EDC 310

Digital Communications

Practical Assignment 1 Report: BPSK and QPSK modulation simulations

Name and Surname	Student Number	Signature	% Contribution
Mohamed Ameen Omar	16055323		100

By signing this assignment we confirm that we have read and are aware of the University of Pretoria's policy on academic dishonesty and plagiarism and we declare that the work submitted in this assignment is our own as delimited by the mentioned policies. We explicitly declare that no parts of this assignment have been copied from current or previous students' work or any other sources (including the internet), whether copyrighted or not. We understand that we will be subjected to disciplinary actions should it be found that the work we submit here does not comply with the said policies.

Wednesday 15^{th} August, 2018

DECLARATION OF ORIGINALITY

UNIVERSITY OF PRETORIA

The Department of .... places great emphasis upon integrity and ethical conduct in the preparation of all written work submitted for academic evaluation.

While academic staff teach you about referencing techniques and how to avoid plagiarism, you too have a responsibility in this regard. If you are at any stage uncertain as to what is required, you should speak to your lecturer before any written work is submitted.

You are guilty of plagiarism if you copy something from another author's work (eg a book, an article or a website) without acknowledging the source and pass it off as your own. In effect you are stealing something that belongs to someone else. This is not only the case when you copy work word-for-word (verbatim), but also when you submit someone else's work in a slightly altered form (paraphrase) or use a line of argument without acknowledging it. You are not allowed to use work previously produced by another student. You are also not allowed to let anybody copy your work with the intention of passing if off as his/her work.

Students who commit plagiarism will not be given any credit for plagiarised work. The matter may also be referred to the Disciplinary Committee (Students) for a ruling. Plagiarism is regarded as a serious contravention of the University's rules and can lead to expulsion from the University.

The declaration which follows must accompany all written work submitted while you are a student of the Department of .... No written work will be accepted unless the declaration has been completed and attached.

Full names of student: ....

Student number: ....

Topic of work: ....

Declaration

1. I understand what plagiarism is and am aware of the University's policy in this regard.

2. I declare that this .... (eg essay, report, project, assignment, dissertation, thesis, etc) is my own original work. Where other people's work has been used (either from a printed source, Internet or any other source), this has been properly acknowledged and referenced in accordance with departmental requirements.

3. I have not used work previously produced by another student or any other person to hand in as my own.

4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

SIGNATURE ....

Table of Contents

1 Introduction

When transmitting digital signals from a transmitter to a receiver, the digital signal needs to be modulated to an analogue signal to allow the signal to travel from transmitter to receiver. Modulation is the process of adjusting the properties of the *carrier* signal, such as the amplitude and phase, thus encoding the symbols to be sent in the carrier signal. This *"modulated"* signal is then sent or transmitted to the receiving antenna.

There are many modulation techniques available to send digital messages through a channel to a receiver, one of which is called the Phase Shift Keying (PSK) digital modulation method.

The Phase Shift Keying (PSK) digital modulation method adjusts or modulates the phase of the carrier signal used to transmit the information over the channel. Binary Phase Shift Keying (BPSK) and Quadrature Phase Shift Keying (QPSK) are two types of PSK modulation methods. BPSK maps a single bit to two possible symbols, namely, 1 or -1. While QPSK maps two bits to four possible symbols in the complex plane, namely, 1, j, -1, -j. The constellation maps are used to translate the bits to symbols and the symbols back to bits ,i.e. they are used for modulation and demodulation of a digital signal. The constellation maps for BPSK and QPSK are given in figures 1 and 2 and respectively.

During the first practical assignment for EDC 310, students were tasked with creating a simulation platform for the Binary Phase Shift Keying (BPSK) and Quadrature Phase Shift Keying (QPSK) digital communication systems. The system would generate a variable number of bits at the receiving end, modulate the signal using the aforementioned PSK modulation techniques, *transmit* the modulated signal to the receiver through a Additive White Gaussian Noise (AWGN) channel, use the optimal detection algorithm at the receiving end to eliminate noise added to the signal by the channel, demodulate the signal and determine the Bit Error Rate for the channel.

In order to create this simulation platform,r students implemented their own uniform random number generator using the Wichmann-Hill algorithm (Question 1) and their own normally distributed random number generator using the Marsagli-Bray algorithm (Question 2). The uniform random number generator was used to generate the random initial bits to be sent over the channel and the normally distributed random number generator was used to add "noise" to the modulated signal sent, thus distorting the signal and mimicking an Additive White Gaussian Noise (AWGN) channel.

The signal to noise ratio (SNR) of the channel was adjudged between -4 and 8 and the resulting Bit Error Rate (BER) of the channel and modulation technique was investigated. This practical allowed students to investigate the performance of the BPSK and QPSK digital modulation techniques through statistical simulation and first-principals implementation.

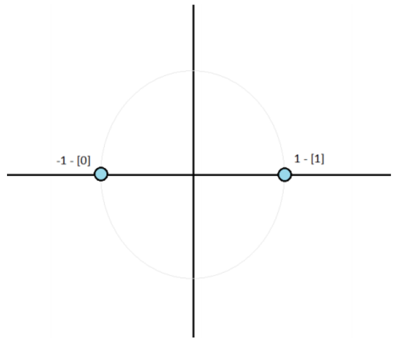


Figure 1. Constellation Map for BPSK modulation technique

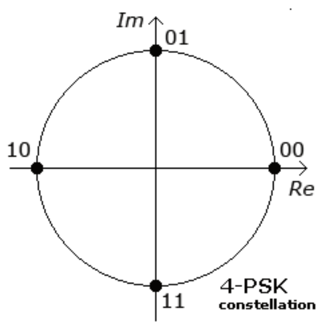


Figure 2. Constellation Map for QPSK modulation technique

2 Question 1

2.1 Introduction and Analysis

Question 1 required students to implement a pseudo-random uniform random number generator, to generate numbers in the range (0,1) using the Wichmann-Hill algorithm. The theoretical mean and variance for the Wichmann-Hill pseudo-random number generator is approximately 0.5 and 0.0833 respectively.

2.2 Design and Implementation

A class called *randomNumber* was created to contain the random number generators for question 1 and question 2. The class accepted three parameters, all of which were the "seeds" used for both the random number generators. By default, if no parameters are passed into the constructor, the class would make use of the *time.time()* library function of the *datetime* Python library. The Uniform random number generator for question 1 was implemented in the *WHill()* class member function. The function implemented the pseudo-code found in figure 3, references [1] and [4].

The function generates two random numbers during execution, however only one random number is returned. The second random number is stored in a class variable called *computed* to simplify the function use in other applications and not waste unnecessary computing time, re-computing or discarding valid random numbers. The function returns the modulo of the result of the algorithm and 1, thus ensuring that the number generates is between 0 and 1.

