfile;
322
323     outfile.open("MPSK(m=8)_Qvalue.dat");
324
325     if(!outfile.is_open()){
326         cout<<"File opening error !!!"<<endl;
327         return;
328     }
329
330     for(int i =0; i<SNR.size(); i++){
```

```
331         outfile<< SNR[i] << " "<<"\t"<<" "<< Qvalue[i]<< endl;
332     }
333 
334     outfile.close();
335 }
336 
337 
338 
339 int main()
340 {
341     // source defination
342     vector<double> sourceBits;
343 
344     // Symbols formation
345     vector<double> Symbols;
346 
347     // Base for conversion
348     int base =3; //3, 4, 5, 6 ...
349     double M = pow(2, base);
350 
351     // Transmiited vectors
352     vector<double> trans1;
353     vector<double> trans2;
354 
355     int SizeOfTransmission = trans1.size();
356 
357     // Noise vector
358     vector<double> gnoise1;
359     vector<double> gnoise2;
360 
361     // Receive bits
362     vector<double> receivedBits1;
363     vector<double> receivedBits2;
364 
365 
366     // Decision block
367     vector<double> decodedBits;
368 
369     vector<double> EnergyVector;
370 
371     vector<double> p_error;
372 
373     double errors=0;
374     double pe;
375     double N_o = 8;
376     double stddev = sqrt(N_o/2);
377 
378 
379    // SNR in dB
380     vector<double> SNR_dB;
381     for(float i =0; i<=20; i=i+0.125)
382     {
383         SNR_dB.insert(SNR_dB.end(), i);
384     }
385 
