Preliminary Exercises: Show that if X is a random variable with mean 0 and variance 1 then

$$Y = aX + b$$

is a random variable with mean b and variance a^2.

Claim: $E[X] = 0$, $E[X^2] = 1$, and $Y = aX + b \implies E[Y] = a$

Proof: First, we compute the mean. Recall that the mean of a random variable Y is defined as the expected value of Y so that

$$\begin{aligned} E[Y] &= E[aX + b] \\ &= aE[X] + b \end{aligned}$$

but because X is zero-mean, then $E[X] = 0$ so that

$$E[Y] = aE[X] + b \implies E[Y] = b.$$

Therefore, the mean of Y is equal to b.

Claim: $Y = aX + b$, $E[X] = 0$, and $E[X^2] = 1 \implies \text{Var}(Y) = a^2$

Proof: Next, we show that the variance is equal to a^2. Begin by recalling that the variance is the expected squared difference from the mean, or that

$$\text{Var}(Y) = E[(Y - b)^2]$$

as the mean is equal to b per our last proof. Therefore,

$$\begin{aligned} \text{Var}(Y) = E[(Y - b)^2] &\implies \text{Var}(Y) = E[Y^2] - 2bE[Y] + b^2 \\ &\implies \text{Var}(Y) = E[(aX + b)^2] - 2bE[aX + b] + b^2 \\ &\implies \text{Var}(Y) = \big[E[a^2X^2] + 2E[abX] + b^2\big] - \big[2abE[X] + 2b^2\big] + b^2 \\ &\implies \text{Var}(Y) = a^2E[X^2] + 2abE[X] - 2abE[X] \\ &\implies \text{Var}(Y) = a^2E[X^2] \\ &\implies \text{Var}(Y) = a^2 \end{aligned}$$

BPSK Simulation

1. *Prompt:* Write a program that will simulate a BPSK communication system with unequal prior bit probabilities. Using your program, create data from which to plot the probability of bit error obtained from your simulation for SNRs in the range from 0 to 10 dB, for the three cases that $P_0 = 0.5$ (in which case your plot should look much like Figure 1.10), $p_0 = 0.25$, and $P_0 = 0.1$. Decide on an appropriate value of N

   Response: We accomplish this objective by constructing a *Source*, *Constellation*, and *ConstellationChannel* class. The source class is responsible for producing input bits which are equal to 1 with probability P_0. The constellation class is responsible for encoding bits to and decoding bits from a constellation of n symbols. Finally, the constellationChannel class adds uncorrelated white gaussian noise with variance σ^2 to an input set of symbols that come from encoding using the Constellation class.

src/Source.h

```cpp
#ifndef ERRORCORRECTIONCODING_SOURCECHANNEL_H
#define ERRORCORRECTIONCODING_SOURCECHANNEL_H
#include<random>
#include<memory>


template <class T>
class Source {
private:
    double pOne;
    std::default_random_engine generator;
    std::uniform_real_distribution<> dist;
    std::vector<T> batch;
    int nBatch;
public:
        // Define constructor
    explicit Source(double probabilityOfOne = 0.5, int inBatchSize = 1): dist{0, 1}, pOne{
        probabilityOfOne}, nBatch{inBatchSize} {
        batch = std::vector<T>(inBatchSize, false);
    }
    void reset(double probabilityOfOne, int inBatchSize = 1)
    {
        pOne = probabilityOfOne;
        nBatch = inBatchSize;
        batch = std::vector<T>(inBatchSize, false);
    }
    std::vector<T> generateInput()
    {
        for(int iBatch = 0; iBatch < nBatch; iBatch++)
        {
            batch[iBatch] = dist(generator) < pOne;
        }
        return batch;
    }
};


#endif //ERRORCORRECTIONCODING_SOURCECHANNEL_H
```