The functionality that meets the requirements for Question 1 is implemented in the function called *question1()*. The function accepted three arguments, the desired sample size, the bin size used during plotting and boolean to indicate whether or not to save the final output. Once the function executes with the desired parameters, the final plot is outputted to the screen, the plot contains the PDF for the uniform random number generated as well as the theoretical uniform PDF for comparison. The built in Python uniform random number generator (*numpy.random.uniform()*) was also used for comparison with it's PDF also being plotted. The standard deviation, variance and mean of the random numbers generated by both random number generators is also printed to the screen during execution. The *matplotlib* Python library was used to plot the final PDFs.

```
[r, s1, s2, s3] = function(s1, s2, s3)
    % s1, s2, s3 should be random from 1 to 30,000. Use clock if available
    s1 = mod ( 171 * s1, 30269 )
    s2 = mod ( 172 * s2, 30307 )
    s3 = mod ( 170 * s3, 30323 )

    r = mod ( s1/30269 + s2/30307 + s3/30323, 1 )
```

Figure 3. Pseudocode for the Wichmann-Hill pseudo-random number generator [4] [1]

2.3 Results

The Python script was executed multiple times, with varying sample size. As the sample size increased, it was observed that the probability distribution of the numbers generated began to perfectly fit the theoretical Uniform distribution. Below are the results obtained with a sample size of 10000000 samples and a bin size for the sample of 200 bins.

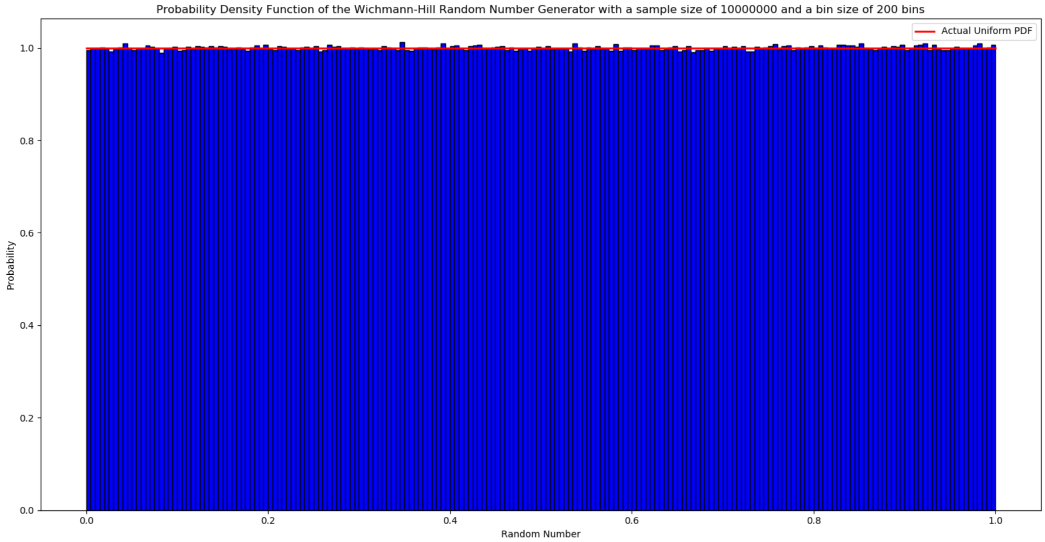


Figure 4. PDF for the Wichmann-Hill pseudo-random uniform random number generator. (Sample size of 10000000)

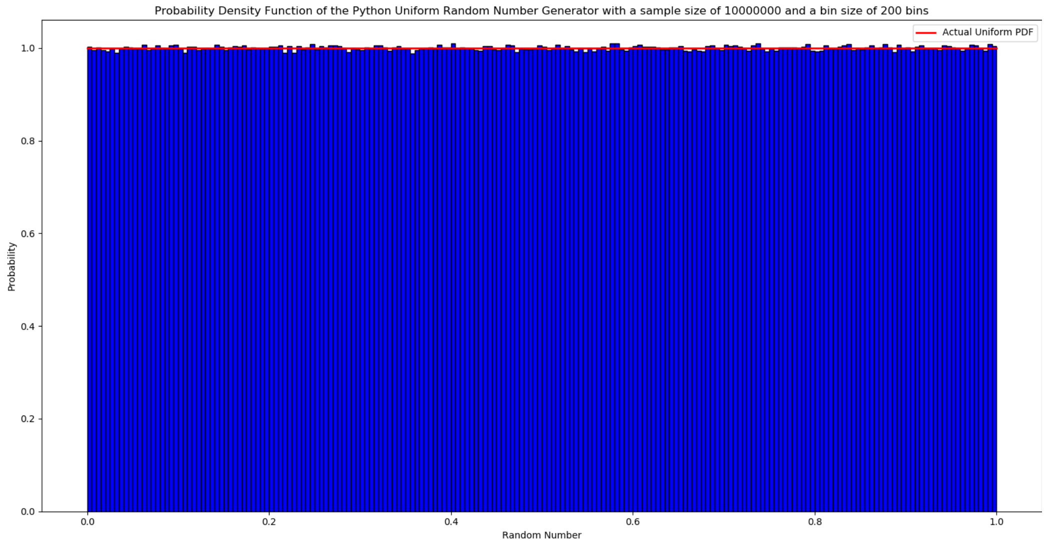
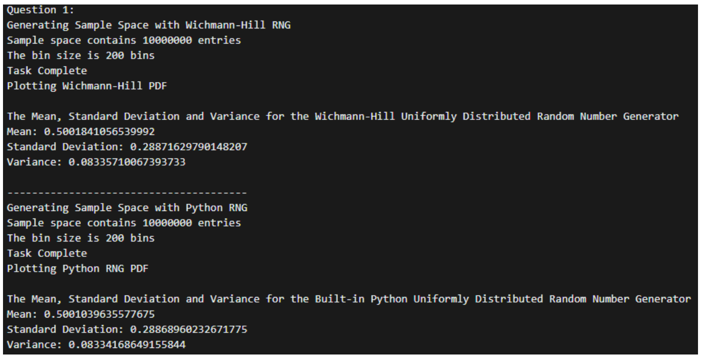


Figure 5. PDF for the Python built-in pseudo-random uniform random number generator. (Sample size of 10000000)



```
Question 1:
Generating Sample Space with Wichmann-Hill RNG
Sample space contains 10000000 entries
The bin size is 200 bins
Task Complete
Plotting Wichmann-Hill PDF

The Mean, Standard Deviation and Variance for the Wichmann-Hill Uniformly Distributed Random Number Generator
Mean: 0.5001841056539992
Standard Deviation: 0.28871629790148207
Variance: 0.08335710067393733

--------------------------------------
Generating Sample Space with Python RNG
Sample space contains 10000000 entries
The bin size is 200 bins
Task Complete
Plotting Python RNG PDF

The Mean, Standard Deviation and Variance for the Built-in Python Uniformly Distributed Random Number Generator
Mean: 0.5001039635577675
Standard Deviation: 0.28868960232671775
Variance: 0.08334168649155844
```

