UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Faculty of Engineering, Built Environment and Information Technology

EDC 310

Digital Communications

Practical Assignment 3: Convolutional Codes

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.

Tuesday 30$^{\text{th}}$ October, 2018

Table of Contents

1 Introduction

During Digital data transmission a bit stream or block of data is sent across a channel from a transmitter to a receiver.During transmission of the signal from the sending antenna to the receiving antenna,the signal undergoes many changes and distortions caused by nature, the medium through which the signal is transmitted and the path followed from the transmitting antenna to the receiving antenna. This distortion causes the signal received at the receiving antenna to no longer mimic the signal sent at the transmitting antenna [1].

Noise in signal processing and Digital Communications is any unwanted signal or frequency added to a signal thus resulting in the original signal being distorted. In particular, additive noise is a concern for Digital signals [1]. Additive noise refers to noise added to a signal regardless of the state of the system and generally arise externally to the system, such as interference from other users of the channel. When such noise and interference occupy the same frequency band as the desired signal, we see distortion in the transmitted signal and we therefore need to process the signal (by using for example a filter) before the signal can be interpreted [2].

Additive White Gaussian noise (AWGN) is a basic noise model used in Information theory and Digital communications to mimic the effect of many random processes that occur in nature and as such is used in simulations to mimic the effects of noise or additive noise to a digital signal transmitted.

Many attempts to minimise distortion and ensure the correct data has been received by the receiver, have been investigated and researched. One of the most promising and popular methods is termed as error-correcting encoding or error-correcting codes [1]. Error-correcting encoding is a method of adding redundancy or parity to set of source information in order to minimise the amount of errors occurring in the data at the receiver [3]. Two of the most popular types of encoding schemes are Linear Block Codes and Convolutional Codes.

This report will focus on Convolutional Codes. Convolutional codes are some of the most widely used codes being used today. GSM and its derivatives for data communications such as EDGE make use of convolutional codes to add redundancy and increase the reliability of data sent across a channel. The reason is that the convolutional codes are easy to implement,simple to decode, and can be efficiently and optimally decoded by the min-sum (Viterbi) algorithm [4].

During Practical 3 for EDC 310, students were required to design and implement a simulation platform in order to investigate the encoding and decoding, of a block of data sent across a channel, using Convolutional codes. Students were required to investigate the performance and illustrate the gain in reliability when error-correcting encoding, specifically Convolutional codes are used. The min-sum Viterbi algorithm coupled with the Viterbi Trellis was used by students to decode the parity bits sent across the channel [5].

The source information was to be encoded using the block and state diagrams in figures 1 and 2 respectively. The received parity symbols were to be decoded using the soft threshold, min-sum, Viterbi Algorithm with the past costs given by $\Delta_t = |r_t^1 - c_t^1| + |r_t^2 - c_t^2| + |r_t^3 - c_t^3|$

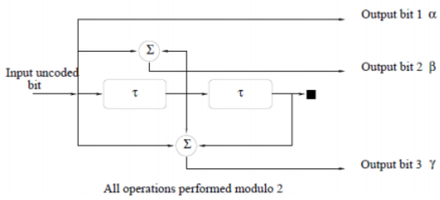

Figure 1. Convolutional Encoder Block diagram [5]

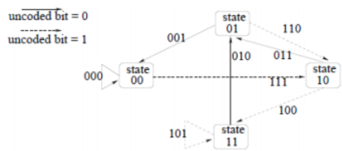

Figure 2. Convolutional Encoder State diagram [5]

2 Theoretical Analysis

A Convolutional Encoder is an extremely powerful algorithm and methodology to ensure redundancy and increase the reliability of digital signals sent across a channel. Convolutional encoders are linear and as such, are extremely efficient for large increases in source data information to be sent. Generally, however, Convolutional encoding is used for smaller streams or blocks of data and Linear Block encoding is used for large amounts of source data.

Unlike Linear Block codes, a Convolutional encoder and transmitter only transmits parity bits or parity symbols across the channel, the source bits are "hidden" within the parity bits [4]. For this reason, the techniques used to decode information on the receiver end, from encoded Convolutional codes back to the original source information are of utmost importance.

A Convolutional encoder is characterised majorly by three parameters, namely, the Code rate($R_c = \frac{n}{k}$), the generator polynomial and the constraint length (K).

A Convolutional encoder has "memory", by making use of a "shift register" as shown in figure 1, in order to encode a given data bit using the current input bit as well as the previous $K - 1$ source input data bits [3]. The code rate signifies that for every n bits to be encoded of source information, k parity bits are generated. The generator polynomial defines how for every bit to be encoded what the output parity bits will be and the constraint length defines the length of the shift register used and determines the amount of memory the encoder has or rather the number of past input data bits used to encode the source information.

Once the source input bits are encoded, the generated block of parity bits are sent across the channel to the receiving antenna. The receiving antenna then decodes the received code-word to determine the source bit block. Many algorithms such as the Viterbi MLSE algorithm and DFE exist that allow for decoding of linear convolutional codes. One would expect the performance of the Viterbi MLSE algorithm to be superior to that of DFE in decoding linear convolutional blocks since, the Viterbi MLSE algorithm takes into account all received bits instead of a single bit at a single time instance. This practical will test this hypothesis through a simulation platform.

3 Design and Implementation

This practical required students to implement a simulation platform to generate 100 uniform random bits, encode the bits using a Linear Convolutional encoder with a generator polynomial given by [4,6,7], a constraint length (K) of 3 and the block and state diagrams given in figures 1 and 2 receptively. The encoded bits where then modulated using the BPSK modulation technique, transmitted over a AWGN channel and decoded at the receiver. The BER was computed and plotted for the MLSE Viterbi decoding algorithm for Convolutional codes, the DFE decoding algorithm for Convolutional codes, the MLSE Viterbi Algorithm for Convolutional codes with Multipath and the optimal detection algorithm for un-coded source bits.

All source code can be found in the attached Appendix.All source code was implemented using Python.

3.1 Transmitter

At the transmitter the source code implemented during Practical 1 was used to generate the 100 un-coded bits using the uniform random number generator. The Linear Convolutional encoder was implemented in the function *encode*, which took in the data stream as well as the generator polynomial for the simulation. The shift register was implemented using a array. The generator polynomial given in decimal was translated to binary in order to encode the source information. The state-register array is initialised to contain all "0's" and has length equivalent that of the given constraint length.

Every data bit is prepended to the array one at a time, with the last element removed from the shift-register for each bit prepended, the generator polynomial is converted to binary, and for every bit to encode, the converted generator polynomial matrix is multiplied by the state-register array, each element of the result is added and the modulus of 2 is used to mimic *XOR'ing*. The encoded bits (3 for every 1 input) is appended to an array and returned to the caller.

3.2 Transmission platform

The transmission platform was implemented in practical 2 and used once again during practical 3.

3.3 Receiver

At the receiver, the MLSE Viterbi and DFE Algorithms were implemented in this practical. The implementation was the same as that of practical 2 with the exception of the path costs being calculated by $\Delta_t = |r_t^1 - c_t^1| + |r_t^2 - c_t^2| + |r_t^3 - c_t^3|$. In order to compute the past cost from a single to state to the next within the *getDelta* function, the received party bits, the time instance and the state transition was taken in as function parameters. The expected parity bits was first obtained for the given transition similar to the method used to encode the data bits.

In order to decode the parity bits received for the MLSE Viterbi, the Trellis was constructed beginning with the first state at (0,0) and the last state also ending at (0,0). The deltas for each transition was calculated as the Trellis was constructed.

