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

EDC 310

Digital Communication

Assignment 4: Group Design
Group G

Name and Surname	Student Number	Signature	% Contribution
Kwaku Bediako	u04465483		20
Lefa Raleting	u14222460		80

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 3, 2020

Contents

1 Introduction

Additive White Gaussian Noise (AWGN) is often used to imitate the random and uncontrollable nature of the world. Phase-Shift Keying (PSK) can be considered as a unique case of Quadrature Amplitude Modulation (QAM), a method that finds frequent use within modern telecommunication systems to transmit information
A simulation platform for BPSK, 4QAM, 8PSK and 16QAM communication systems was developed. Data was transferred through these the systems with AWGN added. The reception part of the communication systems then attempted to demodulate the transmitted data and the received data was compared against the sent data to study the effects of noise on each communication system.

Firstly, two number generators were created. The first was a uniform number generator, and the second one was a Gaussian Number Generator. These were designed using the algorithms published by Wichmann-Hill [1] and Marsaglia-Bray [2]. The uniform number generator was used to generate random bits. Any value less than or equal to 0.5 was considered a 0 bit, while all values greater than 0.5 were considered a 1.

Marsaglia-Bray's algorithm, more also known as the Marsaglia polar method, generates a pair of random numbers along a normal distributive curve about the number. Basic random functions would take a range of numbers and the probability of picking a number randomly is equal. If someone were to pick at random a value within the range 1,10 the odds of picking any one of them would be equal to that of any other number. 10

A gaussian function would make it such that the value would average to a mean, , with a specified standard deviation, .

The bits were then mapped to their respective modulation constellation maps and noise was added to the symbols during their transmission. The transmitted symbols were demodulated, and we compared the detected symbols to the transmitted symbols and tallied the errors. The symbols were then converted back to bits and the Bit Error Rate (BER) was calculated by:

We expected that as the number of bits being mapped increased, the error rate within the transmission would increase. We also expected to see the QAM methods having a higher error rate than the PSK systems.

2 Theoretical background

2.1 Wichmann-Hill

An ideal random number generator should be repeatable and should have good time complexity. The random number generator needs to be portal, be able to run on a 16-bit machine. To do so a prime number less than 32768 and a multiplier of around the square root of the prime (Wichmann. B Hill D,1982).

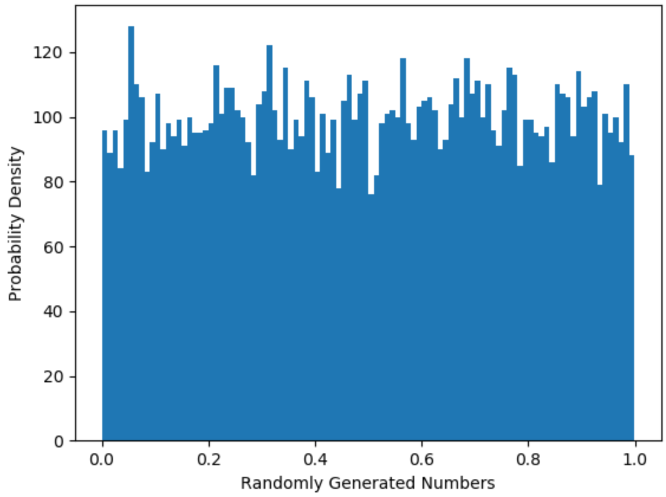

Figure 1: Uniform number Generator with size = 10 000

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

@author: user
"""

import random
import statistics as st
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

def theorwhichman(size):
    rand=[]
    for i in range(0,size):
        rand.append(random.uniform(0, 1))
    return rand
size=10000

a=theorwhichman(size)

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

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

plt.hist(a,100)
plt.ylabel('Probability Density')
plt.xlabel('Randomly Generated Numbers')
plt.show()