Figure 6. Question 1 console output

2.4 Discussion

The PDF for the Wichmann-Hill random number generator implemented is shown in figure 4. As we can see from the figure the PDF is almost uniform with extremely slight deviations ooccuring at the maximum. There are no outliers in the PDF and the output meets the requirements of a Uniform random number generator. Comparing the Wichmann-Hill random number generator and the *numpy* uniform random number generator PDFs in figures 4 and 5 we can see that they are identical, further emphasizing the accuracy of the Wichmann-Hill random number generator implemented. From figure 6, the console output of the script, we can see that the mean and standard deviation of the Wichmann-Hill implementation of 0.5 and 0.288 was achieved which is in accordance with the theoretical uniform distribution parameters. The function thus does produce pseudo random numbers in the range 0 to 1, that follows a Uniform Probability Distribution.

3 Question 2

3.1 Introduction and Analysis

Question 2 required students to implement a pseudo-random number generator in Python that would follow the Gaussian probability distribution using the Marsaglia-Bray algorithm. The algorithm needed to generate random numbers that followed a Gaussian probability distribution with a mean (μ) of 0 and a standard deviation (σ) of 1.

3.2 Design and Implementation

The Marsaglia-Bray random number generator was implemented according to the polar method implementation in figure 7, references [2] and [3]. The random number generator was implemented in the *gaussian()* member function of the *randomNumber* class implemented in the question_1.py Python script.

The function takes in two parameters, the mean and the standard deviation, both of which can be adjusted for the desired output. Question 2 required a mean of 0 and standard deviation of 1. The *time.time()* library function of the *time* Python library was used to generate the seed values for the number generator. The Wichmann-Hill Uniform number generator implemented in Question 1 was used to generate the random numbers required in the Gaussian random number generator. The *matplotlib* Python library was used to plot the final PDF. The *scipy.stats.norm* library function was used to plot the ideal Gaussian Normal Distribution on the same plot, thus enabling comparison between the achieved distribution and the theoretical result.

```
private static double spare;
private static boolean isSpareReady = false;

public static synchronized double getGaussian(double mean, double stdDev) {
    if (isSpareReady) {
        isSpareReady = false;
        return spare * stdDev + mean;
    } else {
        double u, v, s;
        do {
            u = Math.random() * 2 - 1;
            v = Math.random() * 2 - 1;
            s = u * u + v * v;
        } while (s >= 1 || s == 0);
        double mul = Math.sqrt(-2.0 * Math.log(s) / s);
        spare = v * mul;
        isSpareReady = true;
        return mean + stdDev * u * mul;
    }
}
```

Figure 7. Java implementation of the Marsaglia-Bray polar method random number generator [2] [3]

3.3 Results

The Python script was executed multiple times, with varying sample size. As the sample size increased, it was observed that the probability distribution of the numbers generated began to perfectly fit the theoretical Gaussian normal distribution. Below are the results obtained with a sample size of 10000000 samples and a bin size for the sample of 250 bins.

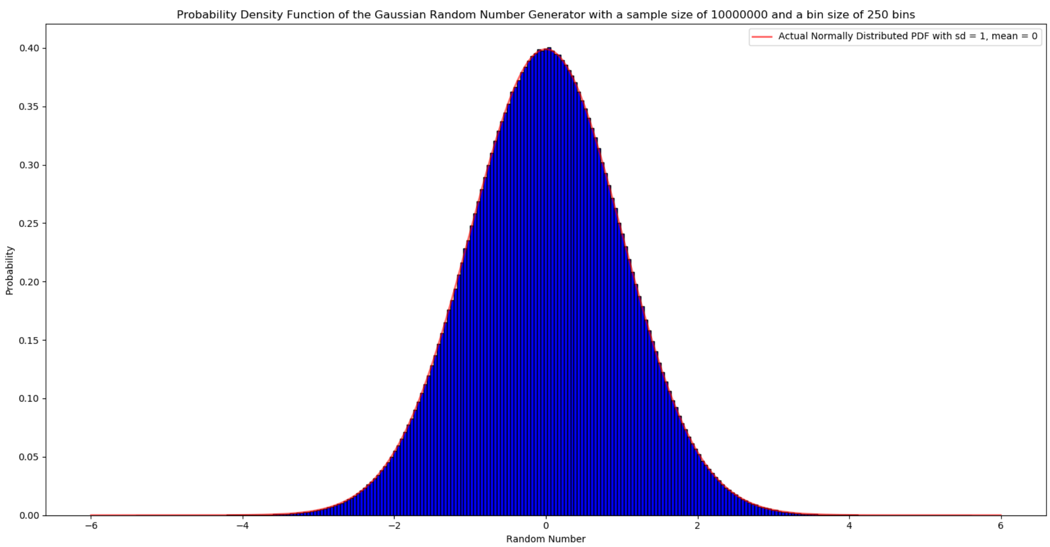


Figure 8. PDF for the Marsaglia-Bray pseudo-random Gaussian random number generator. (Sample size of 10000000)

```
Question 2
Generating Sample Space with Gaussian Random Number Generator
Sample space contains 10000000 entries
The bin size is 250 bins
Done

Plotting the Gaussian Random Number Generator PDF
Complete

The Mean, Standard Deviation and Variance for the Gaussian Normally Distirbitued Random Number Generator
Mean = 6.768549144052215e-06
Standard Deviation = 0.9999772740637718
Variance = 0.9999545486440118
```

Figure 9. Question 2 console output

3.4 Discussion

As we can see from figure 8, the numbers generated by the implemented Gaussian random number generator, do follow the theoretical Gaussian Normal distribution. The mean value achieved of approximately 0.00000676 and standard deviation of 0.9999 is extremely close to the theoretical mean standard deviation of 0 and 1 respectively. The difference in the practical results and the expected theoretical statistics is essentially non-existent. I have thus successfully implemented the Marsaglia-Bray Gaussian random number generated as required.

4 Question 3

4.1 Introduction and Analysis

Question 3 required students to evaluate the performance of BPSK and QPSK modulation through a AWGN channel, implemented from first principals using the Python programming language. This question required students to do the following:

1. Use the uniform random number generator implemented in Question 1 to generate a sample of bits.
2. Implement both the QPSK and BPSK modulation techniques and map the randomly generated bits from (1) to symbols according to the respective constellation map given in figures 1 and 2.
3. Simulate the transmission of the bits over a AWGN channel by adding noise to the modulated bits. The noise was to be added using the Gaussian random number generator implemented in Question 2.
4. Using the optimal detection algorithm, detect the symbols received.
5. Demodulate the symbols received by converting the symbols back to bits.
6. Compare the originally generated bits (sent bits) to the received bits and count the number of errors.
7. Calculate the Bit Error Rate given as $BER = \frac{NumberofErrors}{Numberofbits}$
8. Plot the BER as a function of the signal to noise ratio (SNR) of the AWGN channel varied on the range [4,8] dB.

