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

EDC 310

Digital Communication

Practical 2: DFE and MLSE

Name and Surname	Student Number	Signature	% Contribution
Kwaku Bediako	u04465483		30
Lefa Raleting	u14222460		70

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.

October 10, 2020

Contents

1 Introduction

In the ideal case radio waves travel in a straight line (**Line of sight**) between the transmitter and receiver, meaning all transmitted signals arrive at the same time, however that's not always the case in the practical world, there are object/obstacles, with different material compositions that have different characteristic's.

When the radio waves come in contact with these objects they exhibit different behaviours namely, **reflection, refraction, absorption, scattering and diffraction**. This leads to an inherent characteristics of communication systems namely **Delay spread**. Delay spread is a result of a received signal that is dispersed in time, where each transmitted signal has its own path and every path has its on delay. This leads to inter-symbol interference.

The purpose of this paper is to establish the effect of multi-path channel in the received signal, where the Channel Impulse Response(**CIR**) is of length 3 (L=3). It distinguishes between the effect that a static and dynamic CIR has on the demodulation of the signal on the receiver end.

Two algorithms that are of interest when it comes to reversing the effect of multi-path in communication system are, Decision feedback equaliser(**DFE**) and Maximum likelihood sequence estimation(**MLSE**), which is discussed in the theoretical background. These algorithms will be tested on three modulation schemes namely **BPSK**(Binary Phase shift keying), **4-QAM**(Quadrature amplitude modulation) and **8PSK**(eight phase shit keying).

The experiment will be conducted with transmitted symbols being limited to data blocks of length 200(N=200). The Bit Error Rates(**BER**) will be compared to give a fair analysis and comparison between the different algorithms on the respective modulation schemes.

2 Theoretical background

2.1 DFE

DFE is a non linear equaliser, instead of using a simple scalar multiple of the received bits for detection, it uses past symbols(Memory) to better estimate signals that are passing through a channel. DFE reduces the interference of past symbols on the current symbol being detected on the receiver end.

For detection the DFE selects the estimate with the lowest cost and this cost is calculated as follows:

$$cost(t) = |r_t - \sum_{n=0}^{L-1} s_{t-n} * c_n|^2 \tag{1}$$

Where:

- r_t - Received bit at position t
- s_{t-n} - Transmitted bit at positing t-n
- c_n - Channel Impulse Response (CIR)
- L - Length of the CIR list

DFE is a sub-optimal algorithm, because it only works if the the channel response the symbols pass is of a minimum phase, this entails that the channel delay spread ratio to the power delay profile is decreasing.

2.2 MLSE

Maximum likelihood sequence estimation (**MLSE**) is a mathematical algorithm to extract useful data out of a noisy data stream. MLSE makes estimations on what a stream of data was, and tries to minimise the number of incorrect symbols

MLSE determines the whole transmitted symbol block at the receiver output , to determine which symbol has the highest probability to have been transmitted, by observing the entire received signal by the receiver end.

What gives MLSE an advantage is the nonlinear processing, this advantage can properly be realised through applying the viterbi algorithm . MLSE performance is measure by the rate of correct estimations. To increase this performance , it maximizes the probability that the correct sequence of symbols is found and decreasing the total path cost.

MSLE is a optimal algorithm, it will always provide the highest sequence estimation probaility even through a dynamic CIR.

3 Design/Method

3.1 DFE

To simulate communication between a transmitter and receiver of a multi-path environment, bits are generated randomly using a uniform number generator rated and mapped to symbols based on their respective modulation schemes constellation maps.

To simulate the noise that's normally added to the signal during transmission, additive white Gaussian noise (**AGWN**) is used to simulate this. The following formula is used to emulate this:

$$r_t = [\sum_{n=0}^{L-1} s_{t-n} * c_n] + \sigma * (\alpha + j\alpha) \tag{2}$$

Where:

- $\sigma = \frac{1}{\sqrt{10^{\frac{SNR}{10}} * 2 * log_2(M)}}$ and M is the number of symbols in the respective constellation map.
- α is a sample from a zero mean Gaussian distribution
- L is the length of the CIR

To detect the transmitted symbol, DFE algorithm is used. The code can be found bellow:

```
#---------------------------------------------------------------
#                        Sigma calculation
#-----------------------------------------------------------------

def sigma(domain,M):# M is the number of symbols
    sigma=[]
    for i in domain:
        sigma.append(1/np.sqrt(math.pow(10, (i/ 10)) * 2 * math.log2(M)))
    return sigma


#----------------------------------------------------------------------------
#                        create noise
#----------------------------------------------------------------------------

def noise(size,sigma):
    noiseList = np.random.normal(0,sigma,size)
    return noiseList

#------------------------------------------------------------------------
#                        add noise
```

```
# ---------------------------------------------------------------------------

def addnoise(transmitted,channels,L,m,snr):# assuming transmited comes with the memory symbols padded
    recieved=[]
    M=m
    k=snr

    sigma_=1/np.sqrt(math.pow(10, (k/ 10)) * 2 * math.log2(M))
    #print(sigma_)


    for i in range(L-1,len(transmitted)):
        sample=np.random.normal(0,1,1)[0]
        recieved.append(transmitted[i]*channels[0]+transmitted[i-1]*channels[0+1]
                        +transmitted[i-2]*channels[2]+sigma_*(sample+(sample)*1j))
    return recieved #without the padding

# ---------------------------------------------------------------------------
#                        DFE function
# ---------------------------------------------------------------------------

def DFE(recieved,channels,L,options,memory):
    Options =options# [1,-1]# these are the option available for bpsk
    symbols = memory#[1]*(L-1) #the first 1 is the memory symbols
    s=0 # symbol mover
    for i in range(0,len(recieved)):
        guess=[]
        n=len(channels)-1# length of the chanel L-1
        #calculating the product but from second position
        sumof=0
        for j in range(1,n):
            sumof+= symbols[n-1+s]*channels[j]
            n-=1

        for k in Options:
            #guess.append(np.abs(recieved[i]-((k)*channels[0]+symbols[1]*channels[1]+symbols[0]*channels[2]))**2)
            guess.append(np.abs(recieved[i]-((k)*channels[0]+sumof))**2)
        estimate=Options[guess.index(min(guess))]
        symbols.append(estimate)
        s+=1
    return symbols[L-1:] #final
# ---------------------------------------------------------------------------
```

3.2 MLSE

MLSE was designed for BPSK, 4QAM and 8PSK transmission. The deltas between from each symbol, to every other symbol was calculated and then starting at the appended symbols, the most likely bit was selected until the prepended symbols were reached.

