**Part II)**

**2)**

**Matlab Code:**

1 – Communication System

import numpy as np

from math import erfc, sqrt

**# # Transmitter**

# Involves a binary source generating random 0s and 1s and a pulse shape filter that fits pulses on the bitstream.

def BinarySource(n):

return np.random.randint(0,2,n)

def binarycode\_to\_signal(bitstream, step):

T = 1 # assume the pulse period is 1

A = 1 # assume the amplitude is 1

pulse = np.ones(int(T/step))

pulse = pulse\*A

signal = np.zeros(len(bitstream)\*len(pulse))

# Polar nonreturn to zero

for i in range(len(bitstream)):

if bitstream[i] == 1:

signal[i\*len(pulse):(i+1)\*len(pulse)] = 1\*pulse

else:

signal[i\*len(pulse):(i+1)\*len(pulse)] = -1\*pulse

return signal

**# # Channel**

# Only adds additive white Gaussian noise on the signal

def AWGN(n, sigma):

return np.random.normal(0,sigma,n)

**# # Receiver**

# Receive signal from the channel then pass it by receive felter, sampler and decision maker.

def receive\_filter(signal\_noise, filter\_num, step):

filter\_num-=1 # Get next filter

filters = [np.ones(int(1/step)), np.ones(1), np.sqrt(3)\*np.arange(0, 1, step)]

filter = filters[filter\_num]

filter = np.concatenate((filter, np.zeros(int(1/step)-len(filter))))

signal\_noise\_filter=np.convolve(signal\_noise, filter)

return signal\_noise\_filter

def sampler(sampling\_period, signal\_noise\_filtered, n=10):

samples = np.zeros(n)

for i in range(len(samples)):

samples[i] = signal\_noise\_filtered[sampling\_period-1+i\*sampling\_period]

return samples

Q = lambda x : 0.5 \* erfc(x/sqrt(2))

def decision\_maker(samples, λ):

return (samples>λ)\*1

# The python files “communication-system” and library includes these functions along with a free simulation on small n.

2 – Simulation:

import numpy as np

import matplotlib.pyplot as plt

from library import AWGN, BinarySource, binarycode\_to\_signal, receive\_filter, Q, decision\_maker, sampler

# simulation parameters

n = 100000 # bitstream length

step = 0.05 # 20 Samples per pulse of duration 1

T=1 # pulse period

## Generate a random bitstream and cascade it through the communication system:

# generate the binary symbols (1)

input\_bitstream = BinarySource(n)

# generate the binary signal (2)

g\_t = binarycode\_to\_signal(input\_bitstream, step)

# The following function simulates the channel and receiver for different noise variances/filter it then returns the probability of error associated with the transmission

def compute\_BER(σₙ, f): # σₙ is the noise's variance and f is the filter's E\_Nₒ. (1 or 2 or 3)

# generate the noise

w\_t = AWGN(len(g\_t), σₙ)

# add the Nₒise to the signal (3)

s\_t = g\_t + w\_t

# apply the filter to the signal (4)

y\_t = receive\_filter(s\_t,f, step)

# sample the filtered signal (5)

sampling\_period = int(T/step)

y\_iT = sampler(sampling\_period, y\_t,n) # y\_T has all samples

# decode the samples (6)

λ = 0 # holds for all 3 cases.

bitstream\_output = decision\_maker(y\_iT, λ)

# compare

return np.sum(input\_bitstream != bitstream\_output)/len(input\_bitstream)

# plot BER VS. E/No for each filter

E\_Nₒ=np.arange(-10, 20, 1) # E\_Nₒ range

Nₒ = 1/(10\*\*(E\_Nₒ/10))

σₙ = np.sqrt(Nₒ/2) # the corresponding range of sigma.