4.2 Design and Implementation

In order to implement the simulation platform required, two classes were created. *BPSKmodulationSimulation* and *QPSKmodulationSimulation*, wherein the BPSK modulation performance and the QPSK modulation performance was simulated respectively. Each class contained a number of member functions to aid the simulation. Both classes required a single parameter into the constructor which was the number of bits to *transmit* and receive over the AWGN channel.

The *simulate()* member function for each class executes the simulation for the respective modulation technique. The functioning of the *simulate()* function is as follows:
It generates a random number of bits according the parameter passed into the class constructor, over a loop that varies the **SNR** of the AWGN channel on the range [4,8] dB, the bits are converted to symbols via the *BpskModulate()* and the *QpskModulate()* functions, according the respective constellation maps given in section 1 of this document.

Noise was added to the modulated bits according to the equation $r_k = s_k + n_k$, thus simulating the transmission of the signal over a AWGN channel. The QPSK version adds "noise" in the form of a real and imaginary number, while the BPSK version only adds "noise" to the symbols in the form of a real number. The Gaussian random number generator implemented in Question 2 was used to generate the noise added, with the standard deviation calculated as $\frac{1}{\sqrt{10^{\frac{SNR}{10}} 2 f_{bit}}}$. An f_{bit} value of 1 was used for BPSK and 2 for QPSK.

The bits were "received" and detected by implementing the optimal detection algorithm in the function *detectSignal()*. The symbols were then trasnalted back into bits via the *BpskDemodulate()* and *QpskDemodulate()* functions, both converting the symbols to bits according to the constellation maps given in section 1 of this document.

The *"received"* bits were then compared to the *"sent"* bits and the number of errors and bit error ratio was calculated. At the end of the loop, the **BER** was plotted against the corresponding **SNR** after which the list of **BER's** was returned from function. The *numpy* and *maplotlib* library functions were used throughout the implementation to pefrom the mathematical computations and plot the final output.

A function called *plotToCompare()* was implemented to plot the BER vs SNR plots of both the simulations by passed in the BER list of each simulation as function parameters.

4.3 Results

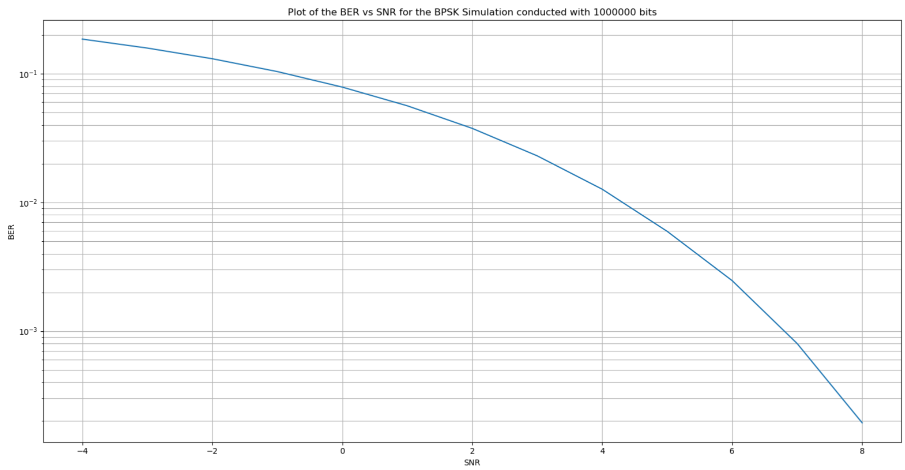


Figure 10. Plot illustrating the performance of the BPSK simulation

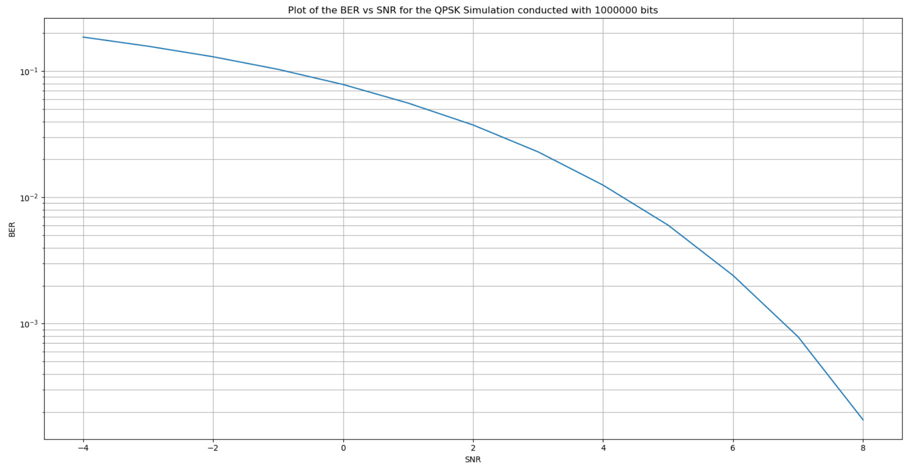


Figure 11. Plot illustrating the performance of the QPSK simulation

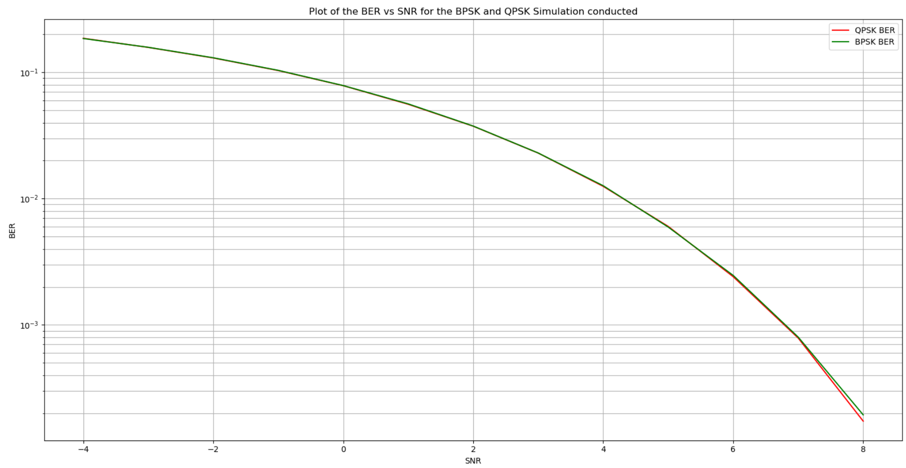


Figure 12. Plot illustrating the performance of both the BPSK and QPSK simulations

