Cairo University - Faculty of Engineering
Computer Engineering Department

M-ARY AMPLITIUDE SHIFT MODULATION

Subject: Digital Communication

Submitted to:

Dr. Hala Abdel Kader

Dr. Mai Badawi

Presented by:

Evram Youssef : Sec: 1 BN: 9

Year
2019/2020

Contents

List of Figures

0.1 Part 1: Digital Communication

0.1.1 Problem 1

Figure 1 below showing the comparison between simulated BER and theoritical (analytical) BER VS the Eb/N0 in db.

Please notice, you'll have to input the no. of bits you wish to be transmitted, and it has to be divisible by 3.

0.1.2 Problem 2

The constellation of the 8-ary with decision region pf each symbol.

Boundaries are at:

$$-6\sqrt{E}, -4\sqrt{E}, -2\sqrt{E}, 0, 2\sqrt{E}, 4\sqrt{E}, 6\sqrt{E}$$

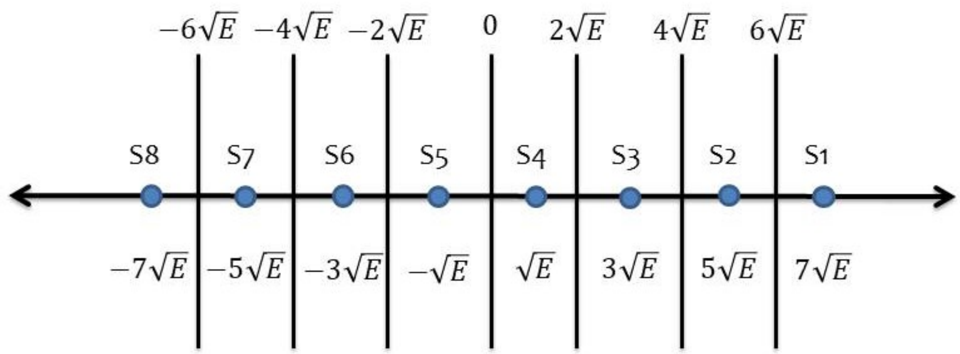

Figure 1: Symbols Boundary

0.1.3 Problem 3

The derivation of theoritical bit error rate using 8-ary.

$$Pe = \frac{1}{8}\sum_{i=0}^{7} P(e|Si) \tag{1}$$

$$Pe(e|S0) = Pe(e|S7) \tag{2}$$

$$Pe(e|S1) = Pe(e|S2) = Pe(e|S3) = Pe(e|S4) = Pe(e|S5) = Pe(e|S6) \tag{3}$$

Using Union bound S0, S7 only one neighbour and S1, S2,...S6 has two neighbours.

$$Pe(e|S0) = \frac{1}{2}erfc(\frac{\sqrt{E}}{\sqrt{N}}) \tag{4}$$

$$Pe(e|S1) = \frac{1}{2}erfc(\frac{\sqrt{E}}{\sqrt{N}}) + \frac{1}{2}erfc(\frac{\sqrt{E}}{\sqrt{N}}) \tag{5}$$

$$Pe(e|S1) = erfc(\frac{\sqrt{E}}{\sqrt{N}}) \tag{6}$$

$$Pe = \frac{1}{8*3}(2*\frac{1}{2}erfc(\frac{\sqrt{E}}{\sqrt{N}}) + 6*erfc(\frac{\sqrt{E}}{\sqrt{N}})) \tag{7}$$

$$Pe = \frac{7}{24}(erfc(\frac{\sqrt{E}}{\sqrt{N}})) \tag{8}$$

0.1.4 Probelm 4

Figure 1 below showing the comparison between simulated BER and theoritical (analytical) BER VS the Eb/N0 in db.