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

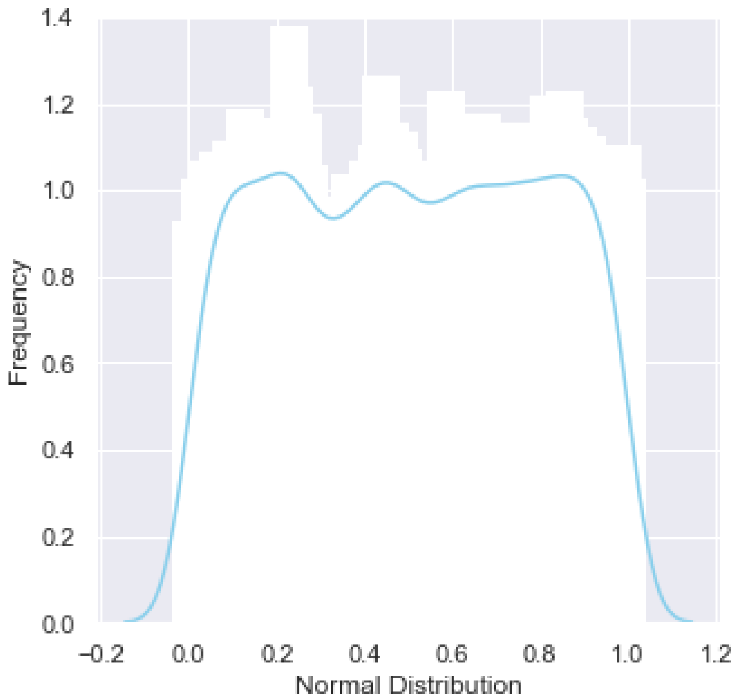

Figure 2: normal distribution with size = 10 000

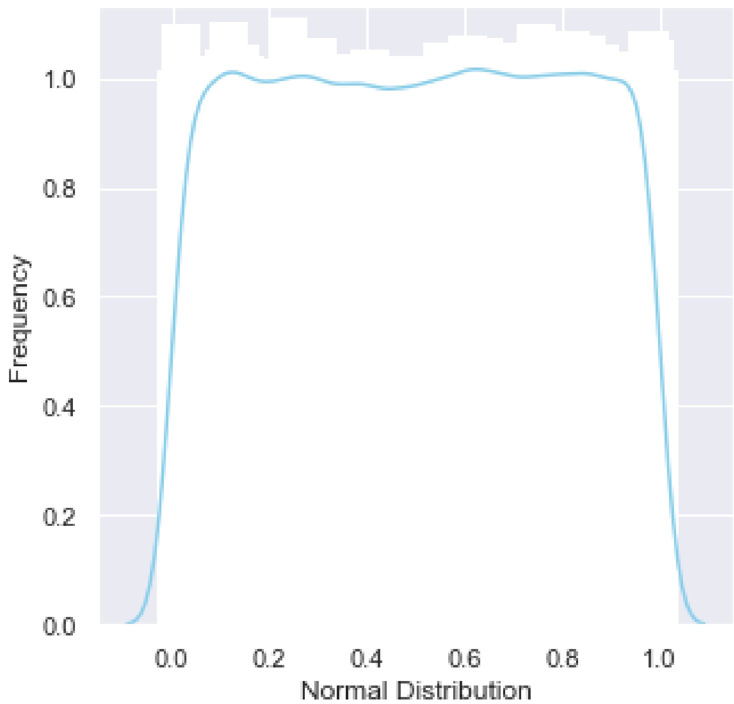

Figure 3: normal distribution with size = 10 0000

The ideal Mean is 0.5 and standard deviation of 0.3.

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

@author: user
"""

import random
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)})

def theorwhichman(size):
    rand=[]
    for i in range(0,size):
        rand.append(random.uniform(0, 1))
    return rand
size=100000

