



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 30th October, 2018

Table of Contents

1	Introduction	1
2	Theoretical Analysis	2
3	Design and Implementation	3
3.1	Transmitter	3
3.2	Transmission platform	4
3.3	Receiver	4
3.4	Simulation Platform	4
4	Results	5
5	Discussion	6
6	Conclusion	8
7	References	8
A	Appendix A: How to operate Python source code	10
A.1	Dependencies	10
A.2	Execution of the script	10
B	Python source code	10

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 $\Delta_t = |r_t^1 - c_t^1| + |r_t^2 - c_t^2| + |r_t^3 - c_t^3|$

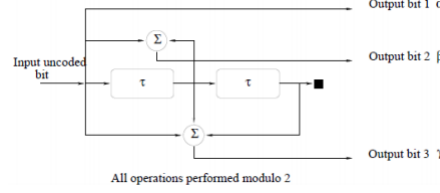


Figure 1. Convolutional Encoder Block diagram [5]

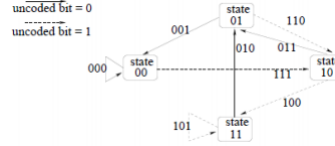


Figure 2. Convolutional Encoder State diagram [5]

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 to 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 $\frac{1}{\sqrt{10^{-\frac{SNR}{10}} 2f_{bit}}}$. 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 computed 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.

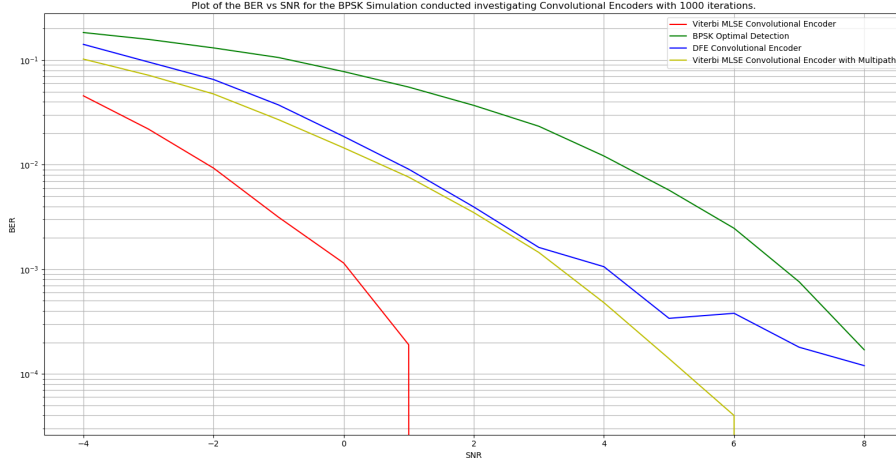


Figure 3. Simulation results for the entire simulation platform 1000 iterations

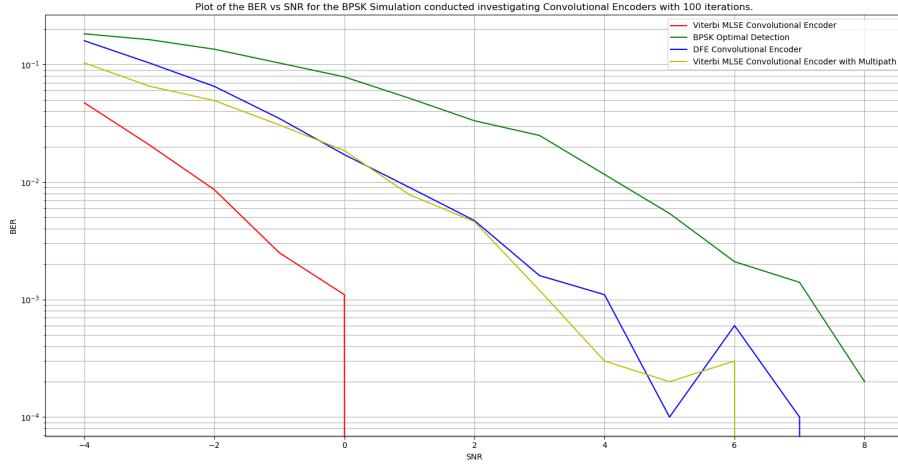


Figure 4. Simulation results for the entire simulation platform 100 iterations

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 algorithm proves to be best algorithm in decoding and estimating the information received at the receiver. 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 reliability 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 parity 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 Theory of 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 Assignment 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                    ###
```

```

6  ###          Convolutional Codes          ###
7  ###          Simulation                    ###
8  #####
9
10 import numpy as np
11 from question_3 import BPSKmodulationSimulation
12 from Simulation import Simulation
13 import copy
14 import matplotlib.pyplot as plt
15 from MLSE import MLSE
16
17 # Class to represent a single node in the Viterbi Trellis
18 # used for the MLSE Equalizer
19 # Contains the time, the alpha value(sum of deltas along cheapest path to
    node),
20 # the state of the node, all previous states it is connected to and all
    next states (time > node's time)
21 # it is connected to
22 class trellisNode:
23     def __init__(self, state, time):
24         self.nextStates = []
25         self.previousStates = []
26         self.time = time
27         self.state = state
28         self.deltas = [] # delta array corresponding to the order of
    previous States
29         self.alpha = 0
30         #Add a new state that the current node is connected to.
31
32     def addNext(self, state):
33         self.nextStates.append(state)
34         # Add a previous state that the node is connected to
35
36     def addPrevious(self, state):
37         self.previousStates.append(state)
38
39
40 ##### begin of class #####
41
42
43 # Class to perform a simulation to determine the
44 # average BER for the Viterbi MLSE and DFE symbol
45 # estimation methods for Convolutional Encoders as
46 # well as the performance of a linear Convolutional encoder
47 # taking into account the effects of Multipath with a linearly declining
    CIR
48 # The class takes in the number of data
49 # bits the generator polynomial, the constraint length and the number of
    Iterations to average the results over
50 # as constructor parameters
51 class convolutionSim:
52     def __init__(self, n = 5, generator = [4,6,7], K = 3, numIter = 50):
53         self.symbols = [1, -1]
54         self.BpskSimulation = BPSKmodulationSimulation(n)
55         self.K = K #constraint length
56         self.n = n #uncoded data bits

```

```

57     self.generator = generator #generator matrix - length of which is
    number of outputs
58     self.numIter = numIter
59
60     # Function to encode the given source bitStream using the
61     # given generator polynomial with a Linear Convolutional encoding
    scheme
62     # Returns the encoded parity bits
63     def encode(self, generator = [4,6,7], bitStream = []):
64         shiftRegister = []
65         genBinary = []
66         #get binary representation of generator
67         for x in range(0, len(generator)):
68             genBinary.append("{0:b}".format(generator[x]))
69
70         #sort shift register to begin in state 0
71         for x in range(0, len(genBinary[x])):
72             shiftRegister.append(0)
73         # list to contain all parity bits
74         parity = []
75         #encode the data
76         # for every bit in the bitStream, insert into shift register
77         # take shift register elements, multiply by generator in binary%2
78         for x in range(0, len(bitStream)):
79             #print("Inserting", bitStream[x])
80             shiftRegister.insert(0, bitStream[x])
81             shiftRegister.pop()
82             #print("Shift after insert:", shiftRegister)
83             temp = 0
84             for y in range(0, len(generator)):
85                 #print("Output bit ", y)
86                 temp = 0
87                 for z in range(0, len(shiftRegister)):
88                     temp += shiftRegister[z]*int(genBinary[y][z]).
    __round__()
89                 #print("TEMP", temp)
90                 temp = temp % 2
91                 #print("temp after mod 2", temp)
92                 parity.append(temp)
93                 #print("Parity so far", parity)
94         return parity
95
96     #Function to build the Viterbi Trellis for decoding a Linear
    Convolutional encoded
97     # block of bits.
98     # The parity bits received are passed in as function parameters as
    well as the
99     # desired start state
100    # Returns the constructed Viterbi Trellis
101    def buildTrellis(self, parity, startState = [-1, -1]):
102        #build the trellis
103        Trellis = np.empty(shape=(2*(self.K-1), self.n+self.K), dtype=
    trellisNode)
104        # print(len(Trellis))
105        #print(len(Trellis[0]))
106        Trellis[0][0] = trellisNode(startState, 0)

```