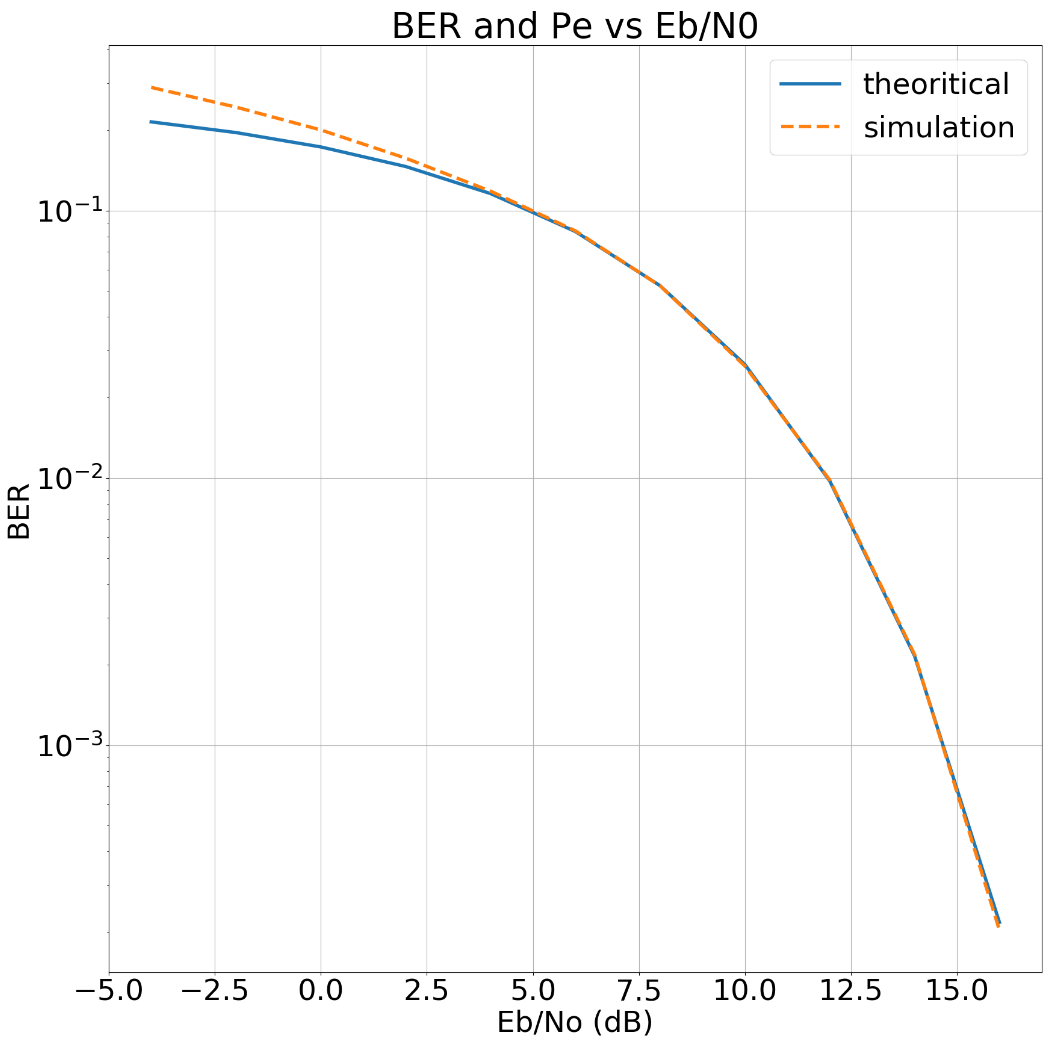

Figure 2: BER vs Eb/N0

0.1.5 Probelm 5

The answer is NO, We can't transmit at Rate 1 Mbps with bandwidth 0.5 MHz in passband transmition.

The minimum M required: 16 Only bit by bit transmittion is allowed.

GIVEN:

$$Bt = 2Rs$$

$$Rb = 1Mbps$$

$$BW = 0.5MHz$$

$$M = 3$$

REQUIRED:

$$BW =?2 * Rs$$

$$BW =?2 * \frac{Rb}{\log_2 M}$$

$$BW =?1Mbps * \frac{2}{3}$$

The Answer is: NO it can not be transmitted

$$0.5MHz =?1Mbps * \frac{2}{\log_2 M}$$

$$4MHz =? \log_2 M$$

and Minimum M alowed is:

$$M = 16$$

0.1.6 Probelm 6

Both of them satisfy the Gray Encoding criterion.