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)
```

Code for theoretical simulations

2.2 Gaussian generator

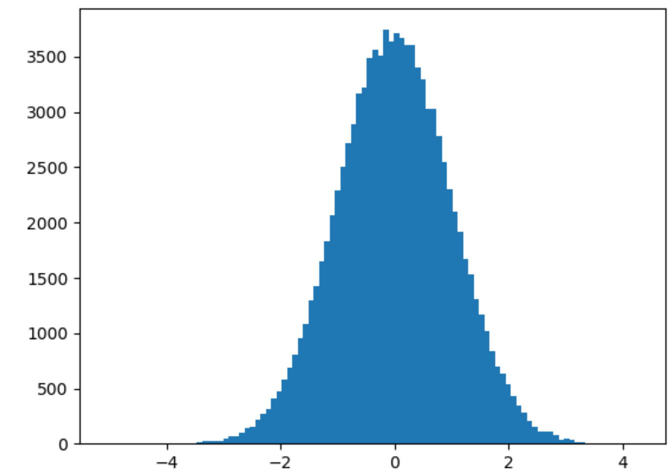

Figure 4: Python standard Gaussian generator after 100 000 iterations

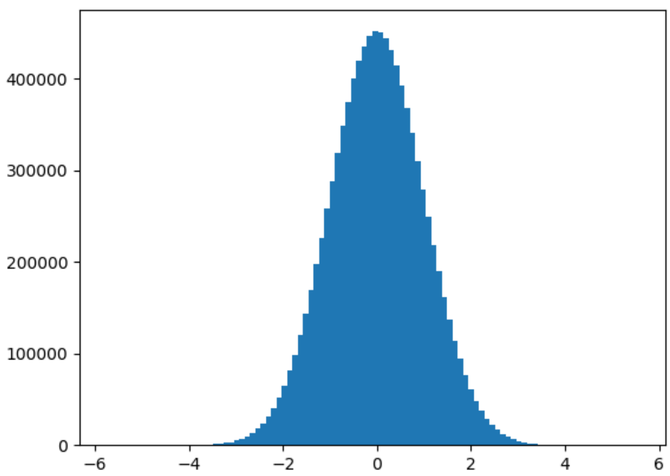

Figure 5: Python standard Gaussian Gaussian generator after 10 000 000 iterations

The ideal Mean is 0, standard deviation is 1, and variance is 1.

3 Design/Method

Two number generators were created. The first was a uniform number generator, and the second one was a Gaussian Number Generator. These were designed using the algorithms published by Wichmann-Hill and Marsaglia-Bray (The code python code can be found in Appendix A, Task 1 and Task 2 respectively). The uniform number generator was used to generate random bits. Any value less than or equal to 0.5 was considered a 0 bit, while all values greater than 0.5 were considered a 1. The Uniform Number

The bits were mapped to their respective modulation constellation maps, after which noise was added to the Symbols using the Gaussian random number generator through the formula:

$$r_k = s_k + \sigma n_k \tag{1}$$

where n_k is the k_th complex zero mean, unity variance, Gaussian random variable. The Signal to Noise Ratio (SNR) was calculated by:

$$SNR = \frac{E_B}{N_o} \tag{2}$$

The received symbols were demodulated by calculating the Euclidean distance between the received symbol and all possible symbols on the respective constellation map. The symbol with the smallest Euclidean distance taken as the demodulated bits. Next, we compared the detected symbols to the transmitted symbols and tallied the errors. SER was calculated by:

$$SER = \frac{no.of symbolerrors}{no.oftransmittedsymbols} \tag{3}$$

The symbols were then converted back to bits and the BER was calculated by:

$$BER = \frac{no.of biterrors}{no.oftransmittedbits} \tag{4}$$