The algorithms used to estimate the transmitted bits can be found in Appendix A

4 Simulations and Results

4.1 DFE

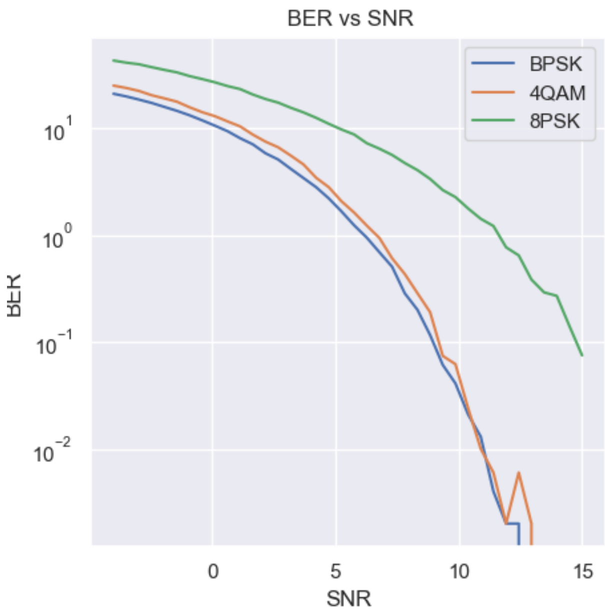

Figure 1: BER vs SNR : Static CIR

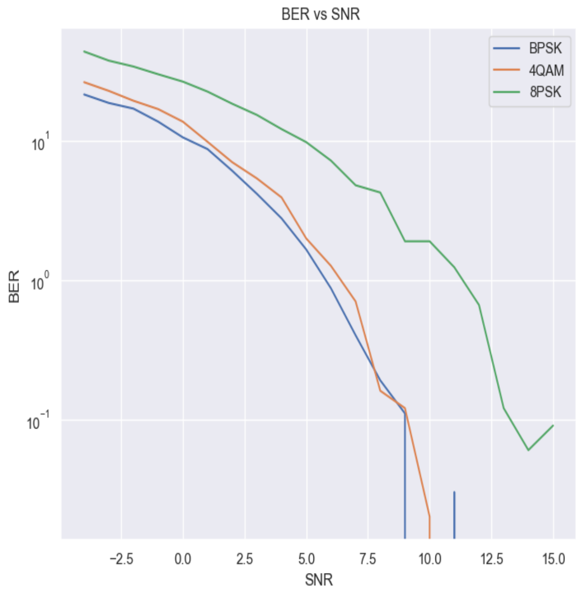

Figure 2: BER vs SNR : Dynamic CIR

4.2 MLSE

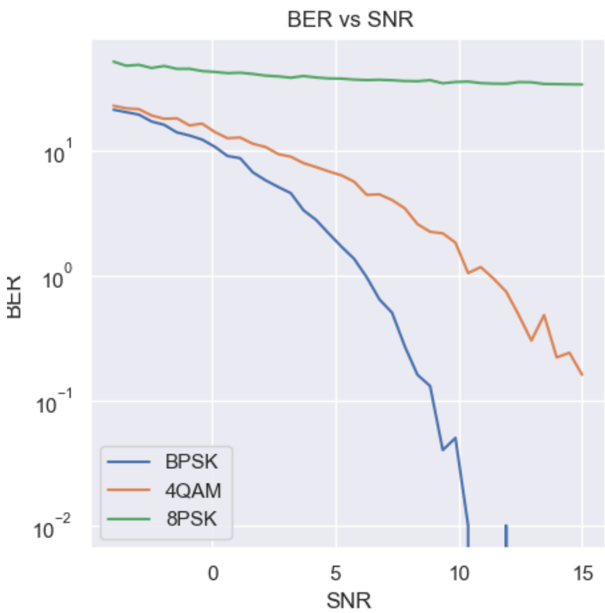

Figure 3: BER vs SNR : Static CIR

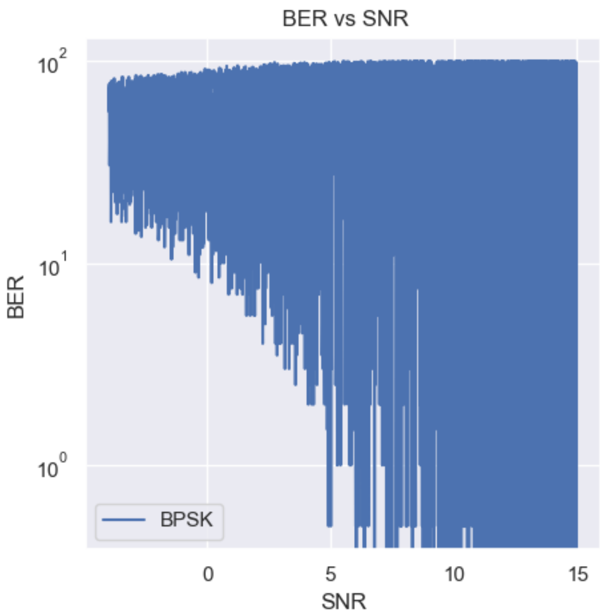

Figure 4: BER vs SNR : Dynamic CIR BPSK

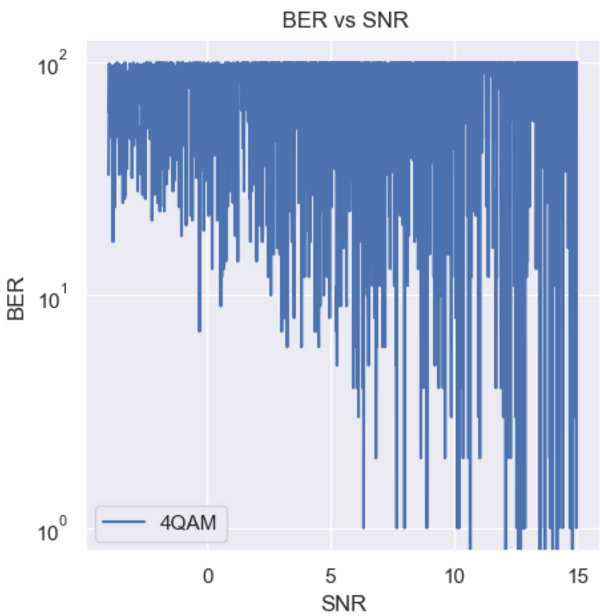

Figure 5: BER vs SNR : Dynamic CIR 4QAM

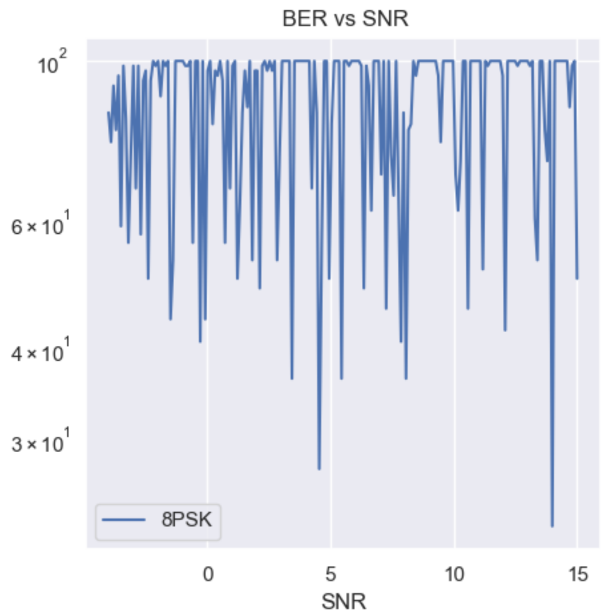

Figure 6: BER vs SNR : Dynamic CIR 8PSK

5 Discussion

DFE is a sub-optimal algorithm, because it only works if the the channel response the symbols pass is of a minimum phase, this entails that the channel delay spread ratio to the power delay profile is decreasing. Figure 2 shows the DFE BER vs SNR(Signal to noise Ration) for a dynamic CIR, the graph decays less smoother and a bit later.

If we compare compare it to the DFE BER vs SNR for a static CIR, it can be seen clearly that on a dynamic channel impulse response DFE performs much worse than it does on a static channel. However DFE is is fast because it needs not to check all possible paths, DFE performs much better with a decaying CIR.

It was found that with MLSE, that the bit error rate decreases more quickly for BPSL than when compared to DFE. Unexpectedly 4QAM was about as accurate for MLSE as it was for DFE, and 8PSK had a steady drop in error rate, but at a much slower rate than in DFE, remaining above 10 for all SNR values tested. As expected, BPSK continued the trend of having the lowest BER at all SNRs followed by 4QAM and then 8PSK.

With the dynamic channel, we see constant oscillation between a bit error rate of 100, and a chaninging minimum. This happens as a result of the channels switching after each block. Looking at the minimum BER, we can see faster drops in BPSK than in 4QAM, or 8PSK, but for SNR's of over 5, we register a BER of less than 1%. The BER for 8PSK also follows these general trends, however it never reaches a BER of less than 10%.

6 Conclusion

From the results, it was seen that DFE runs quicker than MLSE, which was expected, but surprisingly DFE gave a better more accurate results FOR static CIR. MLSE was more accurate when dealing with dynamic channels. Due to the fact that it is rare, for two wireless communicating devices to remain perfectly still through the entire transmission of data, we would expect MLSE to have superior practical use. The slower demodulation speed of MLSE is also not too concerning when one considers the current speed of processors, versus wireless transmission rates. Ways to improve Signal Accuracy, is to use techniques to boost SNR. These include use more power signals, but also allows for methods such as noise filtering.

7 References

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

[2] G. Marsaglia and T.A. Bray, "A convenient method for generating normal variables", SIAM Rev., Vol. 6, pp 260-264, 1964.

[3] A.Grami,"Baseband digital transmission, "Introduction to digital communication",(2016), pp 257-293

8 Appendix A

8.1 Code

8.1.1 MLSE: Static CIR

```
# -*- coding: utf-8 -*-
"""
Created on Thu Oct  1 08:28:37 2020

@author: Kwaku
"""
import random
import math
import statistics as st
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
import seaborn as sns




# settings for seaborn plotting style
sns.set(color_codes=True)
# settings for seaborn plot sizes
sns.set(rc={'figure.figsize':(5,5)})
np.random.seed(20)#just the to add a little bit of repeatbilty in the
    randomnes
#np.random.seed()
def theorwhichman(size):
    rand=[]
    for i in range(0,size):
        rand.append(random.uniform(0, 1))
    return rand
size=100

a=theorwhichman(size)

mu = st.mean(a)
sigma = st.stdev(a)

x = np.linspace(-1, 1, size)
a.sort()

print("Sigma:", sigma)
print("Mu:", mu)


#Creating bits to be transmitted
#
    ----------------------------------------------------------------------------
#                               Bit generator
#
    ----------------------------------------------------------------------------
```

```python
# random bits generator
def bits_gen(values): # function takes a list of values between 0 and 1
    data = []

    for i in values:
        #if the values is less than 0.5 append a 0
        if i < 0.5:
            data.append(0)
        else:
            #if the value found in the list is greater than 0.5 append 1
            data.append(1)

    return data

# ------------------------------------------------------------------------------
#                 mapping of bits to symbol using constellation maps
# ------------------------------------------------------------------------------
def BPSK(bits):
    bpsk = []
    for k in bits:
        if k == 1:
            bpsk.append(1)
        else:
            bpsk.append(-1)
    return bpsk

def fourQAM(bits):
    FQAM = []
    M = 2
    subList = [bits[n:n + M] for n in range(0, len(bits), M)]
    for k in subList:
        if k == [0, 0]:
            FQAM.append(complex(1 / np.sqrt(2), 1 / np.sqrt(2)))
        elif k == [0, 1]:
            FQAM.append(complex(-1 / np.sqrt(2), 1 / np.sqrt(2)))
        elif k == [1, 1]:
            FQAM.append(complex(-1 / np.sqrt(2), -1 / np.sqrt(2)))
        # elif(k==[1,0]):
        elif k == [1, 0]:
            FQAM.append(complex(1 / np.sqrt(2), -1 / np.sqrt(2)))

    return FQAM

def eight_PSK(bits):
    EPSK = []
    M = 3
    subList = [bits[n:n + M] for n in range(0, len(bits), M)]
    for k in subList:
        if k == [0, 0, 0]:
            EPSK.append(complex(1 , 0))
        elif k == [0, 0, 1]:
```

```
            EPSK.append((1+1j)/np.sqrt(2))
        elif k == [0, 1, 1]:
            EPSK.append( 1j)
        elif k == [0, 1, 0]:
            EPSK.append((-1+1j)/np.sqrt(2))
        elif k == [1, 1, 0]:
            EPSK.append(-1)
        elif k == [1, 1, 1]:
            EPSK.append((-1-1j)/np.sqrt(2))
        elif k == [1, 0, 1]:
            EPSK.append(-1j)
        elif k == [1, 0, 0]:
            EPSK.append((1-1j)/np.sqrt(2))
    return EPSK

#_______________________________________________________________
#                       Sigma calculation
#_________________________________________________________________

def sigma(domain,M):# M is the number of symbols
    sigma=[]
    for i in domain:
        sigma.append(1/np.sqrt(math.pow(10, (i/ 10)) * 2 * math.log2(M)))
    return sigma


#----------------------------------------------------------------------------
#                       create noise
#----------------------------------------------------------------------------

def noise(size,sigma):
    noiseList = np.random.normal(0,sigma,size)
    return noiseList

#------------------------------------------------------------------------
#                       add noise
#----------------------------------------------------------------------------

def addnoise(transmitted,channels,L,m,snr):# assuming transmited comes with the memory symbols padded
    recieved=[]
    M=m
    k=snr

    sigma_=1/np.sqrt(math.pow(10, (k/ 10)) * 2 * math.log2(M))
    #print(sigma_)