```

107     #print("FIRST ONE", Trellis[0][0].state)
108     numBits = len(startState)
109     #for every node in the trellis compute the nodes it
110     # will transition to
111     for t in range(0, self.n+self.K-1):
112         for s in range(0, (2**self.K-1)):
113             #print(Trellis[0][0].state)
114             # check if this is none
115             if Trellis[s][t] is None:
116                 continue
117             else:
118                 #get all states:
119                 allStates = []
120                 tempState = []
121                 # #only upward transitions
122                 if (t >= self.n+self.K-3):
123                     #print("HERE")
124                     tempState.append(-1)
125                     for x in range(0, numBits-1):
126                         tempState.append(Trellis[s][t].state[x])
127                     allStates.append(tempState)
128
129                 else:
130                     for x in range(0, len(self.symbols)):
131                         tempState = []
132                         tempState.append(self.symbols[x])
133                         #print(tempState, "TEMPSTATE")
134                         for y in range(0, numBits-1):
135                             tempState.append(Trellis[s][t].state[y])
136                         allStates.append(tempState)
137
138                 #print(allStates)
139                 #now we have all states
140                 for index in range(0, len(allStates)):
141                     #print("HERE")
142                     #print(t)
143                     newState = allStates[index]
144                     #print(newState, "NEW STATE")
145                     stateIndex = self.getStateIndex(newState)
146                     prevNode = Trellis[s][t]
147                     #print("Start Index", stateIndex)
148                     if Trellis[stateIndex][t+1] is None:
149                         Trellis[stateIndex][t+1] = trellisNode(
150 newState, t+1)
151
152                         #print("newStateHERE", newState)
153
154                         newNode = Trellis[stateIndex][t+1]
155                         prevNode.addNext(newNode)
156                         delta = self.getDelta(t+1, prevNode.state,
157 newState, parity)
158
159                         newNode.deltas.append(delta)
160                         newNode.alpha = delta
161                         newNode.addPrevious(prevNode)
162
163         return Trellis
164
165 #Function to decode the recieved parity bits using the Viterbi MLSE
166 algorithm for a Linear Convolutional encoded

```

```

160 # block of bits.
161 # The parity bits recieved are passed in as function paramters as
    well as the
162 # desired start state
163 # Returns the decoded source information
164 def mlseDecode(self ,parity ,startState = [-1,-1]):
165     Trellis = self.buildTrellis(parity ,startState)
166     for t in range(0, self.n+self.K):
167         for s in range(0, (2**self.K)):
168             if (Trellis[s][t] is None):
169                 continue
170             #if more than one state connected (contending)
171             if (len(Trellis[s][t].previousStates) > 1):
172                 #get smallest delta
173                 #delete all other previous states, all other deltas,
store complete path alpha
174                 bestIndex = 0 #index of node with the best or
shortest path so far
175                 #just set to first
176                 bestAlpha = Trellis[s][t].previousStates[0].alpha +
Trellis[s][t].deltas[0]
177                 for x in range(0, len(Trellis[s][t].previousStates)):
178                     tempAlpha = Trellis[s][t].previousStates[x].alpha
+ Trellis[s][t].deltas[x]
179                     if (tempAlpha <= bestAlpha):
180                         bestIndex = x
181                         bestAlpha = tempAlpha
182                         Trellis[s][t].alpha = bestAlpha
183                         Trellis[s][t].deltas = [ Trellis[s][t].deltas[
bestIndex] ]
184                         Trellis[s][t].previousStates = [ Trellis[s][t].
previousStates[bestIndex] ]
185                     else:
186                         #compute the alpha
187                         if (len(Trellis[s][t].previousStates) == 0):
188                             continue
189                         Trellis[s][t].alpha += Trellis[s][t].previousStates
[0].alpha
190                         myTemp = Trellis[0][-1]
191
192                         #print(myTemp.state)
193                         detected = [myTemp.state[0]] #this is just from the trellis
194                         #print("State left at time", myTemp.time, "is", myTemp.state)
195                         myTemp = myTemp.previousStates[0]
196                         while (myTemp.time > 0):
197                             #print("State left at time", myTemp.time, "is", myTemp.state)
198                             detected = [myTemp.state[0]] + detected
199                             myTemp = myTemp.previousStates[0]
200                             #detected = [myTemp.state[0]] + detected #add t=0
201                             #print("detected", detected)
202                             return detected
203
204 # Return the state index [-1,-1] = 0 and [1,-1] = 1 , and so forth
205 def getStateIndex(self , state):
206     if (state == [1, 1]):
207         return 3

```



```

208         if(state == [1, -1]):
209             return 2
210         if(state == [-1, 1]):
211             return 1
212         if(state == [-1, -1]):
213             return 0
214     # Returns the delta value for time = time, from
215     # state1 to state2 for the given parity bits recieved
216     def getDelta(self, time, state1, state2, parityRec):
217         #first need to get parity symbols at time
218         #get output symbols
219         # print("THE PARITY AFTER SENT", parityRec)
220         # print("Getting Delta for time", time)
221         # print("State1", state1)
222         # print("state2", state2)
223         outputSymbols = self.getOutputSymbols(state1, state2)
224         #print("Output Symbols", outputSymbols)
225         recParity = []
226         for x in range((time-1)*len(self.generator), (time-1)*len(self.
generator)+len(self.generator)):
227             recParity.append(parityRec[x])
228             temp = 0
229             #print("Rec parity", recParity)
230             for x in range(0, len(recParity)):
231                 temp += np.abs(recParity[x]-outputSymbols[x])**2
232             return temp
233
234     # Returns the output symbols as per the state diagram
235     # for a transition from state 1 to state2
236     #state 1 is the original, state 2 is the new state
237     def getOutputSymbols(self, state1, state2):
238         shiftRegister = copy.deepcopy(state1)
239         #insert the first bit of state
240         shiftRegister.insert(0, copy.deepcopy(state2[0]))
241         # shiftRegister.pop()
242         genBinary = []
243         for x in range(0, len(shiftRegister)):
244             if(shiftRegister[x] == -1):
245                 shiftRegister[x] = 0
246         #get binary representation of generator
247         for x in range(0, len(self.generator)):
248             genBinary.append("{0:b}".format(self.generator[x]))
249         # list to contain all parity bits
250         parity = []
251         temp = 0
252         #print("Shift register", shiftRegister)
253         for y in range(0, len(self.generator)):
254             temp = 0
255             for z in range(0, len(shiftRegister)):
256                 #print("temp +=" , shiftRegister[z], "*", int(genBinary[y]
][z]).__round__())
257                 temp += shiftRegister[z]*int(genBinary[y][z]).__round__()
258             #print("TEMP", temp)
259             temp = temp % 2
260             if(temp == 0):
261                 temp = -1

```

```

262         #print("temp after mod 2", temp)
263         parity.append(temp)
264         #print("Parity so far", parity)
265     #print()
266     return parity
267 # Function to perform the simulation using the MLSE Viterbi algorithm
268 # to decode
269 # No multipath effects taken into account
270 # Returns the BER list
271 def SimulateMLSE(self):
272     numIter = self.numIter
273     print("Conducting a MLSE BPSK simulation with Convolutional
274 Encoding")
275     BER = [] # store the average BER for each SNR
276     for SNR in range(-4, 9): #9
277         tempBER = [] #to store all the iterations for one SNR
278         for count in range(0, numIter):
279             myDataBits = self.BpskSimulation.generateBits()
280             #append tail
281             for x in range(0, self.K-1):
282                 myDataBits.append(0)
283             encodedBits = self.encode(self.generator, myDataBits)
284             modulatedSignal = self.BpskSimulation.BpskModulate(
285 encodedBits)
286             signalSent = self.BpskSimulation.addNoise(SNR,
287 modulatedSignal)
288             decodedSymbols = self.mlseDecode(signalSent)
289             decodedBits = self.BpskSimulation.BpskDemodulate(
290 decodedSymbols)
291             tempBER.append(self.BpskSimulation.getNumErrors(
292 myDataBits, decodedBits)/(self.n))
293             BER.append((sum(tempBER)/len(tempBER)))
294         print(BER)
295     return BER
296 # Function to perform the simulation using BPSK modulation, no
297 # multipath, no Convolutional encoding
298 # Returns the BER list
299 def simulateBPSK(self):
300     numIter = self.numIter
301     print("Conducting a BPSK simulation with no Convolutional
302 Encoding")
303     BER = [] #store the average BER for each SNR
304     for SNR in range(-4, 9): #9
305         tempBER = [] #to store all the iterations for one SNR
306         for count in range(0, numIter):
307             # generate 300 bits
308             myDataBits = self.BpskSimulation.generateBits() #raw data
309             bits
310             # map bits to symbols
311             modulatedSignal = self.BpskSimulation.BpskModulate(
312 myDataBits)
313             signalSent = self.BpskSimulation.addNoise(SNR,
314 modulatedSignal)
315             detectedSignal = self.BpskSimulation.detectSignal(SNR,
316 signalSent)
317             # demodulate

```