4 Simulations and Results

4.1 Task 1

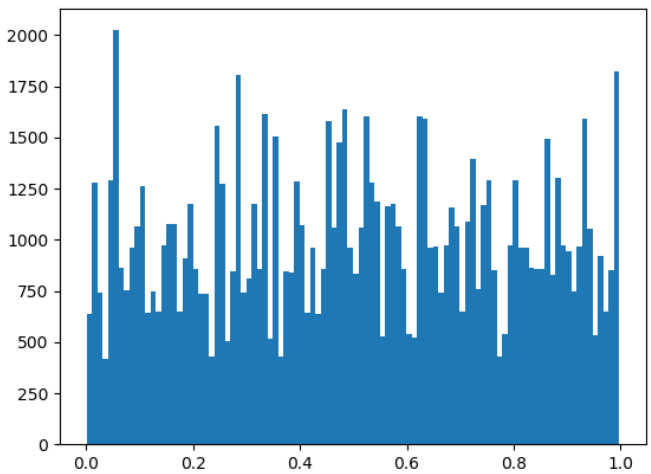

Figure 6: Uniform number Generator with size = 10 000

The mean found here was 0.497 and the standard deviation was 0.296. This is close to the expected results as obtained in the theoretical background.

4.2 Task 2

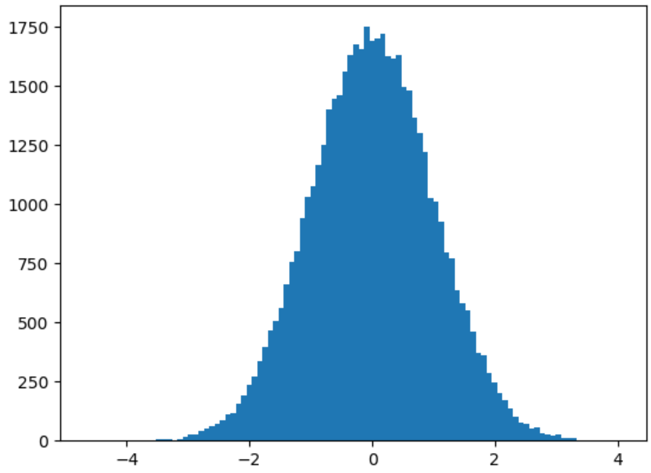

Figure 7: Gaussian generator after 100 000 iterations

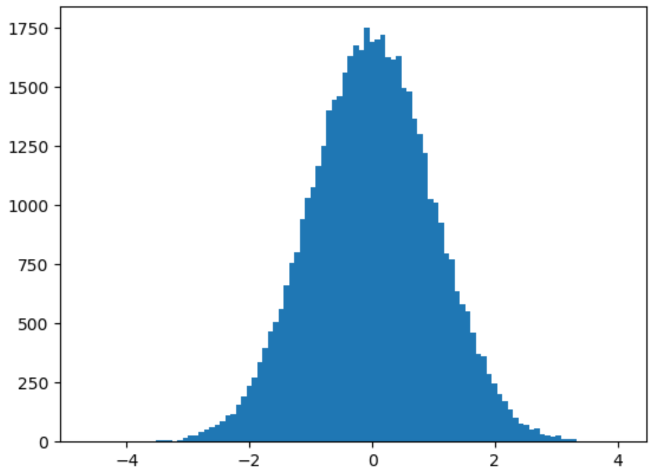

Figure 8: Gaussian generator after 10 000 000 iterations

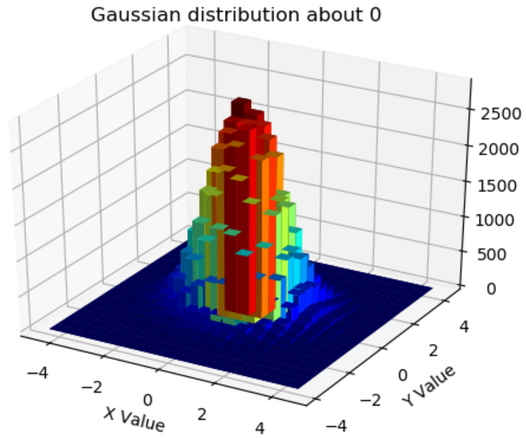

Figure 9: Gaussian generator mapped to 3D space

From Figures 4 and 5, we see that increasing the iterations improves the "shape" of the data. That is as we increase the number of iterations, the data better falls into a normally distributed curve.

The Average tends to 0 as the number iterations is increased and the standard deviation approaches 1. Although the standard python Gaussian's distributor was more accurate than the Marsaglia-Bray algorithm, it took considerably longer to compile.

4.3 Task 3

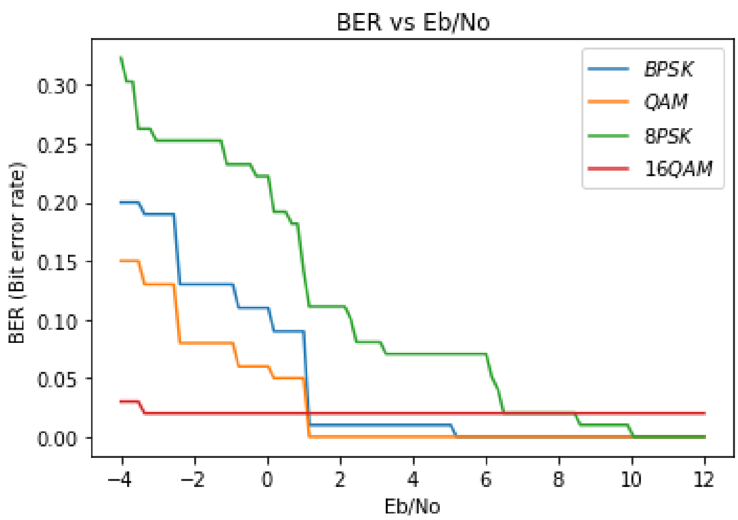

Figure 10: Signal Error Rate as a function of Signal to Noise Ratio(SNR)

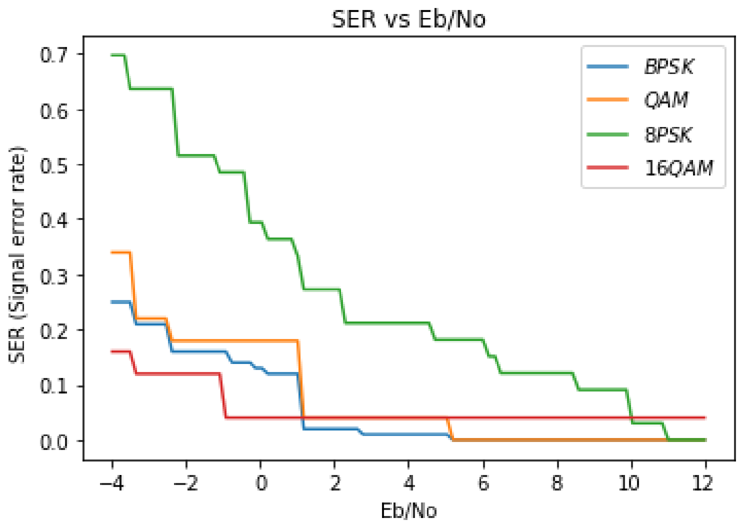

Figure 11: Bit Error Rate as a function of SNR

5 Conclusion

We saw that the effects of noise became more prominent as the number of bits being mapped increased. From this we can deduce the advantages of using low QAM, but we also see how effective higher orders of QAM become in transmitting speed, as the lower orders take significantly more time to transfer and decode. In an environment in which noise can be minimized reliably, it would be recommended to use higher orders of QAM and PSK, but in areas in which the noise cannot be reliably mitigated the use of lower order transfer systems is recommended.

6 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.

7 Appendix A

7.1 Task 1