4.4 Discussion

An increase in the **SNR** of the channel, results in less noise distorting the signal transmitted. This is clearly visible from the outputs of the simulation in figures 10, 11 and 12. Both the QPSK and BPSK follow this trend with the **BER** tending to zero as the **SNR** is increased. From figure 12 we can see that the performance of the BPSK and QPSK modulation techniques over the channel are very similar. They both show alternating intervals of steep and gradual progression towards zero as the SNR increases. However, we can see that QPSK seems to be the better choice, by a very small margin, separating itself in the fact that it settles to lower **BER** compared to BPSK at a **SNR** of 8. We can see that both modulation techniques are extremely reliable with very low **BERs** even at a **SNR** of as low as -4 dB. A higher **SNR** means more power is needed to transmit the signal over the channel, however, from the results obtained, one can see that whether one uses BPSK or QPSK to modulate the signal, the **BER** seems to be extremely low and thus, in practice we may not need to use as much power since the **SNR** needed, does not need to be very high for the signal that is transmitted to contain few errors.

5 Conclusion

Question 1 and Question 2 provided the building blocks needed to implement the desired simulation platform. They were tested and verified to be accurate enough to use in the simulation platform implemented in Question 3.

The Wichmann-Hill algorithm implemented in Question 1 was successful in generating random numbers in the range (0,1) that followed a Uniform probability distribution. The Marsaglia-Bray algorithm implemented in question 2 was also successful in generating the random numbers required. The random number generator, generated random numbers that followed a Gaussian Normal Probability Distribution, with parameters reflecting those passed in (the mean and standard deviation). We observed that as the number of samples increased for Question 1 and Question 2, the PDFs generated more closely reflected the theoretical Probability Density Functions. Comparisons were done for both questions between the theoretical result and the practical achieved result. These comparisons confirmed that our implementation was extremely close to the theoretical result.

From the BPSK and QPSK simulations, we can see that the higher the SNR the lower the BER. This conforms to what is expected, since the higher the SNR of the channel, the more power is used and thus the greater the amplitude of the actual signal as compared to the amplitude of the noise added to the signal within the channel. BPSK and QPSK perform identically over the AWGN channel, with QPSK settling to a lower BER as compared to the BPSK simulation. Thus, QPSK would be more ideal to use, as it can send twice as many bits per second, due to its modulation constellation map and it results in a very similar BER for the same SNR as compared with BPSK.

The building blocks were successfully used to successfully implemented the simulation platform. This practical has given students tremendous insight into the process undergone to module, transmit, demodulate and translate bits sent over a digital channel.

6 References

[1] B. Wichmann and D. Hill, "Building a random number generator," *Byte*, pp. 127–128, 1987.

[2] G. Marsaglia and T. Bray, "a convenient method for generating normal variables," *SIAM Rev*, vol. 6, pp. 260–264, 1964.

[3] "Marsaglia polar method." [Online]. Available: https://en.wikipedia.org/wiki/Marsaglia_polar_method

[4] "Wichmannhill." [Online]. Available: https://en.wikipedia.org/wiki/Wichmann%E2%80%93Hill

A Appendix A: How to operate Python source code

A.1 Dependencies

The following Python packages are required to execute the scripts:

- numpy
- matplotlib
- time
- datetime
- scipy (scipy.stats)

All the packages can be installed using *pip* in the Linux terminal or *conda* if Anaconda is installed on a Windows machine.

The scripts question_2.py and question_3.py both require that the question_1.py source file is in the same directory as they are in order to execute.

A.2 Execution of the scripts

A.2.1 question_1.py

To run the script, scroll to the bottom of the script file and uncomment line 165. The parameters may be adjusted at will by the user, these parameters include the sample size, bin size and a boolean indicating whether or not to save the output plot.

The function question1() (executed on line 165) executes the script meeting the requirements for Question 1.

The sample size parameter indicates the number of samples to generate for the output and bin size indicates the size of the groups into which the samples are grouped for the final plot. The entire script is adequately commented explaining any variable or block of code that is not self-explanatory. Once the script executes, the statistical parameters will be printed to the console and the final PDF will be shown.

Once the script has been executed please ensure that line 165 is commented again before proceeding to evaluate the next two scripts.

A.2.2 question_2.py

To run the script, ensure that all dependencies have been met and that line 165 in the question_1.py script is commented.

Line 74 contains the line of code that calls the question2() function. This function executes the script to meet the requirements for Question 2. The function parameters are the same as that for the question1() function in the question_1.py script.

The source file is adequately commented to explain any variable or block of code that may not be self-explanatory.

A.2.3 question_3.py

To run the script, ensure that all dependencies have been met and that line 165 in the question_1.py script is commented.

The script contains three lines of interest, line 381, line 384 and line 387. Line 381 executes the BPSK simulation and outputs the performance plot as required for Question 3. Line 384 executes the QPSK simulation and outputs the performance plot as required of Question 3. Line 387 plots the performance of both the BPSK and QPSK simulation on a single plot for comparison. Line 387 requires that line 381 and line 384 are NOT commented out and execute to completion.

The only parameter that may be changed for final output of the simulation is the number of bits generated during the simulation given by the *numBits* variable on line 376. The entire script is adequately commented explaining any variable or block of code that is not self-explanatory.