```

```
    for i in range(L-1,len(transmitted)):
        sample=np.random.normal(0,1,1)[0]
        recieved.append(transmitted[i]*channels[0]+transmitted[i-1]*channels[0+1]
                        +transmitted[i-2]*channels[2]+sigma_*(sample+(sample)*1j))
    return recieved #without the padding

#--------------------------------------------------------------------------
#                    DFE function
#------------------------------------------------------------------------------

def DFE(recieved,channels,L,options,memory):
    Options =options# [1,-1]# these are the option available for bpsk
    symbols = memory#[1]*(L-1) #the first 1 is the memory symbols
    s=0 # symbol mover
    for i in range(0,len(recieved)):
        guess=[]
        n=len(channels)-1# length of the chanel L-1
        #calculating the product but from second position
        sumof=0
        for j in range(1,n):
           sumof+= symbols[n-1+s]*channels[j]
           n-=1

        for k in Options:
            guess.append(np.abs(recieved[i]-((k)*channels[0]+sumof))**2)
        estimate=Options[guess.index(min(guess))]
        symbols.append(estimate)
        s+=1
    return symbols[L-1:] #final
#--------------------------------------------------------------------------
#                    Options and memory generator
#------------------------------------------------------------------------------

def OptMemGen(i,L):
    #BPSK=1 #4QAM=2 8PSK=3
    if i==1:
        Options = [1,-1]# these are the option available for bpsk
        memory= [1]*(L-1) #the first 1 is the memory symbols
        return Options,memory
    elif(i==2):
        Options = [(1+1j)/np.sqrt(2), (-1+1j)/np.sqrt(2), (-1-1j)/np.sqrt(2), (1-1j)/np.sqrt(2)]
        memory=[(1+1j)/np.sqrt(2)]*(L-1)
        return Options,memory
    elif(i==3):
         Options=[1, (1+1j)/np.sqrt(2), 1j, (-1+1j)/np.sqrt(2), -1, (-1-1j)/np.sqrt(2), -1j, (1-1j)/np.sqrt(2)]
```

```python
         memory=[(1+1j)/np.sqrt(2)]*(L-1)
         return Options,memory

# ------------------------------------------------------------------------
#                        Bit Error calculation
# ----------------------------------------------------------------------------

def bit_errors(sent, recieved):
    error = 0
    for k in range(0,len(recieved)):
        if sent[k] != recieved[k]:
            error += 1
    BER = error / len(recieved)*100
    return BER

#transmitted=[1,1,1,-1,1,-1,1]
#channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
#Recieved=addnoise(transmitted,channels,3)#[1.5,1.2,1,-1.2,-1.5,0.2]
#print(Recieved)


# ############################################################################
"""

Lefa's graph function















"""
# ############################################################################



sqrt2 = np.sqrt(2)
c = [0.89 + 0.92j, 0.42 - 0.37j, 0.19 + 0.12j]
N = 200


```

```
def Reverse(x):
  return x [::-1]

# generate deltas for BPSK
def findDeltaBPSK(recieved, Symbols =[[]],i=0):
    Options = [-1,1]
    delta = []
    for s in Symbols:
        for j in Options:
            delta.append(np.abs(recieved[i] - (j*c[0] + s[1]*c[1] + s
    [0]*c[2]) )**2)
    return delta

# generate deltas for 4QAM
def findDelta4QAM(recieved, Symbols =[[]],i=0):
    Options = [(1+1j)/sqrt2, (-1+1j)/sqrt2, (-1-1j)/sqrt2, (1-1j)/
    sqrt2]
    delta = []
    for s in Symbols:
        for j in Options:
            delta.append(np.abs(recieved[i] - (j*c[0] + s[1]*c[1] + s
    [0]*c[2]) )**2)

    return delta

# generate deltas for 8PSK
def findDelta8PSK(recieved, Symbols =[[]],i=0):
    Options = [1, (1+1j)/sqrt2, 1j, (-1+1j)/sqrt2, -1, (-1-1j)/sqrt2,
    -1j, (1-1j)/sqrt2]
    delta = []
    for s in Symbols:
        for j in Options:
            delta.append(np.abs(recieved[i] - (j*c[0] + s[1]*c[1] + s
    [0]*c[2]) )**2)
    return delta


def BPSK_MLSE(recieved, N):
    # Generate the bits, then get their symbols from the constellation
    map
    Bits = []
    for i in range(8):
        Bits.append(format(8+i, 'b'))
    Symbols = []
    # print(Bits)
    for i in range(8):
        Sym = []
        for j in range(3):
            if Bits[i][j+1] == "1":
                Sym.append(1)
            else:
                Sym.append(-1)
        Symbols.append(Sym)
    # print(Symbols)
    # Get all the deltas
    deltas = []
    for i in range(N):
```

```
        deltas.append(findDeltaBPSK(recieved, Symbols ,i))

    # Using the deltas, work backwards and determine the transmitted sequence
    transmitted = []
    for i in range(N):
        cost = min(deltas[N-1-i])
        bit = deltas[N-1-i].index(cost)

        # print(bit)
        if bit % 2 == 0:
            transmitted.append(-1)
        else:
            transmitted.append(1)
    transmitted = Reverse(transmitted)
    return transmitted

def MLSE_4QAM(recieved, N):
    # Generate the bits, then get their symbols from the constellation map
    Bits = []
    for i in range(64):
        Bits.append(format(64+i, 'b'))
    Symbols = []
    i = 0
    while i < 64:
        Sym = []
        for j in range(6):
            if Bits[i][j+1:j+3] == "00":
                Sym.append((1+1j)/sqrt2)
            elif Bits[i][j+1:j+3] == "01":
                Sym.append((-1+1j)/sqrt2)
            elif Bits[i][j+1:j+3] == "11":
                Sym.append((-1-1j)/sqrt2)
            elif Bits[i][j+1:j+3] == "10":
                Sym.append((1-1j)/sqrt2)
        Symbols.append(Sym)
        i += 2

    # Get all the deltas
    deltas = []
    for i in range(N):
        deltas.append(findDelta4QAM(recieved, Symbols ,i))

    # print(deltas)
    # Using the deltas, work backwards and determine the transmitted sequence
    transmitted = []
    for i in range(N):
        cost = min(deltas[N-1-i])
        bit = deltas[N-1-i].index(cost)

        if bit % 4 == 0: #recieved (1+1j)/sqrt2
            transmitted.append((1+1j)/sqrt2)
        elif bit % 4 == 1: #recieved (-1+1j)/sqrt2
            transmitted.append((-1+1j)/sqrt2)
        elif bit % 4 == 2: #recieved (-1-1j)/sqrt2
```

```
            transmitted.append((-1-1j)/sqrt2)
        elif bit % 4 == 3: #recieved (1-1j)/sqrt2
            transmitted.append((1-1j)/sqrt2)

    transmitted = Reverse(transmitted)

    return transmitted

def MLSE_8PSK(recieved = [], N = 4):
    # Generate the bits, then get their symbols from the constellation map
    L=3
    Bits = []

    for i in range(512):
        Bits.append(format(512+i, 'b'))
    Symbols = []
    i = 0
    Sym = []
    while i < 8**L:
        Sym = []
        for j in range(9):
            if Bits[i][j+1:j+4]  == "111":
                Sym.append(1)
            elif Bits[i][j+1:j+4] == "110":
                Sym.append((1+1j)/sqrt2)
            elif Bits[i][j+1:j+4] == "010":
                Sym.append(1j)
            elif Bits[i][j+1:j+4] == "011":
                Sym.append((-1+1j)/sqrt2)
            elif Bits[i][j+1:j+4] == "001":
                Sym.append(-1)
            elif Bits[i][j+1:j+4] == "000":
                Sym.append((-1-1j)/sqrt2)
            elif Bits[i][j+1:j+4] == "100":
                Sym.append(-1j)
            elif Bits[i][j+1:j+4] == "101":
                Sym.append((1-1j)/sqrt2)
        Symbols.append(Sym)
        i += 3


    # Get all the deltas
    deltas = []
    for i in range(N):
        deltas.append(findDelta8PSK(recieved, Symbols ,i))

    # print(deltas)
    # Using the deltas, work backwards and determine the transmitted sequence
    transmitted = []
    for i in range(N):
        cost = min(deltas[N-1-i])
        bit = deltas[N-1-i].index(cost)

        if bit % 8 == 0:
            transmitted.append(1)
```

```
        elif bit % 8 == 1:
            transmitted.append((1+1j)/sqrt2)
        elif bit % 8 == 2:
            transmitted.append(1j)
        elif bit % 8 == 3:
            transmitted.append((-1+1j)/sqrt2)
        elif bit % 8 == 4:
            transmitted.append(-1)
        elif bit % 8 == 5:
            transmitted.append((-1-1j)/sqrt2)
        elif bit % 8 == 6:
            transmitted.append(-1j)
        elif bit % 8 == 7:
            transmitted.append((1-1j)/sqrt2)

    transmitted = Reverse(transmitted)
    return transmitted

def Graph():
    size=N
    randomValues= theorwhichman(size)
    bits=bits_gen(randomValues)
    BPSK_bits=BPSK(bits)
    FourQAM_bits=fourQAM(bits)
    EBPSK_bits=eight_PSK(bits)
    channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
    transmitted=[]
    xValues = np.linspace(-4, 15, 38)
    yvalues=[]
    #bpsk

    L=3
    M=2
    a,transmitted=OptMemGen(1,3)
    transmitted.extend(BPSK_bits)
    k=-4
    while (k<15):


        Recieved=addnoise(transmitted,channels,L,M,k)#[1.5,1.2,1,-1.2,-1.5,0.2]
        #print(Recieved)
        Options,memory= OptMemGen(1,L)#bpsk 1, 4Qam,8psk
        Detected=BPSK_MLSE(Recieved,size)
        yvalues.append(bit_errors(transmitted[L-1:],Detected))
        k+=0.5
    plt.semilogy(xValues,yvalues, label="BPSK")
    plt.ylabel('BER')
    plt.xlabel('SNR')


    yvalues=[]

    #4Qam
    L=3
    M=4
    a,transmitted=OptMemGen(2,3)
```

```
    transmitted.extend(FourQAM_bits)
    k=-4
    while (k<15):


        Recieved=addnoise(transmitted,channels,L,M,k)#
    [1.5,1.2,1,-1.2,-1.5,0.2]
        Options,memory= OptMemGen(2,L)#bpsk 1, 4Qam,8psk
        Detected=MLSE_4QAM(Recieved,int(size/2))
        yvalues.append(bit_errors(transmitted[L-1:],Detected))
        k+=0.5


    plt.semilogy(xValues,yvalues, label="4QAM")
    plt.ylabel('BER')
    plt.xlabel('SNR')
    yvalues=[]

    L=3
    M=8 #BPSK=2 4Qam=4 8psk=8
    #8psk
    a,transmitted=OptMemGen(3,3)
    transmitted.extend(EBPSK_bits)

    k=-4
    while (k<15):


        Recieved=addnoise(transmitted,channels,L,M,k)#
    [1.5,1.2,1,-1.2,-1.5,0.2]
        Options,memory= OptMemGen(3,L)#bpsk 1, 4Qam,8psk
        Detected=MLSE_8PSK(Recieved,int(size/3))
        yvalues.append(bit_errors(transmitted[L-1:],Detected))
        k+=0.5

    plt.semilogy(xValues,yvalues, label="8PSK")
    plt.ylabel('BER')
    plt.xlabel('SNR')
    plt.title(" BER vs SNR")
    plt.legend()
# print(BPSK_MLSE(r,N))
#
#print(MLSE_4QAM(r,N))
#
Graph()
```