```
# -*- coding: utf-8 -*-
"""
Created on Thu Aug 27 17:09:43 2020

@author: Boris
"""
import random
import math
```

```python
from datetime import datetime
import statistics as st
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from matplotlib import cm
import mpl_toolkits.mplot3d
from operator import add
import copy


def Whichman():
    # creating the random values based on the above seed
    x = random.randint(1, 30000)
    y = random.randint(1, 30000)
    z = random.randint(1, 30000)

    # first generation
    x = 171 * (x % 177) - 2 * (x / 177)

    if x < 0:
        x = x + 30629
        # first generation
    y = 172 * (y % 176) - 2 * (y / 176)
    if y < 0:
        y = y + 30307
    # first Generation
    z = 170 * (z % 178) - 2 * (z / 178)
    if z < 0:
        z = z + 30323
    temp = x / 30269 + y / 30307 + z / 30323

    return temp - math.trunc(temp)


def Whichman_Random_Generator(size):
    values = []
    # creating a random seed based on the current date and time now
    random.seed(datetime.now())
    for i in range(0, size):
        values.append(Whichman())
    return values


# plotting
# size of the random values
size = 98
randomValues = Whichman_Random_Generator(size)
laterValues = copy.deepcopy(randomValues)
mu = st.mean(randomValues)
sigma = st.stdev(randomValues)

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

plt.plot(randomValues, norm(mu, sigma).pdf(randomValues))
plt.ylabel('Probability Density')
```

```python
plt.xlabel('Randomly Generated Numbers')
plt.show()