src/Constellation.h

```cpp
#ifndef ERRORCORRECTIONCODING_CONSTELLATION_H
#define ERRORCORRECTIONCODING_CONSTELLATION_H
enum constellationType{n2PSK, n4PSK, n8PSK};
#include <complex>
#include <vector>
#include "SimulationParameters.h"

// Constellation class: this defines how bits are encoded/decoded from constellation values.
class Constellation {
private:
    SimulationParameters SimParam;
    std::vector<std::complex<double>> table;
    const int forwardIndexTable[8] = {0, 1, 3, 2, 6, 7, 5, 4};
    int nSymbol;
```

```
    int nBitPerSymbol;
    double threshold;
    double sigma;
public:
    // Define Constructor
    Constellation(SimulationParameters SimParam): SimParam{SimParam}
    {
        // Initialize parameters based on how many symbols are given
        double amplitude = std::sqrt(SimParam.Eb);
        auto encodeMethod = SimParam.nPsk;
        switch(encodeMethod)
        {
            case n2PSK:
                nSymbol = 2;
                break;
            case n4PSK:
                nSymbol = 4;
                break;
            case n8PSK:
                nSymbol = 8;
                break;
        }
        nBitPerSymbol = std::log2(nSymbol);
        double degPerSymbol = 2*M_PI/nSymbol;

        // Give symbols for each set of bits in gray-code order
        for(int iSymbol = 0; iSymbol < nSymbol; iSymbol++)
        {
            double deg = degPerSymbol*forwardIndexTable[iSymbol];
            double r = std::cos(deg)*amplitude;
            double i = std::sin(deg)*amplitude;
            table.emplace_back(r,i);
        }

        // compute decoding threshold
        computeThreshold();


    }
    void computeThreshold()
    {
        double p1 = SimParam.bitProbabilityOfOne;
        double logRatio = std::log(p1/(1.0 - p1));
        double var = std::pow(sigma,2.0);
        double terms = 2.0*std::sqrt(SimParam.Eb);
        threshold = logRatio*var/terms;
    }

    // Encodes a set of bool values to symbols, number must be 0 moduls number of bits per symbol
    [[nodiscard]] std::vector<std::complex<double>> encode(const std::vector<bool>& input) const
    {
        // error checking
        int nInput = input.size();
        if(nInput%nBitPerSymbol != 0)
        {
            throw std::invalid_argument("number of input values must equal the number of bits per symbol.");
        }

        // initialize input parameters
        int nTransmitted = nInput/nBitPerSymbol;
        std::vector<std::complex<double>> output(nTransmitted);

        // loop over data and convert to constellation
        int idx;
        int iData = 0;
        for(int iTransmitted = 0; iTransmitted < nTransmitted; iTransmitted++)
        {
            idx = 0;
            for(int iBit= 0; iBit< nBitPerSymbol; iBit++)
```

```
            {
                idx += input[iData] << iBit;
                iData ++;
            }
            output[iTransmitted] = table[idx];
        }
        return output;
    }

    // Decodes symbols to the index spelled by the binary of the codeword.
    [[nodiscard]] std::vector<int> decodeToIndex(std::vector<std::complex<double>> symbol)
        const
    {
        int nInputSymbol = symbol.size();
        auto output = std::vector<int>(nInputSymbol);
        for(int iSymbol = 0; iSymbol < nInputSymbol; iSymbol++)
        {
            int idx = 0;
            double minDist = std::abs((symbol[iSymbol] - table[0]));
            for(int iSymbolOption = 0; iSymbolOption < nSymbol; iSymbolOption++)
            {
                double dist = std::abs(symbol[iSymbol] - table[iSymbolOption]);
                idx = dist < minDist? iSymbolOption:idx;
                minDist = std::min(dist,minDist);
            }
            output[iSymbol] = idx;
        }
        return output;
    }

    // Decodes symbols to binary
    [[nodiscard]] double getThreshold() const
    {
        return threshold;
    }

    void setNoiseSigma(double sigmaIn)
    {
        sigma = sigmaIn;
        computeThreshold();
    }
    [[nodiscard]] std::vector<int> n2PskDecodeToIndex(std::vector<std::complex<double>>
        symbols) const
    {
        int nInputSymbol = symbols.size();
        auto output = std::vector<int>(nInputSymbol);
        for(int iSymbol = 0; iSymbol < nInputSymbol; iSymbol ++)
        {
            output[iSymbol] = symbols[iSymbol].real() > threshold?0:1;
        }
        return output;
    }
    bool isUsingBpsk() const
    {
        return SimParam.nPsk == n2PSK;
    }
    [[nodiscard]] std::vector<bool> decode(const std::vector<std::complex<double>>& symbol)
        const
    {
        auto indices = isUsingBpsk()? n2PskDecodeToIndex(symbol):decodeToIndex(symbol);
        int nInputSymbol = symbol.size();
        auto output = std::vector<bool>(nInputSymbol*nBitPerSymbol);
        for(int iSymbol = 0; iSymbol < nInputSymbol; iSymbol++)
        {
            auto idx = indices[iSymbol];
            for(int iBit = 0; iBit < nBitPerSymbol; iBit++)
            {
                output[iSymbol*nBitPerSymbol + iBit] = (idx%2 != 0);
                idx = idx >> 2;
            }
        }
        return output;
```

```
    }

    // Define getter functions
    [[nodiscard]] int getNSymbol() const
    {
        return nSymbol;
    }

    [[nodiscard]] int getNBitPerSymbol() const
    {
        return nBitPerSymbol;
    }

};


#endif //ERRORCORRECTIONCODING_CONSTELLATION_H
```

src/ConstellationChannel.h

```
#ifndef ERRORCORRECTIONCODING_CONSTELLATIONCHANNEL_H
#define ERRORCORRECTIONCODING_CONSTELLATIONCHANNEL_H

#include <vector>
#include <complex>
#include <random>

class ConstellationChannel {
private:
    double sigma;
    std::default_random_engine generator;
    std::normal_distribution<double> dist;
public:

    // Constructor
    explicit ConstellationChannel(double sigma = 1): dist(0,sigma), sigma(sigma){}
    void reset(double sigma)
    {
        dist = std::normal_distribution<double>(0, sigma);
        sigma = sigma;
    }

    // Encorporates channel effects to a set of input symbols
    std::vector<std::complex<double>> addChannelEffects(std::vector<std::complex<double>>
        symbols)
    {
        int nSymbol = symbols.size();
        std::vector<std::complex<double>> output;
        output.reserve(nSymbol);
        double r, i;
        for(int iSymbol = 0; iSymbol < nSymbol; iSymbol++)
        {
            r = dist(generator);
            i = dist(generator);
            output.push_back(symbols[iSymbol] + std::complex<double>(r,i));
        }
        return output;
    }
};

#endif //ERRORCORRECTIONCODING_CONSTELLATIONCHANNEL_H
```

Error metrics are computed using a *PerformanceMetrics* class which computs both theoretical and actual error metrics.

