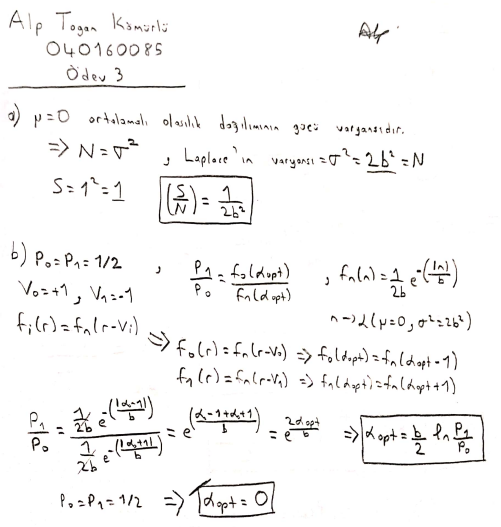
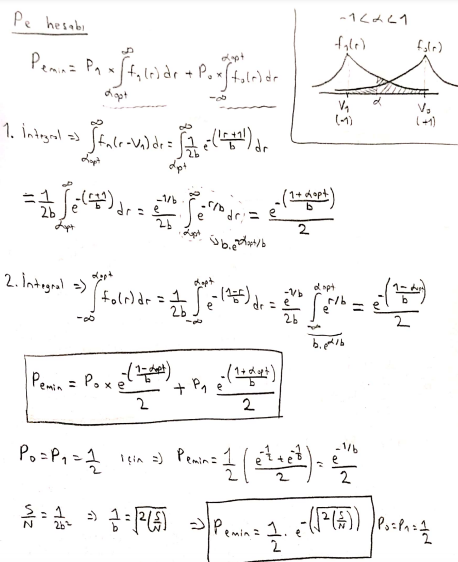
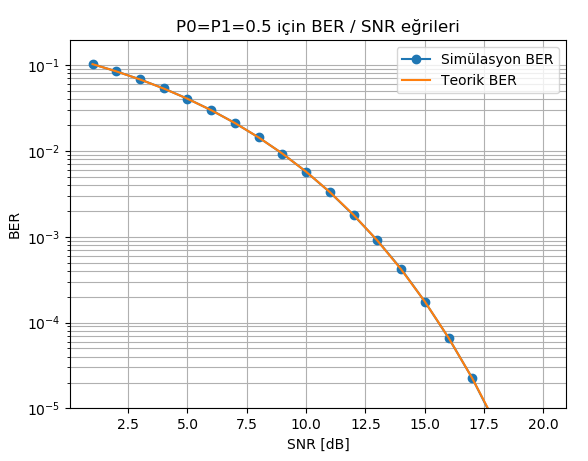
ALP TOGAN KÖMÜRLÜ

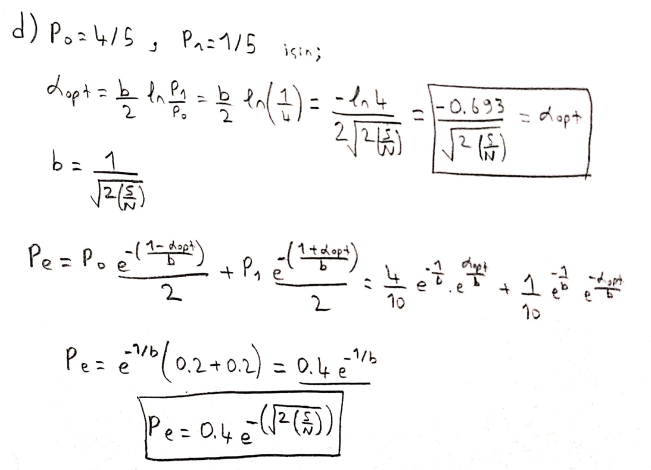
040160085



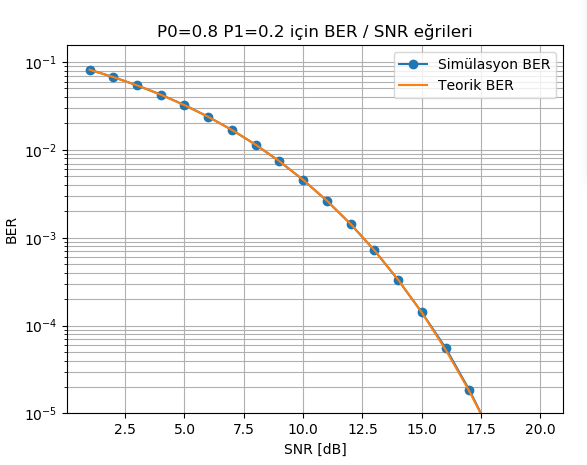


C)





E)



F)

Teorik sonuçtan beklendiği üzere ve simülasyonda elde edilen sonuçlarla birlikte bit hata olasılığının(BER), SNR yükseldikçe düşeceği gözlemlenmiştir. Bunun sebebi ise yüksek SNR’ın gürültünün varyansı düşürmesi ve bu da onun gücünü azaltmasıdır. Az gürültü ise işaretin algılanmasında daha az hataya sebep olur.