Once the Trellis was constructed the Trellis is traverse to compute the alpha values and remove any contending paths, thereafter the Trellis is traversed again to obtained the cheapest path and the resulting source information is returned.

In order to decode the received information using DFE, every path cost for every valid state transition (as per figure 2) is computed and the state with the lowest delta value for a given time instance is held, with all other transitions for the respective time instance ignored. Once all transitions for every time instance has been inspected, the decoded source information is returned.

3.4 Simulation Platform

The BPSK simulation platform developed in Practical 1 adn 2 was used to develop a simulation platform that includes the effect of Convolutional encoding in the received signal and compared the results to the BER of un-coded transmission, the effect of a Linearly declining channel impulse response (Multipath) and the DFE and MLSE Vitberbi decoding algorithms. The simulation platform would simulate the transmission and detection of 100 data symbols over a AWGN channel that included the effect of multipath as well as excluded the effect of Multipath and Convolutional encoding.

The simulation platform was implemented in the *convolutionSim* class which took in the number of data bits to generate, the generator polynomial, the constraint length and the number of iterations to average the results over The class contained a number of member functions to encode and decode as well as to perform the simulation for both the DFE and MLSE decoding algorithms

The simulation platform can be executed using the *plotAll()* member function which would simulate all 4 simulations plot all results on the same plot.
Each simulation 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()* (implemented in Practical 1) function.

Noise was added to the modulated bits, thus simulating the transmission of the signal over a AWGN multipath channel. The Gaussian random number generator implemented in practical 1 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 since BPSK modulated and demodulation was being conducted. If needed for the simulation the tail symbols were appended and noise was added. The effect of multipath was added in the same manner as practical 2 if needed. The bits were "received" and decoded using the appropriate decoding algorithms and translated back into bits using the *BpskDemodulate()* function. The original bits generated were compared to the "detected" bits. An average of a user defined number of iterations was compute and the average BER for each SNR was returned.

Once conducted for both the DFE and MLSE algorithms the 4 curves were plotted on a single plot for comparison.

4 Results

The simulations were performed and the results were averaged over a variable number of iterations. A SNR (dB) range of between -4 and 8 dB was used to simulate the effects of a AGWN channel for the symbols sent across the channel. The decoded bits on the receiver end was compared to the original source information generated and the number of errors were counted. These errors were plotted as the ratio of the number of errors to the number of source information generated.

The results of the simulations performed for 1000 and 100 iterations are given in figures 3 and 4 respectively.

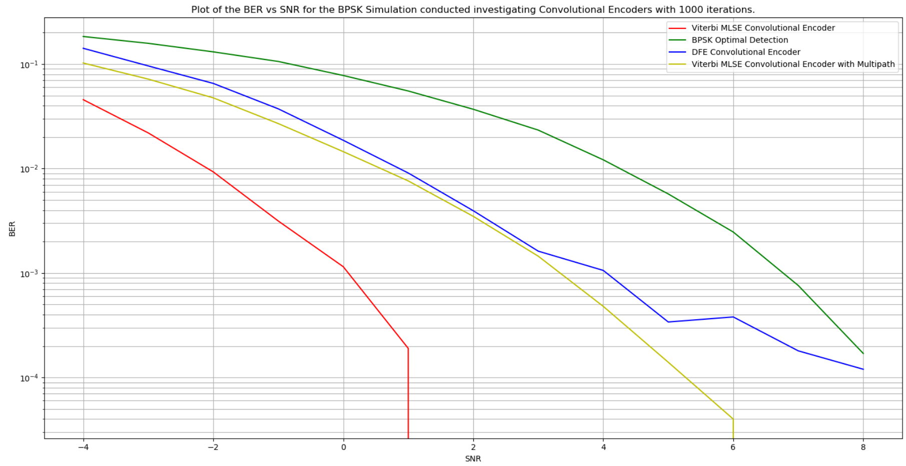

Figure 3. Simulation results for the entire simulation platform 1000 iterations

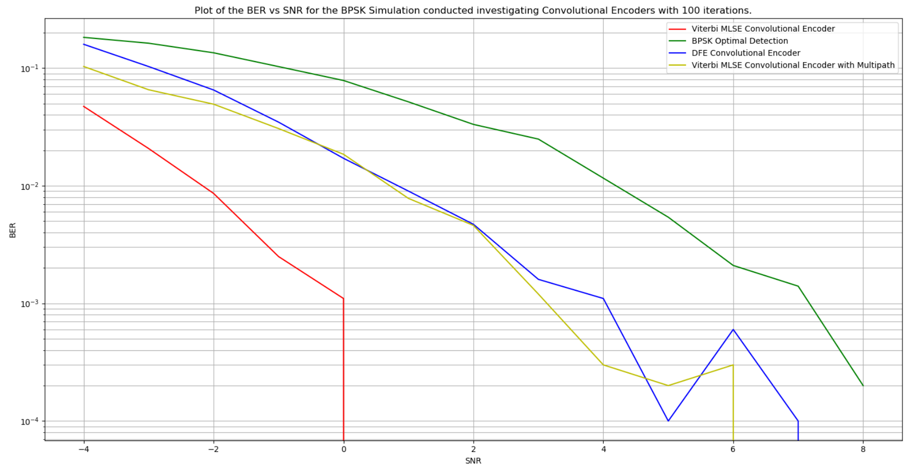

Figure 4. Simulation results for the entire simulation platform 100 iterations

5 Discussion

The results in figures 3 and 4 allow us to visualise and judge the performance of Convolutional encoding in the transmission of data over a channel. In order to gain a conceptual understanding as well as to truly grasp the effects of Convolutional encoding, simulations were performed with standard BPSK modulation of un-coded bits sent across a channel, the DFE algorithm for decoding Linear Convolutional codes as well as Multipath effects coupled with Convolutional encoding.

From the results obtained, one can see that the more iterations averaged over, the smoother the curves plotted and as such, the greater reliability the reader may have in the results obtained.

From figure 3, it is evident that the MLSE Viterbi algorithm used to decode a block of transmitted encoded bits sees an exponential increase in BER as the SNR increases. The MLSE Vitebri algorithm results in less than 1% of errors for a SNR of greater than 0. Therefore one can conclude that the performance of the Viterbi MLSE algorithm is extremely reliable in decoding Convolutional codes transmitted over a AWGN channel.

The DFE algorithm used for decoding Convolutional codes transmitted over the simulated AWGN channel, performs as expected. It sees an exponential decrease in the number of errors made in decoding the information, with an increase in the SNR. Once the SNR is greater than 6, the amount of noise causing distortion in the signal sent across the channel is low and the DFE algorithm results in a BER of less than 1% for this SNR range.

The un-coded source information sent across the channel is estimated at the receiver using the Optimal Detection algorithm. One can see from the results obtained, that the curve for un-coded source information sees an initial gradual decrease in BER as the SNR increases, slowly increasing in the rate of decrease as the SNR increases.

The Viterbi MLSE with Multipath and Convolutional encoding response, sees a gradual decrease in BER as the SNR increases. The rate of decrease increases as the SNR increases. It is evident that the Viterbi MLSE with Multipath and Convolutional encoding simulation results in a BER of less than 1% for a SNR greater than 6.