src/PerformanceMetric.h

```
//
// Created by danie on 9/5/2023.
//

```

```cpp
#ifndef ERRORCORRECTIONCODING_PERFORMANCEMETRIC_H
#define ERRORCORRECTIONCODING_PERFORMANCEMETRIC_H

#include <utility>
#include <vector>
#include <complex>
#include <cassert>
#include <memory>
#include "Constellation.h"

// Encapsulates the theoretical error estimates
struct ErrorBounds{
    ErrorBounds(double bit, double symbol): bit(bit), symbol(symbol) {}
    double bit;
    double symbol;
};
template <class T>
class PerformanceMetric {
private:
    int nError = 0;
    int nTotal = 0;
    double probabilityError = 0;
    double nSymbolError = 0;
    double probabilitySymbolError = 0;
    double nSymbolCount = 0;
public:
    explicit PerformanceMetric(){}
    [[nodiscard]] static double stdNormalCdf(const double& x)
    {
        return std::erfc(-x/std::sqrt(2))/2;
    }
    [[nodiscard]] ErrorBounds computeProbabilityErrorBounds(const double Eb, const double N0,
        const double pOne, const double sigma, const std::shared_ptr<Constellation>&
        decoder_ptr) const
    {
        double pBitErrorBound;
        double pSymbolErrorBound;
        if(decoder_ptr->getNBitPerSymbol() == 1)
        {
            double threshold = decoder_ptr -> getThreshold();
            double pErrorGiven0 = stdNormalCdf((threshold - sqrt(Eb))/sqrt(N0/2));
            double pErrorGiven1 = 1 - stdNormalCdf((threshold + sqrt(Eb))/sqrt(N0/2));
            pBitErrorBound = pErrorGiven0*(1 - pOne) + pErrorGiven1*pOne;
            pSymbolErrorBound = pBitErrorBound;
        }
        else{

            double degPerSymbol = 2*M_PI/decoder_ptr -> getNSymbol();
            double minDist = std::pow(Eb,2)*(2 - 2*std::cos(degPerSymbol));
            pSymbolErrorBound = 2*(1 - stdNormalCdf(minDist/(sigma*2)));
            pBitErrorBound = pSymbolErrorBound/decoder_ptr -> getNBitPerSymbol();
        }
        return ErrorBounds{pBitErrorBound,pSymbolErrorBound};
    }

    // This function computes the rolling probability of bit error for a set of samples
     double evaluateBits(const std::vector<T>& p1, const std::vector<T>& p2)
    {
        auto nPoint = p1.size();
        for(int iPoint = 0; iPoint < nPoint; iPoint++)
        {
            nError += (p1[iPoint] + p2[iPoint])%2;
        }
        nTotal += nPoint;
        probabilityError = double(nError)/double(nTotal);
        return probabilityError;
    }

    // This function computes the rolling probability of symbol error for a set of samples
    double evaluateSymbols(const std::vector<std::complex<double>>& symbolTransmitted, const
        std::vector<std::complex<double>>& symbolReceived, const std::shared_ptr<Constellation
        >& decoder_ptr)
```

```
    {
        auto transmittedIdx = decoder_ptr -> decodeToIndex(symbolTransmitted);
        auto receivedIdx = decoder_ptr -> decodeToIndex(symbolReceived);
        auto nInput = transmittedIdx.size();
        for(int iInput=0; iInput< nInput; iInput++)
        {
            nSymbolError += transmittedIdx[iInput] != receivedIdx[iInput];
            nSymbolCount += 1;
        }
        probabilitySymbolError = nSymbolError/nSymbolCount;

        return probabilitySymbolError;
    }

    // Used to reset the error counts between simulations
    void reset()
    {
        nTotal = 0;
        probabilityError = 0;
        nError = 0;
        nSymbolCount = 0;
        nSymbolError = 0;
        probabilitySymbolError = 0;
    }

    // Define getter functions
    [[nodiscard]] double getSymbolErrorProbability() const
    {
        return probabilitySymbolError;
    }
    [[nodiscard]] int getNBitError() const
    {
        return nError;
    }
    [[nodiscard]] double getBitErrorProbability() const
    {
        return probabilityError;
    }

};


#endif //ERRORCORRECTIONCODING_PERFORMANCEMETRIC_H
```

Each class is used together as follows:

src/main.cpp

```
#include <iostream>
#include "Source.h"
#include "Constellation.h"
#include "ConstellationChannel.h"
#include "PerformanceMetric.h"
#include "SimulationParameters.h"
#include "HammingCode74.h"
#include "HammingCode1511.h"
#include "BinarySymmetricChannel.h"
#include <sstream>
#include <string>
template <class T>
std::string toReportString(const SimulationParameters&, const std::shared_ptr<
    PerformanceMetric<T>>&, const ErrorBounds&, double);
void nPskSimulation(const SimulationParameters&);
void hammingSimulation(const SimulationParameters&);

// main
int main() {

    // BPSK for 0.5 probability of one in prior bits
    std::cout<<"Simulation with p=0.5 with BPSK"<<std::endl;
    auto P = SimulationParameters{};
    P.nRequiredError = 50;
```

```
    P.Eb = 1;
    P.bitProbabilityOfOne = 0.5;//, 0.25, 0.1};
    P.snrDb = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // in db
    P.nPsk = n2PSK;
    hammingSimulation(P);
//    nPskSimulation(P);
//    std::cout<<"End Simulation"<<std::endl<<std::endl;
//
//    // BPSK for 0.25 probability of one in prior bits
//    std::cout<<"Simulation with p=0.25 with BPSK"<<std::endl;
//    P.bitProbabilityOfOne = 0.25;//, 0.25, 0.1};
//    nPskSimulation(P);
//    std::cout<<"End Simulation"<<std::endl<<std::endl;
//
//    // BPSK for 0.1 probability of one in prior bits
//    std::cout<<"Simulation with p=0.1" << std::endl;
//    P.bitProbabilityOfOne = 0.1;//, 0.25, 0.1};
//    nPskSimulation(P);
//    std::cout<<"End Simulation"<<std::endl<<std::endl;
//
//    // 8PSK for 0.5 probability of one in prior bits
//    std::cout<<"Simulation with p=0.5 with 8PSK" <<std::endl;
//    P.nRequiredError = 10000;
//    P.bitProbabilityOfOne = 0.5;//, 0.25, 0.1};
//    P.nPsk = n8PSK;
//    nPskSimulation(P);
//    std::cout<<"End Simulation"<<std::endl<<std::endl;

    return 0;
}
void hammingSimulation(const SimulationParameters& P)
{
    unsigned int nInput = 11;
    auto ConstellationMap = std::make_shared<Constellation>(P);
    auto BitSource = std::make_shared<Source<unsigned int>>(0.5, nInput);
    auto Code = std::make_shared<HammingCode1511>();
    auto Metric = std::make_shared<PerformanceMetric<unsigned int>>();
    for(double snrDb: P.snrDb){
        double snr = std::pow(10, snrDb/10);
        double N0 = P.Eb/snr;
        double sigma = sqrt(N0/2);
        double pError = (1 - Metric->stdNormalCdf(std::sqrt(2*P.Eb/N0)));
        auto Channel = std::make_shared<BinarySymmetricChannel>(pError);
        Metric->reset();

        // test until minimum number of errors has been met
        while (Metric -> getNBitError() < P.nRequiredError) {
            auto transmittedBits = BitSource -> generateInput();
            auto encodedBits = Code->encode(transmittedBits);
//            auto receivedBits = encodedBits;
//            receivedBits[2] += receivedBits[2] == 0?1:-1;
            auto receivedBits = Channel->addChannelEffects(encodedBits);
            auto decodedBits = Code->decode(receivedBits);
            Metric -> evaluateBits(transmittedBits, decodedBits);
        }

        auto Error = Metric -> computeProbabilityErrorBounds(P.Eb, N0, P.bitProbabilityOfOne, sigma, ConstellationMap);
        auto output = toReportString(P,Metric,Error,snrDb);
        std::cout << output << std::endl;

    }
}
// This function runs a single simulation over different SNR values
void nPskSimulation(const SimulationParameters& P){
    auto BitSource = std::make_shared<Source<bool>>();
    auto ConstellationMap = std::make_shared<Constellation>(P);
    auto Channel = std::make_shared<ConstellationChannel>();
    auto Metric = std::make_shared<PerformanceMetric<bool>>();

    // compute probability of error for different parameters
    for(double snrDb : P.snrDb) {
```

