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

EDC 310

Digital Communications

Practical Assignment 2 Report: MLSE and DFE modulation simulations

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

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

Friday 28$^{\text{th}}$ September, 2018

Table of Contents

1 Introduction

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. Two sources of the distortion incurred by the signal are noise and the effects of multipath.

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. 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.

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. During Practical 1, students implemented a Additive white Gaussian noise (AWGN) simulation platform which will be used as a critical component for the simulations conducted during practical 2.

The second common source of distortion of a digital signal being sent is as a result of the effects a Multipath channel. A channel in Digital communications refers to the medium through which the transmitted signal is sent from the transmitting antenna to the receiving antenna. A Multipath channel is a medium in which the signal reaches the receiving antenna via more than one path. Sources of multipath include reflection and diffraction of the signal from the surface of water, the ionosphere of the earth, buildings and other physical objects. Multipath can cause errors such as Inter Signal Interference (ISI) as well as affect the quality of communication.

Due to the fact that there are multiple electromagnetic paths to the receiver, more than one instance of the same signal may be received at different time periods, thus causing distortion of the signal received at the antenna. In order to model the effects of Multipath and determine the correct signal at the receiver, a channel impulse response model is produced. The Channel Impulse response is a model that shows how the signal is distorted within the channel as a function of frequency and time. Figures 1 to 3 illustrate the concept of Multipath propagation and the Channel Impulse Response.

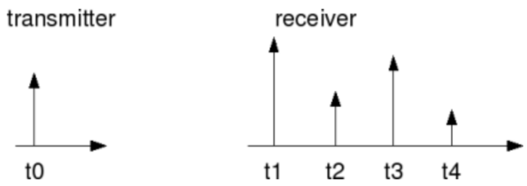

Figure 1. Channel Impulse Response over a Multipath channel

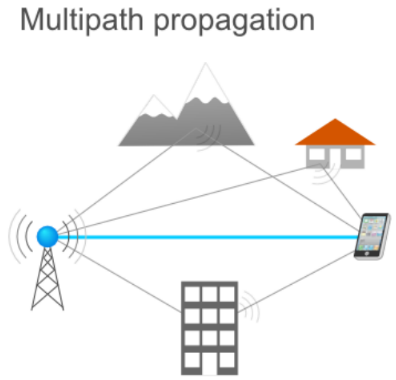

Figure 2. Multipath propagation for a signal from an Antenna to a cellphone

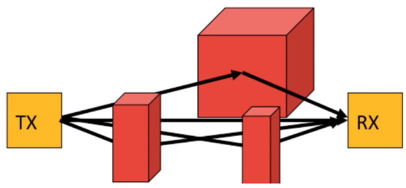

Figure 3. Illustration of a multipath channel

The signal received at the receiving antenna can be interpreted by the general expression for a Multipath symbol: $r_k = s_k C_0 + s_{k-1} C_1 ... + s_{k-n} C_n + noise$, where noise the additive noise added to the signal, r_k is the symbol received at time k and C_n is the channel impulse response for the nth frequency harmonic of the channel.
Two equalizers may be used to solve this problem. The two equalizers that will be tested to determine the most probable sequence of symbols transmitted, are the Viterbi Maximum Likelihood Sequence Estimation (MLSE) algorithm and the Decision Feedback Equalization algorithm (DFE).

During Practical 2 for EDC 320, students will investigate the effectiveness of Viterbi Maximum Likelihood Sequence Estimation (MLSE) algorithm and the Decision Feedback Equalization algorithm (DFE) in determining the correct bits being sent over an AWGN Multipath channel using BPSK modulation. In addition, the effects of a linearly declining Channel Impulse Response and a randomly generated Channel Impulse Response will be compared.

