**Tittle: Basic function of Code Division Multiple Access (CDMA).**

import numpy as np

c1=[1,1,1,1]

c2=[1,-1,1,-1]

c3=[1,1,-1,-1]

c4=[1,-1,-1,1]

rc=[]

print("Enter the data bits :")

d1=int(input("Enter D1 :"))

d2=int(input("Enter D2 :"))

d3=int(input("Enter D3 :"))

d4=int(input("Enter D4 :"))

r1=np.multiply(c1,d1)

r2=np.multiply(c2,d2)

r3=np.multiply(c3,d3)

r4=np.multiply(c4,d4)

resultant\_channel=r1+r2+r3+r4;

print("Resultant Channel",resultant\_channel)

Channel=int(input("Enter the station to listen for C1=1 ,C2=2, C3=3 C4=4 : "))

if Channel==1:

rc=c1

elif Channel==2:

rc=c2

elif Channel==3:

rc=c3

elif Channel==4:

rc=c4

inner\_product=np.multiply(resultant\_channel,rc)

print("Inner Product",inner\_product)

res1=sum(inner\_product)

data=res1/len(inner\_product)

print("Data bit that was sent",data

the code you provided implements a simple example of a Code Division Multiple Access (CDMA) system using spreading codes. Let's break down the code step by step:

1. Import the **numpy** library for numerical operations.
2. Define spreading codes **c1**, **c2**, **c3**, and **c4**, which are sequences of numbers representing the spreading codes for each user. Each spreading code is a binary sequence.
3. Create an empty list **rc** to store the selected spreading code.
4. Prompt the user to enter data bits (**d1**, **d2**, **d3**, and **d4**) to be transmitted. These are the user-specific data bits that will be multiplied by the spreading codes.
5. Perform element-wise multiplication between each spreading code and its respective data bit using **np.multiply()**, generating **r1**, **r2**, **r3**, and **r4**.
6. Calculate the resultant channel by adding up all the encoded data bits: **resultant\_channel = r1 + r2 + r3 + r4**.
7. Prompt the user to enter the station (channel) they want to listen to, represented by the variable **Channel**.
8. Based on the chosen station, select the corresponding spreading code and assign it to **rc**.
9. Perform element-wise multiplication between the resultant channel and the selected spreading code using **np.multiply()**, generating **inner\_product**.
10. Compute the sum of the elements in **inner\_product** using **sum(inner\_product)** and assign it to **res1**.
11. Calculate the data bit that was sent by dividing **res1** by the length of **inner\_product**, and assign it to **data**.
12. Print the resultant channel, inner product, and the data bit that was sent.

The code simulates a simplified scenario of a CDMA system where the user's data bits are encoded with spreading codes and transmitted over a channel. The receiver selects a specific station and correlates the received signal with the corresponding spreading code to decode the original data bit.

Please note that this code assumes user input for data bits and channel selection. In a real-world implementation, these values would typically be received from the physical layer of the system, such as through a receiver or demodulator. Additionally, error detection and correction mechanisms, synchronization, and other aspects are not addressed in this simplified example

**Server Program:**

import socket

HOST = 'localhost' # Server IP address

PORT = 12345 # Server port number

def receive\_file():

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as server\_socket:

server\_socket.bind((HOST, PORT))

server\_socket.listen(1)

print('Waiting for connection...')

client\_socket, addr = server\_socket.accept()

print('Connected to:', addr)

with open('received\_file', 'wb') as file:

while True:

data = client\_socket.recv(1024)

if not data:

break

file.write(data)

print('File received successfully.')

receive\_file()

**Client Program:**

**import socket**

**HOST = 'localhost' # Server IP address**

**PORT = 12345 # Server port number**

**FILE\_PATH = 'file\_to\_send.txt' # Path of the file to be sent**

**def send\_file():**

**with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as client\_socket:**

**client\_socket.connect((HOST, PORT))**

**with open(FILE\_PATH, 'rb') as file:**

**for data in file:**

**client\_socket.sendall(data)**

**print('File sent successfully.')**

**send\_file()**

Make sure to change the HOST, PORT, and FILE\_PATH variables according to your specific setup. The server will save the received file as 'received\_file' in the same directory.