From the results obtained all simulation responses show a decrease in BER as the SNR increases. The MLSE Viterbi Algorithm for decoding source information that has been encoded with a Linear Convolutional encoder results in a much lower BER as compared to the un-coded bits transmitted. Using Convolutional encoding, resulted in an initial BER lower than that of the un-coded simulation. The rate of decrease of the BER as the SNR increased was also much higher for the encoded bit stream as opposed to the un-coded bit stream which saw an initial gradual increase. The final BER for both simulation results also differ with the Convolutional code word transmitted exhibiting a much lower BER for an equivalent SNR for the channel

The reason for the encoded data resulting in a lower BER for an equivalent SNR as compared to the un-coded data is due to the fact that the Convolutional encoder adds redundancy to the transmitted message, therefore, resulting in bits sent over channel having a lower chance of experiencing a large degree of distortion. The presence of the parity bits in the received information results in the noise added to the signal to be spread over the bit stream, resulting in lower distortion to the individual source bits "hidden" within or behind the parity bits. In addition, the MLSE algorithm takes into account past received bits and makes use of "backtracking" to reduce the effects of errors made in decoding the received information.

The MLSE Viterbi Algorithm with Convolutional encoding preforms the best from all simulations conducted. With the effects of Multipath taken into account the MLSE Viterbi algorithms proves to be best algorithm in decoding and estimating the information received at the received. This is evident from the results obtained with both MLSE Viterbi algorithms resulting in the two lowest BER for an equivalent SNR as compared to the other simulations performed. The DFE algorithm used for decoding the transmitted cod-word performs worse than the MLSE, however, does perform better than estimation of the un-coded data bits through optimal detection. This coincides with our assumptions and results obtained in Practical 2. The fact that DFE decoding of Convolutional code words transmitted across a channel performed better than estimating of un-coded data, further emphasises the effectiveness of using Convolutional encoding to add redundancy and increase the reliability of interpreting information received at a receiver.

The un-coded data simulation performed faster than the Convolutional encoding simulation as expected, since the Convolutional encoder chosen for this practical produces a factor three more bits to be sent, processed and decoded as compared to the un-coded simulation.

6 Conclusion

A Convolutional encoder is extremely powerful in increasing the reliability of data sent across a channel. It adds redundancy and parity to the source information in order to minimise the amount of errors made at a receiver in estimating the bits that have been sent. This practical allowed students to conceptually, logically and systemically conceive the process of transmitting digital signal across a channel, interpreting them and ensuring increased relatively in the process of estimating the distorted signal received.

In conclusion, Convolutional codes do prove extremely useful in minimising the errors made at the receiver. Use of a Convolutional encoder results in a much lower BER as compared to transmitting data over a channel without encoding. This was true for the entire spectrum of SNR values simulated.

The process of decoding the encoded distorted information received is slower than interpreting un-coded data received. This, however, is a trade-off that must be made between reliability as opposed to speed. The higher the code rate of the encoder, the greater number of party bits generated and the longer the process of encoding and decoding.

7 References

[1] S. Haykin, *Communications Systems*, Third ed. New York: John Wiley and Sons, 1994.

[2] C. Shannon, “A Mathematical Theoryof Communications,” *Bell Systems Technical Journal*, vol. 27, October 1948.

[3] S. Holub, *Introduction to convolutional codes.* MIT - Massachusetts Institute of Technology, 2016.

[4] P. J. Olivier, *Class Notes: Digital Communications*, 3rd ed. Department of Electrical, Electronic and Computer Engineering - University of Pretoria, 2008.

[5] H. M. . A. Yan, *EDC 310 - Practical Assigment 3.* Department of Electrical, Electronic and Computer Engineering - University of Pretoria, 2018.

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)
- copy

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

A.2 Execution of the script

All scripts must be in the same working directory in order to execute the simulation. The simulation may be executed from the *ConvolutionalCodes.py* python script. Please ensure that lines 421 to 426 of the script is not commented out.

The script will automatically simulate the DFE and MLSE Viterbi algorithms for a Convolutional encoding scheme, the MLSE Viterbi Algorithm for Multipath combined with the MLSE Viterbi for decoding Convolutional codes as well as a standard BPSK Optimal detection simulation on a uniformly randomly generated set of 100 bits (by default, may be changed by the user) and plot the results on a single plot. The number of iterations to average the results over may be changed by the user for more accurate results (see source code comments).