```

306         demodulatedSignal = self.BpskSimulation.BpskDemodulate(
detectedSignal)
307         # compare
308         tempBER.append(self.BpskSimulation.getNumErrors(
myDataBits, demodulatedSignal)/self.n)
309         BER.append( (sum(tempBER)/len(tempBER) ))
310         print(BER)
311         return BER
312
313     #Function to decode the recieved parity bits using the DFE algorithm
for a Linear Convolutional encoded
314     # block of bits.
315     # The parity bits recieved are passed in as function paramters as
well as the
316     # desired start state
317     # Returns the decoded source information
318     def dfeDecode(self, parity, startState = [-1,-1]):
319         detected = []
320         detectedState = startState
321         #detected.append(detectedState[0])
322         for x in range(0, self.n+self.K):
323             tempState1 = detectedState
324             if(self.getDelta(x, tempState1, [-1, tempState1[0]], parity)
> self.getDelta(x, tempState1, [1, tempState1[0]], parity)):
325                 detectedState = [1, tempState1[0]]
326             else:
327                 detectedState = [-1, tempState1[0]]
328             detected.append(detectedState[0])
329         return detected[1:]
330
331     # Function to perform the simulation using the DFE alorhtm to decode
332     # No multipath effects taken into account
333     # Returns the BER list
334     def SimulateDfe(self):
335         numIter = self.numIter
336         print("Conducting a DFE BPSK simulation with Convolutional
Encoding")
337         BER = [] # store the average BER for each SNR
338         for SNR in range(-4, 9): #9
339             tempBER = [] #to store all the interations for one SNR
340             for count in range(0, numIter):
341                 myDataBits = self.BpskSimulation.generateBits()
342                 #print("Data Bits:", myDataBits)
343                 #append tail
344                 for x in range(0, self.K-1):
345                     myDataBits.append(0)
346                 #print("Data Bits after append:", myDataBits)
347                 #print("Encoding")
348                 encodedBits = self.encode(self.generator, myDataBits)
349                 #print("Encoded bits", encodedBits)
350                 modulatedSignal = self.BpskSimulation.BpskModulate(
encodedBits)
351                 #print("Modulated Encoded", modulatedSignal)
352                 signalSent = self.BpskSimulation.addNoise(SNR,
modulatedSignal)
353                 decodedSymbols = self.dfeDecode(signalSent)

```

```

354         #print("Decoded Symbols", decodedSymbols)
355         decodedBits = self.BpskSimulation.BpskDemodulate(
decodedSymbols)
356         #print("decoded Bits", decodedBits)
357
358         tempBER.append(self.BpskSimulation.getNumErrors(
myDataBits, decodedBits)/(self.n))
359         BER.append((sum(tempBER)/len(tempBER)))
360         print(BER)
361         return BER
362     # Function to perform the simulation using the MLSE Viterbi algorithm
to decode
363     # Multipath effects taken into account
364     # Returns the BER list
365     def simulateMultiPath(self):
366         numIter = self.numIter
367         multiSim = Simulation(n=self.n*len(generator), linC=[0.89, 0.42,
0.12])
368         print("Conducting a Multipath BPSK simulation with Convolutional
Encoding")
369         BER = [] # store the average BER for each SNR
370         for SNR in range(-4, 9): #9
371             tempBER = [] #to store all the iterations for one SNR
372             for count in range(0, numIter):
373                 myDataBits = self.BpskSimulation.generateBits()
374                 #print("Original Data", myDataBits)
375                 #append tail
376                 for x in range(0, self.K-1):
377                     myDataBits.append(0)
378                 #print("ORIGINAL WITH APPEND", myDataBits)
379                 encodedBits = self.encode(self.generator, myDataBits)
380                 #print("Encoded", encodedBits)
381                 modulatedSignal = self.BpskSimulation.BpskModulate(
encodedBits)
382                 convolutedSignal = multiSim.channelModification(multiSim.
linC, modulatedSignal)
383                 signalSent = self.BpskSimulation.addNoise(SNR,
convolutedSignal)
384                 myMLSE = MLSE(self.n*len(generator), signalSent, [0.89,
0.42, 0.12])
385                 myMLSE.buildTrellis()
386                 myMLSE.cheapestPath()
387                 # get data bits
388                 dataDetected = myMLSE.dataDetected
389                 #print("Data Detected", dataDetected)
390                 decodedSymbols = self.mlseDecode(signalSent)
391                 decodedBits = self.BpskSimulation.BpskDemodulate(
decodedSymbols)
392                 #print("Decoded bits", decodedBits)
393                 tempBER.append(self.BpskSimulation.getNumErrors(
myDataBits, decodedBits)/(self.n))
394                 BER.append((sum(tempBER)/len(tempBER)))
395                 print(BER)
396                 return BER
397     # Fuction to perform and plot all the implemented simulations on a
single curve

```

```

398     def plotAll(self):
399         print("Plotting all for", self.numIter, "iterations")
400         SNR = np.linspace(-4, 8, 13)
401         MlseConv = self.SimulateMLSE()
402         bpskSim = self.simulateBPSK()
403         dfeConv = self.SimulateDfe()
404         mlseMultiPath = self.simulateMultiPath()
405         plt.semilogy(SNR, MlseConv, 'r-', label='Viterbi MLSE
Convolutional Encoder')
406         plt.semilogy(SNR, bpskSim, 'g-', label='BPSK Optimal Detection')
407         plt.semilogy(SNR, dfeConv, 'b-', label='DFE Convolutional Encoder
')
408         plt.semilogy(SNR, mlseMultiPath, 'y-', label='Viterbi MLSE
Convolutional Encoder with Multipath')
409         plt.grid(which='both')
410         plt.ylabel("BER")
411         plt.xlabel("SNR")
412         title = "Plot of the BER vs SNR for the BPSK Simulation conducted
investigating Convolutional Encoders with " + str(self.numIter) + "
iterations."
413         plt.title(title)
414         plt.legend(loc='best')
415         plt.show()
416         print("Complete")
417         return
418 ##### end of class #####
419
420 #uncomment last line to conduct a simulation of all simulations
implemented
421 n = 100 #number unenconded bits
422 numIter = 15 #number of iterations to avergae the results over
423 generator = [4, 6, 7] #generator polynomial
424 K = 3 #constraint length
425 tempSim = convolutionSim(n,generator,K, numIter)
426 tempSim.plotAll()

```

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

```

1 # Mohamed Ameen Omar
2 # 16055323
3 #####
4 ###      EDC 310 Practical 2      ###
5 ###      2018                    ###
6 ###      BPSK DFE Algotihm      ###
7 #####
8
9 import numpy as np
10 import copy
11
12 # Class to run a DFE equalizer to dermine the bits sent over a AGWN
channel
13 # Constructor paramaters:
14 # @param N = the number of data bits of data being sent
15 # @param r = a vector with all the recieved symbols
16 # @param c = the channel impulse response vector

```

```

17 class DFE:
18     def __init__(self, N = 0, r = [], c = []):
19         self.n = N #number of data bits
20         self.r = r #recieved vector - only data bits len(r) = self.n
21         self.c = c #convolution matrix
22         self.L = len(c) #length of the c vector
23         self.numHeader = self.L-1 #number of header bits
24         self.symbols = [1,-1]
25         self.dataDetected = [] #just the data bits detected
26
27     # Function to return the data symbols detected.
28     # It will detect or estimate or symbols recieved and return
29     # a vector with those symbols
30     def getDataSymbols(self):
31         if(self.dataDetected == []):
32             self.detectSymbols()
33             return self.dataDetected
34         else:
35             return self.dataDetected
36
37     # Function to detect the symbols in the recieved vector
38     # using the DFE Equalizer
39     def detectSymbols(self):
40         for x in range(0,len(self.r)):
41             self.dataDetected.append(self.getSymbol(x))
42     # Function to detect a single sumbol using
43     # DFE Equalizer algorithm
44     def getSymbol(self,time):
45         if(self.getDelta(time, 1) > self.getDelta(time, -1)):
46             return -1
47         else:
48             return 1
49
50     # Function to get the delta value for a single symbol
51     # symbol is the symbol we are estimating
52     # time is the time instance for the symbol we are getting the delta
53     def getDelta(self,time,symbol):
54         temp = 0
55         t = 0
56         for x in range(0,self.L):
57             if(x == 0):
58                 temp += self.c[x]*symbol
59             else:
60                 #if we havent detected the first l symbols
61                 if(len(self.dataDetected)+t <0):
62                     temp += self.c[x]*1
63                 else:
64                     temp += self.c[x]*self.dataDetected[len(self.
dataDetected)+t]
65                 t = t-1
66         temp = np.abs(self.r[time] - temp) **2
67         return temp
68 ##### END OF CLASS IMPLEMENTATION

```

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