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

7.2 Task 2

```python

def Gaussian(Seed):
    length = Seed
    loop = 0

    GaussX = []
    GaussY = []


    while loop < length:
        v = [random.random(), random.random()]

        v[0] = 2*v[0]-1
        v[1] = 2*v[1]-1
        while (v[0]**2 + v[1]**2 > 1) or (v[0]**2+ v[1]**2 == 0):
            v[0] = random.random()
            v[1] = random.random()

            v[0] = 2*v[0]-1
            v[1] = 2*v[1]-1
        X = v[0]*(-2*np.log(v[0]**2+v[1]**2)/(v[0]**2+v[1]**2))**0.5
        Y = v[1]*(-2*np.log(v[0]**2+v[1]**2)/(v[0]**2+v[1]**2))**0.5
        GaussX.append(X)
        GaussY.append(Y)
        loop = loop + 1

    return GaussX, GaussY

#plotting
def Plot2D(X = [],Y=[],density=100):
    A = X+Y
    plt.hist(A, density)


# 3D map plotting
# source: ArtifexR, https://stackoverflow.com/questions/8437788/how-to
    -correctly-generate-a-3d-histogram-using-numpy-or-matplotlib-built-
    in-func
def Plot3D(GaussX = [], GaussY = []):
    xAmplitudes = GaussX
    yAmplitudes = GaussY

    x = np.array(xAmplitudes)   #turn x,y data into numpy arrays
    y = np.array(yAmplitudes)

    fig = plt.figure()          #create a canvas, tell matplotlib it's
     3d
    ax = fig.add_subplot(111, projection='3d')

```

```
    #make histogram stuff - set bins - I choose 20x20 because I have a
     lot of data
    hist, xedges, yedges = np.histogram2d(x, y, bins=(20,20))
    xpos, ypos = np.meshgrid(xedges[:-1]+xedges[1:], yedges[:-1]+
   yedges[1:])

    xpos = xpos.flatten()/2.
    ypos = ypos.flatten()/2.
    zpos = np.zeros_like (xpos)

    dx = xedges [1] - xedges [0]
    dy = yedges [1] - yedges [0]
    dz = hist.flatten()

    cmap = cm.get_cmap('jet') # Get desired colormap - you can change
   this!
    max_height = np.max(dz)   # get range of colorbars so we can
   normalize
    min_height = np.min(dz)
    # scale each z to [0,1], and get their rgb values
    rgba = [cmap((k-min_height)/max_height) for k in dz]

    ax.bar3d(xpos, ypos, zpos, dx, dy, dz, color=rgba, zsort='average'
   )
    plt.title("Gaussian distribution about 0")
    plt.xlabel("X Value")
    plt.ylabel("Y Value")
    plt.savefig("Gaussian distribution about 0")
    plt.show()
```