B Python source code

```
1 #Mohamed Ameen Omar
2 #16055323
3 #####################################
4 ###        EDC 310 Practical 1      ###
5 ###              2018               ###
6 ###           Question 1            ###
7 #####################################
8 
9 import numpy as np
10 import matplotlib.pyplot as plt
11 import time
12 import datetime
13 from scipy.stats import norm
14 
15 ############ Random Number Class #####################
16 # The random number class contains the two random number generators
```

```python
class randomNumber:
    #Constructor
    def __init__(self, seed1 = time.time(), seed2 = time.time()-10, seed3 = time.time()+10):
        self.seed1 = seed1
        self.seed2 = seed2
        self.seed3 = seed3
        self.computed = None

    #Uniform Distribution Wichmann-Hill algorithm
    def WHill(self):
        # Seed values must be greater than zero

        self.seed1 = 171 * (self.seed1 % 177) - (2*(self.seed1/177))
        if self.seed1 < 0:
            self.seed1 = self.seed1 + 30269

        self.seed2 = 172 * (self.seed2 % 176) - (35*(self.seed2/176))
        if self.seed2 < 0:
            self.seed2 = self.seed2 + 30307

        self.seed3 = 170 * (self.seed3 % 178) - (63*(self.seed2/178))
        if self.seed3 < 0:
            self.seed3 = self.seed3 + 30323

        temp = float(self.seed1/30269) + float(self.seed2/30307) + float(self.seed3/30323)
        # So that the output is between 0 and 1
        return (temp%1)

    # Normal Distribituion random number generator.
    # Only used in Question 2 and Question 3.
    def gaussian(self, mean = 0, stdDeviation = 1):
        if(self.computed is not None):
            returnVal = self.computed
            self.computed = None
            return returnVal
        else:
            squaredSum = -1.0
            temp1 = 0.0
            temp2 = 0.0
            # find two numbers such that their squared sum falls within the
            # boundaries of the square.
            while( (squaredSum >= 1) or squaredSum == -1.0 ):
                temp1 = (2*self.WHill())-1
                temp2 = (2*self.WHill())-1
                squaredSum = (temp1**2) + (temp2 **2)

            mul = np.sqrt(-2.0*np.log(squaredSum)/squaredSum)

            #store the second point to avoid wasting computation time
            self.computed = mean + (stdDeviation * temp1 * mul)
            return (mean + (stdDeviation*temp2*mul))

################## End of class ###########################
```

```
# function to plot the PDF for the Uniformly distributed random number generator
def plotUniformPDF(sample = None, binSize = 200, save = True, iswHill = True):
    if(sample is None):
        print("Error, sample to plot not provided")
        return

    title = ("Probability Density Function of the ")
    # if the sample passed in was generated from the Wichmann-Hill algorithm
    if(iswHill is True):
        title = title + "Wichmann-Hill Random Number Generator"
    # if the sample passed in was generated from the python uniform random number generator
    else:
        title = title + "Python Uniform Random Number Generator"
    fileName = title
    title = title + " with a sample size of " + str(len(sample)) + " and a bin size of " + str(binSize) + " bins"
    fig = plt.figure()
    plt.hist(sample,color = "Blue", edgecolor = "black", bins = binSize, density = True)
    plt.xlabel("Random Number")
    plt.ylabel("Probability")
    plt.title(title)
    #Plot a real Uniform PDF for comparison
    plt.plot([0.0, 1.0], [1,1],'r-', lw=2, label='Actual Uniform PDF')
    plt.legend(loc='best')
    plt.show()

    #to save the plot
    if(save is True):
        fileName = fileName + ".png"
        fig.savefig(fileName , dpi=fig.dpi)

# function to print the relevant statistics for the sample passed in
# The mean, standard deviation and variance of the sample is computed
# and displayed to the screen
def printStats(sample = None, isWHill = True):
    if(sample is None):
        print("Error, sample not provided, no statistics to print")
        return
    message = ("The Mean, Standard Deviation and Variance for the ")
    if(isWHill is True):
        message = message + "Wichmann-Hill Uniformly Distributed Random Number Generator"
    else:
        message = message + "Built-in Python Uniformly Distributed Random Number Generator"
    print(message)
    print("Mean: " + str(np.mean(sample)))
    print("Standard Deviation: " + str(np.std(sample)))
    print("Variance: " + str(np.var(sample)))

```

```python

def question1(sampleSize = 10000000, binSize = 200, save = True):
    #input parameter validation
    #all paramters must have a postive value
    if(sampleSize < 0):
        sampleSize = 10000000
    if(binSize < 0):
        binSize = 200
    #create a randomGenerator object to get Uniform distribution random numbers
    # For consistent results, pass seed paramters into the constructor
    randomGenerator = randomNumber() #create a random number object to generate uniform random number
    sampleSpace = []

    print("Question 1:")

    #Generate Sample Space for Wichmann-Hill
    print("Generating Sample Space with Wichmann-Hill RNG")
    print("Sample space contains", sampleSize, "entries")
    print("The bin size is", binSize, "bins")
    for x in range(0, sampleSize):
        sampleSpace.append(randomGenerator.WHill())
    print("Task Complete")

    print("Plotting Wichmann-Hill PDF")
    plotUniformPDF(sampleSpace, binSize, save, True)
    print()
    printStats(sampleSpace, True)
    print()

    print("----------------------------------------")
    print("Generating Sample Space with Python RNG")
    print("Sample space contains", sampleSize, "entries")
    print("The bin size is", binSize, "bins")
    sampleSpace = np.random.uniform(size = sampleSize)
    print("Task Complete")

    print("Plotting Python RNG PDF")
    plotUniformPDF(sampleSpace, binSize, save, False)
    print()
    printStats(sampleSpace, False)

# adjust parameters at will
# Sample size = First Parameter
# Bin size = Second Parameter
# boolean to save the plots = Third Paramter

# Uncomment next line to run
#question1(1000000,250,save = False)
```

Listing 1. Python Source code for question 1 (Uniform Random Number Generator).

```python
#Mohamed Ameen Omar
#16055323
```

```python
#####################################
###      EDC 310 Practical 1      ###
###             2018              ###
###          Question 2           ###
#####################################

import numpy as np
import matplotlib.pyplot as plt
import time
import datetime
from scipy.stats import norm
from question_1 import randomNumber


# Ensure that the source file question_1.py is in the current dorectory
# before running the script
# Refer to question_1.py for the parameters of the random number
    generator

def question2(sampleSize=10000000, binSize=200, save=True):
    # input parameter validation
    # all paramters must have a postive value
    if(sampleSize < 0):
        print("Sample size set to 10000000 entries")
        sampleSize = 10000000
    if(binSize < 0):
        print("Bin Size set to 200 bins")
        binSize = 200
    # create a randomGenerator object to get random numbers
    # For consistent results, pass seed paramters into the constructor
    randomGenerator = randomNumber()
    # list to store the values generated
    sampleSpace = []

    #Generate Sample Space
    print("Question 2")
    print("Generating Sample Space with Gaussian Random Number Generator"
    )
    print("Sample space contains",sampleSize, "entries")
    print("The bin size is",binSize, "bins")
    for x in range(0, sampleSize):
        # adjust gaussian function paramters for differing outputs
        sampleSpace.append(randomGenerator.gaussian())
    print("Done")
    print()

    # Plot the PDF
    title = ("Probability Density Function of the Gaussian Random Number
    Generator")
    title = title + " with a sample size of " + str(sampleSize) + " and a
     bin size of " + str(binSize) + " bins"
    print("Plotting the Gaussian Random Number Generator PDF")
    plt.hist(sampleSpace, color="Blue", edgecolor="black", bins=binSize,
    density=True)
    plt.ylabel("Probability")
    plt.xlabel("Random Number")
```

```
    x = np.linspace(norm.ppf(0.000000001), norm.ppf(0.999999999), binSize)
    plt.plot(x, norm.pdf(x), 'r-', lw=2, alpha=0.6, label='Actual Normally Distributed PDF with sd = 1, mean = 0')
    plt.legend(loc='best')
    plt.title(title)
    plt.show()
    print("Complete")
    print()

    # The mean, standard deviation and variance of the sample is computed
    # and displayed to the screen
    print("The Mean, Standard Deviation and Variance for the Gaussian Normally Distirbitued Random Number Generator")
    print("Mean =",np.mean(sampleSpace))
    print("Standard Deviation =", np.std(sampleSpace))
    print("Variance =",np.var(sampleSpace))

#adjust parameters at will
# Sample size = First Parameter
# Bin size = Second Parameter
# boolean to save the plots in the current directory= Third Paramter
# uncomment next line to run
#question2(10000000, 200, save = False)
```