Because at the two examples only one bit is changed in each transition from symbol to the next one.

0.2 Part 2: Information Theory

Cyclic Codes.

0.2.1 Definition of Cyclic Codes

A Cyclic Codes is a block codes that follow the property of linearity where the circular i shift to the left (or n-1 shift to the right) always result in a word that belongs to the code words.

They are defined as $C(x) = (n, k)$ it means the message has k bits and the code vector is n bits.

$C(x)$ are defined with the help of generator polynomial $g(x)$.

The degree of $g(x)$ is equal to the number of parity-check digits of the code.

There are total of 2^k code polynomials in $C(x)$.

They are error-correction codes, used earlier to transmit images of planets.

They have algebric properties that help detecting and correcting errors.

Cyclic codes can be used to correct errors, it can be generalized to correct burst of errors, not just one bit.

For example the set of [000, 1111, 0110, 1001]:

They follow the Linearity property, but not the Cyclic Shift property, therefore they can't be considered a valide Cyclic Codes.

Another Example the set of [0000, 1111, 0101, 1010]:

They follow both the linearity and cyclic shift property. Because, when adding any two of them it will result in another (third) codeword that lies in the finite list.

0.2.2 Systematic CodeWords:

In Systematic Codewords $C(x) = [message, parity]$ this means, each generated codeword's first k bits are the message that got us that codeword while the remaining $n - k$ bits are the parity check of the codeword.

Systematic Codewords follow the following conditions:

$$C(x) = x^{n-k}m(x) + p(x)$$

Where:

$$p(x) = Rem\left[\frac{x^{n-k}m(x)}{g(x)}\right]$$

and,
$C(x)$ is codeword polynomial
$m(x)$ is message polynomial
$g(x)$ is generator polynomial

Example: Constructing a Systematic Cyclic Codes (7,4), using generator polynomial $g(x) = x^3 + x^2 + 1$, with message [1010].

Answer:

$m(x) = x^3 + x$
$p(x) = Rem\left[\frac{x^3 * (x^3+x)}{(x^3+x^2+1)}\right]$
From division:
$p(x) = 1$
therefore,
$C(x) = x^3 * (x^3 + x) + 1$
$C(x) = x^6 + x^4 + 1$
$C(x) = [1010001]$

You may notice that the message $m(x) = 1010$ and the first k bits in the generated codeword is also 1010

Will the rest of the codeword is the parity check.

0.2.3 Relation between generator polynomial and generator matrix:

At the last section we've introduced the difinition of generator polynomial such that $g(x) = x^3 + x^2 + 1$, this polynomial is unique and we can derive all the other codewords from it by multiplying with the various messages allowed.

In cyclic code C, the generator matrix's dimentsions is $[n, k]$ n number of columns and k number of rows.

Generator Matric $[G]$ is composed of $[I, P]$, I is the identity matrix with dimentsions of $[k, k]$ while P is the parity matrix with the dimentsions of $[k, (n-k)]$

So, the main core of our problem is identifying the parity matrix, and it is identified as follows:

$$k^{th} Row = Rem\left[\frac{x^{n-k}}{g(x)}\right]$$

continuing on the previous example the generator matrix will be something like this:

$$G(x) = \begin{bmatrix} 1 & 0 & 0 & 0 & - & - & - \\ 0 & 1 & 0 & 0 & - & - & - \\ 0 & 0 & 1 & 0 & - & - & - \\ 0 & 0 & 0 & 1 & - & - & - \end{bmatrix}$$

so, to calculate the parity matrix we'll solve the following equations:

$$1^{st} Row = Rem\left[\frac{x^6}{x^3 + x^2 + 1}\right]$$
$$1^{st} Row = x^2 + 1 = [101]$$

$$2^{nd} Row = Rem\left[\frac{x^5}{x^3 + x^2 + 1}\right]$$
$$2^{st} Row = x^2 + x + 1 = [111]$$

$$3^{rd} Row = Rem\left[\frac{x^4}{x^3 + x^2 + 1}\right]$$

$$3^{rd}Row = x^2 + 1 = [101]$$

$$4^{th}Row = Rem\left[\frac{x^3}{x^3 + x^2 + 1}\right]$$

$$4^{th}Row = x + 1 = [011]$$

therefore the final matrix will be:

$$G(x) = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

0.3 Appendix: Main Code for part 1

Listing 1: Main Code for part 1

```
import math
import random

import matplotlib.pyplot as plt
import numpy as np
from scipy.special import erfc


def generateRandomProcess(elementCount):
    '''
    This function returns random process of streaming bits 0 an 1.

    eg. [1, 0, 0, 1,....]
    and we know for sure that the main function is checking that the elementCount
    is divisible by 3
    '''
    XofT = []
    for _ in range(elementCount):
        XofT.append(math.floor(0.5 + random.uniform(0, 1)))
    return XofT


def generateTimeSteps(elementCount, width):
    '''
    This function returns time steps
    '''
    return np.linspace(0, elementCount, elementCount, endpoint=False)


def mapper():
    '''
        Function:
            Here you can find the Mapper Code and Logic

            Since M = 3 and Eb = 1, then E0 = 1/7

        Logic:
            Simply reading an input from the user count of bits
            Making sure its divisible by three
            Generating random bits of that size
            Returns M-ary of size bits.size/3

        Returns:
            Bits: the randomly generated bits
            Mapped: the mapped M-ary values
    '''
    E0 = 1/7
    elemCount = 1
    while elemCount%3 != 0:
        elemCount = int(input("Please enter the desired count of elements, and make sure

    Bits = generateRandomProcess (elemCount)

    Symbols = [-7,-5,-3,-1,1,3,5,7]
    Symbols = [i * math.sqrt(E0) for i in Symbols]

    Bits_to_Symbols = []
    for i in range (0,elemCount, 3):
        stack_of_elements = ""
```

```
        stack_of_elements += str(Bits[i])
        stack_of_elements += str(Bits[i+1])
        stack_of_elements += str(Bits[i+2])

        if stack_of_elements == "000":
            Bits_to_Symbols.append([Symbols[4]])
        elif stack_of_elements == "001":
            Bits_to_Symbols.append([Symbols[5]])
        elif stack_of_elements == "010":
            Bits_to_Symbols.append([Symbols[7]])
        elif stack_of_elements == "011":
            Bits_to_Symbols.append([Symbols[6]])
        elif stack_of_elements == "100":
            Bits_to_Symbols.append([Symbols[3]])
        elif stack_of_elements == "101":
            Bits_to_Symbols.append([Symbols[2]])
        elif stack_of_elements == "110":
            Bits_to_Symbols.append([Symbols[0]])
        elif stack_of_elements == "111":
            Bits_to_Symbols.append([Symbols[1]])
        else:
            print ("error occured in mapping")
    return Bits, Bits_to_Symbols, elemCount


def Channel (mean, variance, length):
    '''
    Function:
        This Function is responsible for simulating the AWGN channel effect of adding ra
    transmitted signal.

    Inputs:
        mean: the mean of awgn channel
        variance: the varinace of awgn channel
        length: Length of the time steps
    Outputs:
        generated random noise
    '''
    return math.sqrt(variance/2)*np.random.randn(length)
    # return np.random.normal(mean, np.sqrt(variance), length)


def DeMapper(Noisy_Bits_to_Symbols):
    '''
    Function:
        Demap/Decode the symbols into their actual bits

    Inputs:
        Noisy_Bits_to_Symbols: Symbols representing the bits added to noise

    Output:
        Received_Bits: The mapped bits
    '''
    Received_Bits = []
    E0 = 1/7
    Symbols_boundry = [-6,-4,-2,0,2,4,6]
    Symbols_boundry = [i * math.sqrt(E0) for i in Symbols_boundry]

    for symbole in Noisy_Bits_to_Symbols:
        if symbole <= Symbols_boundry[0]:
            Received_Bits.append(1)
            Received_Bits.append(1)