7.3 Task 3

```

#
    ---------------------------------------------------------------------------
#                                Bit generator
#
    ---------------------------------------------------------------------------
# random bits generator
def bits_gen(values):
    data = []
    # values= Whichman_Random_Generator(number)
    for i in values:
        if i < 0.5:
            data.append(0)
        else:
            data.append(1)

    return data


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

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


def fourQAM(data):
    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(data):
    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 / np.sqrt(2), 0))
        elif k == [0, 0, 1]:
            EPSK.append(complex(1 / 2, 1 / 2))
        elif k == [0, 1, 1]:
            EPSK.append(complex(0, 1 / np.sqrt(2)))
        elif k == [0, 1, 0]:
            EPSK.append(complex(-1 / 2, 1 / 2))
        elif k == [1, 1, 0]:
            EPSK.append(complex(-1 / np.sqrt(2), 0))
        elif k == [1, 1, 1]:
            EPSK.append(complex(-1 / 2, -1 / 2))
        elif k == [1, 0, 1]:
            EPSK.append(complex(0, -1 / np.sqrt(2)))
        elif k == [1, 0, 0]:
            EPSK.append(complex(1 / 2, -1 / 2))
    return EPSK


def sixteenQAM(data):
    sixtQAM = []
```

```
    M = 4
    subList = [bits[n:n + M] for n in range(0, len(bits), M)]
    for k in subList:
        if k == [0, 0, 0, 0]:
            sixtQAM.append(complex(-3, -3))
        elif k == [0, 0, 0, 1]:
            sixtQAM.append(complex(-3, -1))
        elif k == [0, 0, 1, 1]:
            sixtQAM.append(complex(-3, 1))
        elif k == [0, 0, 1, 0]:
            sixtQAM.append(complex(-3, 3))
        elif k == [0, 1, 1, 0]:
            sixtQAM.append(complex(-1, 3))
        elif k == [0, 1, 1, 1]:
            sixtQAM.append(complex(-1, 1))
        elif k == [0, 1, 0, 1]:
            sixtQAM.append(complex(-1 - 1))
        elif k == [0, 1, 0, 0]:
            sixtQAM.append(complex(-1, -3))
        elif k == [1, 1, 0, 0]:
            sixtQAM.append(complex(1, -3))
        elif k == [1, 1, 0, 1]:
            sixtQAM.append(complex(1, -1))
        elif k == [1, 1, 1, 1]:
            sixtQAM.append(complex(1, 1))
        elif k == [1, 1, 1, 0]:
            sixtQAM.append(complex(1, 3))
        elif k == [1, 0, 1, 0]:
            sixtQAM.append(complex(3, 3))
        elif k == [1, 0, 1, 1]:
            sixtQAM.append(complex(3, 1))
        elif k == [1, 0, 0, 1]:
            sixtQAM.append(complex(3, -1))
        elif k == [1, 0, 0, 0]:
            sixtQAM.append(complex(3, -3))
    return sixtQAM


#
    ------------------------------------------------------------------------------
#                            Noise addition
#
    ------------------------------------------------------------------------------

def Add_noise(transmitted, Gnoise, M, SNR):
    gama = 1 / np.sqrt(math.pow(10, (SNR / 10)) * 2 * math.log2(M))
    # print(gama)
    new = [i * gama for i in Gnoise]
    R = list(map(add, transmitted, new))
    return R


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

```
#                               Detection
# ------------------------------------------------------------------------------

def BPSKDetection(comp):
    points = [-1, 1]
    Bpoints = [[0], [1]]
    recieved = -1
    minDistance = 99
    decoded = []
    Bdecoded = []
    for y in comp:
        for x in range(len(points)):
            distance = (y - points[x]) ** 2
            if distance <= minDistance:
                minDistance = distance
                recieved = x
        decoded.append(points[recieved])
        Bdecoded.append(Bpoints[recieved])
    return decoded, Bdecoded  # recieved


def QAM4Detection(comp):
    points = [(1 + 1j) / np.sqrt(2), (-1 - 1j) / np.sqrt(2),
              (1 - 1j) / np.sqrt(2), (-1 + 1j) / np.sqrt(2)]
    Bpoints = [[0, 0], [1, 1], [1, 0], [0, 1]]
    recieved = -1
    minDistance = 99
    decoded = []
    Bdecoded = []
    for y in comp:
        for x in range(len(points)):
            distance = (points[x] - y) ** 2
            if np.abs(distance) <= np.abs(minDistance):
                minDistance = distance
                recieved = x
        decoded.append(points[recieved])
        Bdecoded.append(Bpoints[recieved])
    return decoded, Bdecoded


def PSK8Detection(comp):
    # points = [(-1 - 1j) / np.sqrt(2), -1, 1j,(-1 + 1j) / np.sqrt(2), -1j, (1 - 1j) / np.sqrt(2), (1 + 1j) /
    # np.sqrt(2), 1]
    points = [complex(1 / np.sqrt(2), 0), complex(1 / 2, 1 / 2),
              complex(0, 1 / np.sqrt(2)), complex(-1 / 2, 1 / 2),
              complex(-1 / np.sqrt(2), 0), complex(-1 / 2, -1 / 2),
              complex(0, -1 / np.sqrt(2)), complex(1 / 2, -1 / 2)]
    Bpoints = [[0, 0, 0], [0, 0, 1],
               [0, 1, 1], [0, 1, 0],
               [1, 1, 0], [1, 1, 1],
               [1, 0, 1], [1, 0, 0]]
    recieved = -1
    minDistance = 99
    decoded = []
```