B Python source code

```
1 # Mohamed Ameen Omar
2 # 16055323
3 #####################################
4 ###       EDC 310 Practical 3      ###
5 ###             2018               ###
```

```python
###       Convolutional Codes      ###
###            Simulation          ###
######################################

import numpy as np
from question_3 import BPSKmodulationSimulation
from Simulation import Simulation
import copy
import matplotlib.pyplot as plt
from MLSE import MLSE

# Class to represent a single node in the Viterbi Trellis
# used for the MLSE Equalizer
# Contains the time, the alpha value(sum of deltas along cheapest path to
    node),
# the state of the node, all previous states it is connected to and all
    next states (time > node's time)
# it is connected to
class trellisNode:
    def __init__(self, state, time):
        self.nextStates = []
        self.previousStates = []
        self.time = time
        self.state = state
        self.deltas = []  # delta array corresponding to the order of
    previous States
        self.alpha = 0
    #Add a new state that the current node is connected to.

    def addNext(self, state):
        self.nextStates.append(state)
    # Add a previous state that the node is connected to

    def addPrevious(self, state):
        self.previousStates.append(state)


########################## begin of class #######################


# Class to perform a simulation to determine the
# average BER for the Viterbi MLSE and DFE symbol
# estimation methods for Convolutional Encoders as
# well as the performance of a linear Convolutional encoder
# taking into account the effects of Multipath with a linearly declining
    CIR
# The class takes in the number of data
# bits the generator polynomial, the constraint length and the number of
    Iterations to average the results over
# as constructor parameters
class convolutionSim:
    def __init__(self, n = 5, generator = [4,6,7], K = 3, numIter = 50):
        self.symbols = [1, -1]
        self.BpskSimulation = BPSKmodulationSimulation(n)
        self.K = K #constraint length
        self.n = n #uncoded data bits
```

```
        self.generator = generator #generator matrix - length of which is number of outputs
        self.numIter = numIter

    # Function to encode the given source bitStream using the
    # given geneator polynomial with a Linear Convolutional encoding scheme
    # Returns the encoded parity bits
    def encode(self,generator = [4,6,7], bitStream = []):
        shiftRegister = []
        genBinary = []
        #get binary representation of generator
        for x in range(0, len(generator)):
            genBinary.append("{0:b}".format(generator[x]))

        #sort shift register to begin in state 0
        for x in range(0,len(genBinary[x])):
            shiftRegister.append(0)
        # list to contain all parity bits
        parity = []
        #encode the data
        # for every bit in the bitStream, insert into shfit register
        # take shift reggister elements, multiply by genertor in binar%2
        for x in range(0,len(bitStream)):
            #print("Inserting", bitStream[x])
            shiftRegister.insert(0,bitStream[x])
            shiftRegister.pop()
            #print("Shift after insert:", shiftRegister)
            temp = 0
            for y in range(0,len(generator)):
                #print("Output bit ", y)
                temp = 0
                for z in range(0,len(shiftRegister)):
                    temp += shiftRegister[z]*int(genBinary[y][z]).__round__()
                #print("TEMP",temp)
                temp = temp % 2
                #print("temp after mod 2", temp)
                parity.append(temp)
                #print("Parity so far", parity)
        return parity

    #Function to build the Viterbi Trellis for decoding a Linear Convolutional encoded
    # block of bits.
    # The parity bits recieved are passed in as function paramters as well as the
    # desired start state
    # Returns the constructed Viterbi Trellis
    def buildTrellis(self,parity,startState = [-1,-1]):
        #build the trellis
        Trellis = np.empty(shape=(2**(self.K-1), self.n+self.K), dtype=trellisNode)
       # print(len(Trellis))
        #print(len(Trellis[0]))
        Trellis[0][0] = trellisNode(startState, 0)
```

```
        #print("FIRST ONE", Trellis[0][0].state)
        numBits = len(startState)
        #for every node in the trellis compute the nodes it
        # will transition to
        for t in range(0,  self.n+self.K-1):
            for s in range(0, (2**(self.K-1))):
                #print(Trellis[0][0].state)
                # check if this is none
                if Trellis[s][t] is None:
                    continue
                else:
                    #get all states:
                    allStates = []
                    tempState = []
                    # #only upward transitions
                    if(t >= self.n+self.K-3):
                        #print("HERE")
                        tempState.append(-1)
                        for x in range(0, numBits-1):
                            tempState.append(Trellis[s][t].state[x])
                        allStates.append(tempState)

                    else:
                        for x in range(0, len(self.symbols)):
                            tempState = []
                            tempState.append(self.symbols[x])
                            #print(tempState, "TEMPSTATE")
                            for y in range(0, numBits-1):
                                tempState.append(Trellis[s][t].state[y])
                            allStates.append(tempState)

                    #print(allStates)
                    #now we have all states
                    for index in range(0, len(allStates)):
                        #print("HERE")
                        #print(t)
                        newState = allStates[index]
                        #print(newState, "NEW STATE")
                        stateIndex = self.getStateIndex(newState)
                        prevNode = Trellis[s][t]
                        #print("Start Index", stateIndex)
                        if Trellis[stateIndex][t+1] is None:
                            Trellis[stateIndex][t +1] = trellisNode(newState, t+1)
                            #print("newStateHERE", newState)

                        newNode = Trellis[stateIndex][t+1]
                        prevNode.addNext(newNode)
                        delta = self.getDelta(t+1, prevNode.state, newState,parity)
                        newNode.deltas.append(delta)
                        newNode.alpha = delta
                        newNode.addPrevious(prevNode)
        return Trellis
    #Function to decode the recieved parity bits using the Viterbi MLSE algorithm for a Linear Convolutional encoded
```

```
    # block of bits.
    # The parity bits recieved are passed in as function paramters as well as the
    # desired start state
    # Returns the decoded source information
    def mlseDecode(self,parity,startState = [-1,-1]):
        Trellis = self.buildTrellis(parity,startState)
        for t in range(0,  self.n+self.K):
            for s in range(0, (2**(self.K-1))):
                if(Trellis[s][t] is None):
                    continue
                #if more than one state connected (contending)
                if(len(Trellis[s][t].previousStates) > 1):
                    #get smallest delta
                    #delete all other previous states, all other deltas, store complete path alpha
                    bestIndex = 0 #index of node with the best or shortest path so far
                    #just set to first
                    bestAlpha = Trellis[s][t].previousStates[0].alpha + Trellis[s][t].deltas[0]
                    for x in range(0, len(Trellis[s][t].previousStates)):
                        tempAlpha = Trellis[s][t].previousStates[x].alpha + Trellis[s][t].deltas[x]
                        if(tempAlpha <= bestAlpha):
                            bestIndex = x
                            bestAlpha = tempAlpha
                    Trellis[s][t].alpha = bestAlpha
                    Trellis[s][t].deltas = [ Trellis[s][t].deltas[bestIndex] ]
                    Trellis[s][t].previousStates = [ Trellis[s][t].previousStates[bestIndex] ]
                else:
                    #compute the alpha
                    if(len(Trellis[s][t].previousStates)  == 0):
                        continue
                    Trellis[s][t].alpha += Trellis[s][t].previousStates[0].alpha
        myTemp = Trellis[0][-1]

        #print(myTemp.state)
        detected = [myTemp.state[0]]  #this is just from the trellis
        #print("State left at time", myTemp.time, "is", myTemp.state)
        myTemp = myTemp.previousStates[0]
        while(myTemp.time > 0):
            #print("State left at time", myTemp.time, "is", myTemp.state)
            detected = [myTemp.state[0]] + detected
            myTemp = myTemp.previousStates[0]
        #detected = [myTemp.state[0]] + detected #add t=0
        #print("detected", detected)
        return detected

    # Return the state index [-1,-1] = 0 and [1,-1] = 1 , and so forth
    def getStateIndex(self, state):
        if(state == [1, 1]):
            return 3
```

```
        if(state == [1, -1]):
            return 2
        if(state == [-1, 1]):
            return 1
        if(state == [-1, -1]):
            return 0
    # Returns the delta value for time = time, from
    # state1 to state2 for the given parity bits recieved
    def getDelta(self,time,state1,state2,parityRec):
        #first need to get parity symbsols at time
        #get output symbols
        # print("THE PARITY AFTER SENT", parityRec)
        # print("Getting Delta for time", time)
        # print("State1", state1)
        # print("state2", state2)
        outputSymbols = self.getOutputSymbols(state1,state2)
        #print("Output Symbols", outputSymbols)
        recParity = []
        for x in range( (time-1)*len(self.generator), (time-1)*len(self.generator)+len(self.generator)):
            recParity.append(parityRec[x])
        temp = 0
        #print("Rec parity", recParity)
        for x in range(0,len(recParity)):
            temp += np.abs(recParity[x]-outputSymbols[x])**2
        return temp

    # Returns the output symbols as per the state diagram
    # for a transition from state 1 to state2
    #state 1 is the orignial, state 2 is the new state
    def getOutputSymbols(self,state1,state2):
        shiftRegister = copy.deepcopy(state1)
        #insert the first bit of state
        shiftRegister.insert(0, copy.deepcopy(state2[0]))
        # shiftRegister.pop()
        genBinary = []
        for x in range(0,len(shiftRegister)):
            if(shiftRegister[x] == -1):
                shiftRegister[x] = 0
        #get binary representation of generator
        for x in range(0, len(self.generator)):
            genBinary.append("{0:b}".format(self.generator[x]))
        # list to contain all parity bits
        parity = []
        temp = 0
        #print("Shift register", shiftRegister)
        for y in range(0, len(self.generator)):
            temp = 0
            for z in range(0, len(shiftRegister)):
                #print("temp +=" , shiftRegister[z], "*", int(genBinary[y][z]).__round__())
                temp += shiftRegister[z]*int(genBinary[y][z]).__round__()
            #print("TEMP",temp)
            temp = temp % 2
            if(temp == 0):
                temp = -1
```

```
            #print("temp after mod 2", temp)
            parity.append(temp)
            #print("Parity so far", parity)
        #print()
        return parity
    # Function to perform the simulation using the MLSE Viterbi algorhtm to decode
    # No multipath effects taken into account
    # Returns the BER list
    def SimulateMLSE(self):
        numIter = self.numIter
        print("Conducting a MLSE BPSK simulation with Convolutional Encoding")
        BER = []  # store the average BER for each SNR
        for SNR in range(-4, 9): #9
            tempBER = [] #to store all the interations for one SNR
            for count in range(0, numIter):
                myDataBits = self.BpskSimulation.generateBits()
                #append tail
                for x in range(0, self.K-1):
                    myDataBits.append(0)
                encodedBits = self.encode(self.generator, myDataBits)
                modulatedSignal = self.BpskSimulation.BpskModulate(encodedBits)
                signalSent = self.BpskSimulation.addNoise(SNR, modulatedSignal)
                decodedSymbols = self.mlseDecode(signalSent)
                decodedBits = self.BpskSimulation.BpskDemodulate(decodedSymbols)
                tempBER.append(self.BpskSimulation.getNumErrors(myDataBits,decodedBits)/(self.n))
            BER.append((sum(tempBER)/len(tempBER)))
        print(BER)
        return BER
    # Function to perform the simulation using BPSK modulation, no multipath, no Convoltuional encoding
    # Returns the BER list
    def simulateBPSK(self):
        numIter = self.numIter
        print("Conducting a BPSK simulation with no Convolutional Encoding")
        BER = [] #store the average BER for each SNR
        for SNR in range(-4, 9): #9
            tempBER = [] #to store all the interations for one SNR
            for count in range(0, numIter):
                # generate 300 bits
                myDataBits = self.BpskSimulation.generateBits() #raw data bits
                # map bits to symbols
                modulatedSignal = self.BpskSimulation.BpskModulate(myDataBits)
                signalSent = self.BpskSimulation.addNoise(SNR, modulatedSignal)
                detectedSignal = self.BpskSimulation.detectSignal(SNR, signalSent)
                # demodulate
```

```
                demodulatedSignal = self.BpskSimulation.BpskDemodulate(detectedSignal)
                # compare
                tempBER.append(self.BpskSimulation.getNumErrors(myDataBits,demodulatedSignal)/self.n)
            BER.append( (sum(tempBER)/len(tempBER) ))
        print(BER)
        return BER

    #Function to decode the recieved parity bits using the DFE algorithm for a Linear Convolutional encoded
    # block of bits.
    # The parity bits recieved are passed in as function paramters as well as the
    # desired start state
    # Returns the decoded source information
    def dfeDecode(self,parity,startState = [-1,-1]):
        detected = []
        detectedState = startState
        #detected.append(detectedState[0])
        for x in range(0,self.n+self.K):
            tempState1 = detectedState
            if(self.getDelta(x, tempState1, [-1, tempState1[0]], parity) > self.getDelta(x, tempState1, [1, tempState1[0]], parity)):
                detectedState = [1, tempState1[0]]
            else:
                detectedState = [-1, tempState1[0]]
            detected.append(detectedState[0])
        return detected[1:]

    # Function to perform the simulation using the DFE algorhtm to decode
    # No multipath effects taken into account
    # Returns the BER list
    def SimulateDfe(self):
        numIter = self.numIter
        print("Conducting a DFE BPSK simulation with Convolutional Encoding")
        BER = []  # store the average BER for each SNR
        for SNR in range(-4, 9): #9
            tempBER = [] #to store all the interations for one SNR
            for count in range(0, numIter):
                myDataBits = self.BpskSimulation.generateBits()
                #print("Data Bits:", myDataBits)
                #append tail
                for x in range(0, self.K-1):
                    myDataBits.append(0)
                #print("Data Bits after append:", myDataBits)
                #print("Encoding")
                encodedBits = self.encode(self.generator, myDataBits)
                #print("Encoded bits", encodedBits)
                modulatedSignal = self.BpskSimulation.BpskModulate(encodedBits)
                #print("Modulated Encoded", modulatedSignal)
                signalSent = self.BpskSimulation.addNoise(SNR, modulatedSignal)
                decodedSymbols = self.dfeDecode(signalSent)
```

```
                #print("Decoded Symbols", decodedSymbols)
                decodedBits = self.BpskSimulation.BpskDemodulate(decodedSymbols)
                #print("decoded Bits", decodedBits)

                tempBER.append(self.BpskSimulation.getNumErrors(myDataBits,decodedBits)/(self.n))
            BER.append((sum(tempBER)/len(tempBER)))
        print(BER)
        return BER
    # Function to perform the simulation using the MLSE Viterbi algorhtm to decode
    # Multipath effects taken into account
    # Returns the BER list
    def simulateMultiPath(self):
        numIter = self.numIter
        multiSim = Simulation(n=self.n*len(generator), linC=[0.89, 0.42, 0.12])
        print("Conducting a Multipath BPSK simulation with Convolutional Encoding")
        BER = []  # store the average BER for each SNR
        for SNR in range(-4, 9): #9
            tempBER = [] #to store all the interations for one SNR
            for count in range(0, numIter):
                myDataBits = self.BpskSimulation.generateBits()
                #print("Original Data", myDataBits)
                #append tail
                for x in range(0, self.K-1):
                    myDataBits.append(0)
                #print("ROIGNAL WITH APPEND", myDataBits)
                encodedBits = self.encode(self.generator, myDataBits)
                #print("Encoded", encodedBits)
                modulatedSignal = self.BpskSimulation.BpskModulate(encodedBits)
                convolutedSignal = multiSim.channelModification(multiSim.linC, modulatedSignal)
                signalSent = self.BpskSimulation.addNoise(SNR, convolutedSignal)
                myMLSE = MLSE(self.n*len(generator), signalSent, [0.89, 0.42, 0.12])
                myMLSE.buildTrellis()
                myMLSE.cheapestPath()
                # get data bits
                dataDetected = myMLSE.dataDetected
                #print("Data Detected", dataDetected)
                decodedSymbols = self.mlseDecode(signalSent)
                decodedBits = self.BpskSimulation.BpskDemodulate(decodedSymbols)
                #print("Decoded bits", decodedBits)
                tempBER.append(self.BpskSimulation.getNumErrors(myDataBits,decodedBits)/(self.n))
            BER.append((sum(tempBER)/len(tempBER)))
        print(BER)
        return BER
    # Fucntion to perform and plot all the implemented simulations on a single curve
```

```python
    def plotAll(self):
        print("Plotting all for", self.numIter, "iterations")
        SNR = np.linspace(-4, 8, 13)
        MlseConv = self.SimulateMLSE()
        bpskSim = self.simulateBPSK()
        dfeConv = self.SimulateDfe()
        mlseMultiPath = self.simulateMultiPath()
        plt.semilogy(SNR, MlseConv, 'r-', label='Viterbi MLSE Convolutional Encoder')
        plt.semilogy(SNR, bpskSim, 'g-', label='BPSK Optimal Detection')
        plt.semilogy(SNR, dfeConv, 'b-', label='DFE Convolutional Encoder')
        plt.semilogy(SNR, mlseMultiPath, 'y-', label='Viterbi MLSE Convolutional Encoder with Multipath')
        plt.grid(which='both')
        plt.ylabel("BER")
        plt.xlabel("SNR")
        title = "Plot of the BER vs SNR for the BPSK Simulation conducted investigating Convolutional Encoders with " + str(self.numIter) + " iterations."
        plt.title(title)
        plt.legend(loc='best')
        plt.show()
        print("Complete")
        return
######################### end of class #######################

#uncomment last line to conduct a simulation of all simulations implemented
n = 100 #number unenconded bits
numIter = 15 #number of iterations to avergae the results over
generator = [4, 6, 7] #generator polynomial
K = 3 #constraint length
tempSim = convolutionSim(n,generator,K, numIter)
tempSim.plotAll()
```