2 Viterbi MLSE

2.1 Introduction

The Maximum Likelihood Sequence Estimation (MLSE) algorithm is used to detect the most likely sequence of symbols received at a receiving transmitter. A block of symbols transmitted from a transmitting antenna to a receiving antenna contains three sets of symbols, the header symbols, the data symbols, and the tail symbols. The header and tail symbols are prepended and appended to the data symbols before transmission and by convention are chosen as *"1"*. The number of header and tail symbols appended and prepended is equal to the number of Channel Impulse Response element minus 1.

The Viterbi MLSE algorithm begins with constructing a Viterbi Trellis, which is a variation of a tree, wherein each node represents a possible "state". The goal of the MLSE algorithm is to choose a sequence of symbols that maximizes the conditional probability error across the entire state space. The Viterbi-algorithm is the optimal and at the same time the most efficient method for the estimation of the input symbol sequence of a finite state machine from its output symbol sequence (distorted signal symbols), that has been distorted with white Gaussian noise [1].

Using the Viterbi trellis, the MLSE algorithm computes the cost of every path through the trellis and returns the cheapest path, with the lowest cost from the start state to the end state. An example of the Viterbi Trellis is given in figure 4. The cost from one node in the trellis to another is referred to as the Delta value is calculated as $\Delta_n^{sk} = |s_k C_0 + s_{k-1} C_1 ... + s_{k-n} C_n|^2$

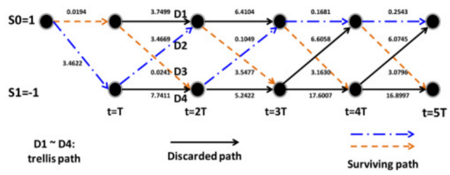

Figure 4. Example of a Viterbi Trellis

2.2 Design and Implementation

In order to perform the Viterbi MLSE algorithm on a set of bits received two classes have been defined. The *trellisNode* and *MLSE* classes. The *trellisNode* class represents a single node in the Viterbi Trellis and is used by the *MLSE* class to build the Viterbi Trellis.

The*trellisNode* class holds members to store the state of the node, all connected states, the delta values for each connected state and the alpha value (representing the sum of delta values along cheapest path to node)

The *MLSE* class holds members to store the the CIR, the received symbols, the number of data bits and a matrix to store the trellis. The class builds the Viterbi Trellis by creating a node for each state in the trellis, beginning with the first node at time = 0, being a symbol = 1. All valid state transitions are connected and new nodes for each state and time are created.

Once the Viterbi Trellis is built, the algorithm computes all the delta values (using the formula given above) for every transition and chooses the cheapest path from the first bit to the last bit and returns the symbol sequence. All code is written in the Python programming language and the source code for the Viterbi MLSE implementation in the attached Appendix.

3 DFE

3.1 Introduction

The Decision Feedback Equalizer (DFE) algorithm is a filter that uses previously detected symbols in conjunction with a linear equalizer to reduce Inter Signal Interference (ISI) and detect the transmitted symbols. A block of symbols transmitted from a transmitting antenna to a receiving antenna contains three sets of symbols, the header symbols, the data symbols, and the tail symbols. The header and tail symbols are prepended and appended to the data symbols before transmission and by convention are chosen as *"1"*. The number of header and tail symbols appended and prepended is equal to the number of Channel Impulse Response element minus 1. In general however, the DFE algorithm does not make use of these tail symbols and only considers the header symbols when estimating the sequence of symbols received.

The DFE algorithm is similar to the MLSE algorithm in the light that both equalizers compute the costs or *deltas* for each state by $\Delta_n^{sk} = |s_k C_0 + s_{k-1} C_1 ... + s_{k-n} C_n|^2$. Where DFE is set apart from MLSE is that, when estimating the symbol received, DFE only considers the cost from one node to another, as opposed to the entire path cost and as such is more susceptible to bear the consequences for errors made in estimating symbols.

3.2 Design and Implementation

The DFE algorithm is implemented by creating a *DFE* class to compute the DFE algorithm and return the detected symbols. The class takes in the number of data bits, the Channel Impulse Response vector and the received bit sequence as constrictor parameters. For each bit received, the DFE algorithm computes the delta values using the equation above, for each possible state.