8.1.2 DFE: Static CIR

```
# -*- coding: utf-8 -*-
"""
Created on Fri Oct  2 22:11:23 2020

@author: user
"""

import random
import math
import statistics as st
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
import seaborn as sns
# settings for seaborn plotting style
sns.set(color_codes=True)
# settings for seaborn plot sizes
sns.set(rc={'figure.figsize':(5,5)})
np.random.seed(20)#just the to add a little bit of repeatbilty in the randomnes
#np.random.seed()
def theorwhichman(size):
    rand=[]
    for i in range(0,size):
        rand.append(random.uniform(0, 1))
    return rand
size=100

a=theorwhichman(size)

mu = st.mean(a)
sigma = st.stdev(a)

x = np.linspace(-1, 1, size)
a.sort()



"""ax = sns.distplot(a,
                  bins=100,
                  kde=True,
                  color='skyblue',
                  hist_kws={"linewidth": 15,'alpha':1})
ax.set(xlabel='Normal Distribution', ylabel='Frequency')
#[Text(0,0.5,u'Frequency'), Text(0.5,0,u'Normal Distribution')]
"""
#plt.plot(a, norm(0.5, 1).pdf(a))
#plt.ylabel('Probability Density')
#plt.xlabel('Randomly Generated Numbers')
#plt.show()

print("Sigma:", sigma)
print("Mu:", mu)


#Creating bits to be transmitted
# ----------------------------------------------------------------------------
#                              Bit generator
# ----------------------------------------------------------------------------
# random bits generator
def bits_gen(values): # function takes a list of values between 0 and 1
    data = []
```

```
    for i in values:
        #if the values is less than 0.5 append a 0
        if i < 0.5:
            data.append(0)
        else:
            #if the value found in the list is greater than 0.5 append 1
            data.append(1)

    return data

# --------------------------------------------------------------------------------
#                mapping of bits to symbol using constellation maps
# ------------------------------------------------------------------------------
def BPSK(bits):
    bpsk = []
    for k in bits:
        if k == 1:
            bpsk.append(1)
        else:
            bpsk.append(-1)
    return bpsk

def fourQAM(bits):
    FQAM = []
    M = 2
    subList = [bits[n:n + M] for n in range(0, len(bits), M)]
    for k in subList:
        if k == [0, 0]:
            FQAM.append(complex(1 / np.sqrt(2), 1 / np.sqrt(2)))
        elif k == [0, 1]:
            FQAM.append(complex(-1 / np.sqrt(2), 1 / np.sqrt(2)))
        elif k == [1, 1]:
            FQAM.append(complex(-1 / np.sqrt(2), -1 / np.sqrt(2)))
        # elif(k==[1,0]):
        elif k == [1, 0]:
            FQAM.append(complex(1 / np.sqrt(2), -1 / np.sqrt(2)))

    return FQAM

def eight_PSK(bits):
    EPSK = []
    M = 3
    subList = [bits[n:n + M] for n in range(0, len(bits), M)]
    for k in subList:
        if k == [0, 0, 0]:
            EPSK.append(complex(1 , 0))
        elif k == [0, 0, 1]:
            EPSK.append((1+1j)/np.sqrt(2))
        elif k == [0, 1, 1]:
            EPSK.append( 1j)
        elif k == [0, 1, 0]:
```

```
            EPSK.append((-1+1j)/np.sqrt(2))
        elif k == [1, 1, 0]:
            EPSK.append(-1)
        elif k == [1, 1, 1]:
            EPSK.append((-1-1j)/np.sqrt(2))
        elif k == [1, 0, 1]:
            EPSK.append(-1j)
        elif k == [1, 0, 0]:
            EPSK.append((1-1j)/np.sqrt(2))
    return EPSK

#----------------------------------------------------------------
#                        Sigma calculation
#------------------------------------------------------------------

def sigma(domain,M):# M is the number of symbols
    sigma=[]
    for i in domain:
        sigma.append(1/np.sqrt(math.pow(10, (i/ 10)) * 2 * math.log2(M)))
    return sigma


# ------------------------------------------------------------------------------
#                        create noise
# ------------------------------------------------------------------------------

def noise(size,sigma):
    noiseList = np.random.normal(0,sigma,size)
    return noiseList

# --------------------------------------------------------------------------
#                        add noise
# ------------------------------------------------------------------------------

def addnoise(transmitted,channels,L,m,snr):# assuming transmited comes with the memory symbols padded
    recieved=[]
    M=m
    k=snr

    sigma_=1/np.sqrt(math.pow(10, (k/ 10)) * 2 * math.log2(M))
    #print(sigma_)


    for i in range(L-1,len(transmitted)):
        sample=np.random.normal(0,1,1)[0]
        recieved.append(transmitted[i]*channels[0]+transmitted[i-1]*channels[0+1]
```

```
                        +transmitted[i-2]*channels[2]+sigma_*(sample+(sample)*1j))
    return recieved #without the padding

# ----------------------------------------------------------------------
#                       DFE function
# ------------------------------------------------------------------------------

def DFE(recieved,channels,L,options,memory):
    Options =options# [1,-1]# these are the option available for bpsk
    symbols = memory#[1]*(L-1) #the first 1 is the memory symbols
    s=0 # symbol mover
    for i in range(0,len(recieved)):
        guess=[]
        n=len(channels)-1# length of the chanel L-1
        #calculating the product but from second position
        sumof=0
        for j in range(1,n):
           sumof+= symbols[n-1+s]*channels[j]
           n-=1

        for k in Options:
            #guess.append(np.abs(recieved[i]-((k)*channels[0]+symbols[1]*channels[1]+symbols[0]*channels[2]))**2)
            guess.append(np.abs(recieved[i]-((k)*channels[0]+sumof))**2)
        estimate=Options[guess.index(min(guess))]
        symbols.append(estimate)
        s+=1
    return symbols[L-1:] #final
# ----------------------------------------------------------------------
#                       Options and memory generator
# ------------------------------------------------------------------------------

def OptMemGen(i,L):
    #BPSK=1 #4QAM=2 8PSK=3
    if i==1:
        Options = [1,-1]# these are the option available for bpsk
        memory= [1]*(L-1) #the first 1 is the memory symbols
        return Options,memory
    elif(i==2):
        Options = [(1+1j)/np.sqrt(2), (-1+1j)/np.sqrt(2), (-1-1j)/np.sqrt(2), (1-1j)/np.sqrt(2)]
        memory=[(1+1j)/np.sqrt(2)]*(L-1)
        return Options,memory
    elif(i==3):
         Options=[1, (1+1j)/np.sqrt(2), 1j, (-1+1j)/np.sqrt(2), -1, (-1-1j)/np.sqrt(2), -1j, (1-1j)/np.sqrt(2)]
         memory=[(1+1j)/np.sqrt(2)]*(L-1)
         return Options,memory
```

```
# --------------------------------------------------------------------------
#                        Bit Error calculation
# ------------------------------------------------------------------------------

def bit_errors(sent, recieved):
    error = 0
    for k in range(0,len(recieved)):
        if sent[k] != recieved[k]:
            error += 1
    BER = error / len(recieved)*100

    return BER

#transmitted=[1,1,1,-1,1,-1,1]
#channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
#Recieved=addnoise(transmitted,channels,3)#[1.5,1.2,1,-1.2,-1.5,0.2]
#print(Recieved)

def yvalueCal(transmitted):
    channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
    L=3
    M=2 #for bpsk=2
    #SNR=15
    yvalues=[]
    #print(transmitted)

    for k in range (-4,16):


        Recieved=addnoise(transmitted,channels,L,M,k)#[1.5,1.2,1,-1.2,-1.5,0.2]
        #print(Recieved)
        Options,memory= OptMemGen(1,L)
        Detected=DFE(Recieved,channels,L,Options,memory)
        yvalues.append(bit_errors(transmitted[L-1:],Detected))
    return yvalues
#print(yvalueCal(transmitted))

def grapghs():
    size=100000
    randomValues= theorwhichman(size)
    bits=bits_gen(randomValues)
    BPSK_bits=BPSK(bits)
    FourQAM_bits=fourQAM(bits)
    EBPSK_bits=eight_PSK(bits)
    channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]

    transmitted=[]
    xValues = np.linspace(-4, 15, 38)
    yvalues=[]
    #bpsk