```
    Bdecoded = []
    for y in comp:
        for x in range(len(points)):
            distance = (points[x] - y) ** 2
            if np.abs(distance) <= np.abs(minDistance):
                minDistance = distance
                recieved = x
        decoded.append(points[recieved])
        Bdecoded.append(Bpoints[recieved])
    return decoded, Bdecoded


def QAM16Detection(comp):
    # points = [-1 + 1j, -1 + 1j / 3, -1 - 1j, -1 - 1j / 3,
    #           -1 / 3 + 1j, (-1 + 1j) / 3, -1 / 3 - 1j, (-1 + 1j) / 3,
    #           1 + 1j, 1 + 1j / 3, 1 - 1j, 1 - 1j / 3,
    #           1 / 3 + 1j, (1 + 1j) / 3, 1 / 3 - 1j, (1 - 1j) / 3]
    points = [complex(-3, -3), complex(-3, -1), complex(-3, 1), complex(-3, 3),
              complex(-1, 3), complex(-1, 1), complex(-1, -1), complex(-1, -3),
              complex(1, -3), complex(1, -1), complex(1, 1), complex(1, 3),
              complex(3, 3), complex(3, 1), complex(3, -1), complex(3, -3)]
    Bpoints = [[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 1], [0, 0, 1, 0],
               [0, 1, 1, 0], [0, 1, 1, 1], [0, 1, 0, 1], [0, 1, 0, 0],
               [1, 1, 0, 0], [1, 1, 0, 1], [1, 1, 1, 1], [1, 1, 1, 0],
               [1, 0, 1, 0], [1, 0, 1, 1], [1, 0, 0, 1], [1, 0, 0, 0]]
    recieved = -1
    minDistance = 99
    decoded = []
    Bdecoded = []
    for y in comp:
        for x in range(len(points)):
            distance = (points[x] - y) ** 2
            if np.abs(distance) <= np.abs(minDistance):
                minDistance = distance
                recieved = x
        decoded.append(points[recieved])
        Bdecoded.append(Bpoints[recieved])
    return decoded, Bdecoded


# -------------------------------------------------------------------------------
#
# -------------------------------------------------------------------------------


# ----------------------------------------------------------------------------
#                                 Transmision and detection
```

```
# -----------------------------------------------------------------------------
def transmission(n, bits):
    if n == 1:
        return BPSK(bits)
    elif n == 2:
        return fourQAM(bits)
    elif n == 3:
        return eight_PSK(bits)
    elif n == 4:
        return sixteenQAM(bits)


def detection(n, bits):
    if n == 1:
        return BPSKDetection(bits)
    elif n == 2:
        return QAM4Detection(bits)
    elif n == 3:
        return PSK8Detection(bits)
    elif n == 4:
        return QAM16Detection(bits)


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

    return BER


def SYM_error(sent, recieved):
    error = 0
    for k in range(len(recieved)):
        if sent[k] != recieved[k]:
            error += 1
    SER = error / len(recieved)
    return SER


# -------------------------------------------------------------------------------------
#                                    Task3
# -------------------------------------------------------------------------------------
# SNR= -4
# bpsk= 2 , 4QAM=4 , 8psk = 8, 16QAM =16
M = 4
bits = bits_gen(laterValues)  # The size is done at the top   of the code
# Select mapping constellation 1=BPSK , 2=4QAM,3= 8PSK, 4=16QAM
```

```
Mode = 2
sent = transmission(Mode, bits)


# Recieved=Add_noise(sent,Ax,M,SNR)
# Detected,Dbits=detection(Mode,Recieved)
# Dbits = [item for sublist in Dbits for item in sublist]
# BER=bit_errors(bits,Dbits)
# SER= SYM_error(sent,Detected)


def SER_BER(Mode, M, bits, SNR):
    BER = []
    SER = []
    for i in SNR:
        Recieved = Add_noise(sent, Ax, M, i)
        Detected, Dbits = detection(Mode, Recieved)
        Dbits = [item for sublist in Dbits for item in sublist]
        BER.append(bit_errors(bits, Dbits))
        SER.append(SYM_error(sent, Detected))
        # print (i)
    return BER, SER


SNR = np.linspace(-4, 12, 50)
# print(list(SNR))
# BER,SER= SER_BER(Mode,M,bits,SNR)
# print(BER)
```