The bit detected at a single time instant is the bit or state that results in the smallest delta value for each time instant. The algorithm estimates all symbols received before storing the result in it's member variable (*dataDetected*). The detected bits can then be retrieved by the user using the *getDataDetected()* class member function. All code is written in the Python programming language and the source code for the Viterbi MLSE implementation in the attached Appendix.

4 Channel Impulse Response

4.0.1 Linear CIR

4.0.2 Theoretical Analysis

A linearly declining Channel impulse response is one wherein as the harmonic number of the Fourier transform of the channel increases, the amplitude or power generated by the channel for that harmonic decreases linearly. Therefore, the effects of a symbol being delayed decline linearly as time increases. A linearly declining CIR is not realistic as in the real world, it is more likely that the CIR of the channel will be random.

4.0.3 Design and implementation

The Linear decline CIR was given as , as used in the simulations for both equalizers. The CIR was stored statically in each implementation since it would not change, as specified by the practical specification.

4.1 Random CIR

4.1.1 Theoretical Analysis

A Random CIR is one wherein there seems to be no correlation between the harmonic number and the power generated in the channel for that harmonic. The effects of such a CIR are that the results or distortions to the symbols sent over the channel are not purely deterministic and would be extremely difficult to estimate the distorted bits. This means that symbols that have been delayed for numerous time instances can still have a significant effect on the symbols detected and cause must greater ISI.

4.1.2 Design and implementation

A single randomly generated CIR element was generated using the equation $\frac{RNG(\sigma)}{\sqrt{3}}$, for both the DFE and MLSE algorithms. Each time a new set of symbols was generated, a new randomly generated CIR vector was generated using the uniform number generator implemented in practical 1.

5 Simulation platform

5.1 Design and implementation

The BPSK simulation platform developed in Practical 1 was used to develop a simulation platform that includes the effect of multipath in the received signal, where the channel impulse response (CIR) length is L = 3. The simulation platform would simulate the transmission and detection of 300 data symbols over a AWGN channel that included the effect of multipath.

The simulation platform was implemented in the *Simulation* class which took in the number of data bits to generate as well as the linearly delving channel impulse response vector. The class contained a number of member functions to perform the simulation for both the DFE and MLSE algorithms using the linearly declining Channel impulse response as C=[0.89, 0.42, 0.12] and the randomly generated CIR generated using one of the member functions.

The randomly generated CIR was generated using the uniform random number generator implemented in practical 1 through the equation $\frac{RNG(\sigma)}{\sqrt{3}}$. The function would return the number of CIR elements as specified by the parameter passed in. The simulation platform can be executed using the *simulate()* member function which would simulate the DFE and MLSE algorithm using both types of CIR vectors and plot all results on the same plot.

The functioning of the *simulate()* function is as follows:
It generates a random number of bits according the parameter passed into the class constructor, over a loop that varies the **SNR** of the AWGN channel on the range [4,8] dB, the bits are converted to symbols via the *BpskModulate()* (implemented in Practical 1) function.

Depending on whether DFE or MLSE was being simulated the tail symbols were added as "1"s, the multipath effects (using the randomly generated CIR or linearly declining CIR) were added to each bit and noise was added to the modulated bits, thus simulating the transmission of the signal over a AWGN multipath channel. The Gaussian random number generator implemented in practical 1 was used to generate the noise added, with the standard deviation calculated as $\frac{1}{\sqrt{10^{\frac{SNR}{10}} 2 f_{bit}}}$. An f_{bit} value of 1 was used since BPSK modulated and demodulation was being conducted. The bits were "received" and detected by using the DFE or MLSE algorithms and translated back into bits using the *BpskDemodulate()* function. The orignal bots generated were compared to the "detected" bits. An average of a user defined number of iterations was compute and the average BER for each SNR was returned.