```

1  # Mohamed Ameen Omar
2  # 16055323
3  #####
4  ###      EDC 310 Practical 2      ###
5  ###              2018              ###
6  ###      MLSE- Viterbi Algotihm      ###
7  #####
8
9  import numpy as np
10 from question_3 import BPSKmodulationSimulation
11 import copy
12
13 '''
14 MY IMPLEMENTATION
15
16
17 trellisNode store the time, the state, the next states, the alpha, the
18     previous states of a node
19
20 first pass in the r,n,c
21
22 then it builds it by:
23 assigniedn fist node to 11 at t=0
24 then for every node, create its transitons and update all
25 including all alphas and deltas
26
27 to get path:
28 from left to right
29 check if there's contending, if not just add previous alpha to current
30 if there is, get alphas for each one, get smallest, remove all other
31     deltas and continue until end
32 '''
33
34 # Class to represent a single node in the Viterbi Trellis
35 # used for the MLSE Equalizer
36 # Contains the time, the alpha value(sum of deltas along cheapest path to
37     node),
38 # the state of the node, all previous states it is connected to and all
39     next states (time > node's time)
40 # it is connected to
41 class trellisNode:
42     def __init__(self, state, time):
43         self.nextStates = []
44         self.previousStates = []
45         self.time = time
46         self.state = state
47         self.deltas = [] #delta array corresponding to the order of
48     previous States
49         self.alpha = 0
50         #Add a new state that the current node is connected to.
51         def addNext(self, state):

```

```

47         self.nextStates.append(state)
48     # Add a previous state that the node is connected to
49     def addPrevious(self, state):
50         self.previousStates.append(state)
51     ##### END OF CLASS #####
52
53 #class that conducts the MLSE algorithm for ANY BPSK modulated signal
54 # Requires:
55 # @param N = the number of data bits in the recieved signal
56 # @param r = the recieved vector of bits
57 # @param c = the Channel Impulse response for the channel
58
59 # it will build the trellis, thereafter calculate all deltas for all
60 # states or nodes in the trellis.
61 # then compute the cheapest path and disregard any nodes eliminated
62 class MLSE:
63     def __init__(self, N = 0, r = [], c = []):
64         self.n = N #data bits
65         self.symbols = [1, -1] # modulation symbols
66         self.r = r #recieved symbols
67         self.c = c #Convolution matrix
68         self.trellisLength = self.n + len(self.c) - 1 #Trellis will be up
69         # to time T
70         self.numStates = len(self.symbols)**(len(self.c) - 1)
71         self.numHT = len(c)-1
72         self.Trellis = np.empty(shape=(self.numStates, self.trellisLength
73 +1), dtype=trellisNode)
74         self.detected = [] #detected symbols from trellis
75         self.entireStream = [] #entire detected stream including head and
76         # tail
77         self.dataDetected = [] #data bits detected
78
79 #print all properties of the problem to which MLSE is applied
80 def printProperties(self):
81     print("Number of data bits:", self.n)
82     print("Signal recieved:", self.r)
83     print("Convolution Matrix (C): ", self.c)
84     print("Trellis Length (L): ", self.trellisLength)
85     print("Number of states:", self.numStates)
86     print("Number of Head and Tail symbols:", self.numHT)
87     print("Modulation Symbols:", self.symbols)
88
89 # Function to build the viterbi trellis
90 # will begin with the first node state being = 1,1; following
91 # convention.
92 # assigning fist node to 11 at t=0
93 # then for every node, create its transits and update all
94 # including all alphas and deltas
95 def buildTrellis(self, startState = [1,1]):
96     self.Trellis[0][0] = trellisNode(startState, 0)
97     numBits = len(startState)
98     #for every node in the trellis compute the nodes it
99     # will transition to
100     for t in range(0, self.trellisLength):
101         for s in range(0, self.numStates):
102             # check if this is none

```



```

98         if self.Trellis[s][t] is None:
99             continue
100     else:
101         #get all states:
102         allStates = []
103         tempState = []
104         #only upward transitions
105         if (t >= self.trellisLength-2):
106             tempState.append(1)
107             for x in range(0,numBits-1):
108                 tempState.append(self.Trellis[s][t].state[x])
109             allStates.append(tempState)
110         #both 1,-1
111         else:
112             for x in range(0,len(self.symbols)):
113                 tempState = []
114                 tempState.append(self.symbols[x])
115                 for y in range(0, numBits-1):
116                     tempState.append(self.Trellis[s][t].state
117 [y])
118                     allStates.append(tempState)
119         #now we have all states
120         for index in range(0,len(allStates)):
121             newState = allStates[index]
122             stateIndex = self.getStateIndex(newState)
123             prevNode = self.Trellis[s][t]
124             if self.Trellis[stateIndex][t+1] is None:
125                 self.Trellis[stateIndex][t+1] = trellisNode(
126 newState,t+1)
127
128                 newNode = self.Trellis[stateIndex][t+1]
129                 prevNode.addNext(newNode)
130                 delta = self.computeDelta(t+1,prevNode.state,
131 newState)
132
133                 newNode.deltas.append(delta)
134                 newNode.alpha = delta
135                 newNode.addPrevious(prevNode)
136
137     # Function to print all the deltas for all the nodes in the Viterbi
138     Trellis
139     def printAllDeltas(self):
140         for t in range(1,self.trellisLength+1):
141             for s in range(0,self.numStates):
142                 myNode = self.Trellis[s][t]
143                 if (myNode is None):
144                     continue
145                 for prev in range(0, len(myNode.previousStates)):
146                     print("Delta", myNode.time, "from", self.
147 getStateIndex(myNode.previousStates[prev].state), "to",self.
148 getStateIndex(myNode.state), "is", myNode.deltas[prev] )
149                 print()
150
151     # Return the state index [1,1] = 0 and [1,-1] = 1 , and so forth
152     def getStateIndex(self, state):
153         if (state == [1,1]):
154             return 0

```

```

148         if(state == [1, -1]):
149             return 1
150
151         if(state == [-1, 1]):
152             return 2
153
154         if(state == [-1, -1]):
155             return 3
156
157
158     # Function to perform a delta calculation from state1 to state 2
159     @time = @param time
160     #state 1 is the state originating, state 2 is the state going to
161     def computeDelta(self, time, state1, state2):
162         temp = 0
163         for x in range(0, len(self.c)):
164             if(x < len(state2)):
165                 temp += self.c[x]*state2[x]
166             else:
167                 temp += self.c[x]*state1[x-len(state2) +1]
168         temp = np.abs(self.r[time-1] - temp)**2
169         return temp
170
171     # Function to calculate the cheapest path and in extension estimate
172     # the
173     # symbols recieved. Must build the trellis before calling this
174     # function
175     # to get path:
176     # from left to right
177     # check if there's contending, if not just add previous alpha to
178     # current
179     # if there is, get alphas for each one, get smallest, remove all
180     # other deltas and continue until end
181     def cheapestPath(self):
182         for t in range(0, self.trellisLength+1):
183             for s in range(0, self.numStates):
184                 if(self.Trellis[s][t] is None):
185                     continue
186                 #if more than one state connected (contending)
187                 if(len(self.Trellis[s][t].previousStates) > 1):
188                     #get smallest delta
189                     #delete all other previous states, all other deltas,
190                     store complete path alpha
191                     bestIndex = 0 #index of node with the best or
192                     shortest path so far
193                     #just set to first
194                     bestAlpha = self.Trellis[s][t].previousStates[0].
195                     alpha + self.Trellis[s][t].deltas[0]
196                     for x in range(1, len(self.Trellis[s][t].
197                     previousStates)):
198                         tempAlpha = self.Trellis[s][t].previousStates[x].
199                         alpha + self.Trellis[s][t].deltas[x]
200                         if(tempAlpha < bestAlpha):
201                             bestIndex = x
202                             bestAlpha = tempAlpha
203                     self.Trellis[s][t].alpha = bestAlpha

```