# Filter 1:

filter1\_BER, filter1\_BER\_th = np.zeros(len(σₙ)), np.zeros(len(σₙ))

for i in range(len(σₙ)):

filter1\_BER[i] = compute\_BER(σₙ[i], 1)

filter1\_BER\_th[i] = Q(1/σₙ[i])

# Filter 2:

filter2\_BER, filter2\_BER\_th = np.zeros(len(σₙ)), np.zeros(len(σₙ))

for i in range(len(σₙ)):

filter2\_BER[i] = compute\_BER(σₙ[i], 2)

filter2\_BER\_th[i] = Q(1/σₙ[i])

# Filter 3:

filter3\_BER, filter3\_BER\_th = np.zeros(len(σₙ)), np.zeros(len(σₙ))

for i in range(len(σₙ)):

filter3\_BER[i] = compute\_BER(σₙ[i], 3)

filter3\_BER\_th[i] = Q(np.sqrt(3)/2\*1/σₙ[i])

plt.semilogy(E\_Nₒ, filter1\_BER, 'r')

plt.semilogy(E\_Nₒ, filter2\_BER, 'g.')

plt.semilogy(E\_Nₒ, filter3\_BER, 'b')

plt.semilogy(E\_Nₒ, filter1\_BER\_th, 'c')

plt.semilogy(E\_Nₒ, filter2\_BER\_th, 'm--')

plt.semilogy(E\_Nₒ, filter3\_BER\_th, 'y')

plt.xlabel('E/Nₒ (db)')

plt.ylabel('BER (log-scale)')

plt.title(' BER VS. E/Nₒ')

plt.legend(['Matched Filter (1)', 'No Filter (2)', 'Linear Filter (3)', 'Theory 1', 'Theory 2', 'Theory 3'])

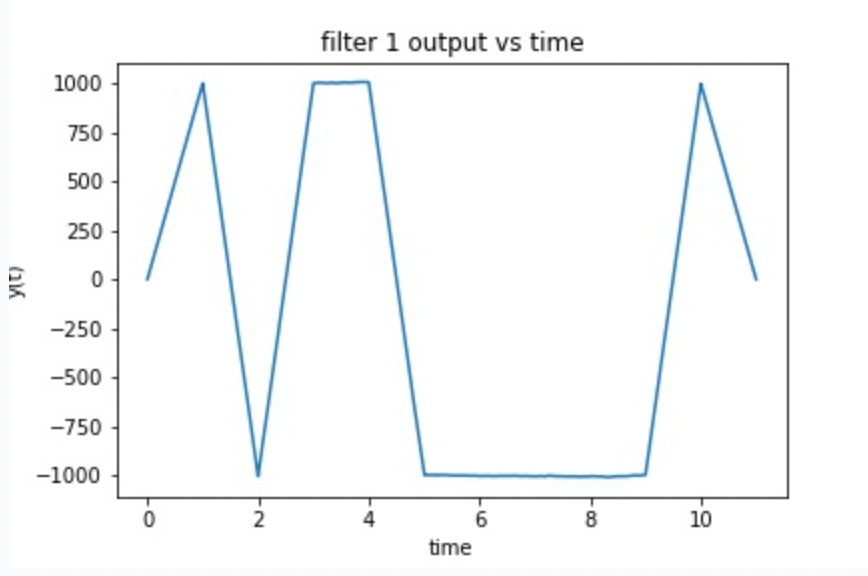
plt.ylim([1/n\*100, 1])

plt.savefig('./BER.png')