```

```
    L=3
    M=2
    a,transmitted=OptMemGen(1,3)
    transmitted.extend(BPSK_bits)
    k=-4
    while (k<15):


        Recieved=addnoise(transmitted,channels,L,M,k)#[1.5,1.2,1,-1.2,-1.5,0.2]
        #print(Recieved)
        Options,memory= OptMemGen(1,L)#bpsk 1, 4Qam,8psk
        Detected=DFE(Recieved,channels,L,Options,memory)
        yvalues.append(bit_errors(transmitted[L-1:],Detected))
        k+=0.5
    plt.semilogy(xValues,yvalues, label="BPSK")
    plt.ylabel('BER')
    plt.xlabel('SNR')


    yvalues=[]
    #4Qam
    L=3
    M=4
    a,transmitted=OptMemGen(2,3)
    transmitted.extend(FourQAM_bits)
    k=-4
    while (k<15):


        Recieved=addnoise(transmitted,channels,L,M,k)#[1.5,1.2,1,-1.2,-1.5,0.2]
        #print(Recieved)
        Options,memory= OptMemGen(2,L)#bpsk 1, 4Qam,8psk
        Detected=DFE(Recieved,channels,L,Options,memory)
        yvalues.append(bit_errors(transmitted[L-1:],Detected))
        k+=0.5


    plt.semilogy(xValues,yvalues, label="4QAM")
    plt.ylabel('BER')
    plt.xlabel('SNR')
    yvalues=[]

    L=3
    M=8 #BPSK=2 4Qam=4 8psk=8
    #8psk
    a,transmitted=OptMemGen(3,3)
    transmitted.extend(EBPSK_bits)

    k=-4
    while (k<15):
        Recieved=addnoise(transmitted,channels,L,M,k)#[1.5,1.2,1,-1.2,-1.5,0.2]
        #print(Recieved)
        Options,memory= OptMemGen(3,L)#bpsk 1, 4Qam,8psk
        Detected=DFE(Recieved,channels,L,Options,memory)
```

```
        yvalues.append(bit_errors(transmitted[L-1:],Detected))
        k+=0.5

    plt.semilogy(xValues,yvalues, label="8PSK")
    plt.ylabel('BER')
    plt.xlabel('SNR')
    plt.title(" BER vs SNR")
    plt.legend()

graphs()  #uncomment this if you want the graphs


def tester():
    size=10000
    #np.random.seed(420)
    print("Generating the random number generator of size 200")
    randomValues= theorwhichman(size)


    print("\nExpected sigma =0.29 and expected mu=0.50")
    print("Sigma",st.mean(randomValues))
    print("mu",st.stdev(randomValues))

    print("\nTesting the bits_gen function of size 200")
    bits=bits_gen(randomValues)
    print("Size:",len(bits))

    print("\nNow testing the mapping of sysmbols for different
   modulation schemes")
    print("BPSK expected length 200")
    BPSK_bits=BPSK(bits)
    print("BPSK length:",len(BPSK_bits))
    print("4QAM expected length =100")
    #FourQAM_bits=fourQAM(bits)
    #print("4QAM length:",len(FourQAM_bits))
    print("8BPSK expected length 66")
    #EBPSK_bits=eight_PSK(bits)
    #print("8BPSK length:",len(EBPSK_bits))
    #SNR = np.linspace(0, 15, 16)
    #print(SNR)
    #print("\nGenerating noise for 8psk")
    #noiseList=noise(len(EBPSK_bits),sigma(SNR,8)[0])
    #print(noiseList)
    #print("Length of noise:",len(noiseList))
    #print("variance:" ,st.variance(noiseList))

    print("\nTesting the DFE function")
    channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
    #bpsk
    transmitted=[1,1]
    transmitted.extend(BPSK_bits)
    #QPSK

    L=3
    M=2 #for bpsk=2
    SNR=-1

```

```
    print("Adding Noise")
    #Recieved=addnoise(transmitted,channels,L,M,SNR)
   #[1.5,1.2,1,-1.2,-1.5,0.2]
    #Options,memory= OptMemGen(1,L)
    #print("Received symbols:",Recieved)
    #Detected=DFE(Recieved,channels,L,Options,memory)
    #print("Detected Symbols",Detected)
    xValues = np.linspace(-4, 15, 20)
    yvalues=yvalueCal(transmitted)
    print(yvalues)
    plt.semilogy(xValues,yvalues)



#tester()
```