```

194         self.Trellis[s][t].deltas = [ self.Trellis[s][t].
deltas[bestIndex] ]
195         self.Trellis[s][t].previousStates = [ self.Trellis[s]
][t].previousStates[bestIndex] ]
196     else:
197         #compute the alpha
198         if(len(self.Trellis[s][t].previousStates) == 0):
199             continue
200         self.Trellis[s][t].alpha += self.Trellis[s][t].
previousStates[0].alpha
201
202     myTemp = self.Trellis[0][-1]
203     self.detected = [myTemp.state[0]] + self.detected #this is just
from the trellis
204     myTemp = myTemp.previousStates[0]
205     while(myTemp.time> 0 ):
206         self.detected = [myTemp.state[0]] + self.detected
207         myTemp = myTemp.previousStates[0]
208         self.detected = [myTemp.state[0]] + self.detected #add t=0
209         self.entireStream = copy.deepcopy(self.detected) #with head and
tail
210     while(len(self.entireStream) != (self.n+self.numHT+self.numHT)):
211         self.entireStream = [1] + self.entireStream
212
213     streamLength = len(self.entireStream)
214     for x in range(0,streamLength):
215         if(x < self.numHT or streamLength-x <= self.numHT):
216             continue
217         else:
218             self.dataDetected.append(self.entireStream[x])
219
220     # Start with building the trellis
221     # have the deltas and all previous and next states for each node
222     # compute the alpha values for all = total path cost including the
current node so far
223     '''
224     go through from the last and add the remaining nodes in the Trellis
that are connected
225     '''
226
227 ##### END OF CLASS #####

```

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

```

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
17 class randomNumber:
18     #Constructor
19     def __init__(self, seed1 = time.time(), seed2 = time.time()-10, seed3
    = time.time()+10):
20         self.seed1 = seed1
21         self.seed2 = seed2
22         self.seed3 = seed3
23         self.computed = None
24
25     #Uniform Distribution Wichmann-Hill algorithm
26     def WHill(self):
27         # Seed values must be greater than zero
28
29         self.seed1 = 171 * (self.seed1 % 177) - (2*(self.seed1/177))
30         if self.seed1 < 0:
31             self.seed1 = self.seed1 + 30269
32
33         self.seed2 = 172 * (self.seed2 % 176) - (35*(self.seed2/176))
34         if self.seed2 < 0:
35             self.seed2 = self.seed2 + 30307
36
37         self.seed3 = 170 * (self.seed3 % 178) - (63*(self.seed2/178))
38         if self.seed3 < 0:
39             self.seed3 = self.seed3 + 30323
40
41         temp = float(self.seed1/30269) + float(self.seed2/30307) + float(
    self.seed3/30323)
42         # So that the output is between 0 and 1
43         return (temp%1)
44
45     # Normal Distribituion random number generator.
46     # Only used in Question 2 and Question 3.
47     def gaussian(self, mean = 0, stdDeviation = 1):
48         if(self.computed is not None):
49             returnVal = self.computed
50             self.computed = None
51             return returnVal
52         else:
53             squaredSum = -1.0
54             temp1 = 0.0
55             temp2 = 0.0
56             # find two numbers such that their squared sum falls within
    the
57             # boundaries of the square.
58             while( (squaredSum >= 1) or squaredSum == -1.0 ):
59                 temp1 = (2*self.WHill())-1
60                 temp2 = (2*self.WHill())-1
61                 squaredSum = (temp1**2) + (temp2 **2)
62
63             mul = np.sqrt(-2.0*np.log(squaredSum)/squaredSum)
64

```

```

65         #store the second point to avoid wasting computation time
66         self.computed = mean + (stdDeviation * temp1 * mul)
67         return (mean + (stdDeviation*temp2*mul))
68
69     def rejection(self, mean = 0, stdDev = 1):
70         a1 = 0.8638
71         a2 = 0.1107
72         a3 = 0.0228002039
73         a4 = 1-a1-a2-a3
74         myProb = self.WHill()
75
76
77         if(myProb < a1):
78             temp = self.WHill()
79             temp2 = self.WHill()
80             temp3 = self.WHill()
81             return (2*(temp+temp2+temp3-1.5))
82
83         if(myProb < (a1+a2)):
84             temp = self.WHill()
85             temp2 = self.WHill()
86             return (1.5*(temp+temp2-1))
87
88         if(myProb < (a1+a2+a3)):
89             temp = self.WHill()
90             temp2 = self.WHill()
91             x = (6*temp) -3
92             y = 0.358*(temp2)
93
94             while(y < self.g3(x)):
95                 temp = self.WHill()
96                 temp2 = self.WHill()
97                 x = (6*temp) - 3
98                 y = 0.358*(temp2)
99             return x
100         else:
101
102             v1 = self.WHill()*2 -1
103             v2 = self.WHill()*2 -1
104             x = 0
105             y = 0
106             while(np.abs(x) < 3 and np.abs(y) < 3):
107                 v1 = self.WHill()*2 - 1
108                 v2 = self.WHill()*2 - 1
109                 print("FUCK")
110                 while(((v1**2) + (v2**2)) > 1 or ((v1**2) + (v2**2)) ==
111 1):
112                     print("FUCK2")
113                     v1 = self.WHill()*2 -1
114                     v2 = self.WHill()*2 -1
115
116                     temp = (v1**2) + (v2**2)
117                     temp = np.log(temp)
118                     temp = -2*temp
119                     temp = temp+9
120                     temp = temp/((v1**2) +(v2**2))

```

```

120         x = v1*np.sqrt(temp)
121         y = v2*np.sqrt(temp)
122         if(np.abs(x) > 3):
123             return x
124         return y
125
126
127
128
129     def g3(self ,x):
130         if(np.abs(x) < 1):
131             return (17.49731196*np.exp(-0.5-(x**2)) -4.73570326*(3-(x**2)
132 ) -2.15787544*(1.5-np.abs(x) ))
133
134         if(np.abs(x) > 1 and (np.abs(x) <1.5)):
135             return (17.49731196*np.exp(-0.5-(x**2)) - 2.36785163*( (3-(x
136 ) **2) - 2.15787544*(1.5-np.abs(x)))
137
138         if(np.abs(x) > 1.5 and (np.abs(x) < 3)):
139             return (17.49731196*np.exp(-0.5-(x**2)) - 2.36785163*( (3-(x
140 ) **2) - 2.15787544*(1.5-np.abs(x)))
141
142         return 0
143
144 ##### End of class #####
145 # function to plot the PDF for the Uniformly distributed random number
146 # generator
147 def plotUniformPDF(sample = None, binSize = 200, save = True, iswHill =
148 True):
149     if(sample is None):
150         print("Error, sample to plot not provided")
151         return
152
153     title = ("Probability Density Function of the ")
154     # if the sample passed in was generated from the Wichmann-Hill
155     algorithm
156     if(iswHill is True):
157         title = title + "Wichmann-Hill Random Number Generator"
158     # if the sample passed in was generated from the python uniform
159     random number generator
160     else:
161         title = title + "Python Uniform Random Number Generator"
162     fileName = title
163     title = title + " with a sample size of " + str(len(sample)) + " and
164     a bin size of " + str(binSize) + " bins"
165     fig = plt.figure()
166     plt.hist(sample,color = "Blue", edgecolor = "black", bins = binSize ,
167 density = True)
168     plt.xlabel("Random Number")
169     plt.ylabel("Probability")
170     plt.title(title)
171     #Plot a real Uniform PDF for comparison
172     plt.plot([0.0 , 1.0], [1,1], 'r-', lw=2, label='Actual Uniform PDF')

```

```

167     plt.legend(loc='best')
168     plt.show()
169
170     #to save the plot
171     if(save is True):
172         fileName = fileName + ".png"
173         fig.savefig(fileName , dpi=fig.dpi)
174
175     # function to print the relevant statistics for the sample passed in
176     # The mean, standard deviation and variance of the sample is computed
177     # and displayed to the screen
178     def printStats(sample = None, isWHill = True):
179         if(sample is None):
180             print("Error, sample not provided, no statistics to print")
181             return
182         message = ("The Mean, Standard Deviation and Variance for the ")
183         if(isWHill is True):
184             message = message + "Wichmann-Hill Uniformly Distributed Random
Number Generator"
185         else:
186             message = message + "Built-in Python Uniformly Distributed Random
Number Generator"
187         print(message)
188         print("Mean: " + str(np.mean(sample)))
189         print("Standard Deviation: " + str(np.std(sample)))
190         print("Variance: " + str(np.var(sample)))
191
192
193     def question1(sampleSize = 10000000, binSize = 200, save = True):
194         #input parameter validation
195         #all paramters must have a postive value
196         if(sampleSize < 0):
197             sampleSize = 10000000
198         if(binSize < 0):
199             binSize = 200
200         #create a randomGenerator object to get Uniform distribution random
numbers
201         # For consistent results , pass seed paramters into the constructor
202         randomGenerator = randomNumber() #create a random number object to
generate uniform random number
203         sampleSpace = []
204
205         print("Question 1:")
206
207         #Generate Sample Space for Wichmann-Hill
208         print("Generating Sample Space with Wichmann-Hill RNG")
209         print("Sample space contains", sampleSize, "entries")
210         print("The bin size is", binSize, "bins")
211         for x in range(0, sampleSize):
212             sampleSpace.append(randomGenerator.WHill())
213         print("Task Complete")
214
215         print("Plotting Wichmann-Hill PDF")
216         plotUniformPDF(sampleSpace, binSize, save, True)
217         print()
218         printStats(sampleSpace, True)

```