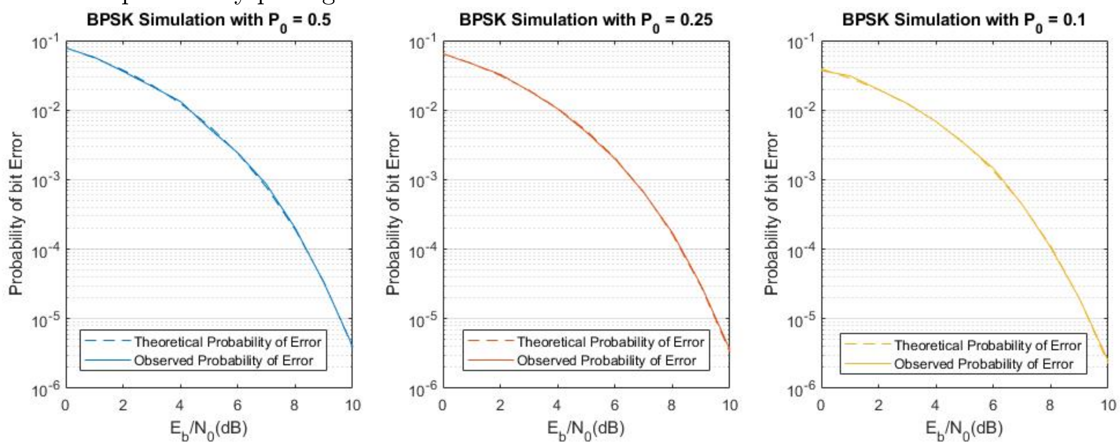
```

        // convert from decibles
        double snr = std::pow(10,snrDb / 10 );
        double N0 = P.Eb / snr;
        double sigma = sqrt(N0/2);

        // set parameters for simulation
        BitSource -> reset(P.bitProbabilityOfOne, ConstellationMap -> getNBitPerSymbol());
        ConstellationMap -> setNoiseSigma(sigma);
        Channel -> reset(sigma);
        Metric -> reset();

        // test until minimum number of errors has been met
        while (Metric -> getNBitError() < P.nRequiredError) {
            auto transmittedBits = BitSource -> generateInput();
            auto transmittedSymbol = ConstellationMap -> encode(transmittedBits);
            auto receivedSymbol = Channel -> addChannelEffects(transmittedSymbol);
            auto receivedBits = ConstellationMap -> decode(receivedSymbol);
            Metric -> evaluateBits(transmittedBits, receivedBits);
            Metric -> evaluateSymbols(transmittedSymbol, receivedSymbol, ConstellationMap);
        }

        // compute lower bound for probability of error
        auto Error = Metric -> computeProbabilityErrorBounds(P.Eb, N0, P.bitProbabilityOfOne,
            sigma, ConstellationMap);
        auto output = toReportString(P,Metric,Error,snrDb);
        std::cout << output << std::endl;
    }
}
template <class T>
std::string toReportString(const SimulationParameters& P, const std::shared_ptr<
    PerformanceMetric<T>>& Metric, const ErrorBounds& Error, double snrDb)
{
    // report error statistics
    std::stringstream sstream;
    sstream.setf(std::ios::scientific);
    sstream.precision(3);

    // convert pBitError to string
    sstream << Metric -> getBitErrorProbability();
    std::string pBitErrorStr = sstream.str();
    sstream.str(std::string());

    sstream << P.bitProbabilityOfOne;
    std::string pOneStr = sstream.str();
    sstream.str(std::string());

    sstream << Error.bit;
    std::string pBitErrorBoundStr = sstream.str();
    sstream.str(std::string());

    sstream << Error.symbol;
    std::string pSymbolErrorBoundStr = sstream.str();
    sstream.str(std::string());

    sstream << Metric -> getSymbolErrorProbability();
    std::string pSymbolErrorStr = sstream.str();
    sstream.str(std::string());

    sstream.setf(std::ios::fixed);
    sstream << snrDb;
    std::string snrStr = sstream.str();
    sstream.str(std::string());

    std::string output = ("pBitErrorBound: " + pBitErrorBoundStr);
    output += (" pBitError: " + pBitErrorStr);
    output += (" pSymbolErrorBound: " + pSymbolErrorBoundStr);
    output += (" pSymbolError: " + pSymbolErrorStr);
    output += (" SNR: " + snrStr);
    return output;
}
```

2. *Prompt:* Prepare data from which to plot the theoretical probability of error (1.24) for the same three values of P_0. (You may want to combinen these first two programs into a single program).

   Response: This was accomplished by implementing equation 1.24 as part of the PerformanceMetric class given in PerformanceMetric.h in the function computeProbabilityErrorBounds.

3. *Prompt:* Plot the simulated probability of error on the same axes as the theoretical probability of error. The plots should have E_b/N_0 in dB as the horiaontal axis and the probability as the vertical axis, plotted on a logarithmic scale.

   Response: See the probability plots given below.