Remember to handle exceptions, perform error checking, and add additional features as needed.

**Explaination for code: Server side**

**Import the socket module:** This module provides the necessary functions and classes for creating and interacting with sockets.

**Define the HOST and PORT variables**: These variables specify the server's IP address (in this case, 'localhost' refers to the current machine) and the port number on which the server will listen for incoming connections.

**Define the receive\_file() function:** This function handles the server-side file receiving logic.

**Create a socket object:** Using socket.socket(), we create a TCP socket for the server. The AF\_INET parameter indicates that we are using IPv4 addressing, and SOCK\_STREAM specifies that we want to use TCP

**Bind the socket to the server address:** Using server\_socket.bind((HOST, PORT)), we associate the socket with the server address and port number. This step is necessary before listening for incoming connections.

**Start listening for connections:** Using server\_socket.listen(1), the server socket begins listening for incoming connections. The argument '1' specifies the maximum number of queued connections.

**Accept incoming connection:** When a client attempts to connect, server\_socket.accept() blocks the execution until a client successfully establishes a connection. It returns a new socket object (client\_socket) representing the connection and the address (addr) of the client.

**Open a file to save the received data:** Using open('received\_file', 'wb'), we open a new file in binary write mode with the name 'received\_file'. This file will store the received data.

**Receive and write data to the file:** Inside the while loop, we continuously receive data from the client using client\_socket.recv(1024). The received data is stored in the data variable. If no data is received (empty data), the loop breaks.

**Write data to the file:** We write the received data to the file using file.write(data).

**Print a success message:** Once the loop completes, we print a success message indicating that the file was received successfully.

**Call the receive\_file() function**: At the end of the code, we call the receive\_file() function to start the server and wait for a client to connect.

**For client:**

The provided code is the client-side portion of the file transfer program using TCP in Python. Let's break down the program:

**Import the socket module:** This module provides the necessary functions and classes for creating and interacting with sockets.

**Define the HOST, PORT, and FILE\_PATH variables:** These variables specify the server's IP address (in this case, 'localhost' refers to the current machine), the server port number, and the path of the file to be sent.

**Define the send\_file() function:** This function handles the client-side file sending logic.

**Create a socket object:** Using socket.socket(socket.AF\_INET, socket.SOCK\_STREAM), we create a TCP socket for the client. The AF\_INET parameter indicates that we are using IPv4 addressing, and SOCK\_STREAM specifies that we want to use TCP.

**Connect to the server:** Using client\_socket.connect((HOST, PORT)), the client socket attempts to establish a connection with the server specified by the HOST and PORT variables.

**Open the file to be sent:** Using open(FILE\_PATH, 'rb'), we open the file specified by the FILE\_PATH variable in binary read mode.

**Iterate over the file data:** Using a for loop, we iterate over the file data line by line.

**Send data to the server:** Inside the loop, we use client\_socket.sendall(data) to send each line of data to the server.

**Print a success message:** Once the loop completes, we print a success message indicating that the file was sent successfully.

**Call the send\_file() function:** At the end of the code, we call the send\_file() function to initiate the file transfer process.

Make sure to have the specified file available in the correct location before running the code. Additionally, ensure that the server is running and configured with the correct HOST and PORT

**Tittle: Simulate BER Performance over Rayleigh Fading wireless channel with BPSK Transmission**

**Program:**

clear

N = 10^6 % number of bits or symbols

rand('state',100); % initializing the rand() function

randn('state',200); % initializing the randn() function

% Transmitter

ip = rand(1,N)>0.5; % generating 0,1 with equal probability

s = 2\*ip-1; % BPSK modulation 0 -> -1; 1 -> 1

n = 1/sqrt(2)\*[randn(1,N) + j\*randn(1,N)]; % white gaussian noise, 0dB variance