386     double normalValue;
```

```
387
388     for(int i =0; i<SNR_dB.size(); i++){
389         normalValue = pow(10, (SNR_dB[i]/10));
390         EnergyVector.insert(EnergyVector.end(), normalValue*N_o);
391     }
392
393
394     for(int step =0; step <SNR_dB.size(); step++){
395
396         sourceBits = Source();
397
398         Symbols = binaryToDecimalConversion(sourceBits, base);
399         // cout<< "Done symbols"<<endl;
400
401         trans1 = SignalVectors1(Symbols, EnergyVector[step], M);
402         trans2 = SignalVectors2(Symbols, EnergyVector[step], M);
403
404         gnoise1 = GnoiseVector(0.0, stddev);
405         gnoise2 = GnoiseVector(0.0, stddev);
406
407         receivedBits1 = channelModel(trans1, gnoise1);
408         receivedBits2 = channelModel(trans2, gnoise2);
409
410
411         decodedBits = decisionBlock(receivedBits1, receivedBits2, M);
412
413
414         errors = errorCount(Symbols, decodedBits);
415         double sizeOfsymbols = Symbols.size();
416         pe = errors/ sizeOfsymbols;
417
418         p_error.insert(p_error.end(), pe);
419
420         cout<< errors<< "\n";
421         cout<< pe << "\n";
422
423         cout<< "\n";
424
425     }
426
427
428     datafile(SNR_dB,p_error);
429
430     vector<double> qvalue = Qfunction(SNR_dB, M);
431     qvalueInFile(SNR_dB, qvalue);
432
433     return 0;
434
435
436 }
```

4.2 Results