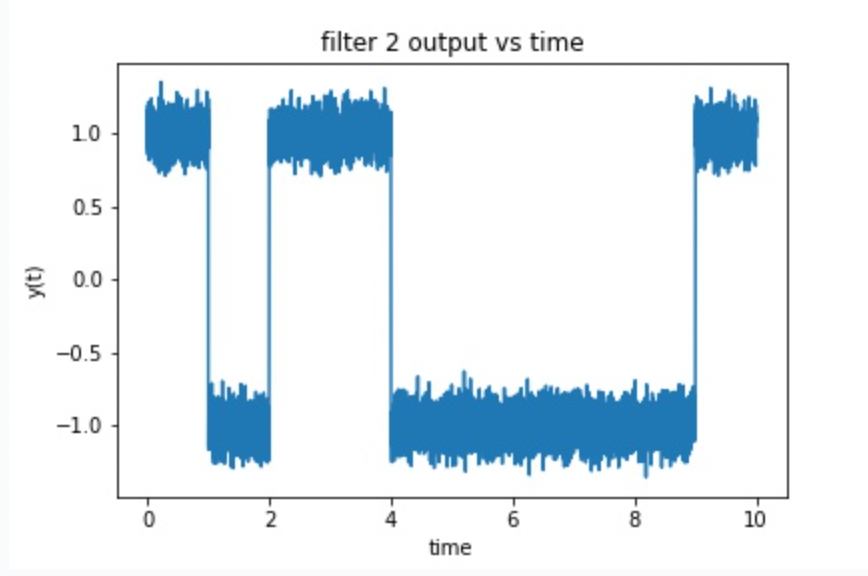
**3)**

**Provided an input bitstream 1011000001**

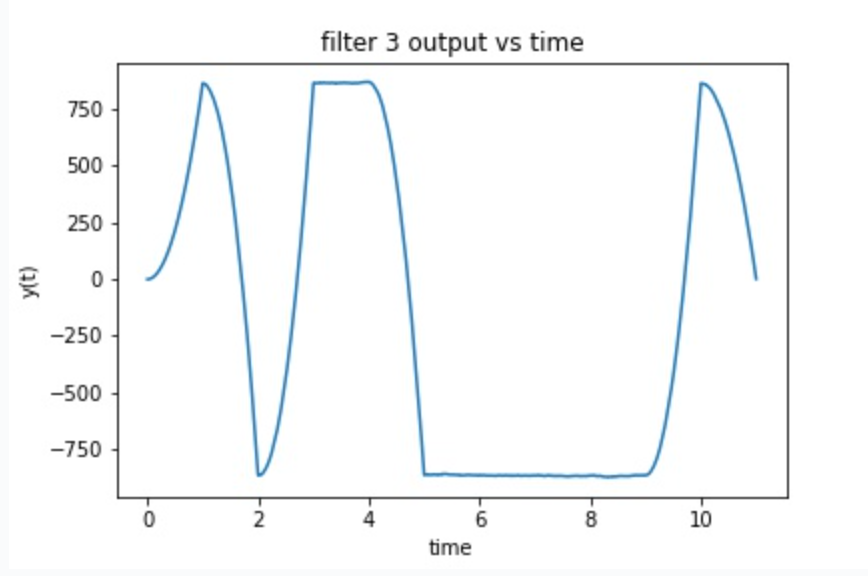
Output due to matched filter:

****

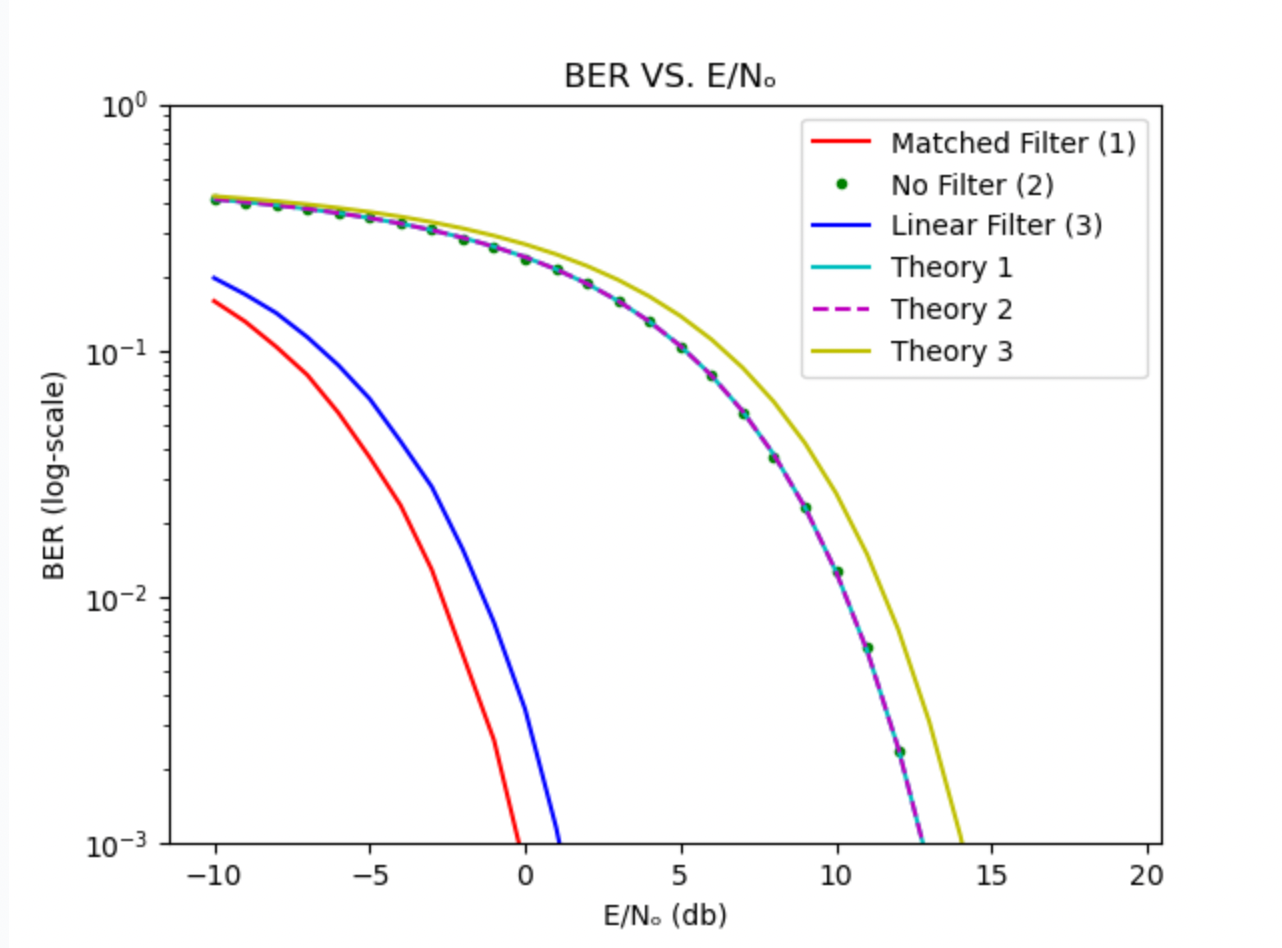
Output due to no filter:



Output due to linear filter:



**4)**



**5)**

The BER is decreasing as a function of E/No as in the plot. We can justify this in different ways:

1 - As E/No increases (here E is constant) No decreases and thus σₙ which means the added AWGN involves less variations (corresponds to a thinner Gaussian distribution) and thus any noise addedcorresponds to small values close to zero which do not affect the signal that much (hence BER decreases)

2 - As in the theoritical expression and knowing that Q is a decreasing function, its clear that Q(a \* sqrt(E/No))

for all the cases above. Hence, BER is a decreasing function of E/No (noting that sqrt is an increasing function)

**6)**

The matched filter case is the one with lowest BER since it uses a filter matched to the pulse to minimize the probability of error (as we have proven in the lecture.) To accompolish this it equivalently maximizes the peak pulse SNR at the sampling instant.

Note that in the theoritical case using a filter or not yields the same expression due to our assumptions on variance and PSD.