8.1.3 MLSE: Dynamic CIR

```
# -*- coding: utf-8 -*-
"""
Created on Thu Oct  1 08:28:37 2020

@author: Kwaku
"""
import random
import math
import statistics as st
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
import seaborn as sns




# settings for seaborn plotting style
sns.set(color_codes=True)
# settings for seaborn plot sizes
sns.set(rc={'figure.figsize':(5,5)})
np.random.seed(20)#just the to add a little bit of repeatbilty in the
   randomnes
#np.random.seed()
def theorwhichman(size):
    rand=[]
    for i in range(0,size):
        rand.append(random.uniform(0, 1))
    return rand
size=100

a=theorwhichman(size)

mu = st.mean(a)
sigma = st.stdev(a)

x = np.linspace(-1, 1, size)
a.sort()

print("Sigma:", sigma)
```

```python
print("Mu:", mu)


#Creating bits to be transmitted
# -----------------------------------------------------------------------------
#                                Bit generator
# -----------------------------------------------------------------------------
# random bits generator
def bits_gen(values): # function takes a list of values between 0 and 1
    data = []

    for i in values:
        #if the values is less than 0.5 append a 0
        if i < 0.5:
            data.append(0)
        else:
            #if the value found in the list is greater than 0.5 append 1
            data.append(1)

    return data

# --------------------------------------------------------------------------------
#                mapping of bits to symbol using constellation maps
# ------------------------------------------------------------------------------
def BPSK(bits):
    bpsk = []
    for k in bits:
        if k == 1:
            bpsk.append(1)
        else:
            bpsk.append(-1)
    bpsk.append(1)
    return bpsk

def fourQAM(bits):
    FQAM = []
    M = 2
    subList = [bits[n:n + M] for n in range(0, len(bits), M)]
    for k in subList:
        if k == [0, 0]:
            FQAM.append(complex(1 / np.sqrt(2), 1 / np.sqrt(2)))
        elif k == [0, 1]:
            FQAM.append(complex(-1 / np.sqrt(2), 1 / np.sqrt(2)))
        elif k == [1, 1]:
            FQAM.append(complex(-1 / np.sqrt(2), -1 / np.sqrt(2)))
        # elif(k==[1,0]):
        elif k == [1, 0]:
```

```
            FQAM.append(complex(1 / np.sqrt(2), -1 / np.sqrt(2)))
    FQAM.append((1+1j)/sqrt2)
    return FQAM

def eight_PSK(bits):
    EPSK = []
    M = 3
    subList = [bits[n:n + M] for n in range(0, len(bits), M)]
    for k in subList:
        if k == [0, 0, 0]:
            EPSK.append(complex(1 , 0))
        elif k == [0, 0, 1]:
            EPSK.append((1+1j)/np.sqrt(2))
        elif k == [0, 1, 1]:
            EPSK.append( 1j)
        elif k == [0, 1, 0]:
            EPSK.append((-1+1j)/np.sqrt(2))
        elif k == [1, 1, 0]:
            EPSK.append(-1)
        elif k == [1, 1, 1]:
            EPSK.append((-1-1j)/np.sqrt(2))
        elif k == [1, 0, 1]:
            EPSK.append(-1j)
        elif k == [1, 0, 0]:
            EPSK.append((1-1j)/np.sqrt(2))
    EPSK.append(1)
    return EPSK

#----------------------------------------------------------------
#                        Sigma calculation
#------------------------------------------------------------------

def sigma(domain,M):# M is the number of symbols
    sigma=[]
    for i in domain:
        sigma.append(1/np.sqrt(math.pow(10, (i/ 10)) * 2 * math.log2(M)))
    return sigma


#
  ----------------------------------------------------------------------------
#                        create noise
#
  ----------------------------------------------------------------------------

def noise(size,sigma):
    noiseList = np.random.normal(0,sigma,size)
    return noiseList

#
  ------------------------------------------------------------------------
#                        add noise
#
```

```
# ------------------------------------------------------------------------------

def addnoise(transmitted,channels,L,m,snr, sigma_):# assuming transmited comes with the memory symbols padded
    recieved=[]
    M=m
    k=snr


    #print(sigma_)


    for i in range(L-1,len(transmitted)):
        sample=np.random.normal(0,1,1)[0]
        recieved.append(transmitted[i]*channels[0]+transmitted[i-1]*channels[0+1]
                        +transmitted[i-2]*channels[2]+sigma_*(sample+(sample)*1j))
    return recieved #without the padding

# --------------------------------------------------------------------------
#                       DFE function
# ------------------------------------------------------------------------------

def DFE(recieved,channels,L,options,memory):
    Options =options# [1,-1]# these are the option available for bpsk
    symbols = memory#[1]*(L-1) #the first 1 is the memory symbols
    s=0 # symbol mover
    for i in range(0,len(recieved)):
        guess=[]
        n=len(channels)-1# length of the chanel L-1
        #calculating the product but from second position
        sumof=0
        for j in range(1,n):
           sumof+= symbols[n-1+s]*channels[j]
           n-=1

        for k in Options:
            guess.append(np.abs(recieved[i]-((k)*channels[0]+sumof))**2)
        estimate=Options[guess.index(min(guess))]
        symbols.append(estimate)
        s+=1
    return symbols[L-1:] #final
# --------------------------------------------------------------------------
#                       Options and memory generator
# ------------------------------------------------------------------------------
def OptMemGen(i,L):
```

```
    #BPSK=1 #4QAM=2 8PSK=3
    if i==1:
        Options = [1,-1]# these are the option available for bpsk
        memory= [1]*(L-1) #the first 1 is the memory symbols
        return Options,memory
    elif(i==2):
        Options = [(1+1j)/np.sqrt(2), (-1+1j)/np.sqrt(2), (-1-1j)/np.
    sqrt(2), (1-1j)/np.sqrt(2)]
        memory=[(1+1j)/np.sqrt(2)]*(L-1)
        return Options,memory
    elif(i==3):
         Options=[1, (1+1j)/np.sqrt(2), 1j, (-1+1j)/np.sqrt(2), -1,
    (-1-1j)/np.sqrt(2), -1j, (1-1j)/np.sqrt(2)]
         memory=[(1+1j)/np.sqrt(2)]*(L-1)
         return Options,memory

#
    ---------------------------------------------------------------------------
#                       Bit Error calculation
#
    -------------------------------------------------------------------------------

def bit_errors(sent, recieved):
    error = 0
    for k in range(0,len(recieved)):
        if sent[k] != recieved[k]:
            error += 1
    BER = error / len(recieved)*100
    return BER

#transmitted=[1,1,1,-1,1,-1,1]
#channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
#Recieved=addnoise(transmitted,channels,3)#[1.5,1.2,1,-1.2,-1.5,0.2]
#print(Recieved)


#
    ###############################################################################
"""

Lefa's graph function
```

```
"""
# ###############################################################################



sqrt2 = np.sqrt(2)
CIR = [0.89 + 0.92j, 0.42 - 0.37j, 0.19 + 0.12j]
N = 200


def Reverse(x):
  return x [::-1]

# generate deltas for BPSK
def findDeltaBPSK(recieved, Symbols =[[]],i=0, c = CIR):
    Options = [-1,1]
    delta = []
    for s in Symbols:
        for j in Options:
            delta.append(np.abs(recieved[i] - (j*c[0] + s[1]*c[1] + s[0]*c[2]) )**2)
    return delta

# generate deltas for 4QAM
def findDelta4QAM(recieved, Symbols =[[]],i=0, c = CIR):
    Options = [(1+1j)/sqrt2, (-1+1j)/sqrt2, (-1-1j)/sqrt2, (1-1j)/sqrt2]
    delta = []
    for s in Symbols:
        for j in Options:
            delta.append(np.abs(recieved[i] - (j*c[0] + s[1]*c[1] + s[0]*c[2]) )**2)

    return delta

# generate deltas for 8PSK
def findDelta8PSK(recieved, Symbols =[[]],i=0, c = CIR):
    Options = [1, (1+1j)/sqrt2, 1j, (-1+1j)/sqrt2, -1, (-1-1j)/sqrt2, -1j, (1-1j)/sqrt2]
    delta = []
    for s in Symbols:
        for j in Options:
            delta.append(np.abs(recieved[i] - (j*c[0] + s[1]*c[1] + s[0]*c[2]) )**2)
    return delta


def BPSK_MLSE(recieved, N, c):
    # Generate the bits, then get their symbols from the constellation map
    Bits = []
    for i in range(8):
        Bits.append(format(8+i, 'b'))
    Symbols = []
```

```
    # print(Bits)
    for i in range(8):
        Sym = []
        for j in range(3):
            if Bits[i][j+1] == "1":
                Sym.append(1)
            else:
                Sym.append(-1)
        Symbols.append(Sym)
    # print(Symbols)
    # Get all the deltas
    deltas = []
    for i in range(N):
        deltas.append(findDeltaBPSK(recieved, Symbols ,i, c))

    # Using the deltas, work backwards and determine the transmitted sequence
    transmitted = []
    for i in range(N):
        cost = min(deltas[N-1-i])
        bit = deltas[N-1-i].index(cost)

        # print(bit)
        if bit % 2 == 0:
            transmitted.append(-1)
        else:
            transmitted.append(1)
    transmitted = Reverse(transmitted)
    return transmitted

def MLSE_4QAM(recieved, N, c):
    # Generate the bits, then get their symbols from the constellation map
    Bits = []
    for i in range(64):
        Bits.append(format(64+i, 'b'))
    Symbols = []
    i = 0
    while i < 64:
        Sym = []
        for j in range(6):
            if Bits[i][j+1:j+3] == "00":
                Sym.append((1+1j)/sqrt2)
            elif Bits[i][j+1:j+3] == "01":
                Sym.append((-1+1j)/sqrt2)
            elif Bits[i][j+1:j+3] == "11":
                Sym.append((-1-1j)/sqrt2)
            elif Bits[i][j+1:j+3] == "10":
                Sym.append((1-1j)/sqrt2)
        Symbols.append(Sym)
        i += 2

    # Get all the deltas
    deltas = []
    for i in range(N):
        deltas.append(findDelta4QAM(recieved, Symbols ,i, c))