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

6 Results

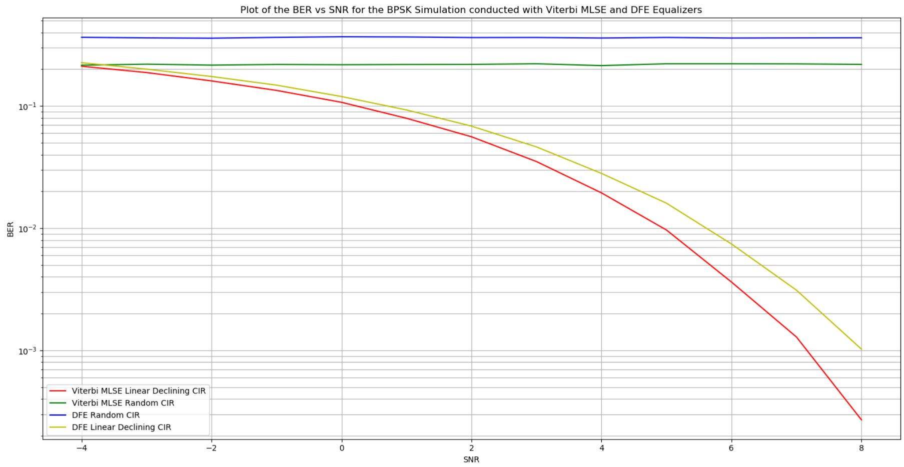

Figure 5. Simulation results of the DFE and Viterbi MLSE algorithms for a linearly declining and randomly generated CIR averaged over 1000 iterations

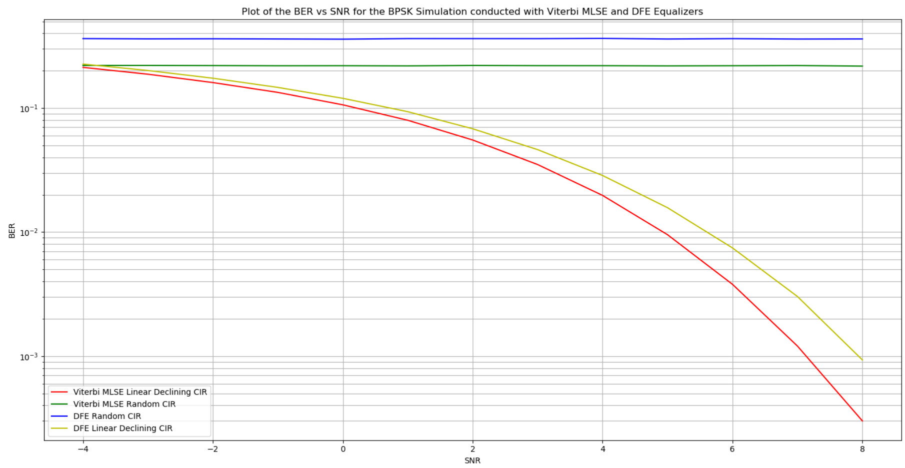

Figure 6. Simulation results of the DFE and Viterbi MLSE algorithms for a linearly declining and randomly generated CIR averaged over 5000 iterations

7 Discussion

The results in figures 5 and 6 allow us to compare the two different equalizers as well as their performance with different types of Channel Impulse Response. In reality, the Channel Impulse Response will not always be linear, randomness does exist in the world around us and it is, therefore, necessary to analyze the effect a randomly generated Channel Impulse Response has on the simulation platform. Referring to figures 5 and 6, we can deduce that a linearly declining Channel Impulse Response performs much better than a randomly generated Channel Impulse Response. This is noted as a result of the exponential decrease in BER for the linearly declining Channel Impulse Response compared to the slight decrease in the randomly generated graph for both the DFE and Viterbi MLSE algorithms.

For the linearly declining Channel Impulse Response, both algorithms see an exponential decline in BER, with an increase in the SNR. This is consistent with our expectations, since the higher the SNR, the less distortion there is to a symbol sequence and as such, one would expect fewer errors to be made in determining the symbols sent.