Simülasyon sonucu teorik değerlere benzerlik gösterdi. Yüksek SNR değerlerinde hata olasılığı hızlanarak azaldığı için örnekleme sayısı yüksek tutuldu. 10^7 mertebedeki örnekleme yeterli olmuştur.

Bir diğer gözlemlenen sonuç ise, p0=4/5, p1=1/5 için p0=p1=0.5 durumuna göre aynı SNR değerlerinde daha düşük hata olasılığı göstermiştir.

Monte Carlo simülasyonu Python kodu:

import numpy as np

import matplotlib.pyplot as plt

# N tane SNR değeri oluşturuldu

N = 20

SNR = np.array([i+1 for i in range(N)])

# snr değerinden varyans hesaplayan fonksiyon

def snr\_to\_variance(snr):

    real = np.power( 10 , snr/10 )

    var = 1 / ( np.power( 2\*real , 1/2 ) )

    return var

# her snr değeri için b bulundu

b = snr\_to\_variance(SNR)

# laplace dağılımlı rastgele değişken oluşturan fonksiyon tanımlandı

def laplace\_noise(u,variance,mean=0):

    lapl = mean - (variance \* np.sign(u) \* np.log(1-2\* np.abs(u) ) )

    return lapl

# düzgün dağılımlı 10000000 tane U rastlantı değişkeni oluşturuldu

sample\_size = 10000000

U = np.random.uniform(-0.5,0.5, (sample\_size) )

# her farklı varyans için 100000 adet "n" gürültüleri oluşturuldu

n = np.zeros((N,sample\_size))

for i in range(np.size(b)):

    n[i] = laplace\_noise(U,b[i])

# BER in hesaplandığı fonksiyon (Monte carlo)

def bit\_error(n,b,sample\_size,p0):

    BER = np.zeros(np.shape(n)[0])

    # hataların hesaplandığı döngü

    for j in range(np.size(b)):

        # gürültüye +1 ve -1 eklendi

        bit\_0 = n[j] +1

        bit\_1 = n[j] -1

        # alfa değeri bulunuyor

        alpha = 0.5\*b[j]\* np.log((1-p0)/p0)

        # her bir işarette yapılan hata sayısı bulundu

        n\_error\_0 = np.count\_nonzero(bit\_0 < alpha)

        n\_error\_1 = np.count\_nonzero(bit\_1 > alpha)

        # bit hata oranı hesaplandı

        BER[j] = p0 \* (n\_error\_0/sample\_size)  + (1-p0) \* (n\_error\_1/sample\_size)

    return BER

# Monte carlo metoduyla Bit hata oranları her iki olasılık için hesaplanıyor

BER\_c = bit\_error(n,b,sample\_size,p0=0.5)

BER\_e = bit\_error(n,b,sample\_size,p0=0.8)

# hem c hem e şıkkı için teorik hata hesaplama kısmı (karşılaştırma için)

Pe\_c = 0.5 \* np.power(np.e , -1\*np.power( 2\*(np.power( 10 , SNR/10 ) ) , 1/2) )

Pe\_e = 0.4 \* np.power(np.e , -1\*np.power( 2\*(np.power( 10 , SNR/10 ) ) , 1/2) )

# karşılaştırma plotları

plt.plot(SNR,BER\_c, "-o",label="Simülasyon BER")

plt.plot(SNR,Pe\_c,label="Teorik BER")

plt.title("P0=P1=0.5 için BER / SNR eğrileri")

plt.yscale("log")

plt.ylim(0.00001)

plt.xlabel("SNR [dB]")

plt.ylabel("BER")

plt.legend()

plt.grid(which="both")

plt.show()

plt.plot(SNR,BER\_e, "-o",label="Simülasyon BER")

plt.plot(SNR,Pe\_e,label="Teorik BER")

plt.title("P0=0.8 P1=0.2 için BER / SNR eğrileri")

plt.yscale("log")

plt.ylim(0.00001)

plt.xlabel("SNR [dB]")

plt.ylabel("BER")

plt.legend()

plt.grid(which="both")

plt.show()