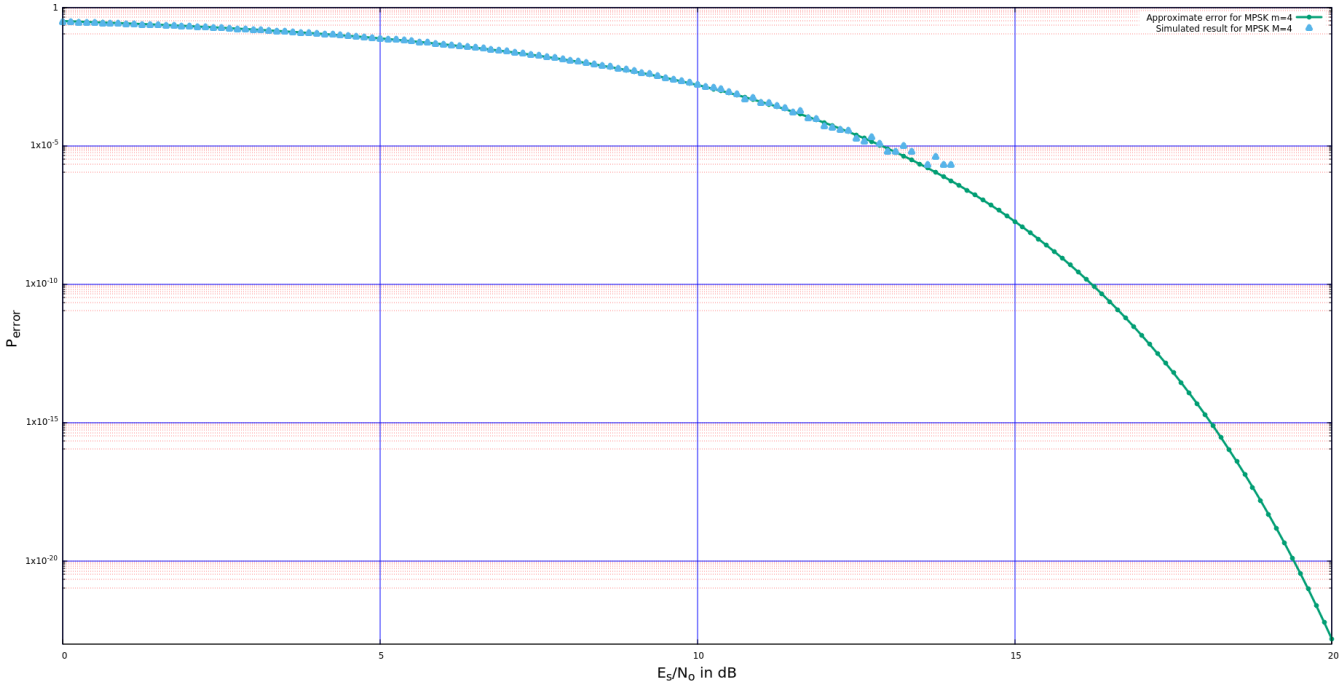

Figure 10: M =4, Result of simulation (4-PSK)

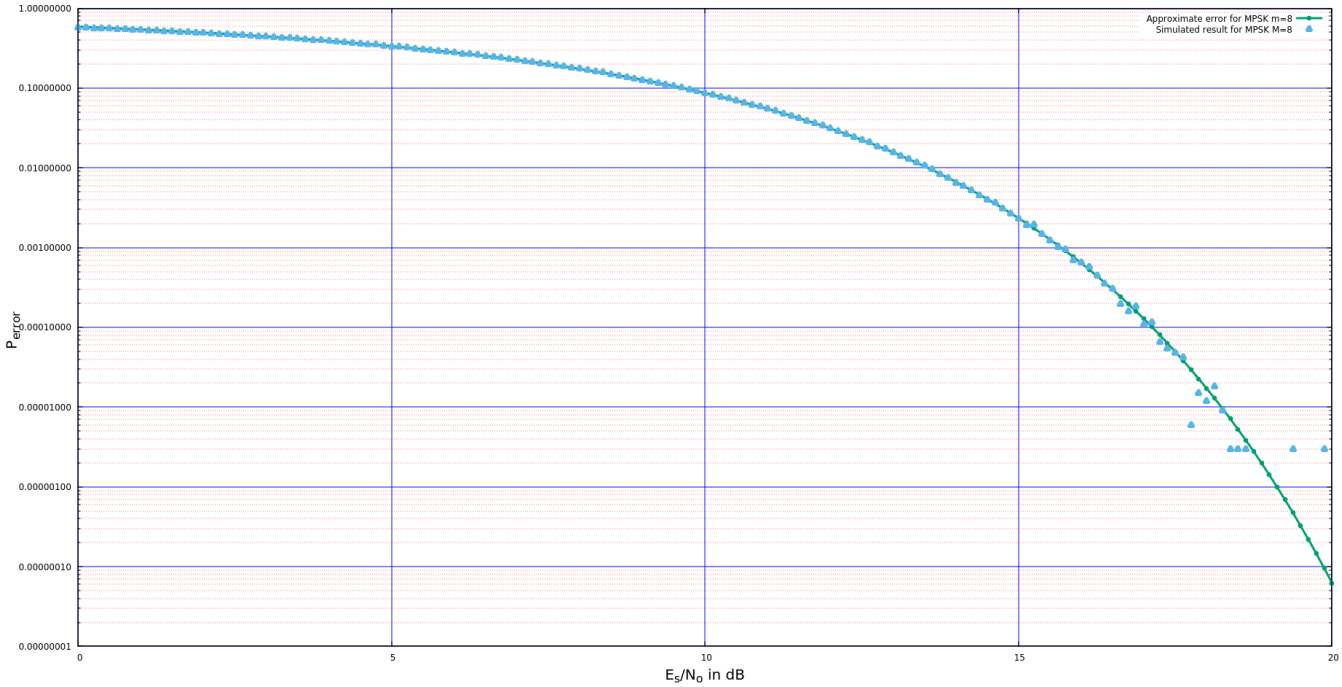

Figure 11: M =8, Result of simulation (8-PSK)

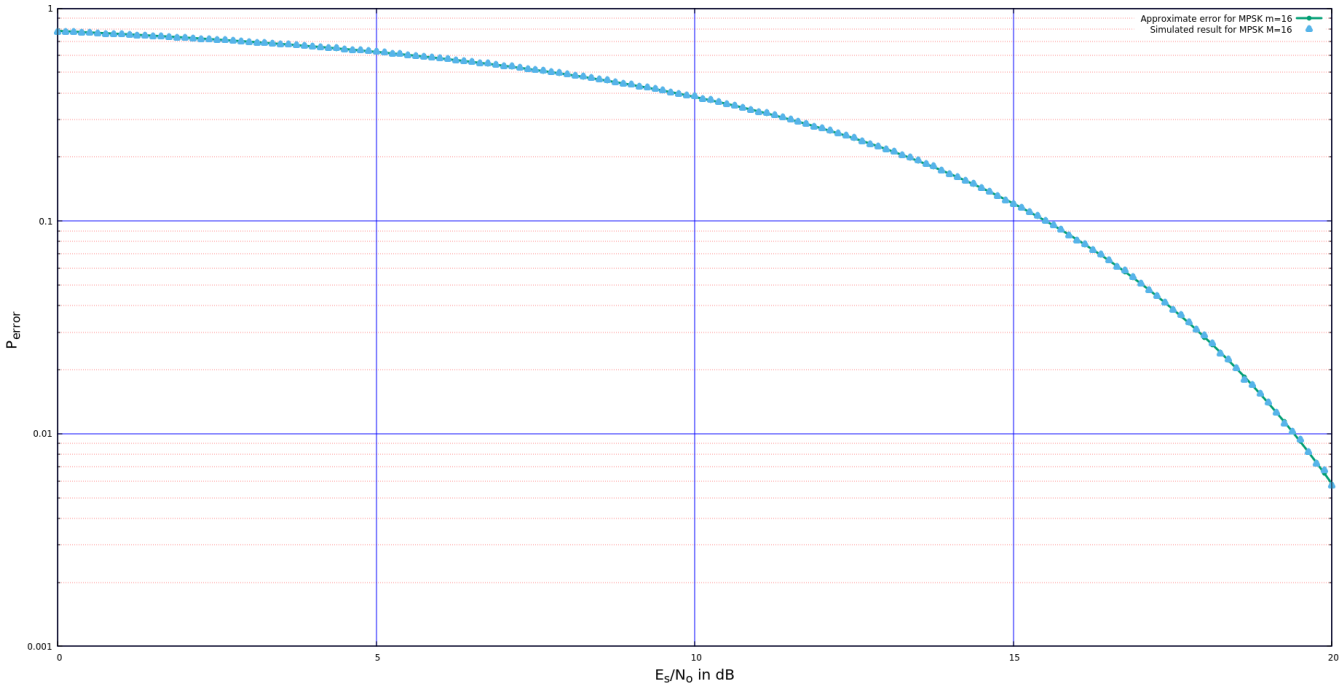

Figure 12: M =16, Result of simulation (16-PSK)

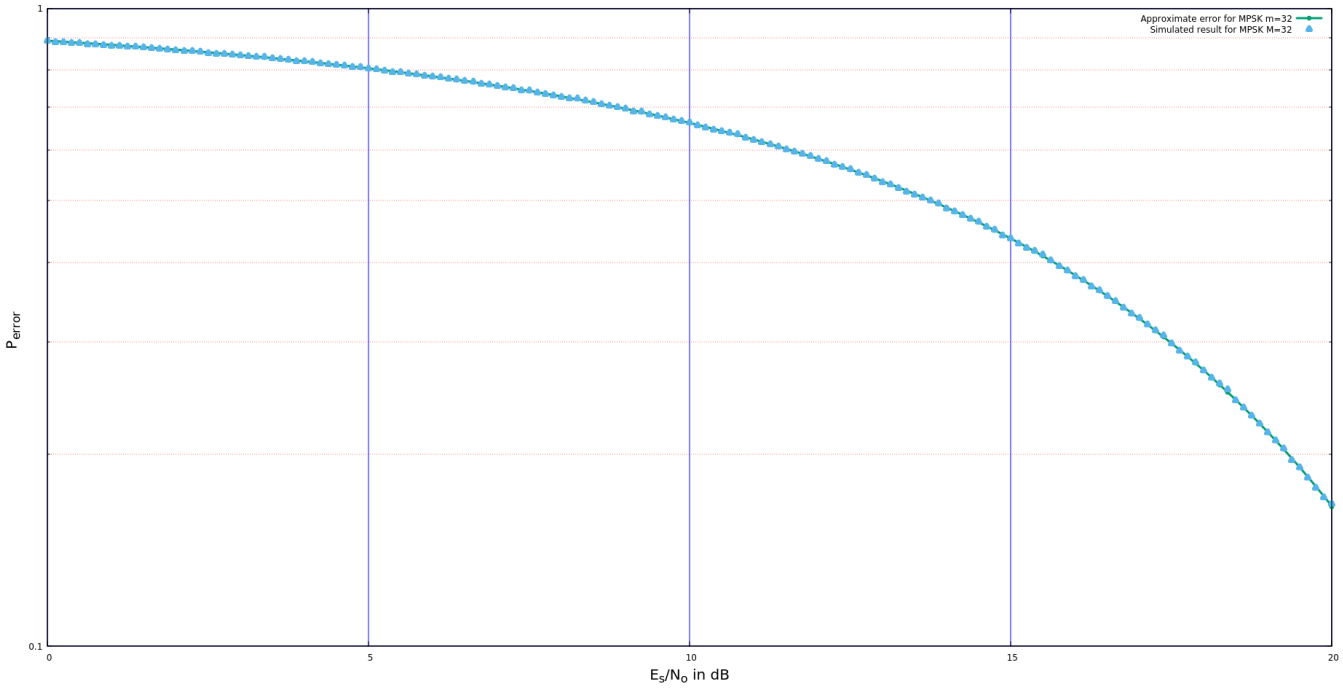

Figure 13: M =32, Result of simulation (32-PSK)

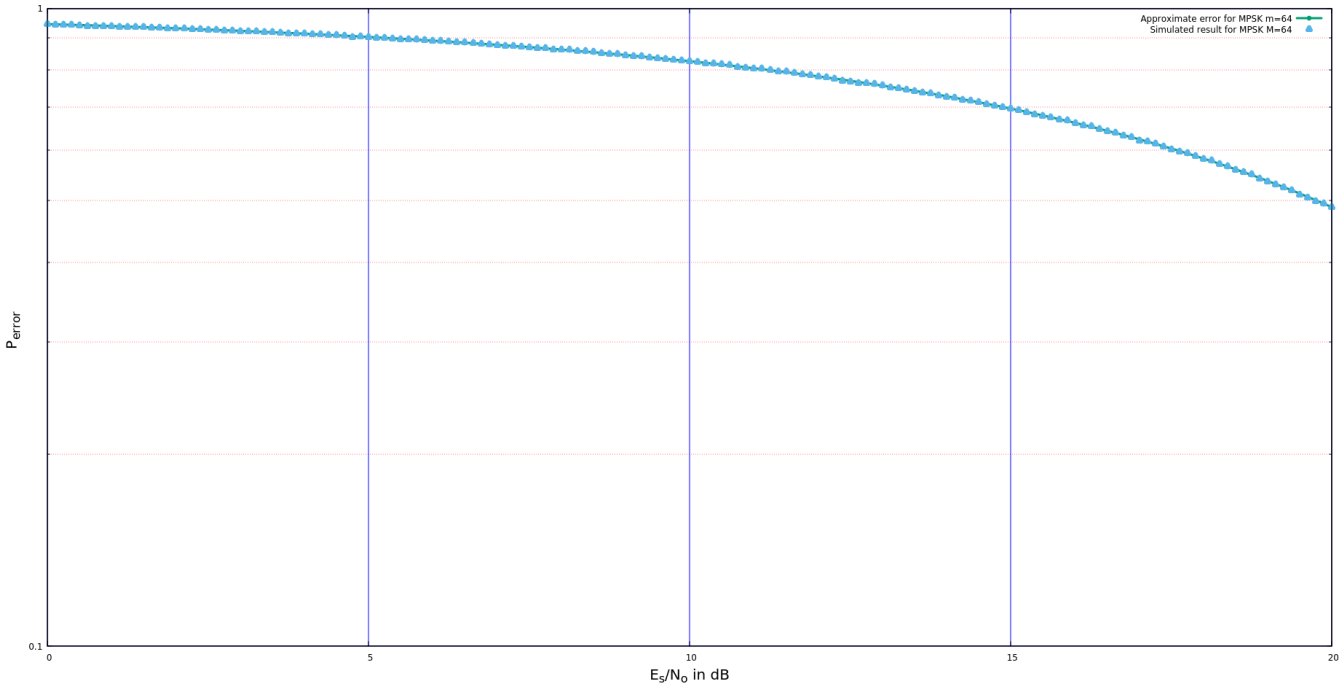

Figure 14: M =64, Result of simulation (64-PSK)

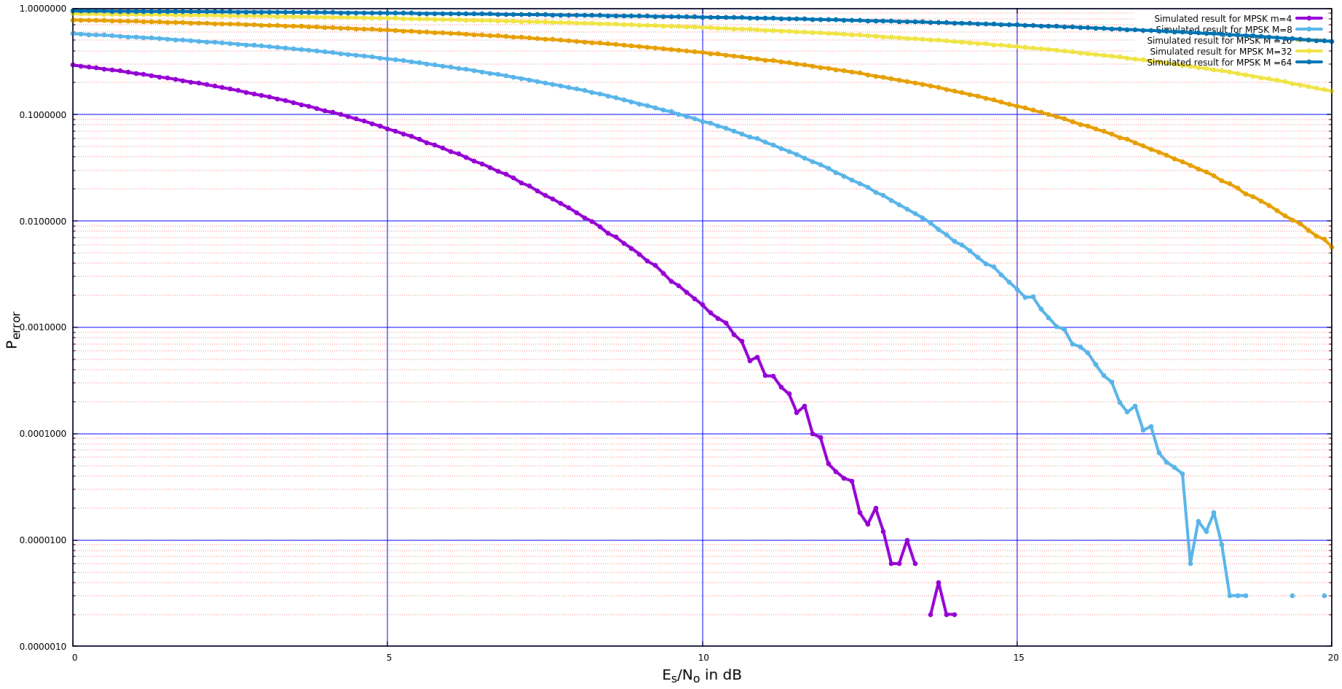

Figure 15: Comparison of all results(4-PSK, 8-PSK, 16-PSK, 32-PSK and 64-PSK)

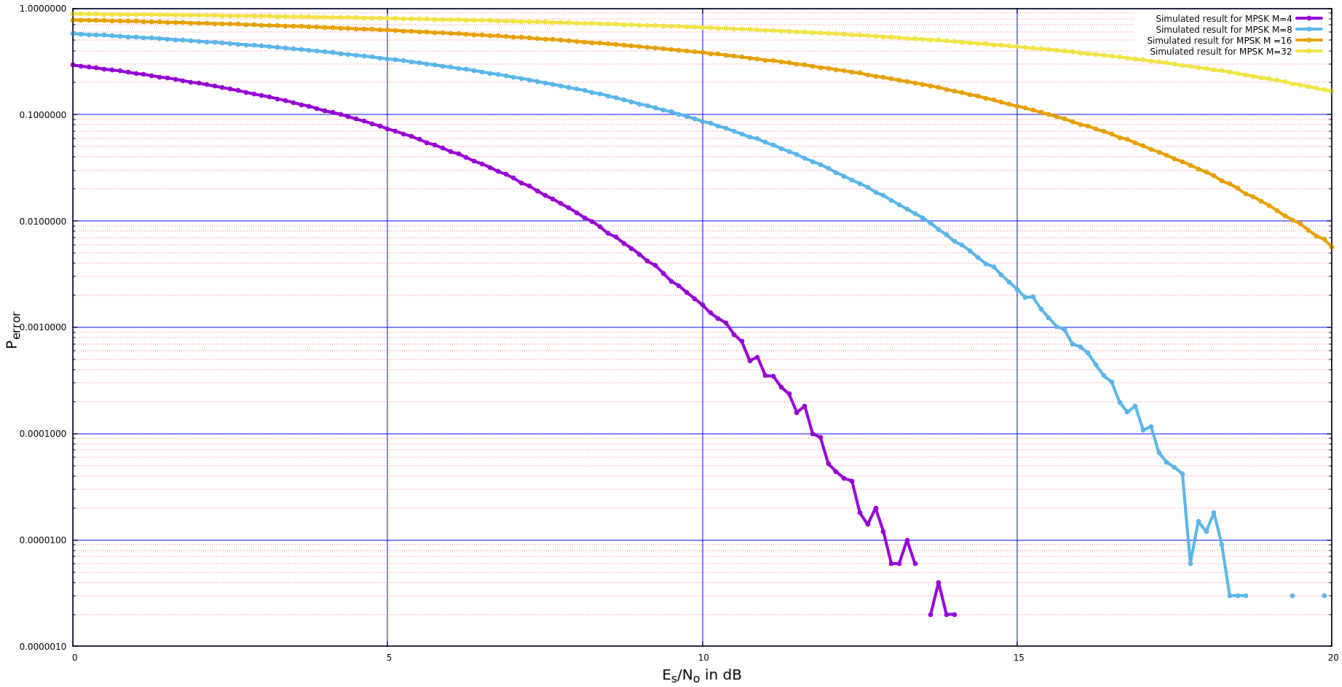

Figure 16: 4-PSK, 8-PSK, 16-PSK and 32-PSK

5 Inferences

1. As $\frac{E_s}{N_o}$ increase, probability of error decreases. But the slope of decrease is different for different M values in MPSK.
2. As M value increases, number of phase requires to modulate message, increases. Therefore, difference between two consecutive phase decreases, that makes difficult to distinguish two consecutive symbols by receiver. Hence, more probability of error.
3. Since $log_2 M$ (M>2) bits can be transmit through this scheme. That implies data rate is higher as compare to BPSK and QPSK. But it comes with high probability of error.
4. Since, simulated results are exactly same as calculated results. That implies the approximations in the derivation of probability of error in any MPSK are valid.