            Received_Bits.append(0)
```

```python
        elif symbole > Symbols_boundry[0] and symbole <= Symbols_boundry[1]:
            Received_Bits.append(1)
            Received_Bits.append(1)
            Received_Bits.append(1)
        elif symbole > Symbols_boundry[1] and symbole <= Symbols_boundry[2]:
            Received_Bits.append(1)
            Received_Bits.append(0)
            Received_Bits.append(1)
        elif symbole > Symbols_boundry[2] and symbole <= Symbols_boundry[3]:
            Received_Bits.append(1)
            Received_Bits.append(0)
            Received_Bits.append(0)
        elif symbole > Symbols_boundry[3] and symbole <= Symbols_boundry[4]:
            Received_Bits.append(0)
            Received_Bits.append(0)
            Received_Bits.append(0)
        elif symbole > Symbols_boundry[4] and symbole <= Symbols_boundry[5]:
            Received_Bits.append(0)
            Received_Bits.append(0)
            Received_Bits.append(1)
        elif symbole > Symbols_boundry[5] and symbole <= Symbols_boundry[6]:
            Received_Bits.append(0)
            Received_Bits.append(1)
            Received_Bits.append(1)
        elif symbole > Symbols_boundry[6]:
            Received_Bits.append(0)
            Received_Bits.append(1)
            Received_Bits.append(0)
        else:
            print("error occured in Demapping")

    return Received_Bits

def BER (Bits, Received_Bits):
    '''
    Function:
        Calculate the BER for each bit sent

    Inputs:
        Bits: The actual transmitted bits
        Received_Bits: The received bits

    Output:
        actual_ber: the sum of all errors occurred
    '''
    error = 0
    #Converting them to ndarray
    for i in range (len(Received_Bits)):
        if (Bits[i] != Received_Bits[i]):
            error += 1

    return error/len(Received_Bits)


if __name__ == "__main__":
    ###VARIABLES###
    Eb_No_dB_Min = -4   # min E/No alowed in db
    Eb_No_dB_Max = 16   # max E/No alowed in db
    '''
        Eb_No_dB: Array of values from Eb_No_dB_Min to Eb_No_dB_Max
        with step_size = 2

        To plot(simulate) the vertical access for BER and Theoritical BER (Pe)
```

```
'''
Eb_No_dB = np.arange(start=Eb_No_dB_Min, stop=Eb_No_dB_Max+1, step=2)

# Linearize Eb/N0
'''
    Liniarizing Eb/No in order to get the Variance of AWGN at this point
'''
Eb_No = 10**(Eb_No_dB/10.0)

#Theoritcal and Actual Errors:
#Bit error rates of different Eb/N0
BERs = []
#Probability of errors of different Eb/N0
PEs = []
#1- Mapper is always the same as E0 = 1/7 all the time
Bits, Bits_to_Symbols, length = mapper()

mean = 0
for E_N0 in Eb_No:
    #2- Channel
    # variance = math.sqrt((1/E_N0)/2)
    variance = (1/E_N0)
    #length//3 ==> floor(length/3)
    Noise = Channel(mean, variance, length//3)
    #Adding the noise to Symbolic Bits ELEMENT WISE
    Noisy_Bits_to_Symbols = []
    for i in range (len(Bits_to_Symbols)):
        Noisy_Bits_to_Symbols.append(Bits_to_Symbols[i] + Noise[i])

    #3- Demapping
    Received_Bits = DeMapper(Noisy_Bits_to_Symbols)

    #4- BER
    actual_ber = BER(Bits, Received_Bits)
    BERs.append(actual_ber)
    PEs.append( (7/8)* (erfc(math.sqrt(E_N0/7))) *(1/3))

#Plotting1/3)
plt.rcParams["figure.figsize"] = (20,20)
plt.rcParams.update({'font.size': 35})

plt.semilogy(Eb_No_dB, PEs,linestyle = 'solid',linewidth=4)
plt.semilogy(Eb_No_dB, BERs,linestyle = 'dashed',linewidth=4)
plt.grid(True)
plt.legend(('theoritical','simulation'))
plt.xlabel('Eb/No (dB)')
plt.ylabel('BER')
plt.title("BER and Pe vs Eb/N0")
# plt.show()
plt.savefig("Figures/Figure_1")
print ("figure is saved at Figures/Figure_1.png")
```