From the results of the simulation a comparison on the performance of the two equalizers can be done. It can be seen that MLSE performs better than DFE for both a linearly declining Channel Impulse Response and a randomly generated Channel Impulse Response. The Viterbi MLSE algorithm using a linearly declining Channel Impulse Response performs slightly better than the DFE, with the rate of decline of the BER of the Viterbi MLSE algorithm increasing much higher than that of the DFE, at an SNR of greater than -1. There is a noticeable difference between the algorithms when using a randomly generated Channel Impulse Response, as both remain relatively constant, the Viterbi MLSE has a lower BER for the entire range of SNR values simulated.

From this, we can infer that the Viterbi MLSE is much more suitable for a randomly generated CIR than the DFE algorithm. The reason why MLSE performs better than DFE is because if the DFE makes an incorrect estimation, the error is compounded as it is used to calculate other bits. This is due to the DFE algorithm bounding it's scope to the current bit estimation, whereas the Viterbi MLSE uses backtracking to take into account the entire data block of bits sent. When there is a random Channel Impulse Response bit estimations become more challenging therefore results in a greater number of errors being made. with MLSE algorithm these errors are dealt with effectively through backtracking however with DFE algorithm it is not dealt with effectively.

8 Conclusion

In conclusion the Viterbi Maximum Likelihood Sequence Estimation performs better than the Decision Feedback Equalizer when using both a linearly declining CIR and a randomly generated Channel Impulse Response. This result is as expected, since the Viterbi MLSE algorithm considers a maximum error possibility over the entire state space whereas the DFE algorithm only considers an error probability of a current state space. DFE will naturally achieve a higher BER than the Viterbi MLSE, this is due to the fact when DFE makes an incorrect bit assumption it compounds its error, which affects future bits that are to be determined. MLSE performs better in comparison because it looks at the sequence as a whole and finds the shortest path cost of the entire sequence of bits.

In the real world there is non-linear CIR, almost random, due to the effect of Multipath, therefore since the MLSE algorithm achieves a much better BER with a randomly generated CIR when compared to DFE, one can conclude that the Viterbi MLSE algorithm is more suitable to real world applications and as such is a better equalizer.

9 References

[1] F. Lenkeit, "Viterbi-algorithm," 2013.

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

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

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

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

A Appendix A: How to operate Python source code

A.1 Dependencies

The following Python packages are required to execute the scripts:

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

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

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 *Simulation.py* python script. Please ensure that lines 238 to 242 of the script is not commented out.

The script will automatically simulate the DFE and MLSE Viterbi algorithms on a randomly generated set of 300 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 1     ###
5 ###             2018              ###
6 ###           Question 1          ###
7 #####################################
8 
9 import numpy as np
10 import matplotlib.pyplot as plt
```

```
import time
import datetime
from scipy.stats import norm

############ Random Number Class #####################
# The random number class contains the two random number generators
class randomNumber:
    #Constructor
    def __init__(self, seed1 = time.time(), seed2 = time.time()-10, seed3
    = time.time()+10):
        self.seed1 = seed1
        self.seed2 = seed2
        self.seed3 = seed3
        self.computed = None

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

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

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

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

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

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

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

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


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

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

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

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

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

```
        message = message + "Built-in Python Uniformly Distributed Random Number Generator"
    print(message)
    print("Mean: " + str(np.mean(sample)))
    print("Standard Deviation: " + str(np.std(sample)))
    print("Variance: " + str(np.var(sample)))


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

    print("Question 1:")

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

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

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

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

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

# Uncomment next line to run
```

```
#question1(1000000,250,save = False)
```