Listing 1. Python Source code to execute the simulation of all 4 simulations

```python
# Mohamed Ameen Omar
# 16055323
####################################
###     EDC 310 Practical 2      ###
###           2018               ###
###      BPSK DFE Algotihm       ###
####################################

import numpy as np
import copy

# Class to run a DFE equalizer to dermine the bits sent over a AGWN channel
# Constructor paramaters:
#   @param N = the number of data bits of data being sent
#   @param r = a vector with all the recieved symbols
#   @param c = the channel impulse response vector
```

```python
class DFE:
    def __init__(self, N = 0, r = [], c = []):
        self.n = N #number of data bits
        self.r = r #recieved vector - only data bits len(r) = self.n
        self.c = c #convolution matrix
        self.L = len(c) #length of the c vector
        self.numHeader = self.L-1 #number of header bits
        self.symbols = [1,-1]
        self.dataDetected = [] #just the data bits detected

    # Function to return the data symbols detected.
    # It will detect or estimate or symbols recieved and return
    # a vector with those symbols
    def getDataSymbols(self):
        if(self.dataDetected == []):
            self.detectSymbols()
            return self.dataDetected
        else:
            return self.dataDetected

    # Function to detect the symbols in the recieved vector
    # using the DFE Equalizer
    def detectSymbols(self):
        for x in range(0,len(self.r)):
            self.dataDetected.append(self.getSymbol(x))
    # Function to detect a single sumbol using
    # DFE Equalizer algorithm
    def getSymbol(self,time):
        if(self.getDelta(time, 1) > self.getDelta(time, -1)):
            return -1
        else:
            return 1

    # Function to get the delta value for a single symbol
    # symbol is the symbol we are estimating
    # time is the time instance for the symbol we are getting the delta
    def getDelta(self,time,symbol):
        temp = 0
        t = 0
        for x in range(0,self.L):
            if(x == 0):
                temp += self.c[x]*symbol
            else:
                #if we havent detected the first l symbols
                if(len(self.dataDetected)+t <0):
                    temp += self.c[x]*1
                else:
                    temp += self.c[x]*self.dataDetected[len(self.
dataDetected)+t]
            t = t-1
        temp = np.abs(self.r[time] - temp) **2
        return temp
################## END OF CLASS IMPLEMENTATION
```

```
    ###############################################
```