Listing 2. Python Source code for question 2 (Normally Distributed Random Number Generator).

```
#Mohamed Ameen Omar
#16055323
####################################
###      EDC 310 Practical 1     ###
###            2018              ###
###         Question 3           ###
####################################

import numpy as np
import matplotlib.pyplot as plt
import time
import datetime
from scipy.stats import norm
from question_1 import randomNumber

# Ensure that the source file question_1.py is in the current dorectory
# before running the script

# Class used for QPSK simulation
class QPSKmodulationSimulation:
    def __init__(self,numBits):
        self.numBits = numBits
        self.numberGenerator = randomNumber()

    def getStdDev(self, SNR):
        return (1/np.sqrt(10**(SNR/10)*2*2))

    #add noise for the real and imaginary parts of the symbol
```

```
    def addNoise(self,SNR,sentBits):
        recieved = []
        standardDeviation = self.getStdDev(SNR)
        for x in range(0,len(sentBits)):
            real = self.numberGenerator.gaussian(stdDeviation=
standardDeviation)
            imag = self.numberGenerator.gaussian(stdDeviation=
standardDeviation) * 1j
            recieved.append(sentBits[x]+real+imag)

        return recieved

    # generate random number using uniform random number generator
    # round the output to a 1 or 0 for a bit
    def generateBits(self):
        print("Generating",self.numBits, "random binary digits")
        toSend = []
        for x in range(0,self.numBits):
            toSend.append(int(np.round(self.numberGenerator.WHill())))
        return(toSend)

    # map an array of bits to a symbols
    # according to the BPSK constellation map
    def QpskModulate(self, original):
        print("Mapping bits to symbols using QPSK modulation")
        mappedMessage = []
        temp = list(map(str,original))

        for x in range(0, len(temp),2):
            toSymbol = temp[x] + temp[x+1]
            mappedMessage.append(self.bitToSymbol(toSymbol))
        return mappedMessage

    # return the Symbol for the
    # the bit passed in
    def bitToSymbol(self, toMap):
        if(toMap == "00"):
            return 1
        elif(toMap == "01"):
            return 1j
        elif(toMap == "11"):
            return -1
        elif(toMap == "10"):
            return -1j

    # demodulate the detected symbols
    # for QPSK
    def QpskDemodulate(self, modulated):
        print("Mapping symbols to bits for QPSK demodulation")
        demod = []
        for x in range(0, len(modulated)):
            temp = self.symbolToBit(modulated[x])
            demod.append(int(temp[0]))
            demod.append(int(temp[1]))
        return demod