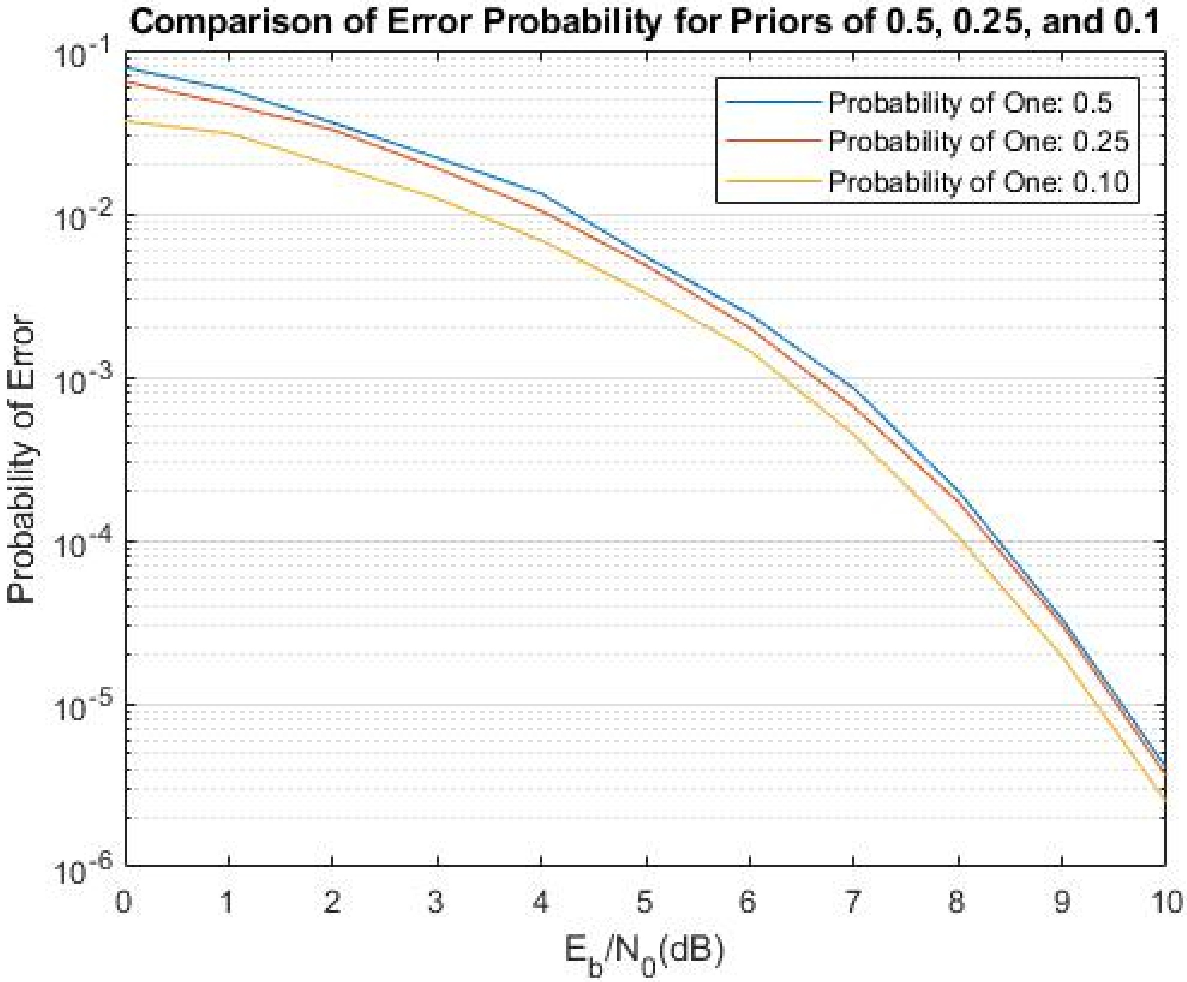


4. *Prompt:* Compare the theoretical and simulated results. Comment on the accuracy of the simiulation and the amount of time it tool to run the simulation. Comment on the importance of the theoretical models (where it is possible to obtain them).

   Response: In the plots above, we see that the simulated results closely resemble the theoretical results. It was helpful to have a mathematical description of what to expect because it helped to thoroughly test the code at a system level and identify errors. These results were also repeatable, although I did require that each simulation achieve at least 100 errors before terminating which helped manage the variability in the results. Fortunately, 100 errors wasn't overly taxing from a compute power perspective, especially with lower SNR values. With SNR values of 0 to 6, the simulation would run in several seconds, however when we started simulating values in the range of 9 or 10 the simulations could take as much as a minute to run. I can see how simulating performance with SNR values of 20 or more could require more time.

5. *Prompt:* Plot the probability of error for $P_0 = 0.1, P_0 = 0.25$ and $P_0 = 0.5$ on the same axes. Compare them and comment.

   Response: Observe the the probability of error for the three scenarios given below:



and how the probability of error seems to decrease as the prior probability becomes more certain. Better performance with a more polarized probability is expected because the system's behavior is more predictable. By modifying the likelihood function to account for the difference in probability distributions we effectively encorporate more information into our system.

8-PSK Simulation

- *Prompt:* Write a program that will simulate an 8-PSK communication system with equal prior bit probabilities. Use a signal constellation in which the pointsnumbered in Gray code order. Make your program so that you can estimate botht eh symbol error probability and the bit error probability. Decide on an appropriate value of N.