Listing 2. Python Source code for the DFE class - to execute a DFE signal estimation - from Practical 2

```
# Mohamed Ameen Omar
# 16055323
########################################
###      EDC 310 Practical 2       ###
###              2018              ###
###    MLSE- Viterbi Algotihm      ###
########################################

import numpy as np
from question_3 import BPSKmodulationSimulation
import copy

'''
MY IMPLEMENTATION


trellisNode store the time, the state, the next states, the alpha, the
    previous states of a node

first pass in the r,n,c

then it builds it by:
assigniedn fist node to 11 at t=0
then for every node, create its transitons and update all
including all alphas and deltas

to get path:
from left to right
check if there's contending, if not just add previous alpha to current
if there is, get alphas for each one, get smallest, remove all other
    deltas and continue until end
'''

# Class to represent a single node in the Viterbi Trellis
# used for the MLSE Equalizer
# Contains the time, the alpha value(sum of deltas along cheapest path to
    node),
# the state of the node, all previous states it is connected to and all
    next states (time > node's time)
# it is connected to
class trellisNode:
    def __init__(self,state,time):
        self.nextStates = []
        self.previousStates = []
        self.time = time
        self.state = state
        self.deltas = [] #delta array corresponding to the order of
    previous States
        self.alpha = 0
    #Add a new state that the current node is connected to.
    def addNext(self,state):
```

```
        self.nextStates.append(state)
    # Add a previous state that the node is connected to
    def addPrevious(self, state):
        self.previousStates.append(state)
######################## END OF CLASS ############################

#class that conducts the MLSE algorithm for ANY BPSK modulated signal
# Requires:
# @param N = the number of data bits in the recieved signal
# @pram r = the recieved vector of bits
# @param c = the Channel Impulse response for the channel

# it will build the trellis, thereafter calculate all deltas for all states or nodes in the trellis.
# then compute the cheapest path and disregard any nodes elliminated
class MLSE:
    def __init__(self, N = 0, r = [], c = []):
        self.n = N #data bits
        self.symbols = [1, -1]  # modulation symbols
        self.r = r #recieved symbols
        self.c = c #Convolution matrix
        self.trellisLength = self.n + len(self.c) - 1 #Trellis will be up to time T
        self.numStates = len(self.symbols)**(len(self.c) -1)
        self.numHT = len(c)-1
        self.Trellis = np.empty(shape=(self.numStates, self.trellisLength+1), dtype=trellisNode)
        self.detected = [] #detected symbols from trellis
        self.entireStream = [] #entire detected stream including head and tail
        self.dataDetected = [] #data bits detected

    #print all properties of the problem to which MLSE is applied
    def printProperties(self):
        print("Number of data bits:", self.n)
        print("Signal recieved:", self.r)
        print("Convolution Matrix (C): ", self.c)
        print("Trellis Length (L): ", self.trellisLength)
        print("Number of states:", self.numStates)
        print("Number of Head and Tail symbols:", self.numHT)
        print("Modulation Symbols:", self.symbols)

    # Function to build the viterbi trellis
    # will begin with the first node state being = 1,1; following convention.
    # assigning fist node to 11 at t=0
    # then for every node, create its transitons and update all
    # including all alphas and deltas
    def buildTrellis(self, startState = [1,1]):
        self.Trellis[0][0] = trellisNode(startState, 0)
        numBits = len(startState)
        #for every node in the trellis compute the nodes it
        # will transition to
        for t in range(0,self.trellisLength):
            for s in range(0,self.numStates):
                # check if this is none
```

```
                if self.Trellis[s][t] is None:
                    continue
                else:
                    #get all states:
                    allStates = []
                    tempState = []
                    #only upward transitions
                    if(t >= self.trellisLength-2):
                        tempState.append(1)
                        for x in range(0,numBits-1):
                            tempState.append(self.Trellis[s][t].state[x])
                        allStates.append(tempState)
                    #both 1,-1
                    else:
                        for x in range(0,len(self.symbols)):
                            tempState = []
                            tempState.append(self.symbols[x])
                            for y in range(0, numBits-1):
                                tempState.append(self.Trellis[s][t].state[y])
                            allStates.append(tempState)
                    #now we have all states
                    for index in range(0,len(allStates)):
                        newState = allStates[index]
                        stateIndex = self.getStateIndex(newState)
                        prevNode = self.Trellis[s][t]
                        if self.Trellis[stateIndex][t+1] is None:
                            self.Trellis[stateIndex][t+1] = trellisNode(newState,t+1)

                        newNode = self.Trellis[stateIndex][t+1]
                        prevNode.addNext(newNode)
                        delta = self.computeDelta(t+1,prevNode.state,newState)
                        newNode.deltas.append(delta)
                        newNode.alpha = delta
                        newNode.addPrevious(prevNode)

    # Function to print all the deltas for all the nodes in the Viterbi Trellis
    def printAllDeltas(self):
        for t in range(1,self.trellisLength+1):
            for s in range(0,self.numStates):
                myNode = self.Trellis[s][t]
                if(myNode is None):
                    continue
                for prev in range(0, len(myNode.previousStates)):
                    print("Delta", myNode.time, "from", self.getStateIndex(myNode.previousStates[prev].state), "to",self.getStateIndex(myNode.state), "is", myNode.deltas[prev] )
            print()

    # Return the state index [1,1] = 0 and [1,-1] = 1 , and so forth
    def getStateIndex(self,state):
        if(state == [1,1]):
            return 0
```

```

        if(state == [1,-1]):
            return 1

        if(state == [-1, 1]):
            return 2

        if(state == [-1, -1]):
            return 3

    # Function to perform a delta calculation from state1 to state 2 @time = @param time
    #state 1 is the state originating, state 2 is the state going to
    def computeDelta(self,time,state1,state2):
        temp = 0
        for x in range(0,len(self.c)):
            if(x < len(state2)):
                temp += self.c[x]*state2[x]
            else:
                temp += self.c[x]*state1[x-len(state2) +1]
        temp = np.abs(self.r[time-1] - temp)**2
        return temp

    # Function to calculate the cheapest path and in extension estimate the
    # symbols recieved. Must build the trellis before calling this function
    # to get path:
    # from left to right
    # check if there's contending, if not just add previous alpha to current
    # if there is, get alphas for each one, get smallest, remove all other deltas and continue until end
    def cheapestPath(self):
        for t in range(0,self.trellisLength+1):
            for s in range(0,self.numStates):
                if(self.Trellis[s][t] is None):
                    continue
                #if more than one state connected (contending)
                if(len(self.Trellis[s][t].previousStates) > 1):
                    #get smallest delta
                    #delete all other previous states, all other deltas, store complete path alpha
                    bestIndex = 0 #index of node with the best or shortest path so far
                    #just set to first
                    bestAlpha = self.Trellis[s][t].previousStates[0].alpha + self.Trellis[s][t].deltas[0]
                    for x in range(1, len(self.Trellis[s][t].previousStates)):
                        tempAlpha = self.Trellis[s][t].previousStates[x].alpha + self.Trellis[s][t].deltas[x]
                        if(tempAlpha < bestAlpha):
                            bestIndex = x
                            bestAlpha = tempAlpha
                    self.Trellis[s][t].alpha = bestAlpha
```

```
                    self.Trellis[s][t].deltas = [ self.Trellis[s][t].deltas[bestIndex] ]
                    self.Trellis[s][t].previousStates = [ self.Trellis[s][t].previousStates[bestIndex] ]
                else:
                    #compute the alpha
                    if(len(self.Trellis[s][t].previousStates) == 0):
                        continue
                    self.Trellis[s][t].alpha += self.Trellis[s][t].previousStates[0].alpha

        myTemp = self.Trellis[0][-1]
        self.detected = [myTemp.state[0]] + self.detected #this is just from the trellis
        myTemp = myTemp.previousStates[0]
        while(myTemp.time> 0 ):
            self.detected = [myTemp.state[0]] + self.detected
            myTemp = myTemp.previousStates[0]
        self.detected = [myTemp.state[0]] + self.detected #add t=0
        self.entireStream = copy.deepcopy(self.detected) #with head and tail
        while(len(self.entireStream) != (self.n+self.numHT+self.numHT)):
            self.entireStream = [1] + self.entireStream

        streamLength = len(self.entireStream)
        for x in range(0,streamLength):
            if(x < self.numHT or streamLength-x <= self.numHT):
                continue
            else:
                self.dataDetected.append(self.entireStream[x])

    # Start with building the trellis
    # have the deltas and all previous and next states for each node
    # compute the alpha values for all = total path cost including the current node so far
    '''
    go through from the last and add the remaining nodes in the Trellis that are connected
    '''

################### END OF CLASS ############################
```