Eb\_N0\_dB = [-3:10]; % multiple Eb/N0 values

for ii = 1:length(Eb\_N0\_dB)

% Noise addition

y = s + 10^(-Eb\_N0\_dB(ii)/20)\*n; % additive white gaussian noise

% receiver - hard decision decoding

ipHat = real(y)>0;

% counting the errors

nErr(ii) = size(find([ip- ipHat]),2);

end

simBer = nErr/N; % simulated ber

theoryBer = 0.5\*erfc(sqrt(10.^(Eb\_N0\_dB/10))); % theoretical ber

% plot

close all

figure

semilogy(Eb\_N0\_dB,theoryBer,'b.-');

hold on

semilogy(Eb\_N0\_dB,simBer,'mx-');

axis([-3 10 10^-5 0.5])

grid on

legend('theory', 'simulation');

xlabel('Eb/No, dB');

ylabel('Bit Error Rate');

title('Bit error probability curve for BPSK modulation');

1. Set the number of bits or symbols (**N**) to 10^6.
2. Initialize the random number generator (**rand()** and **randn()**) states to specific values using **rand('state', 100)** and **randn('state', 200)**.
3. Transmitter section:
   * Generate a random binary sequence of length **N** with equal probability of 0 and 1 using **ip = rand(1, N) > 0.5**.
   * Perform BPSK modulation, mapping 0 to -1 and 1 to 1, by multiplying the binary sequence by 2 and subtracting 1: **s = 2 \* ip - 1**.
4. Add AWGN to the transmitted signal:
   * Generate complex Gaussian noise with zero mean and unit variance using **randn(1, N) + j \* randn(1, N)**. The factor **1/sqrt(2)** scales the noise to have a variance of 0.5.
   * Multiply the noise by **10^(-Eb\_N0\_dB(ii)/20)** to adjust its power based on the desired signal-to-noise ratio (**Eb/N0**) values. This represents the AWGN channel.
   * Add the noise to the transmitted signal: **y = s + 10^(-Eb\_N0\_dB(ii)/20) \* n**.
5. Receiver section:
   * Perform hard decision decoding by thresholding the received signal. The real part of **y** is compared to zero to estimate the transmitted symbols: **ipHat = real(y) > 0**.
6. Calculate the number of bit errors (**nErr**) by comparing the estimated symbols (**ipHat**) to the original transmitted symbols (**ip**).
7. Repeat the simulation for multiple **Eb/N0** values defined in the range **[-3:10]**.
8. Calculate the simulated bit error rate (**simBer**) by dividing the number of bit errors (**nErr**) by the total number of symbols (**N**).
9. Calculate the theoretical bit error rate (**theoryBer**) using the formula **0.5 \* erfc(sqrt(10.^(Eb\_N0\_dB/10)))**, where **erfc()** is the complementary error function.
10. Plot the theoretical and simulated bit error rate curves using **semilogy()** to create a logarithmic scale plot.
11. Set the plot limits, add grid lines, legends, and axis labels.
12. Display the bit error probability curve for the BPSK modulation scheme.

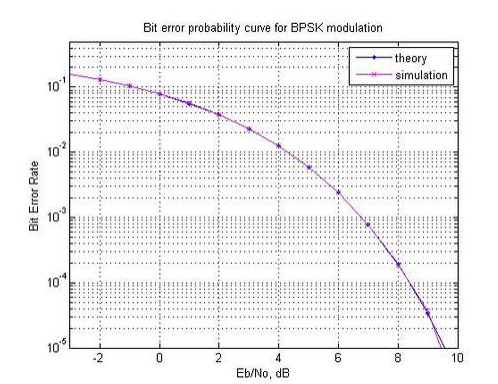
The code allows you to observe how the bit error rate changes with different **Eb/N0** values, representing different levels of signal-to-noise ratio. The simulated results (**simBer**) are compared to the theoretical results (**theoryBer**) to validate the simulation.

Note: The code is written in MATLAB or Octave syntax.

Top of Form

Regenerate response

Output:

Bottom of Form