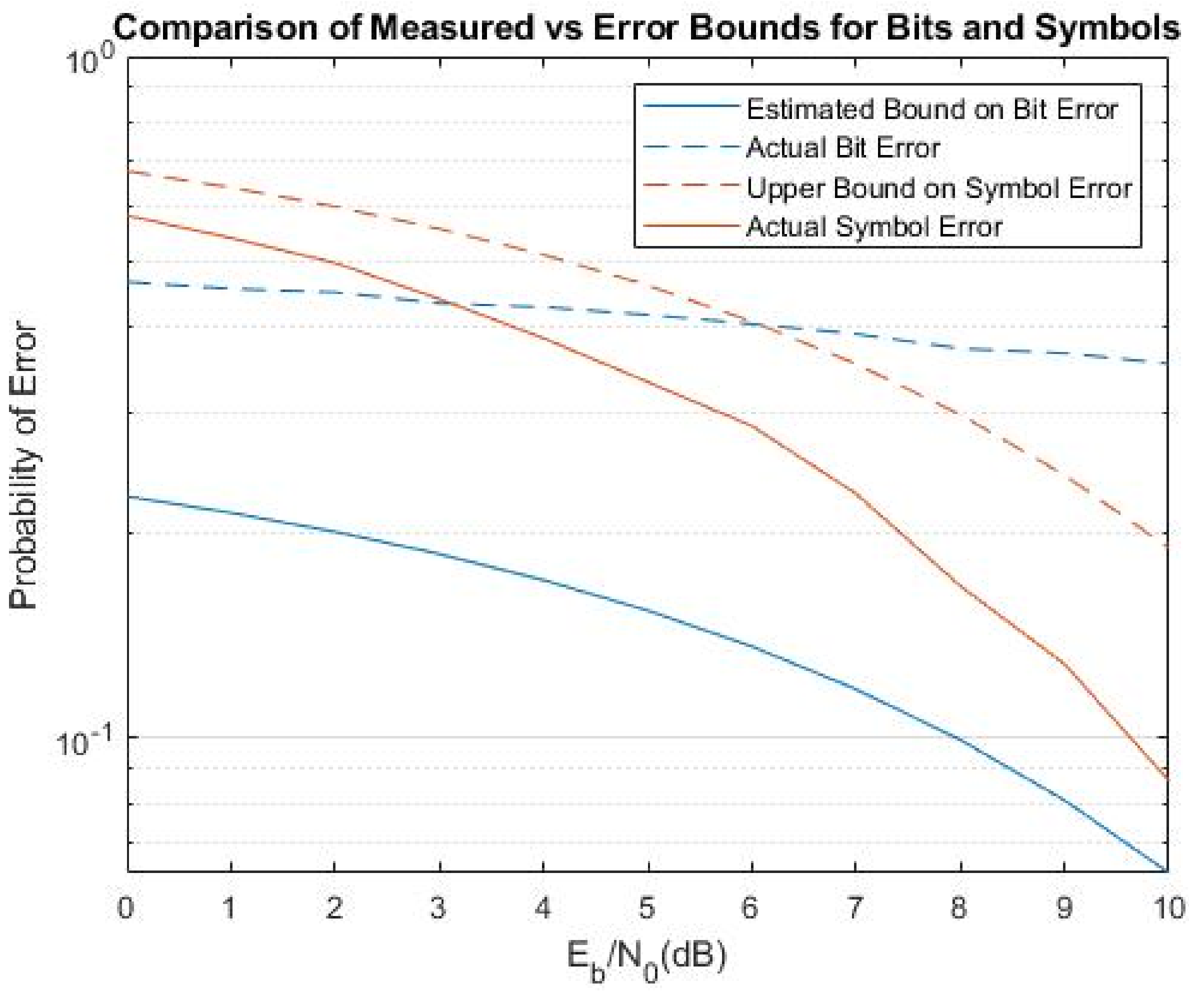
  Response: This was accomplished in tandom with the BPSK code given in the previous section. The only change was to use an N of 10000 because there were more errors in this simulation.

- *Prompt:* Prepare data from which to plot the bound on the probability of symbol error P_s using (1.26) and the probability of bit error P_b using (1.28).

  Response: This was accomplished as part of the PerformanceMetric class given above.

- *Prompt:* Plot the simulated probability of symbol error and bit error on the same axes as the bounds on the probabilities of error.

  Response: A plot containing each of these elements is given below.



- *Prompt:* Compare the theoretical and simulated results. Comment on the accuracy of the bound compared to the simulation and the amount of time it took to run the simulation.

  Response: The theoretical results did act as an upper bound, although the accuracy of the bounds is rather poor. If feels like if these bounds are the only estimate available, then there is a good chance I would end up overdesigning a system because of their lack of accuracy, especially when estimating the error bounds for the probability of bit error. In regards to the simulation compute time, this particular simulation was run with a minimum error count of 10000 and ran relatively quick, requiring 5 - 10 seconds to complete.

Coded BPSK Simulation

1. *Prompt:* Write a program that will simulate performance of the $(7, 4)$ Hamming code over a BSC channel with channel crossover probability $p = Q(\sqrt{2E_b/N_0})$ and plot the probability of error as a function of E_b/N_0 in dB. On the same plot, plot the theoretical probability of error for uncoded BPSK transmission. Identify what the coding gain is for a probability of error $P_b = 10^{-5}$.

   Response: Per the suggestions given in the problem description, I put together a BinarySymmetricChannel and HammingCode class that are shown below:

src/BinarySymmetricChannel.h

```cpp
//
// Created by danie on 9/7/2023.
//

#ifndef ERRORCORRECTIONCODING_BINARYSYMMETRICCHANNEL_H
#define ERRORCORRECTIONCODING_BINARYSYMMETRICCHANNEL_H
#include<random>

class BinarySymmetricChannel {
private:
    double pError;
    std::default_random_engine generator;
    std::uniform_real_distribution<> dist;
public:
    explicit BinarySymmetricChannel(double pError): pError(pError) {}
    unsigned int addChannelEffects(unsigned int bitIn){
       return (bitIn + dist(generator) < pError)%2;
    }
    std::vector<unsigned int> addChannelEffects(const std::vector<unsigned int>& bitIn){
        auto nBit = bitIn.size();
        std::vector<unsigned int>bitOut(nBit);
        for(int iBit = 0; iBit < nBit; iBit++){
            bitOut[iBit] = (bitIn[iBit] + (dist(generator) < pError))%2;
        }
        return bitOut;
    }

};


#endif //ERRORCORRECTIONCODING_BINARYSYMMETRICCHANNEL_H
```

src/HammingCode74.h

```cpp
//
// Created by danie on 9/10/2023.
//

#ifndef ERRORCORRECTIONCODING_HAMMINGCODE74_H
#define ERRORCORRECTIONCODING_HAMMINGCODE74_H
#include<memory>
#include<vector>
#include <cassert>

class HammingCode74 {
public:
    unsigned int nInput{4};
    unsigned int nOutput{7};
    unsigned int nParity{3};
    std::vector<unsigned int> lookup{8};
    std::vector<std::vector<unsigned int>> G;
    std::vector<std::vector<unsigned int>> H;

HammingCode74(){
    G.emplace_back((std::initializer_list<unsigned int>){1, 0, 0, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 1, 0, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 0, 1, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 0, 0, 1});
    G.emplace_back((std::initializer_list<unsigned int>){1, 0, 1, 1});
```

```
    G.emplace_back((std::initializer_list<unsigned int>){1, 1, 1, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 1, 1, 1});
    H.emplace_back((std::initializer_list<unsigned int>){1, 0, 1, 1, 1, 0, 0});
    H.emplace_back((std::initializer_list<unsigned int>){0, 1, 0, 1, 1, 1, 0});
    H.emplace_back((std::initializer_list<unsigned int>){0, 0, 1, 0, 1, 1, 1});

    for(int iSyndrome = 0; iSyndrome < nOutput; iSyndrome ++)
    {
        unsigned int val = 0;
        for(int iParity = 0; iParity < nParity; iParity ++)
        {
            val += (H[iParity][iSyndrome] << iParity);
        }
        lookup[val] = iSyndrome;
    }

}
std::vector<unsigned int> encode(std::vector<unsigned int> input)
{
    std::vector<unsigned int> output(nOutput,0);
    for(int iRow = 0; iRow < nOutput; iRow ++)
    {
        unsigned int temp = 0;
        for(int iCol = 0; iCol < nInput; iCol ++)
        {
            output[iRow] ^= (G[iRow][iCol]&input[iCol]);
        }
    }
    return output;
}
std::vector<unsigned int> decode(std::vector<unsigned int> input)
{
    unsigned int syndrom{0};
    for(int iSyndrom = 0; iSyndrom < nParity; iSyndrom++)
    {
        unsigned int val{0};
        for(int iCol = 0; iCol < nOutput; iCol++)
        {
            val ^= H[iSyndrom][iCol]&input[iCol];
        }
        syndrom += (val << iSyndrom);
    }
    if(syndrom != 0)
    {
        input[lookup[syndrom]] = (input[lookup[syndrom]] + 1)%2;
    }

    return {input.begin(),input.begin() + nInput};
}
};


#endif //ERRORCORRECTIONCODING_HAMMINGCODE74_H
```