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

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

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

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

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

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

    #add noise for the real and imaginary parts of the symbol
    def addNoise(self,SNR,sentBits):
        recieved = []
        standardDeviation = self.getStdDev(SNR)
        for x in range(0,len(sentBits)):
            real = self.numberGenerator.gaussian(stdDeviation=
    standardDeviation)
            imag = self.numberGenerator.gaussian(stdDeviation=
    standardDeviation) * 1j
            recieved.append(sentBits[x]+real+imag)

        return recieved

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

    # map an array of bits to a symbols
    # according to the BPSK constellation map
```

```
    def QpskModulate(self, original):
        print("Mapping bits to symbols using QPSK modulation")
        mappedMessage = []
        temp = list(map(str,original))

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

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

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

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

    # given the SNR and recieved signal
    # use optimum detection algorithm to
    # determine the signal that was sent
    # return the symbols that were sent
    def detectSignal(self,SNR, recieved):
        stdDev = self.getStdDev(SNR)
        detectedBits = []
        #for every bit recieved
        for x in range(0, len(recieved)):
            probabilities = []  # 1,j,-1,-j
            temp = []
```

```
            temp.append(self.getExpProb(recieved[x], 1,stdDev))
            temp.append(self.getExpProb(recieved[x], 1j, stdDev))
            temp.append(self.getExpProb(recieved[x], -1, stdDev))
            temp.append(self.getExpProb(recieved[x], -1j, stdDev))
            probabilities = np.exp(temp)
            beta = self.getBeta(probabilities)
            for prob in range(0,len(probabilities)):
                probabilities[prob] = probabilities[prob] * beta
            ind = np.argmax(probabilities)
            detectedBits.append(self.getSymbol(ind))

        return detectedBits

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

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

    # given the index with the highest probability
    # returnn the QPSK symbol
    def getSymbol(self,index):
        if(index == 0):
            return 1
        elif(index == 1):
            return 1j
        elif(index == 2):
            return -1
        elif(index == 3):
            return -1j

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

    # simulate the sending and recieving
    # plot BER vs SNR as well
    # returns the BER array
    def simulate(self):
        print("Question 3")
        print("QPSK Simulation with", self.numBits, "bits")
```

```
        BER = []
        bits = self.generateBits()
        for SNR in range(-4,9):
            print("SNR set to", SNR)
            #map each bit to a Qpsk symbol
            sentSignal = self.QpskModulate(bits)
            print("Signal Sent")
            #print(sentSignal)
            recievedSignal = self.addNoise(SNR, sentSignal)
            print("Signal Recieved")
            #print(recievedSignal)
            detectedSignal = self.detectSignal(SNR, recievedSignal) #
    still symbols
            #print(detectedSignal)
            #convert symbols to bits
            detectedBits = self.QpskDemodulate(detectedSignal)
            print("Signal Has been demodulated")
            #print(detectedBits)
            BER.append(self.getNumErrors(bits,detectedBits)/self.numBits)
            print()