```

```
    # return the bits for the
    # the symbol passed in
    def symbolToBit(self,symbol):
        if(symbol == 1):
            return "00"
        elif(symbol == 1j):
            return "01"
        elif(symbol == -1):
            return "11"
        elif(symbol == -1j):
            return "10"

    # given the SNR and recieved signal
    # use optimum detection algorithm to
    # determine the signal that was sent
    # return the symbols that were sent
    def detectSignal(self,SNR, recieved):
        stdDev = self.getStdDev(SNR)
        detectedBits = []
        #for every bit recieved
        for x in range(0, len(recieved)):
            probabilities = []  # 1,j,-1,-j
            temp = []
            temp.append(self.getExpProb(recieved[x], 1,stdDev))
            temp.append(self.getExpProb(recieved[x], 1j, stdDev))
            temp.append(self.getExpProb(recieved[x], -1, stdDev))
            temp.append(self.getExpProb(recieved[x], -1j, stdDev))
            probabilities = np.exp(temp)
            beta = self.getBeta(probabilities)
            for prob in range(0,len(probabilities)):
                probabilities[prob] = probabilities[prob] * beta
            ind = np.argmax(probabilities)
            detectedBits.append(self.getSymbol(ind))

        return detectedBits

    # get exponent e is raised to for
    # the symbol recieved and the symbol we think it is
    # need standard deviation for the channel as well
    def getExpProb(self,recieved,actual,stdDev):
        temp = np.abs(recieved-actual)
        temp = (temp**2)*(-1)
        temp = temp/(2*(stdDev**2))
        return temp

    # given an array of conditional probabilities
    # return scaling factor or normalization constant (beta)
    def getBeta(self,probs):
        temp = 0
        for x in range(0,len(probs)):
            temp += probs[x]
        return (1/temp)

    # given the index with the highest probability
    # returnn the QPSK symbol
    def getSymbol(self,index):
```

```
        if(index == 0):
            return 1
        elif(index == 1):
            return 1j
        elif(index == 2):
            return -1
        elif(index == 3):
            return -1j

    #return the number of bit errors for the signal
    def getNumErrors(self,sentBits, recievedBits):
        errors = 0
        for x in range(0,len(sentBits)):
            if(sentBits[x] != recievedBits[x]):
                errors += 1
        return errors

    # simulate the sending and recieving
    # plot BER vs SNR as well
    # returns the BER array
    def simulate(self):
        print("Question 3")
        print("QPSK Simulation with", self.numBits, "bits")
        BER = []
        bits = self.generateBits()
        for SNR in range(-4,9):
            print("SNR set to", SNR)
            #map each bit to a Qpsk symbol
            sentSignal = self.QpskModulate(bits)
            print("Signal Sent")
            #print(sentSignal)
            recievedSignal = self.addNoise(SNR, sentSignal)
            print("Signal Recieved")
            #print(recievedSignal)
            detectedSignal = self.detectSignal(SNR, recievedSignal) # still symbols
            #print(detectedSignal)
            #convert symbols to bits
            detectedBits = self.QpskDemodulate(detectedSignal)
            print("Signal Has been demodulated")
            #print(detectedBits)
            BER.append(self.getNumErrors(bits,detectedBits)/self.numBits)
            print()

        print("The Bit Error rate for each SNR tested is given in the array below:")
        print(BER)
        print()
        print("Plotting the BER vs SNR relationship")
        SNR = np.linspace(-4, 8, 13)
        plt.semilogy(SNR, BER)
        plt.grid(which='both')
        plt.ylabel("BER")
        plt.xlabel("SNR")
        plt.title("Plot of the BER vs SNR for the QPSK Simulation conducted with " + str(self.numBits) + " bits")
```

```
        plt.show()
        print("End of QPSK Simulation")
        return BER
################### End of class ############################

class BPSKmodulationSimulation:
    def __init__(self, numBits):
        self.numBits = numBits
        self.numberGenerator = randomNumber()

    #return standard deviation for SNR with fbit =1
    def getStdDev(self, SNR):
        return (1/np.sqrt(10**(SNR/10)*2*1))

    def addNoise(self, SNR, sentBits):
        recieved = []
        standardDeviation = self.getStdDev(SNR)
        for x in range(0, len(sentBits)):
            real = self.numberGenerator.gaussian(stdDeviation=standardDeviation)
            recieved.append(sentBits[x]+real)
        return recieved

    # generate random number using uniform random number generator
    # round the output to a 1 or 0 for a bit
    def generateBits(self):
        print("Generating", self.numBits, "random binary digits")
        toSend = []
        for x in range(0, self.numBits):
            toSend.append(int(np.round(self.numberGenerator.WHill())))
        return(toSend)

    # map an array of bits to a symbols
    # according to the BPSK constellation map
    def BpskModulate(self, original):
        print("Mapping bits to symbols using BPSK modulation")
        mappedMessage = []
        for x in range(0, len(original)):
            mappedMessage.append(self.bitToSymbol(original[x]))
        return mappedMessage

    # map a single bit to a symbol
    # according to the BPSK constellation map
    def bitToSymbol(self, bit):
        if(bit == 1):
            return 1
        if(bit == 0):
            return -1
        else:
            print("Error occured, bit to map was not zero or one")

    # demodulate the detected symbols
    # for QPSK
    def BpskDemodulate(self, modulated):
        print("Mapping symbols to bits for BPSK demodulation")
        demod = []
```

```
        for x in range(0, len(modulated)):
            demod.append(self.symbolToBit(modulated[x]))
        return demod

    # return the bit for the
    # the symbol passed in
    def symbolToBit(self, symbol):
        if(symbol == 1):
            return (1)

        elif(symbol == -1):
            return 0

    # given the SNR and recieved signal
    # use optimum detection algorithm to
    # determine the signal that was sent
    # return the symbols that were sent
    def detectSignal(self, SNR, recieved):
        stdDev = self.getStdDev(SNR)
        detectedBits = []
        #for every bit recieved
        for x in range(0, len(recieved)):
            probabilities = []  # 1,j,-1,-j
            temp = []
            temp.append(self.getExpProb(recieved[x], 1, stdDev))
            temp.append(self.getExpProb(recieved[x], -1, stdDev))
            probabilities = np.exp(temp)
            beta = self.getBeta(probabilities)
            for prob in range(0, len(probabilities)):
                probabilities[prob] = probabilities[prob] * beta
            ind = np.argmax(probabilities)
            detectedBits.append(self.getSymbol(ind))
        return detectedBits

    # get exponent e is raised to for
    # the symbol recieved and the symbol we think it is
    # need standard deviation for the channel as well
    def getExpProb(self, recieved, actual, stdDev):
        temp = np.abs(recieved-actual)
        temp = (temp**2)*(-1)
        temp = temp/(2*(stdDev**2))
        return temp

    # given an array of conditional probabilities
    # return scaling factor or normalization constant (beta)
    def getBeta(self, probs):
        temp = 0
        for x in range(0, len(probs)):
            temp += probs[x]
        return (1/temp)

    # given the index with the highest probability
    # return the BPSK symbol
    def getSymbol(self, index):
        if(index == 0):
            return 1
```

```python
        elif(index == 1):
            return -1

    #return the number of bit errors for the signal
    def getNumErrors(self, sentBits, recievedBits):
        errors = 0
        for x in range(0, len(sentBits)):
            if(sentBits[x] != recievedBits[x]):
                errors += 1
        return errors

    # simulate the sending and recieving
    # plot BER vs SNR as well
    # returns the BER array
    def simulate(self):
        print("Question 3")
        print("BPSK Simulation with", self.numBits, "bits")
        BER = []
        bits = self.generateBits()
        for SNR in range(-4, 9):
            print("SNR set to", SNR)
            #map each bit to a BPSK symbol
            sentSignal = self.BpskModulate(bits)
            print("Signal Sent")
            #print(sentSignal)
            recievedSignal = self.addNoise(SNR, sentSignal)
            print("Signal Recieved")
            #print(recievedSignal)
            detectedSignal = self.detectSignal(SNR, recievedSignal)  # still symbols
            #print(detectedSignal)
            #convert symbols to bits
            detectedBits = self.BpskDemodulate(detectedSignal)
            print("Signal Has been demodulated")
            #print(detectedBits)
            BER.append(self.getNumErrors(bits, detectedBits)/self.numBits)
            print()

        print("The Bit Error rate for each SNR tested is given in the array below:")
        print(BER)
        print()
        print("Plotting BER vs SNR function for the BPSK Simulation")
        SNR = np.linspace(-4, 8, 13)
        plt.semilogy(SNR, BER)
        plt.grid(which='both')
        plt.ylabel("BER")
        plt.xlabel("SNR")
        plt.title("Plot of the BER vs SNR for the BPSK Simulation conducted with " + str(self.numBits) + " bits")
        plt.show()
        print("End of BPSK Simulation")
        return BER

```

```python
#plots the result of BPSK and QPSK modulation on a single plot
#must pass in BER for both
def plotToCompare(QpskBer, BpskBer):
    print("Plotting both")
    SNR = np.linspace(-4, 8, 13)
    plt.semilogy(SNR, QpskBer, 'r-', label='QPSK BER')
    plt.semilogy(SNR, BpskBer, 'g-', label='BPSK BER')
    plt.grid(which='both')
    plt.ylabel("BER")
    plt.xlabel("SNR")
    plt.title("Plot of the BER vs SNR for the BPSK and QPSK Simulation
   conducted")
    plt.legend(loc='best')
    plt.show()
    print("Complete")

# numBits is the number of bits
# that we would like to use in the simulation of each
# may change the number of bits at will

# create two objects, one for the BPSK simulation
# one for the QPSK Simulation
numBits = 1000000
BpskSimulation = BPSKmodulationSimulation(numBits)
QpskSimulation = QPSKmodulationSimulation(numBits)

#uncomment next line to run the Bpsk Simulation
#bpsk = BpskSimulation.simulate()

#uncomment next line to run the Qpsk Simulation
#qpsk = QpskSimulation.simulate()

#to plot both on the same axis, uncomment next line
#plotToCompare(qpsk,bpsk)
```

Listing 3. Python Source code for question 3 (Simulation).