**DIGITAL COMMUNICATION LAB REPORT (2022-2023)**

**A Project Report submitted in partial fulfillment of the requirements for the award of the degree of**

**Bachelor of Technology**

**in**

**Electronics and Communication Engineering**

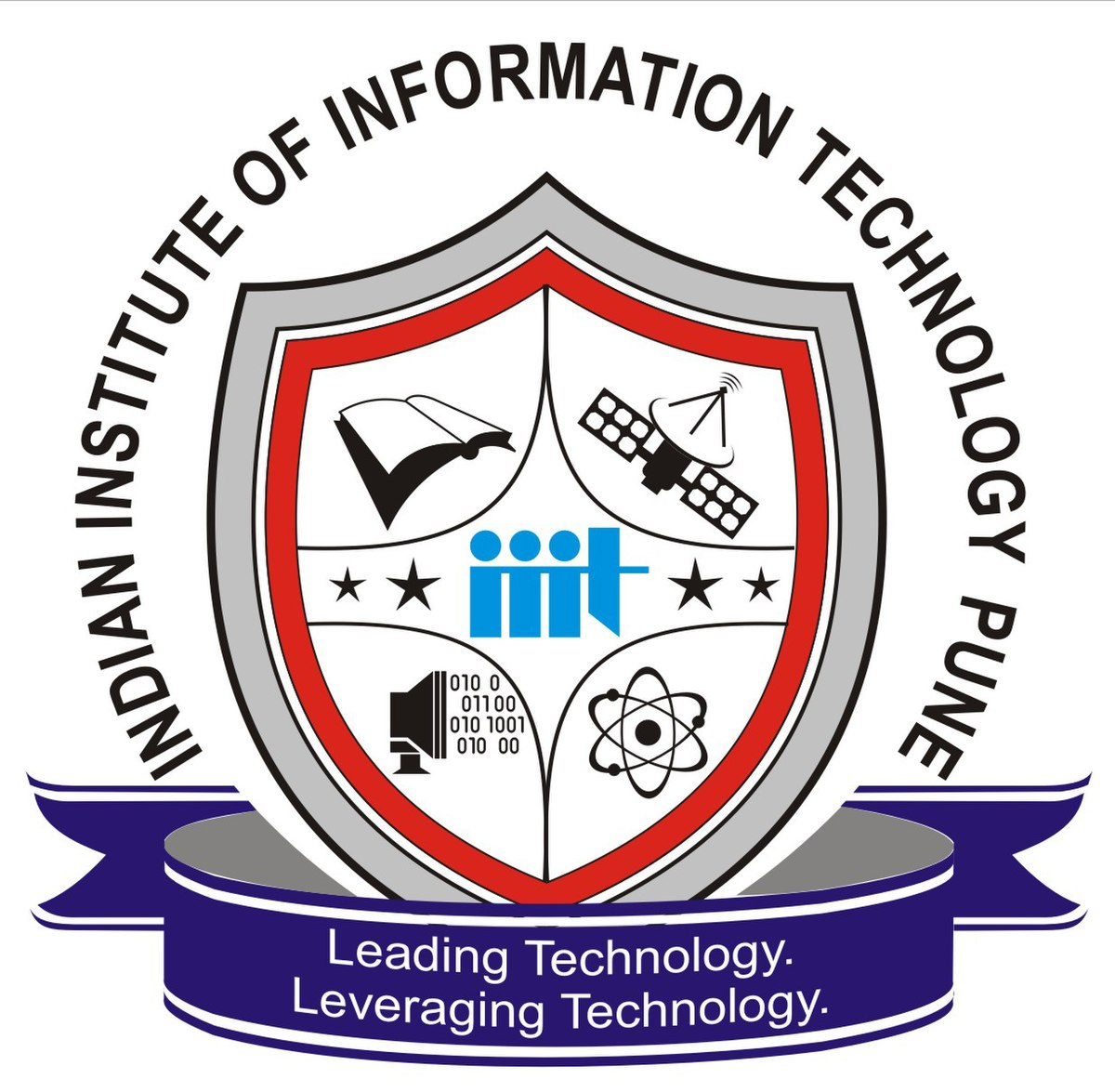
by

**SHIVANSHU VERMA (112116037):**

**Under the Supervision of: Dr. Nagendra Kushwaha**

**Semester: 5**

**Department of Electronics and Communication Engineering**



**Indian Institute of Information And Technology, Pune**

**(An Institute of National Importance by an Act of Parliament)**

**November 2023**

**EXPERIMENT-1**

**Aim**: To explore various random variable generation techniques and manipulate random data, including Gaussian random variables, uniform random variables, random integers, binary sequences, and BPSK symbols. Additionally, to simulate an Additive White Gaussian Noise (AWGN) channel.