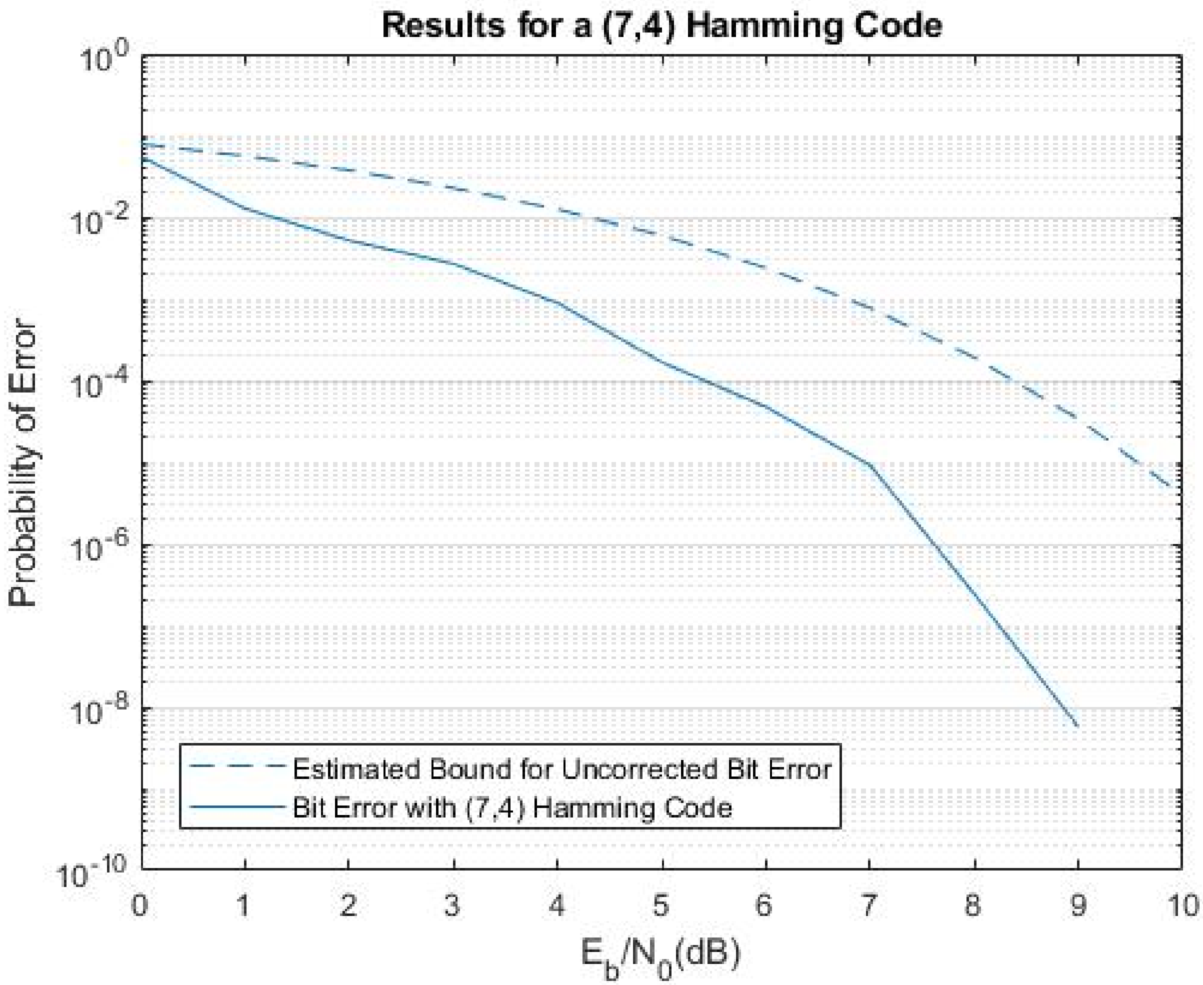
src/HammingCode1511.h

```
//
// Created by danie on 9/10/2023.
//

#ifndef ERRORCORRECTIONCODING_HAMMINGCODE1511_H
#define ERRORCORRECTIONCODING_HAMMINGCODE1511_H
#include<memory>
#include<vector>
#include <cassert>

class HammingCode1511 {
public:
    unsigned int nInput{11};
    unsigned int nOutput{15};
    unsigned int nParity{4};
    std::vector<unsigned int> lookup{16};
```

```cpp
    std::vector<std::vector<unsigned int>> G;
    std::vector<std::vector<unsigned int>> H;

HammingCode1511(){
    G.emplace_back((std::initializer_list<unsigned int>){1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0});
    G.emplace_back((std::initializer_list<unsigned int>){0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1});
    G.emplace_back((std::initializer_list<unsigned int>){1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1});
    G.emplace_back((std::initializer_list<unsigned int>){1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1});
    G.emplace_back((std::initializer_list<unsigned int>){0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1});
    G.emplace_back((std::initializer_list<unsigned int>){0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1});
    H.emplace_back((std::initializer_list<unsigned int>){1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1,
        0, 0, 0});
    H.emplace_back((std::initializer_list<unsigned int>){1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0,
        1, 0, 0});
    H.emplace_back((std::initializer_list<unsigned int>){0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0,
        0, 1, 0});
    H.emplace_back((std::initializer_list<unsigned int>){0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,
        0, 0, 1});

    for(int iSyndrome = 0; iSyndrome < nOutput; iSyndrome ++)
    {
        unsigned int val = 0;
        for(int iParity = 0; iParity < nParity; iParity ++)
        {
            val += (H[iParity][iSyndrome] << iParity);
        }
        lookup[val] = iSyndrome;
    }

}
std::vector<unsigned int> encode(std::vector<unsigned int> input)
{
    std::vector<unsigned int> output(nOutput,0);
    for(int iRow = 0; iRow < nOutput; iRow ++)
    {
        unsigned int temp = 0;
        for(int iCol = 0; iCol < nInput; iCol ++)
        {
            output[iRow] ^= (G[iRow][iCol]&input[iCol]);
        }
    }
    return output;
}
std::vector<unsigned int> decode(std::vector<unsigned int> input)
{
    unsigned int syndrom{0};
    for(int iSyndrom = 0; iSyndrom < nParity; iSyndrom++)
    {
        unsigned int val{0};
        for(int iCol = 0; iCol < nOutput; iCol++)
        {
            val ^= H[iSyndrom][iCol]&input[iCol];
        }
        syndrom += (val << iSyndrom);
    }
    if(syndrom != 0)
    {
        input[lookup[syndrom]] = (input[lookup[syndrom]] + 1)%2;
    }

    return {input.begin(),input.begin() + nInput};
}
};
```

```


#endif //ERRORCORRECTIONCODING_HAMMINGCODE1511_H
```