Listing 3. Python Source code for the MLSE class - to execute a MLSE signal estimation for multipath - from practical 2

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

import numpy as np
import matplotlib.pyplot as plt
import time
```

```
import datetime
from scipy.stats import norm

############ Random Number Class #####################
# The random number class contains the two random number generators
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))

    def rejection(self,mean = 0,stdDev = 1):
        a1 = 0.8638
        a2 = 0.1107
        a3 = 0.0228002039
        a4 = 1-a1-a2-a3
        myProb = self.WHill()


        if(myProb < a1):
            temp = self.WHill()
            temp2 = self.WHill()
            temp3 = self.WHill()
            return (2*(temp+temp2+temp3-1.5))

        if(myProb < (a1+a2)):
            temp = self.WHill()
            temp2 = self.WHill()
            return(1.5*(temp+temp2-1))

        if(myProb < (a1+a2+a3)):
            temp = self.WHill()
            temp2 = self.WHill()
            x = (6*temp) -3
            y  = 0.358*(temp2)

            while(y < self.g3(x)):
                temp = self.WHill()
                temp2 = self.WHill()
                x = (6*temp) - 3
                y = 0.358*(temp2)
            return x
        else:

            v1 = self.WHill()*2 -1
            v2 = self.WHill()*2 -1
            x = 0
            y = 0
            while(np.abs(x) < 3 and np.abs(y) < 3):
                v1 = self.WHill()*2 - 1
                v2 = self.WHill()*2 - 1
                print("FUCK")
                while(((v1**2) + (v2**2)) > 1 or ((v1**2) + (v2**2)) == 1):
                    print("FUCK2")
                    v1 = self.WHill() *2 -1
                    v2 = self.WHill()*2 -1

                temp = (v1**2) + (v2**2)
                temp = np.log(temp)
                temp = -2*temp
                temp = temp+9
                temp = temp/((v1**2) +(v2**2))
```

```python
                x = v1*np.sqrt(temp)
                y = v2*np.sqrt(temp)
            if(np.abs(x) > 3):
                return x
            return y




    def g3(self,x):
        if(np.abs(x) < 1):
            return (17.49731196*np.exp(-0.5-(x**2)) -4.73570326*(3-(x**2)
    ) -2.15787544*(1.5-np.abs(x) ))

        if(np.abs(x) > 1 and (np.abs(x) <1.5)):
            return (17.49731196*np.exp(-0.5-(x**2)) - 2.36785163*( (3-(x)
    )**2) - 2.15787544*(1.5-np.abs(x)))

        if(np.abs(x) > 1.5 and (np.abs(x) < 3)):
            return (17.49731196*np.exp(-0.5-(x**2)) - 2.36785163*( (3-(x)
    )**2) - 2.15787544*(1.5-np.abs(x)))

        return 0



################## 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)))