```

219     print()
220
221     print("-----")
222     print("Generating Sample Space with Python RNG")
223     print("Sample space contains", sampleSize, "entries")
224     print("The bin size is", binSize, "bins")
225     sampleSpace = np.random.uniform(size = sampleSize)
226     print("Task Complete")
227
228     print("Plotting Python RNG PDF")
229     plotUniformPDF(sampleSpace, binSize, save, False)
230     print()
231     printStats(sampleSpace, False)
232
233     # adjust parameters at will
234     # Sample size = First Parameter
235     # Bin size = Second Parameter
236     # boolean to save the plots = Third Paramter
237
238     # Uncomment next line to run
239     #question1(1000000,250,save = False)

```

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

```

1  # Mohamed Ameen Omar
2  # 16055323
3  #####
4  ###      EDC 310 Practical 2      ###
5  ###            2018            ###
6  ###      Simulation      ###
7  #####
8
9  import numpy as np
10 from question_3 import BPSKmodulationSimulation
11 from MLSE import MLSE
12 import copy
13 from DFE import DFE
14 import matplotlib.pyplot as plt
15
16 # Class to perform a simulation of the
17 # BER for the Viterbi MLSE and DFE symbol
18 # estimation methods
19 # The class takes in the number of data
20 # bits and the linear declining channel impulse response
21 # vector as constructor parameters
22 class Simulation:
23     def __init__(self, n=300, linC=[0.89, 0.42, 0.12], numIterations=20):
24         self.BpskSimulation = BPSKmodulationSimulation(n)
25         self.linC = linC
26         self.n = n
27         self.numIterations = numIterations
28
29     # Function to perform the Viterbi MLSE simulation
30     # with a Gaussian random Channel Impulse Response
31     # returns the BER array

```



```

32 # simulation runs for @param self.numIterations, for each
33 # SNR in the range (-4,9)
34 def viterbiRandomCIR(self):
35     numIter = self.numIterations
36     print("Conducting a MLSE BPSK simulation with a Uniform Random
37     CIR")
38     BER = [] #store the average BER for each SNR
39     for SNR in range(-4, 9): #9
40         tempBER = [] #to store all the iterations for one SNR
41         for count in range(0, numIter):
42             # generate 300 bits
43             myDataBits = self.BpskSimulation.generateBits() #raw data
44             bits
45             # map bits to symbols
46             modulatedSignal = self.BpskSimulation.BpskModulate(
47             myDataBits)
48             # add tail
49             for x in range(0, len(self.linC) -1):
50                 modulatedSignal.append(1)
51             # add convolution
52             randomCIR = self.generateRandomCIR(SNR)
53             convolutedSignal = self.channelModification(randomCIR,
54             modulatedSignal)
55             signalSent = self.BpskSimulation.addNoise(SNR,
56             convolutedSignal)
57             # check MLSE
58             myMLSE = MLSE(self.n, signalSent, randomCIR)
59             myMLSE.buildTrellis()
60             myMLSE.cheapestPath()
61             # get data bits
62             dataDetected = myMLSE.dataDetected
63             # demodulate
64             demodulatedSignal = self.BpskSimulation.BpskDemodulate(
65             dataDetected)
66             # compare
67             tempBER.append(self.BpskSimulation.getNumErrors(
68             myDataBits, demodulatedSignal)/self.n)
69             del myMLSE
70             del modulatedSignal
71             del convolutedSignal
72             del demodulatedSignal
73             BER.append( (sum(tempBER)/len(tempBER)) )
74         return BER
75
76 # Function to perform the Viterbi MLSE simulation
77 # with a Linear Declining Channel Impulse Response
78 # returns the BER array
79 # simulation runs for @param self.numIterations, for each
80 # SNR in the range (-4,9)
81 def viterbiLinearCIR(self):
82     numIter = self.numIterations
83     print("Conducting a MLSE BPSK simulation with a linear declining
84     CIR")
85     BER = [] #store the average BER for each SNR
86     for SNR in range(-4, 9): #9
87         tempBER = [] #to store all the iterations for one SNR

```

```

80         for count in range(0, numIter):
81
82             # generate 300 bits
83             myDataBits = self.BpskSimulation.generateBits() #raw data
84             bits
85             # map bits to symbols
86             modulatedSignal = self.BpskSimulation.BpskModulate(
87             myDataBits)
88             # add tail
89             for x in range(0, len(self.linC) -1):
90                 modulatedSignal.append(1)
91             # add convolution
92             convolutedSignal = self.channelModification(self.linC ,
93             modulatedSignal)
94             signalSent = self.BpskSimulation.addNoise(SNR,
95             convolutedSignal)
96             # check MLSE
97             myMLSE = MLSE(self.n, signalSent , self.linC)
98             myMLSE.buildTrellis()
99             myMLSE.cheapestPath()
100             # get data bits
101             dataDetected = myMLSE.dataDetected
102             # demodulate
103             demodulatedSignal = self.BpskSimulation.BpskDemodulate(
104             dataDetected)
105             # compare
106             tempBER.append(self.BpskSimulation.getNumErrors(
107             myDataBits, demodulatedSignal)/self.n)
108             del myMLSE
109             del modulatedSignal
110             del convolutedSignal
111             del demodulatedSignal
112             BER.append( (sum(tempBER)/len(tempBER) ))
113         return BER
114
115     # Function to perform the DFE simulation
116     # with a Linear Declining Channel Impulse Response
117     # returns the BER array
118     # simulation runs for @param self.numIterations, for each
119     # SNR in the range (-4,9)
120     def dfeLinearCIR(self):
121         numIter = self.numIterations
122         print("Conducting a DFE BPSK simulation with a linear declining
123         CIR")
124         BER = [] #store the average BER for each SNR
125         for SNR in range(-4, 9): #9
126             tempBER = [] #to store all the interations for one SNR
127             for count in range(0, numIter):
128                 # generate 300 bits
129                 myDataBits = self.BpskSimulation.generateBits() #raw data
130                 bits
131                 # map bits to symbols
132                 modulatedSignal = self.BpskSimulation.BpskModulate(
133                 myDataBits)
134                 # add convolution
135                 convolutedSignal = self.channelModification(self.linC ,

```

```

modulatedSignal)
127     signalSent = self.BpskSimulation.addNoise(SNR,
convolutedSignal)
128     # check DFE
129     myDFE = DFE(self.n, signalSent, self.linC)
130     # get data bits
131     dataDetected = myDFE.getDataSymbols()
132     # demodulate
133     demodulatedSignal = self.BpskSimulation.BpskDemodulate(
dataDetected)
134     # compare
135     tempBER.append(self.BpskSimulation.getNumErrors(
myDataBits, demodulatedSignal)/self.n)
136     del myDFE
137     del modulatedSignal
138     del convolutedSignal
139     del demodulatedSignal
140     BER.append((sum(tempBER)/len(tempBER)))
141     return BER
142
143     # Function to perform the DFE simulation
144     # with a Gaussian random Channel Impulse Response
145     # returns the BER array
146     # simulation runs for @param self.numIterations, for each
147     # SNR in the range (-4,9)
148     def dfeRandomCIR(self):
149         numIter = self.numIterations
150         print("Conducting a DFE BPSK simulation with a Uniform Random CIR
")
151         BER = [] #store the average BER for each SNR
152         for SNR in np.arange(-4, 9): #9
153             tempBER = [] #to store all the iterations for one SNR
154             for count in range(0, numIter):
155                 # generate 300 bits
156                 myDataBits = self.BpskSimulation.generateBits() #raw data
bits
157                 # map bits to symbols
158                 modulatedSignal = self.BpskSimulation.BpskModulate(
myDataBits)
159                 # add convolution
160                 randomCIR = self.generateRandomCIR(SNR)
161                 convolutedSignal = self.channelModification(randomCIR,
modulatedSignal)
162                 signalSent = self.BpskSimulation.addNoise(SNR,
convolutedSignal)
163                 # check DFE
164                 myDFE = DFE(self.n, signalSent, randomCIR)
165                 # get data bits
166                 dataDetected = myDFE.getDataSymbols()
167                 # demodulate
168                 demodulatedSignal = self.BpskSimulation.BpskDemodulate(
dataDetected)
169                 # compare
170                 tempBER.append(self.BpskSimulation.getNumErrors(
myDataBits, demodulatedSignal)/self.n)
171                 del myDFE

```