```

```
    # print(deltas)
    # Using the deltas, work backwards and determine the transmitted sequence
    transmitted = []
    for i in range(N):
        cost = min(deltas[N-1-i])
        bit = deltas[N-1-i].index(cost)

        if bit % 4 == 0: #recieved (1+1j)/sqrt2
            transmitted.append((1+1j)/sqrt2)
        elif bit % 4 == 1: #recieved (-1+1j)/sqrt2
            transmitted.append((-1+1j)/sqrt2)
        elif bit % 4 == 2: #recieved (-1-1j)/sqrt2
            transmitted.append((-1-1j)/sqrt2)
        elif bit % 4 == 3: #recieved (1-1j)/sqrt2
            transmitted.append((1-1j)/sqrt2)

    transmitted = Reverse(transmitted)

    return transmitted

def MLSE_8PSK(recieved, N, c):
    # Generate the bits, then get their symbols from the constellation map
    L=3
    Bits = []

    for i in range(512):
        Bits.append(format(512+i, 'b'))
    Symbols = []
    i = 0
    Sym = []
    while i < 8**L:
        Sym = []
        for j in range(9):
            if Bits[i][j+1:j+4]  == "111":
                Sym.append(1)
            elif Bits[i][j+1:j+4] == "110":
                Sym.append((1+1j)/sqrt2)
            elif Bits[i][j+1:j+4] == "010":
                Sym.append(1j)
            elif Bits[i][j+1:j+4] == "011":
                Sym.append((-1+1j)/sqrt2)
            elif Bits[i][j+1:j+4] == "001":
                Sym.append(-1)
            elif Bits[i][j+1:j+4] == "000":
                Sym.append((-1-1j)/sqrt2)
            elif Bits[i][j+1:j+4] == "100":
                Sym.append(-1j)
            elif Bits[i][j+1:j+4] == "101":
                Sym.append((1-1j)/sqrt2)
        Symbols.append(Sym)
        i += 3


    # Get all the deltas
    deltas = []
```

```
    for i in range(N):
        deltas.append(findDelta8PSK(recieved, Symbols ,i, c))

    # print(deltas)
    # Using the deltas, work backwards and determine the transmitted
   sequence
    transmitted = []
    for i in range(N):
        cost = min(deltas[N-1-i])
        bit = deltas[N-1-i].index(cost)

        if bit % 8 == 0:
            transmitted.append(1)
        elif bit % 8 == 1:
            transmitted.append((1+1j)/sqrt2)
        elif bit % 8 == 2:
            transmitted.append(1j)
        elif bit % 8 == 3:
            transmitted.append((-1+1j)/sqrt2)
        elif bit % 8 == 4:
            transmitted.append(-1)
        elif bit % 8 == 5:
            transmitted.append((-1-1j)/sqrt2)
        elif bit % 8 == 6:
            transmitted.append(-1j)
        elif bit % 8 == 7:
            transmitted.append((1-1j)/sqrt2)

    transmitted = Reverse(transmitted)
    return transmitted

def Graph():
    size=N
    size2=5
    randomValues= theorwhichman(size)
    bits=bits_gen(randomValues)
    BPSK_bits=BPSK(bits)
    FourQAM_bits=fourQAM(bits)
    EBPSK_bits=eight_PSK(bits)
    channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
    transmitted=[]
    xValues = np.linspace(-4, 15, 38*size2)
    yvalues=[]
    """#bpsk

    L=3
    M=2
    a,transmitted=OptMemGen(1,3)
    transmitted.extend(BPSK_bits)
    k=-4
    while (k<15):

        for i in range(size2):
            sigma_=1/np.sqrt(math.pow(10, (k/ 10)) * 2 * math.log2(M))
            c = [(random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
   .sqrt(6),
                 (random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
```

```
.sqrt(6),
                     (random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
.sqrt(6)]
            Recieved=addnoise(transmitted,channels,L,M,k,sigma_)
#[1.5,1.2,1,-1.2,-1.5,0.2]
            Options,memory= OptMemGen(1,L)#bpsk 1, 4Qam,8psk
            Detected=BPSK_MLSE(Recieved,size,c)
            yvalues.append(bit_errors(transmitted[L-1:],Detected))
        k+= 0.5
    plt.semilogy(xValues,yvalues, label="BPSK")
    plt.ylabel('BER')
    plt.xlabel('SNR')


    yvalues=[]

    #4Qam
    L=3
    M=4
    a,transmitted=OptMemGen(2,3)
    transmitted.extend(FourQAM_bits)
    k=-4
    while (k<15):
        for i in range(size2):
            sigma_=1/np.sqrt(math.pow(10, (k/ 10)) * 2 * math.log2(M))
            c = [(random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
.sqrt(6),
                     (random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
.sqrt(6),
                     (random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
.sqrt(6)]
            Recieved=addnoise(transmitted,channels,L,M,k,sigma_)
#[1.5,1.2,1,-1.2,-1.5,0.2]
            Options,memory= OptMemGen(2,L)#bpsk 1, 4Qam,8psk
            Detected=MLSE_4QAM(Recieved,int(size/2),c)
            yvalues.append(bit_errors(transmitted[L-1:],Detected))
        k+=0.5


    plt.semilogy(xValues,yvalues, label="4QAM")
    plt.ylabel('BER')
    plt.xlabel('SNR')
    yvalues=[]
    """
    L=3
    M=8 #BPSK=2 4Qam=4 8psk=8
    #8psk
    a,transmitted=OptMemGen(3,3)
    transmitted.extend(EBPSK_bits)

    k=-4
    while (k<15):
        for i in range(size2):
            sigma_=1/np.sqrt(math.pow(10, (k/ 10)) * 2 * math.log2(M))
            c = [(random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
.sqrt(6),
                     (random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
```

```
    .sqrt(6),
                    (random.gauss(0,sigma_)+random.gauss(0,sigma_)*1j)/np
    .sqrt(6)]
                Recieved=addnoise(transmitted,channels,L,M,k, sigma_)#
    [1.5,1.2,1,-1.2,-1.5,0.2]
                Options,memory= OptMemGen(3,L)#bpsk 1, 4Qam,8psk
                Detected=MLSE_8PSK(Recieved,int(size/3),c)
                yvalues.append(bit_errors(transmitted[L-1:],Detected))
            k+=0.5

    plt.semilogy(xValues,yvalues, label="8PSK")
    plt.ylabel('BER')
    plt.xlabel('SNR')
    plt.title(" BER vs SNR")
    plt.legend()
# print(BPSK_MLSE(r,N))
#
#print(MLSE_4QAM(r,N))
#
Graph()
```

8.1.4 DFE Dynamic CIR

```
# -*- coding: utf-8 -*-
"""
Created on Sat Oct 10 10:52:03 2020

@author: user
"""


# -*- coding: utf-8 -*-
"""
Created on Fri Oct  2 22:11:23 2020

@author: user
"""

import random
import math
import statistics as st
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
import seaborn as sns
# settings for seaborn plotting style
sns.set(color_codes=True)
# settings for seaborn plot sizes
sns.set(rc={'figure.figsize':(5,5)})
np.random.seed(20)#just the to add a little bit of repeatbilty in the
    randomnes
#np.random.seed()
def theorwhichman(size):
    rand=[]
    for i in range(0,size):
        rand.append(random.uniform(0, 1))
    return rand
size=100
```

```

a=theorwhichman(size)

mu = st.mean(a)
sigma = st.stdev(a)

x = np.linspace(-1, 1, size)
a.sort()



"""ax = sns.distplot(a,
                  bins=100,
                  kde=True,
                  color='skyblue',
                  hist_kws={"linewidth": 15,'alpha':1})
ax.set(xlabel='Normal Distribution', ylabel='Frequency')
#[Text(0,0.5,u'Frequency'), Text(0.5,0,u'Normal Distribution')]
"""
#plt.plot(a, norm(0.5, 1).pdf(a))
#plt.ylabel('Probability Density')
#plt.xlabel('Randomly Generated Numbers')
#plt.show()

print("Sigma:", sigma)
print("Mu:", mu)


#Creating bits to be transmitted
#
    ---------------------------------------------------------------------------
#                               Bit generator
#
    ---------------------------------------------------------------------------
# random bits generator
def bits_gen(values): # function takes a list of values between 0 and
    1
    data = []

    for i in values:
        #if the values is less than 0.5 append a 0
        if i < 0.5:
            data.append(0)
        else:
            #if the value found in the list is greater than 0.5 append
     1
            data.append(1)

    return data

#
    ------------------------------------------------------------------------------
#                mapping of bits to symbol using constellation maps
#
```

```
--------------------------------------------------------------------------------
def BPSK(bits):
    bpsk = []
    for k in bits:
        if k == 1:
            bpsk.append(1)
        else:
            bpsk.append(-1)
    return bpsk

def fourQAM(bits):
    FQAM = []
    M = 2
    subList = [bits[n:n + M] for n in range(0, len(bits), M)]
    for k in subList:
        if k == [0, 0]:
            FQAM.append(complex(1 / np.sqrt(2), 1 / np.sqrt(2)))
        elif k == [0, 1]:
            FQAM.append(complex(-1 / np.sqrt(2), 1 / np.sqrt(2)))
        elif k == [1, 1]:
            FQAM.append(complex(-1 / np.sqrt(2), -1 / np.sqrt(2)))
        # elif(k==[1,0]):
        elif k == [1, 0]:
            FQAM.append(complex(1 / np.sqrt(2), -1 / np.sqrt(2)))

    return FQAM

def eight_PSK(bits):
    EPSK = []
    M = 3
    subList = [bits[n:n + M] for n in range(0, len(bits), M)]
    for k in subList:
        if k == [0, 0, 0]:
            EPSK.append(complex(1 , 0))
        elif k == [0, 0, 1]:
            EPSK.append((1+1j)/np.sqrt(2))
        elif k == [0, 1, 1]:
            EPSK.append( 1j)
        elif k == [0, 1, 0]:
            EPSK.append((-1+1j)/np.sqrt(2))
        elif k == [1, 1, 0]:
            EPSK.append(-1)
        elif k == [1, 1, 1]:
            EPSK.append((-1-1j)/np.sqrt(2))
        elif k == [1, 0, 1]:
            EPSK.append(-1j)
        elif k == [1, 0, 0]:
            EPSK.append((1-1j)/np.sqrt(2))
    return EPSK

#_________________________________________________________________
#                          Sigma calculation
#___________________________________________________________________

def sigma(domain,M):# M is the number of symbols
    sigma=[]
```

```
    for i in domain:
        sigma.append(1/np.sqrt(math.pow(10, (i/ 10)) * 2 * math.log2(M)))
    return sigma


# -----------------------------------------------------------------------------
#                        create noise
# -----------------------------------------------------------------------------

def noise(size,sigma):
    noiseList = np.random.normal(0,sigma,size)
    return noiseList

# -------------------------------------------------------------------------
#                        add noise
# -----------------------------------------------------------------------------

def addnoise(transmitted,channels,L,m,snr):# assuming transmited comes with the memory symbols padded
    recieved=[]
    M=m
    k=snr

    sigma_=1/np.sqrt(math.pow(10, (k/ 10)) * 2 * math.log2(M))
    #print(sigma_)


    for i in range(L-1,len(transmitted)):
        sample=np.random.normal(0,1,1)[0]
        recieved.append(transmitted[i]*channels[0]+transmitted[i-1]*channels[0+1]
                        +transmitted[i-2]*channels[2]+sigma_*(sample+(sample)*1j))
    return recieved #without the padding

# -------------------------------------------------------------------------
#                        DFE function
# -----------------------------------------------------------------------------

def DFE(recieved,channels,L,options,memory):
    Options =options# [1,-1]# these are the option available for bpsk
    symbols = memory#[1]*(L-1) #the first 1 is the memory symbols
    s=0 # symbol mover
    #print(recieved)
```

```
    for i in range(0,len(recieved)):
        guess=[]
        n=len(channels)-1# length of the chanel L-1
        #calculating the product but from second position
        sumof=0
        for j in range(1,n):
           sumof+= symbols[n-1+s]*channels[j]
           n-=1

        for k in Options:
            #guess.append(np.abs(recieved[i]-((k)*channels[0]+symbols[1]*channels[1]+symbols[0]*channels[2]))**2)
            guess.append(np.abs(recieved[i]-((k)*channels[0]+sumof))**2)
        #print(guess[0])
        estimate=Options[guess.index(min(guess))]
        symbols.append(estimate)
        s+=1
    return symbols[L-1:] #final
# -------------------------------------------------------------------------
#                         Options and memory generator
# -----------------------------------------------------------------------------
def OptMemGen(i,L):
    #BPSK=1 #4QAM=2 8PSK=3
    if i==1:
        Options = [1,-1]# these are the option available for bpsk
        memory= [1]*(L-1) #the first 1 is the memory symbols
        return Options,memory
    elif(i==2):
        Options = [(1+1j)/np.sqrt(2), (-1+1j)/np.sqrt(2), (-1-1j)/np.sqrt(2), (1-1j)/np.sqrt(2)]
        memory=[(1+1j)/np.sqrt(2)]*(L-1)
        return Options,memory
    elif(i==3):
         Options=[1, (1+1j)/np.sqrt(2), 1j, (-1+1j)/np.sqrt(2), -1, (-1-1j)/np.sqrt(2), -1j, (1-1j)/np.sqrt(2)]
         memory=[(1+1j)/np.sqrt(2)]*(L-1)
         return Options,memory

# -------------------------------------------------------------------------
#                        Bit Error calculation
# -----------------------------------------------------------------------------

def bit_errors(sent, recieved):
    error = 0
    for k in range(0,len(recieved)):
        if sent[k] != recieved[k]:
            error += 1
    BER = error / len(recieved)*100
```

```

    return BER

#transmitted=[1,1,1,-1,1,-1,1]
#channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
#Recieved=addnoise(transmitted,channels,3)#[1.5,1.2,1,-1.2,-1.5,0.2]
#print(Recieved)

def yvalueCal(transmitted):
    channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
    L=3
    M=2 #for bpsk=2
    #SNR=15
    yvalues=[]
    #print(transmitted)

    for k in range (-4,16):


        Recieved=addnoise(transmitted,channels,L,M,k)#
    [1.5,1.2,1,-1.2,-1.5,0.2]
        #print(Recieved)
        Options,memory= OptMemGen(1,L)
        Detected=DFE(Recieved,channels,L,Options,memory)
        yvalues.append(bit_errors(transmitted[L-1:],Detected))
    return yvalues
#print(yvalueCal(transmitted))

def newchannel(v1,v2,v3):
    c=[]
    b=[v1+v2*1j,(v2+v3*1j),(v3+v1*1j)]/np.sqrt(2.3)
    c.extend(b)
    return c

def grapghs():
    size=1000000
    randomValues= theorwhichman(size)
    bits=bits_gen(randomValues)
    BPSK_bits=BPSK(bits)
    FourQAM_bits=fourQAM(bits)
    EBPSK_bits=eight_PSK(bits)
    channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]

    transmitted=[]
    xValues = np.linspace(-4, 15, 38)
    yvalues=[]
    #bpsk

    L=3
    M=2
    blocks=[BPSK_bits[n:n + 200] for n in range(0, len(BPSK_bits),
   200)]
    a,transmitted=OptMemGen(1,3)
    #transmitted.extend(BPSK_bits)
    a,transmitter=OptMemGen(1,3)
    transmitter.extend(BPSK_bits)