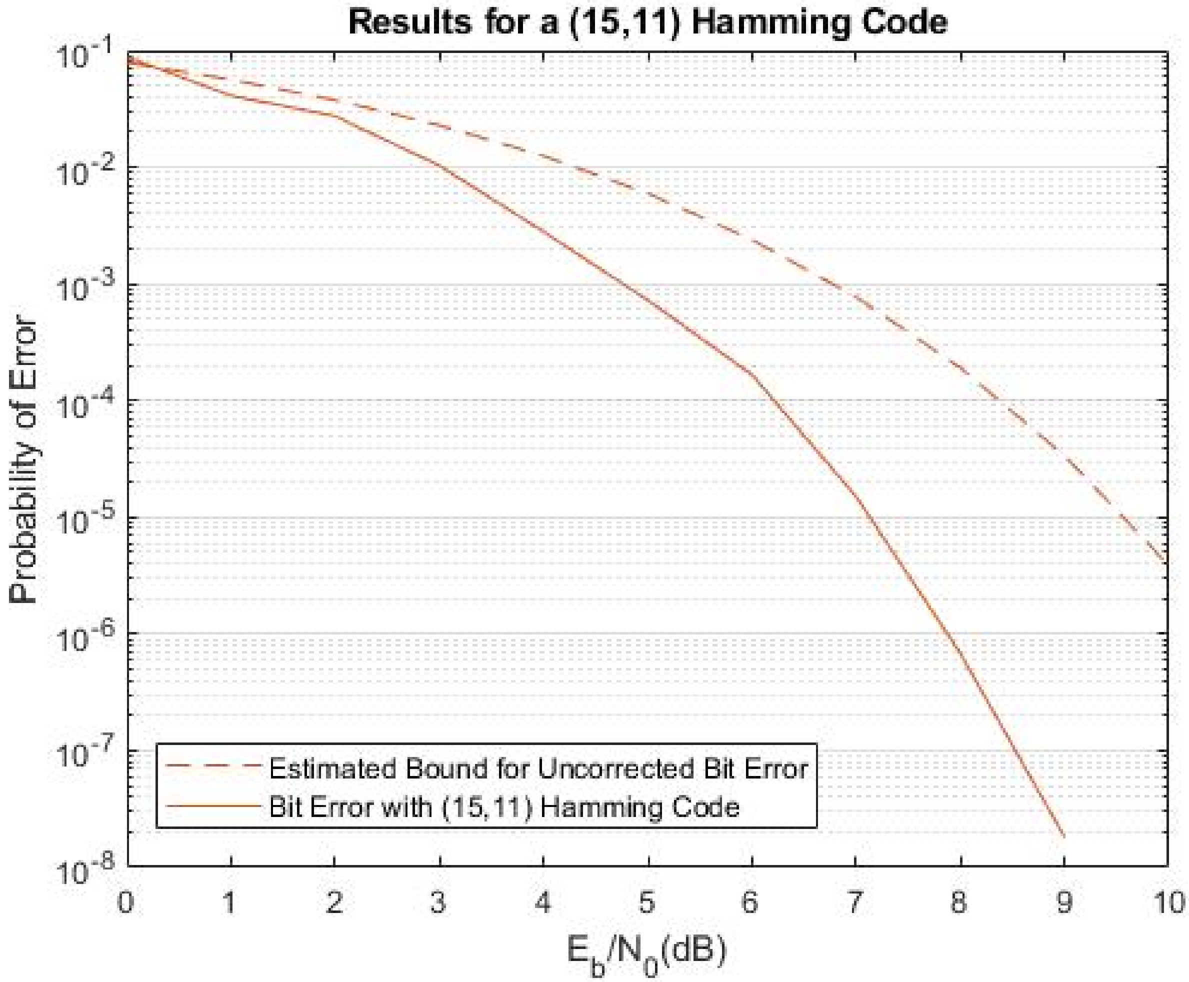
and used the resulting probability of errors to generate the plot given below:



Note the improved performance between the two which demonstrates a coding gain of about 2.8 dB.

2. *Prompt:* Repeat this for a (15,11) Hamming code. (See page 112 and Equations (3.6) and (3.4).)

   Response: The performance of the (15,11) code was much better than the uncoded version although not quite as robust as its (7,4) counterpart with a 2.6 dB coding gain (approximately) as shown below:


Results for a (15,11) Hamming Code
Probability of Error
E_b/N_0(dB)
Estimated Bound for Uncorrected Bit Error
Bit Error with (15,11) Hamming Code
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
0
1
2
3
4
5
6
7
8
9
10