```

172         del modulatedSignal
173         del convolutedSignal
174         del demodulatedSignal
175         BER.append( (sum(tempBER)/len(tempBER) ) )
176     return BER
177
178     #returns a random CIR with "values" elements
179     # Requires the SNR ratio of the channel
180     # uses the Gaussian random number generator implemented in practical
181     1.
182     def generateRandomCIR(self , SNR, values=3):
183         c = []
184         for x in range(0,values):
185             temp = self.BpskSimulation.numberGenerator.gaussian(
stdDeviation=self.BpskSimulation.getStdDev(SNR))
186             temp = temp/(np.sqrt(3))
187             c.append(temp)
188             temp = 0
189         return c
190
191     # pass in just the symbols.
192     # return symbols with channel repsonse.
193     # function to add the effects of the channel to the stream of
194     # symbols being sent. (Apply the CIR)
195     def channelModification(self , cir , stream):
196         returnStream = copy.deepcopy(stream)
197         for x in range(0,len(stream)):
198             temp1 = 0 #sk-2
199             temp2 = 0 #sk-1
200             if(x-2 < 0):
201                 temp1 = 1
202             else:
203                 temp1 = stream[x-2]
204             if(x-1 <0):
205                 temp2 = 1
206             else:
207                 temp2 = stream[x-1]
208             returnStream[x] = (stream[x]*cir[0] + temp2*cir[1] + temp1*
cir[2])
209         return returnStream
210
211     # Function to perform the simulation of all 4 situations
212     # DFE linear CIR, DFE Gaussian Random CIR, MLSE linear CIR and MLSE
213     Gaussian Random CIR.
214     # Plots all four simulations on the same plot
215     def simulate(self):
216         print("Plotting all")
217         SNR = np.linspace(-4, 8, 13)
218         viterbiLinDec = self.viterbiLinearCIR()
219         viterbiRandom = self.viterbiRandomCIR()
220         dfeRandom = self.dfeRandomCIR()
221         dfeLin = self.dfeLinearCIR()
222         plt.semilogy(SNR, viterbiLinDec , 'r-', label='Viterbi MLSE Linear
Declining CIR')
223         plt.semilogy(SNR, viterbiRandom , 'g-', label='Viterbi MLSE Random

```

```

    CIR')
223     plt.semilogy(SNR, dfeRandom, 'b-', label='DFE Random CIR')
224     plt.semilogy(SNR, dfeLin, 'y-', label='DFE Linear Declining CIR')
225     plt.grid(which='both')
226     plt.ylabel("BER")
227     plt.xlabel("SNR")
228     plt.title("Plot of the BER vs SNR for the BPSK Simulation
conducted with Viterbi MLSE and DFE Equalizers")
229     plt.legend(loc='best')
230     plt.show()
231     print("Complete")
232
233 # n = number of data bits (300)
234 # c = linear declining CIR
235 # numIterations = number of iterations to take the average of, before
    plotting - adjust as needed.
236 # if commented - uncomment the last line
237 # "mySim.simulate()" to run the simulation and retrieve the plot
238 # n = 300
239 # c = [0.89,0.42,0.12]
240 # numIterations = 50
241 # mySim = Simulation(n,c, numIterations)
242 # mySim.simulate()

```

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

```

1 #Mohamed Ameen Omar
2 #16055323
3 #####
4 ###      EDC 310 Practical 1      ###
5 ###      2018                    ###
6 ###      Question 3              ###
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 from question_1 import randomNumber
15
16 # Ensure that the source file question_1.py is in the current dorectory
17 # before running the script
18
19 # Class used for QPSK simulation
20 class QPSKmodulationSimulation:
21     def __init__(self, numBits):
22         self.numBits = numBits
23         self.numberGenerator = randomNumber()
24
25     def getStdDev(self, SNR):
26         return (1/np.sqrt(10**((SNR/10)*2*2)))
27
28     #add noise for the real and imaginary parts of the symbol
29     def addNoise(self, SNR, sentBits):

```

```

30         recieved = []
31         standardDeviation = self.getStdDev(SNR)
32         for x in range(0, len(sentBits)):
33             real = self.numberGenerator.gaussian(stdDeviation=
standardDeviation)
34             imag = self.numberGenerator.gaussian(stdDeviation=
standardDeviation) * 1j
35             recieved.append(sentBits[x]+real+imag)
36
37         return recieved
38
39     # generate random number using uniform random number generator
40     # round the output to a 1 or 0 for a bit
41     def generateBits(self):
42         print("Generating", self.numBits, "random binary digits")
43         toSend = []
44         for x in range(0, self.numBits):
45             toSend.append(int(np.round(self.numberGenerator.WHill())))
46         return(toSend)
47
48     # map an array of bits to a symbols
49     # according to the BPSK constellation map
50     def QpskModulate(self, original):
51         print("Mapping bits to symbols using QPSK modulation")
52         mappedMessage = []
53         temp = list(map(str, original))
54
55         for x in range(0, len(temp), 2):
56             toSymbol = temp[x] + temp[x+1]
57             mappedMessage.append(self.bitToSymbol(toSymbol))
58         return mappedMessage
59
60     # return the Symbol for the
61     # the bit passed in
62     def bitToSymbol(self, toMap):
63         if(toMap == "00"):
64             return 1
65         elif(toMap == "01"):
66             return 1j
67         elif(toMap == "11"):
68             return -1
69         elif(toMap == "10"):
70             return -1j
71
72     # demodulate the detected symbols
73     # for QPSK
74     def QpskDemodulate(self, modulated):
75         print("Mapping symbols to bits for QPSK demodulation")
76         demod = []
77         for x in range(0, len(modulated)):
78             temp = self.symbolToBit(modulated[x])
79             demod.append(int(temp[0]))
80             demod.append(int(temp[1]))
81         return demod
82
83     # return the bits for the

```

```

84 # the symbol passed in
85 def symbolToBit(self, symbol):
86     if(symbol == 1):
87         return "00"
88     elif(symbol == 1j):
89         return "01"
90     elif(symbol == -1):
91         return "11"
92     elif(symbol == -1j):
93         return "10"
94
95 # given the SNR and recieved signal
96 # use optimum detection algorithm to
97 # determine the signal that was sent
98 # return the symbols that were sent
99 def detectSignal(self, SNR, recieved):
100     stdDev = self.getStdDev(SNR)
101     detectedBits = []
102     #for every bit recieved
103     for x in range(0, len(recieved)):
104         probabilities = [] # 1,j,-1,-j
105         temp = []
106         temp.append(self.getExpProb(recieved[x], 1, stdDev))
107         temp.append(self.getExpProb(recieved[x], 1j, stdDev))
108         temp.append(self.getExpProb(recieved[x], -1, stdDev))
109         temp.append(self.getExpProb(recieved[x], -1j, stdDev))
110         probabilities = np.exp(temp)
111         beta = self.getBeta(probabilities)
112         for prob in range(0, len(probabilities)):
113             probabilities[prob] = probabilities[prob] * beta
114         ind = np.argmax(probabilities)
115         detectedBits.append(self.getSymbol(ind))
116
117     return detectedBits
118
119 # get exponent e is raised to for
120 # the symbol recieved and the symbol we think it is
121 # need standard deviation for the channel as well
122 def getExpProb(self, recieved, actual, stdDev):
123     temp = np.abs(recieved - actual)
124     temp = (temp**2)*(-1)
125     temp = temp/(2*(stdDev**2))
126     return temp
127
128 # given an array of conditional probabilities
129 # return scaling factor or normalization constant (beta)
130 def getBeta(self, probs):
131     temp = 0
132     for x in range(0, len(probs)):
133         temp += probs[x]
134     return (1/temp)
135
136 # given the index with the highest probability
137 # returnn the QPSK symbol
138 def getSymbol(self, index):
139     if(index == 0):

```