```

```
    k=-4
    while (k<15):
        Recieved=[]
        Detected=[]

        for block in blocks:
            v1=np.random.normal(0,1,1)[0]
            v2= np.random.normal(0,1,1)[0]
            v3 =np.random.normal(0,1,1)[0]
            channels= newchannel(v1,v2,v3)
            a,transmitted=OptMemGen(1,3)
            transmitted.extend(block)

            Recieved=(addnoise(transmitted,channels,L,M,k))
            Options,memory= OptMemGen(1,L)#bpsk 1, 4Qam,8psk
            Detected.extend(DFE(Recieved,channels,L,Options,memory))

        yvalues.append(bit_errors(transmitter[L-1:],Detected))
        k+=0.5
    plt.semilogy(xValues,yvalues, label="BPSK")
    plt.ylabel('BER')
    plt.xlabel('SNR')


    yvalues=[]
    #4Qam
    L=3
    M=4
    blocks=[FourQAM_bits[n:n + 200] for n in range(0, len(FourQAM_bits), 200)]
    a,transmitter=OptMemGen(2,3)
    transmitter.extend(FourQAM_bits)

    k=-4
    while (k<15):
        Recieved=[]
        Detected=[]

        for block in blocks:
            v1=np.random.normal(0,1,1)[0]
            v2= np.random.normal(0,1,1)[0]
            v3 =np.random.normal(0,1,1)[0]
            channels= newchannel(v1,v2,v3)
            a,transmitted=OptMemGen(2,3)
            transmitted.extend(block)

            Recieved=(addnoise(transmitted,channels,L,M,k))
            Options,memory= OptMemGen(2,L)#bpsk 1, 4Qam,8psk
            Detected.extend(DFE(Recieved,channels,L,Options,memory))

        yvalues.append(bit_errors(transmitter[L-1:],Detected))
        k+=0.5
    plt.semilogy(xValues,yvalues, label="4QAM")
    plt.ylabel('BER')
    plt.xlabel('SNR')

    yvalues=[]
```

```
    L=3
    M=8 #BPSK=2 4Qam=4 8psk=8
    #8psk
    blocks=[EBPSK_bits[n:n + 200] for n in range(0, len(EBPSK_bits), 200)]
    a,transmitter=OptMemGen(3,3)
    transmitter.extend(EBPSK_bits)

    k=-4
    while (k<15):
        Recieved=[]
        Detected=[]

        for block in blocks:
            v1=np.random.normal(0,1,1)[0]
            v2= np.random.normal(0,1,1)[0]
            v3 =np.random.normal(0,1,1)[0]
            channels= newchannel(v1,v2,v3)
            a,transmitted=OptMemGen(3,3)
            transmitted.extend(block)

            Recieved=(addnoise(transmitted,channels,L,M,k))
            Options,memory= OptMemGen(3,L)#bpsk 1, 4Qam,8psk
            Detected.extend(DFE(Recieved,channels,L,Options,memory))

        yvalues.append(bit_errors(transmitter[L-1:],Detected))
        k+=0.5
    plt.semilogy(xValues,yvalues, label="8PSK")
    plt.ylabel('BER')
    plt.xlabel('SNR')
    plt.title(" BER vs SNR")
    plt.legend()

grapghs()



def tester():
    size=10000
    #np.random.seed(420)
    print("Generating the random number generator of size 200")
    randomValues= theorwhichman(size)


    print("\nExpected sigma =0.29 and expected mu=0.50")
    print("Sigma",st.mean(randomValues))
    print("mu",st.stdev(randomValues))

    print("\nTesting the bits_gen function of size 200")
    bits=bits_gen(randomValues)
    print("Size:",len(bits))

    print("\nNow testing the mapping of sysmbols for different modulation schemes")
    print("BPSK expected length 200")
    BPSK_bits=BPSK(bits)
```

```
    print("BPSK length:",len(BPSK_bits))
    print("4QAM expected length =100")
    #FourQAM_bits=fourQAM(bits)
    #print("4QAM length:",len(FourQAM_bits))
    print("8BPSK expected length 66")
    #EBPSK_bits=eight_PSK(bits)
    #print("8BPSK length:",len(EBPSK_bits))
    #SNR = np.linspace(0, 15, 16)
    #print(SNR)
    #print("\nGenerating noise for 8psk")
    #noiseList=noise(len(EBPSK_bits),sigma(SNR,8)[0])
    #print(noiseList)
    #print("Length of noise:",len(noiseList))
    #print("variance:" ,st.variance(noiseList))

    print("\nTesting the DFE function")
    channels = [0.89+0.92j,0.42-0.37j,0.19+0.12j]
    #bpsk
    transmitted=[1,1]
    transmitted.extend(BPSK_bits)
    #QPSK

    L=3
    M=2 #for bpsk=2
    SNR=-1

    print("Adding Noise")
    #Recieved=addnoise(transmitted,channels,L,M,SNR)
   #[1.5,1.2,1,-1.2,-1.5,0.2]
    #Options,memory= OptMemGen(1,L)
    #print("Received symbols:",Recieved)
    #Detected=DFE(Recieved,channels,L,Options,memory)
    #print("Detected Symbols",Detected)
    xValues = np.linspace(-4, 15, 20)
    yvalues=yvalueCal(transmitted)
    print(yvalues)
    plt.semilogy(xValues,yvalues)



#tester()
```