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 4. Python Source code for question 1 (Uniform Random Number Generator) - From practical 1

```
# Mohamed Ameen Omar
# 16055323
#######################################
###      EDC 310 Practical 2      ###
###             2018              ###
###          Simulation           ###
#######################################

import numpy as np
from question_3 import BPSKmodulationSimulation
from MLSE import MLSE
import copy
from DFE import DFE
import matplotlib.pyplot as plt

# Class to perform a simulation of the
# BER for the Viterbi MLSE and DFE symbol
# estimation methods
# The class takes in the number of data
# bits and the linear declining channel impulse response
# vector as constructor parameters
class Simulation:
    def __init__(self, n=300, linC=[0.89, 0.42, 0.12], numIterations=20):
        self.BpskSimulation = BPSKmodulationSimulation(n)
        self.linC = linC
        self.n = n
        self.numIterations = numIterations

    # Function to perform the Viterbi MLSE simulation
    # with a Gaussian random Channel Impulse Response
    # returns the BER array
```

```python
    # simulation runs for @param self.numIterations, for each
    # SNR in the range (-4,9)
    def viterbiRandomCIR(self):
        numIter = self.numIterations
        print("Conducting a MLSE BPSK simulation with a Uniform Random CIR")
        BER = [] #store the average BER for each SNR
        for SNR in range(-4, 9): #9
            tempBER = [] #to store all the interations for one SNR
            for count in range(0, numIter):
                # generate 300 bits
                myDataBits = self.BpskSimulation.generateBits() #raw data bits
                # map bits to symbols
                modulatedSignal = self.BpskSimulation.BpskModulate(myDataBits)
                # add tail
                for x in range(0,len(self.linC) -1):
                    modulatedSignal.append(1)
                # add convolution
                randomCIR = self.generateRandomCIR(SNR)
                convolutedSignal = self.channelModification(randomCIR, modulatedSignal)
                signalSent = self.BpskSimulation.addNoise(SNR, convolutedSignal)
                # check MLSE
                myMLSE = MLSE(self.n, signalSent, randomCIR)
                myMLSE.buildTrellis()
                myMLSE.cheapestPath()
                # get data bits
                dataDetected = myMLSE.dataDetected
                # demodulate
                demodulatedSignal = self.BpskSimulation.BpskDemodulate(dataDetected)
                # compare
                tempBER.append(self.BpskSimulation.getNumErrors(myDataBits,demodulatedSignal)/self.n)
                del myMLSE
                del modulatedSignal
                del convolutedSignal
                del demodulatedSignal
            BER.append( (sum(tempBER)/len(tempBER) ))
        return BER

    # Function to perform the Viterbi MLSE simulation
    # with a Linear Declining Channel Impulse Response
    # returns the BER array
    # simulation runs for @param self.numIterations, for each
    # SNR in the range (-4,9)
    def viterbiLinearCIR(self):
        numIter = self.numIterations
        print("Conducting a MLSE BPSK simulation with a linear declining CIR")
        BER = [] #store the average BER for each SNR
        for SNR in range(-4, 9): #9
            tempBER = [] #to store all the interations for one SNR
```

```
            for count in range(0, numIter):

                # generate 300 bits
                myDataBits = self.BpskSimulation.generateBits() #raw data bits
                # map bits to symbols
                modulatedSignal = self.BpskSimulation.BpskModulate(myDataBits)
                # add tail
                for x in range(0,len(self.linC) -1):
                    modulatedSignal.append(1)
                # add convolution
                convolutedSignal = self.channelModification(self.linC, modulatedSignal)
                signalSent = self.BpskSimulation.addNoise(SNR, convolutedSignal)
                # check MLSE
                myMLSE = MLSE(self.n, signalSent, self.linC)
                myMLSE.buildTrellis()
                myMLSE.cheapestPath()
                # get data bits
                dataDetected = myMLSE.dataDetected
                # demodulate
                demodulatedSignal = self.BpskSimulation.BpskDemodulate(dataDetected)
                # compare
                tempBER.append(self.BpskSimulation.getNumErrors(myDataBits,demodulatedSignal)/self.n)
                del myMLSE
                del modulatedSignal
                del convolutedSignal
                del demodulatedSignal
            BER.append( (sum(tempBER)/len(tempBER) ))
        return BER

    # Function to perform the DFE simulation
    # with a Linear Declining Channel Impulse Response
    # returns the BER array
    # simulation runs for @param self.numIterations, for each
    # SNR in the range (-4,9)
    def dfeLinearCIR(self):
        numIter = self.numIterations
        print("Conducting a DFE BPSK simulation with a linear declining CIR")
        BER = [] #store the average BER for each SNR
        for SNR in range(-4, 9): #9
            tempBER = [] #to store all the interations for one SNR
            for count in range(0, numIter):
                # generate 300 bits
                myDataBits = self.BpskSimulation.generateBits() #raw data bits
                # map bits to symbols
                modulatedSignal = self.BpskSimulation.BpskModulate(myDataBits)
                # add convolution
                convolutedSignal = self.channelModification(self.linC,
```

```
modulatedSignal)
                signalSent = self.BpskSimulation.addNoise(SNR, convolutedSignal)
                # check DFE
                myDFE = DFE(self.n, signalSent, self.linC)
                # get data bits
                dataDetected = myDFE.getDataSymbols()
                # demodulate
                demodulatedSignal = self.BpskSimulation.BpskDemodulate(dataDetected)
                # compare
                tempBER.append(self.BpskSimulation.getNumErrors(myDataBits,demodulatedSignal)/self.n)
                del myDFE
                del modulatedSignal
                del convolutedSignal
                del demodulatedSignal
            BER.append((sum(tempBER)/len(tempBER)))
        return BER

    # Function to perform the DFE simulation
    # with a Gaussian random Channel Impulse Response
    # returns the BER array
    # simulation runs for @param self.numIterations, for each
    # SNR in the range (-4,9)
    def dfeRandomCIR(self):
        numIter = self.numIterations
        print("Conducting a DFE BPSK simulation with a Uniform Random CIR")
        BER = [] #store the average BER for each SNR
        for SNR in np.arange(-4, 9): #9
            tempBER = [] #to store all the interations for one SNR
            for count in range(0, numIter):
                # generate 300 bits
                myDataBits = self.BpskSimulation.generateBits() #raw data bits
                # map bits to symbols
                modulatedSignal = self.BpskSimulation.BpskModulate(myDataBits)
                # add convolution
                randomCIR = self.generateRandomCIR(SNR)
                convolutedSignal = self.channelModification(randomCIR, modulatedSignal)
                signalSent = self.BpskSimulation.addNoise(SNR, convolutedSignal)
                # check DFE
                myDFE = DFE(self.n, signalSent, randomCIR)
                # get data bits
                dataDetected = myDFE.getDataSymbols()
                # demodulate
                demodulatedSignal = self.BpskSimulation.BpskDemodulate(dataDetected)
                # compare
                tempBER.append(self.BpskSimulation.getNumErrors(myDataBits,demodulatedSignal)/self.n)
                del myDFE
```

```
                del modulatedSignal
                del convolutedSignal
                del demodulatedSignal
            BER.append( (sum(tempBER)/len(tempBER) ))
        return BER

    #returns a random CIR with "values" elements
    # Reqires the SNR ratio of the channel
    # uses the Gaussian random number generator implemented in practical
    1.
    def generateRandomCIR(self, SNR, values=3):
        c = []
        for x in range(0,values):
            temp = self.BpskSimulation.numberGenerator.gaussian(
    stdDeviation=self.BpskSimulation.getStdDev(SNR))
            temp = temp/(np.sqrt(3))
            c.append(temp)
            temp = 0
        return c

    # pass in just the symbols.
    # return symbols with channel repsonse.
    # function to add the effects of the channel to the stream of
    # symbols being sent. (Apply the CIR)
    def channelModification(self, cir, stream):
        returnStream = copy.deepcopy(stream)
        for x in range(0,len(stream)):
            temp1 = 0 #sk-2
            temp2 = 0 #sk-1
            if(x-2 < 0):
                temp1 = 1
            else:
                temp1 = stream[x-2]
            if(x-1 <0):
                temp2 = 1
            else:
                temp2 = stream[x-1]
            returnStream[x] = (stream[x]*cir[0] + temp2*cir[1] + temp1*
    cir[2])
        return returnStream


    # Function to perform the simulation of all 4 situations
    # DFE linear CIR, DFE Gaussian Random CIR, MLSE linear CIR and MLSE
    Gaussian Random CIR.
    # Plots all four simulations on the same plot
    def simulate(self):
        print("Plotting all")
        SNR = np.linspace(-4, 8, 13)
        viterbiLinDec = self.viterbiLinearCIR()
        viterbiRandom = self.viterbiRandomCIR()
        dfeRandom = self.dfeRandomCIR()
        dfeLin = self.dfeLinearCIR()
        plt.semilogy(SNR, viterbiLinDec, 'r-', label='Viterbi MLSE Linear
     Declining CIR')
        plt.semilogy(SNR, viterbiRandom, 'g-', label='Viterbi MLSE Random
```

```
    CIR')
        plt.semilogy(SNR, dfeRandom, 'b-', label='DFE Random CIR')
        plt.semilogy(SNR, dfeLin, 'y-', label='DFE Linear Declining CIR')
        plt.grid(which='both')
        plt.ylabel("BER")
        plt.xlabel("SNR")
        plt.title("Plot of the BER vs SNR for the BPSK Simulation
    conducted with Viterbi MLSE and DFE Equalizers")
        plt.legend(loc='best')
        plt.show()
        print("Complete")

# n = number of data bits (300)
# c = linear declining CIR
# numIterations = number of iterations to take the average of, before
    plotting - adjust as needed.
# if commented - uncomment the last line
# "mySim.simulate()" to run the simulation and retrieve the plot
# n = 300
# c = [0.89,0.42,0.12]
# numIterations = 50
# mySim = Simulation(n,c, numIterations)
# mySim.simulate()
```

Listing 5. Python Source code to execute the simulation for Multipath - Practical 2

```
#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()
```

```python
        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

```

```
        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

#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 6. Python Source code for question 3 (Simulation Platform) - From practical 1