```

140         return 1
141     elif(index == 1):
142         return 1j
143     elif(index == 2):
144         return -1
145     elif(index == 3):
146         return -1j
147
148     #return the number of bit errors for the signal
149     def getNumErrors(self , sentBits , recievedBits):
150         errors = 0
151         for x in range(0, len(sentBits)):
152             if(sentBits[x] != recievedBits[x]):
153                 errors += 1
154         return errors
155
156     # simulate the sending and recieving
157     # plot BER vs SNR as well
158     # returns the BER array
159     def simulate(self):
160         print("Question 3")
161         print("QPSK Simulation with", self.numBits, "bits")
162         BER = []
163         bits = self.generateBits()
164         for SNR in range(-4,9):
165             print("SNR set to", SNR)
166             #map each bit to a Qpsk symbol
167             sentSignal = self.QpskModulate(bits)
168             print("Signal Sent")
169             #print(sentSignal)
170             recievedSignal = self.addNoise(SNR, sentSignal)
171             print("Signal Recieved")
172             #print(recievedSignal)
173             detectedSignal = self.detectSignal(SNR, recievedSignal) #
174             #print(detectedSignal)
175             #convert symbols to bits
176             detectedBits = self.QpskDemodulate(detectedSignal)
177             print("Signal Has been demodulated")
178             #print(detectedBits)
179             BER.append(self.getNumErrors(bits , detectedBits)/self.numBits)
180             print()
181
182         print("The Bit Error rate for each SNR tested is given in the
183 array below:")
184         print(BER)
185         print()
186         print("Plotting the BER vs SNR relationship")
187         SNR = np.linspace(-4, 8, 13)
188         plt.semilogy(SNR, BER)
189         plt.grid(which='both')
190         plt.ylabel("BER")
191         plt.xlabel("SNR")
192         plt.title("Plot of the BER vs SNR for the QPSK Simulation
193 conducted with " + str(self.numBits) + " bits")
194         plt.show()

```



```

193         print("End of QPSK Simulation")
194         return BER
195 ##### End of class #####
196
197 class BPSKmodulationSimulation:
198     def __init__(self, numBits):
199         self.numBits = numBits
200         self.numberGenerator = randomNumber()
201
202     #return standard deviation for SNR with fbit =1
203     def getStdDev(self, SNR):
204         return (1/np.sqrt(10**((SNR/10)*2*1)))
205
206     def addNoise(self, SNR, sentBits):
207         recieved = []
208         standardDeviation = self.getStdDev(SNR)
209         for x in range(0, len(sentBits)):
210             real = self.numberGenerator.gaussian(stdDeviation=
standardDeviation)
211             recieved.append(sentBits[x]+real)
212         return recieved
213
214     # generate random number using uniform random number generator
215     # round the output to a 1 or 0 for a bit
216     def generateBits(self):
217         #print("Generating", self.numBits, "random binary digits")
218         toSend = []
219         for x in range(0, self.numBits):
220             toSend.append(int(np.round(self.numberGenerator.WHill())))
221         return(toSend)
222
223     # map an array of bits to a symbols
224     # according to the BPSK constellation map
225     def BpskModulate(self, original):
226         #print("Mapping bits to symbols using BPSK modulation")
227         mappedMessage = []
228         for x in range(0, len(original)):
229             mappedMessage.append(self.bitToSymbol(original[x]))
230         return mappedMessage
231
232     # map a single bit to a symbol
233     # according to the BPSK constellation map
234     def bitToSymbol(self, bit):
235         if(bit == 1):
236             return 1
237         if(bit == 0):
238             return -1
239         else:
240             print("Error occured, bit to map was not zero or one")
241
242     # demodulate the detected symbols
243     # for QPSK
244     def BpskDemodulate(self, modulated):
245         #print("Mapping symbols to bits for BPSK demodulation")
246         demod = []
247         for x in range(0, len(modulated)):

```

```

248         demod.append(self.symbolToBit(modulated[x]))
249     return demod
250
251     # return the bit for the
252     # the symbol passed in
253     def symbolToBit(self, symbol):
254         if(symbol == 1):
255             return (1)
256
257         elif(symbol == -1):
258             return 0
259
260     # given the SNR and recieved signal
261     # use optimum detection algorithm to
262     # determine the signal that was sent
263     # return the symbols that were sent
264     def detectSignal(self, SNR, recieved):
265         stdDev = self.getStdDev(SNR)
266         detectedBits = []
267         #for every bit recieved
268         for x in range(0, len(recieved)):
269             probabilities = [] # 1,j,-1,-j
270             temp = []
271             temp.append(self.getExpProb(recieved[x], 1, stdDev))
272             temp.append(self.getExpProb(recieved[x], -1, stdDev))
273             probabilities = np.exp(temp)
274             beta = self.getBeta(probabilities)
275             for prob in range(0, len(probabilities)):
276                 probabilities[prob] = probabilities[prob] * beta
277             ind = np.argmax(probabilities)
278             detectedBits.append(self.getSymbol(ind))
279         return detectedBits
280
281     # get exponent e is raised to for
282     # the symbol recieved and the symbol we think it is
283     # need standard deviation for the channel as well
284     def getExpProb(self, recieved, actual, stdDev):
285         temp = np.abs(recieved-actual)
286         temp = (temp**2)*(-1)
287         temp = temp/(2*(stdDev**2))
288         return temp
289
290     # given an array of conditional probabilities
291     # return scaling factor or normalization constant (beta)
292     def getBeta(self, probs):
293         temp = 0
294         for x in range(0, len(probs)):
295             temp += probs[x]
296         return (1/temp)
297
298     # given the index with the highest probability
299     # return the BPSK symbol
300     def getSymbol(self, index):
301         if(index == 0):
302             return 1
303

```

```

304         elif(index == 1):
305             return -1
306
307     #return the number of bit errors for the signal
308     def getNumErrors(self, sentBits, recievedBits):
309         errors = 0
310         for x in range(0, len(sentBits)):
311             if(sentBits[x] != recievedBits[x]):
312                 errors += 1
313         return errors
314
315     # simulate the sending and recieving
316     # plot BER vs SNR as well
317     # returns the BER array
318     def simulate(self):
319         print("Question 3")
320         print("BPSK Simulation with", self.numBits, "bits")
321         BER = []
322         bits = self.generateBits()
323         for SNR in range(-4, 9):
324             print("SNR set to", SNR)
325             #map each bit to a BPSK symbol
326             sentSignal = self.BpskModulate(bits)
327             print("Signal Sent")
328             #print(sentSignal)
329             recievedSignal = self.addNoise(SNR, sentSignal)
330             print("Signal Recieved")
331             #print(recievedSignal)
332             detectedSignal = self.detectSignal(SNR, recievedSignal) #
333             still symbols
334             #print(detectedSignal)
335             #convert symbols to bits
336             detectedBits = self.BpskDemodulate(detectedSignal)
337             print("Signal Has been demodulated")
338             #print(detectedBits)
339             BER.append(self.getNumErrors(bits, detectedBits)/self.numBits
340         )
341
342         print()
343
344         print("The Bit Error rate for each SNR tested is given in the
345         array below:")
346         print(BER)
347         print()
348         print("Plotting BER vs SNR function for the BPSK Simulation")
349         SNR = np.linspace(-4, 8, 13)
350         plt.semilogy(SNR, BER)
351         plt.grid(which='both')
352         plt.ylabel("BER")
353         plt.xlabel("SNR")
354         plt.title("Plot of the BER vs SNR for the BPSK Simulation
355         conducted with " + str(self.numBits) + " bits")
356         plt.show()
357         print("End of BPSK Simulation")
358         return BER
359
360     #plots the result of BPSK and QPSK modulation on a single plot

```

```

356 #must pass in BER for both
357 def plotToCompare(QpskBer, BpskBer):
358     print("Plotting both")
359     SNR = np.linspace(-4, 8, 13)
360     plt.semilogy(SNR, QpskBer, 'r-', label='QPSK BER')
361     plt.semilogy(SNR, BpskBer, 'g-', label='BPSK BER')
362     plt.grid(which='both')
363     plt.ylabel("BER")
364     plt.xlabel("SNR")
365     plt.title("Plot of the BER vs SNR for the BPSK and QPSK Simulation
366               conducted")
367     plt.legend(loc='best')
368     plt.show()
369     print("Complete")
370
371 # numBits is the number of bits
372 # that we would like to use in the simulation of each
373 # may change the number of bits at will
374
375 # create two objects, one for the BPSK simulation
376 # one for the QPSK Simulation
377 #numBits = 1000000
378 #BpskSimulation = BPSKmodulationSimulation(numBits)
379 #QpskSimulation = QPSKmodulationSimulation(numBits)
380
381 #uncomment next line to run the Bpsk Simulation
382 #bpsk = BpskSimulation.simulate()
383
384 #uncomment next line to run the Qpsk Simulation
385 #qpsk = QpskSimulation.simulate()
386
387 #to plot both on the same axis, uncomment next line
388 #plotToCompare(qpsk, bpsk)

```

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