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

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

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

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

```

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

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

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

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

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

        elif(symbol == -1):
            return 0

    # given the SNR and recieved signal
    # use optimum detection algorithm to
    # determine the signal that was sent
    # return the symbols that were sent
    def detectSignal(self, SNR, recieved):
        stdDev = self.getStdDev(SNR)
        detectedBits = []
        #for every bit recieved
        for x in range(0, len(recieved)):
            probabilities = []  # 1,j,-1,-j
```

```
                temp = []
                temp.append(self.getExpProb(recieved[x], 1, stdDev))
                temp.append(self.getExpProb(recieved[x], -1, stdDev))
                probabilities = np.exp(temp)
                beta = self.getBeta(probabilities)
                for prob in range(0, len(probabilities)):
                    probabilities[prob] = probabilities[prob] * beta
                ind = np.argmax(probabilities)
                detectedBits.append(self.getSymbol(ind))
        return detectedBits

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

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

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

        elif(index == 1):
            return -1

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

    # simulate the sending and recieving
    # plot BER vs SNR as well
    # returns the BER array
    def simulate(self):
        print("Question 3")
        print("BPSK Simulation with", self.numBits, "bits")
        BER = []
        bits = self.generateBits()
        for SNR in range(-4, 9):
            print("SNR set to", SNR)
            #map each bit to a BPSK symbol
```

```
            sentSignal = self.BpskModulate(bits)
            print("Signal Sent")
            #print(sentSignal)
            recievedSignal = self.addNoise(SNR, sentSignal)
            print("Signal Recieved")
            #print(recievedSignal)
            detectedSignal = self.detectSignal(SNR, recievedSignal)  # still symbols
            #print(detectedSignal)
            #convert symbols to bits
            detectedBits = self.BpskDemodulate(detectedSignal)
            print("Signal Has been demodulated")
            #print(detectedBits)
            BER.append(self.getNumErrors(bits, detectedBits)/self.numBits)
            print()

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

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

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

# create two objects, one for the BPSK simulation
# one for the QPSK Simulation
numBits = 1000000
```

```
BpskSimulation = BPSKmodulationSimulation(numBits)
QpskSimulation = QPSKmodulationSimulation(numBits)

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

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

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

Listing 2. Python Source code for question 3 (Simulation Platform) - From practical 1

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

import numpy as np
import copy

# Class to run a DFE equalizer to dermine the bits sent over a AGWN
    channel
# Constructor paramaters:
#   @param N = the number of data bits of data being sent
#   @param r = a vector with all the recieved symbols
#   @param c = the channel impulse response vector
class DFE:
    def __init__(self, N = 0, r = [], c = []):
        self.n = N #number of data bits
        self.r = r #recieved vector - only data bits len(r) = self.n
        self.c = c #convolution matrix
        self.L = len(c) #length of the c vector
        self.numHeader = self.L-1 #number of header bits
        self.symbols = [1,-1]
        self.dataDetected = [] #just the data bits detected

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

    # Function to detect the symbols in the recieved vector
    # using the DFE Equalizer
    def detectSymbols(self):
        for x in range(0,len(self.r)):
```

```python
                self.dataDetected.append(self.getSymbol(x))
    # Function to detect a single sumbol using
    # DFE Equalizer algorithm
    def getSymbol(self,time):
        if(self.getDelta(time, 1) > self.getDelta(time, -1)):
            return -1
        else:
            return 1

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

Listing 3. Python Source code for the DFE class - to execute a DFE signal estimation

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

import numpy as np
from question_3 import BPSKmodulationSimulation
import copy

'''
MY IMPLEMENTATION


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

first pass in the r,n,c

then it builds it by:
```

```python
assigniedn fist node to 11 at t=0
then for every node, create its transitons and update all
including all alphas and deltas

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

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

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

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

```
        self.detected = [] #detected symbols from trellis
        self.entireStream = [] #entire detected stream including head and tail
        self.dataDetected = [] #data bits detected

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

    # Function to build the viterbi trellis
    # will begin with the first node state being = 1,1; following convention.
    # assigning fist node to 11 at t=0
    # then for every node, create its transitons and update all
    # including all alphas and deltas
    def buildTrellis(self, startState = [1,1]):
        self.Trellis[0][0] = trellisNode(startState, 0)
        numBits = len(startState)
        #for every node in the trellis compute the nodes it
        # will transition to
        for t in range(0,self.trellisLength):
            for s in range(0,self.numStates):
                # check if this is none
                if self.Trellis[s][t] is None:
                    continue
                else:
                    #get all states:
                    allStates = []
                    tempState = []
                    #only upward transitions
                    if(t >= self.trellisLength -2):
                        tempState.append(1)
                        for x in range(0,numBits-1):
                            tempState.append(self.Trellis[s][t].state[x])
                        allStates.append(tempState)
                    #both 1,-1
                    else:
                        for x in range(0,len(self.symbols)):
                            tempState = []
                            tempState.append(self.symbols[x])
                            for y in range(0, numBits-1):
                                tempState.append(self.Trellis[s][t].state[y])
                            allStates.append(tempState)
                    #now we have all states
                    for index in range(0,len(allStates)):
                        newState = allStates[index]
                        stateIndex = self.getStateIndex(newState)
                        prevNode = self.Trellis[s][t]
                        if self.Trellis[stateIndex][t+1] is None:
```

```
                            self.Trellis[stateIndex][t+1] = trellisNode(newState,t+1)

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

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

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

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

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

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

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

    # Function to calculate the cheapest path and in extension estimate the
    # symbols recieved. Must build the trellis before calling this function
```

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

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

        streamLength = len(self.entireStream)
        for x in range(0,streamLength):
            if(x < self.numHT or streamLength-x <= self.numHT):
```

```
                    continue
                else:
                    self.dataDetected.append(self.entireStream[x])

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

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

Listing 4. Python Source code for - to execute the MLSE Viterbi algorithm for signal estimation

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

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

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

    # Function to perform the Viterbi MLSE simulation
    # with a Gaussian random Channel Impulse Response
    # returns the BER array
    # simulation runs for @param self.numIterations, for each
    # SNR in the range (-4,9)
    def viterbiRandomCIR(self):
        numIter = self.numIterations
        print("Conducting a MLSE BPSK simulation with a Uniform Random CIR")
        BER = [] #store the average BER for each SNR
```

```
        for SNR in range(-4, 9): #9
            tempBER = [] #to store all the interations for one SNR
            for count in range(0, numIter):
                # generate 300 bits
                myDataBits = self.BpskSimulation.generateBits() #raw data bits
                # map bits to symbols
                modulatedSignal = self.BpskSimulation.BpskModulate(myDataBits)
                # add tail
                for x in range(0,len(self.linC) -1):
                    modulatedSignal.append(1)
                # add convolution
                randomCIR = self.generateRandomCIR(SNR)
                convolutedSignal = self.channelModification(randomCIR, modulatedSignal)
                signalSent = self.BpskSimulation.addNoise(SNR, convolutedSignal)
                # check MLSE
                myMLSE = MLSE(self.n, signalSent, randomCIR)
                myMLSE.buildTrellis()
                myMLSE.cheapestPath()
                # get data bits
                dataDetected = myMLSE.dataDetected
                # demodulate
                demodulatedSignal = self.BpskSimulation.BpskDemodulate(dataDetected)
                # compare
                tempBER.append(self.BpskSimulation.getNumErrors(myDataBits,demodulatedSignal)/self.n)
                del myMLSE
                del modulatedSignal
                del convolutedSignal
                del demodulatedSignal
            BER.append( (sum(tempBER)/len(tempBER) ))
        return BER

    # Function to perform the Viterbi MLSE simulation
    # with a Linear Declining Channel Impulse Response
    # returns the BER array
    # simulation runs for @param self.numIterations, for each
    # SNR in the range (-4,9)
    def viterbiLinearCIR(self):
        numIter = self.numIterations
        print("Conducting a MLSE BPSK simulation with a linear declining CIR")
        BER = [] #store the average BER for each SNR
        for SNR in range(-4, 9): #9
            tempBER = [] #to store all the interations for one SNR
            for count in range(0, numIter):

                # generate 300 bits
                myDataBits = self.BpskSimulation.generateBits() #raw data bits
                # map bits to symbols
                modulatedSignal = self.BpskSimulation.BpskModulate(
```

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

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

```
                # demodulate
                demodulatedSignal = self.BpskSimulation.BpskDemodulate(dataDetected)
                # compare
                tempBER.append(self.BpskSimulation.getNumErrors(myDataBits,demodulatedSignal)/self.n)
                del myDFE
                del modulatedSignal
                del convolutedSignal
                del demodulatedSignal
            BER.append((sum(tempBER)/len(tempBER)))
        return BER

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

    #returns a random CIR with "values" elements
```

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

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


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

```
    conducted with Viterbi MLSE and DFE Equalizers")
        plt.legend(loc='best')
        plt.show()
        print("Complete")

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

Listing 5. Python Source code to execute the simulation of all 4 simulations