**Apparatus Required:**

1. Python environment with NumPy library for random variable generation and manipulation.

**Theory:**

Generation of Gaussian Random Variable:

Gaussian (normal) random variables are characterized by their mean (average) and standard deviation (a measure of spread). Generating a Gaussian random variable allows us to model a wide range of real-world phenomena.

Generation of Array of Gaussian Random Variables:

This operation demonstrates how to create a 2D array of Gaussian random variables, useful for modeling multivariate data.

Generation of White Gaussian Noise:

White Gaussian noise is characterized by its randomness and constant power spectral density. It's commonly used to simulate the effects of noise in communication systems.

Generation of Random Integers:

Random integers are often used in applications such as simulations and modeling where discrete values are needed.

Generation of a Random Sequence of 1s and 0s:

This operation is useful for simulating digital data sequences, making it relevant for communication systems modeling.

Generation of BPSK Symbols:

BPSK is a digital modulation scheme where binary data is represented by phase shifts of a carrier signal. The symbols are +A and -A, where A is the amplitude.

Generation of AWGN and Adding to BPSK Symbols:

AWGN is a common model for noise in communication channels. Adding it to transmitted symbols simulates the effect of noise on the received signal.

**PROGRAM:**

import numpy.random as nr

import numpy as np

# Generate a single Gaussian random variable with mean 0 and standard deviation 1

print(nr.normal())

# Generate a single Gaussian random variable with mean 1 and standard deviation 1

b = nr.normal(1, 1, 1)

print(b)

# Generate a 2x2 array of Gaussian random variables with mean 3 and standard deviation 5^0.5

c = []

for i in range(2):

    k = []

    for j in range(2):

        k.append(nr.normal(3, 5\*\*0.5, 1))

    c.append(k)

print(c)

# Generate white Gaussian noise

# Generate a uniform random variable between 2 and 5, repeated 5 times

d = nr.uniform(2, 5, 5)

print(d)

# Generate random integers between 0 and 4, repeated 5 times

e = nr.randint(4, size=5)

# Generate a random sequence of 1s and 0s of length 7

f = nr.randint(0, 2, size=7)

print(f)

# Generate BPSK symbols with amplitude A=5

A = 5

g = nr.randint(0, 2, size=10)

h = [1\*A if i == 0 else i\*(-A) for i in g]

print(h)

# Alternative way to generate BPSK symbols using array operations

bpsk\_symbols = 2 \* g - 1

i = A \* bpsk\_symbols

print(i)

# Generate Additive White Gaussian Noise (AWGN)

snr\_db = 10  # Signal-to-Noise Ratio in dB

snr = 10 \*\* (snr\_db / 10.0)  # Convert SNR from dB to linear scale

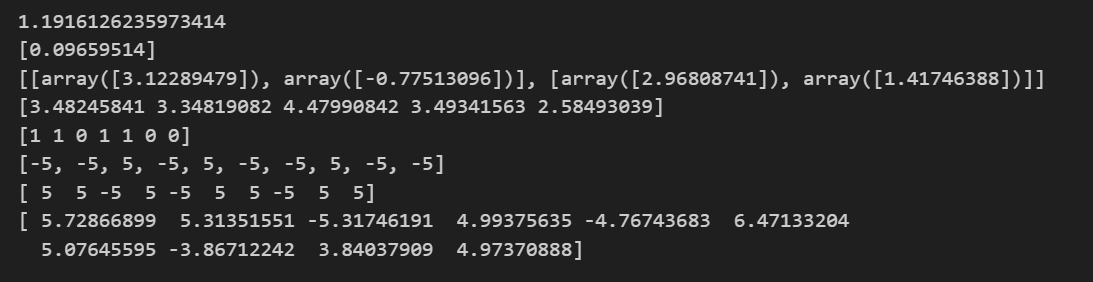
noise\_power = A\*\*2 / (2 \* snr)  # Calculate noise power based on SNR

awgn = np.random.normal(0, np.sqrt(noise\_power), len(i))

received\_symbols = i + awgn

print(received\_symbols)

OUTPUT:



**EXPERIMENT-2**

**Aim**: To simulate Binary Phase Shift Keying (BPSK) modulation, visualize the waveforms, and understand the modulation process.

**Apparatus Required:**

1. Python environment with NumPy, SciPy, and Matplotlib libraries for data manipulation and visualization.

**Theory:**

In this practical, we aim to simulate Binary Phase Shift Keying (BPSK) modulation and visualize the involved waveforms. BPSK is a digital modulation scheme that represents binary data using two different phase shifts of a carrier signal. The process involves:

1. Unipolar to Bipolar Conversion:
   * The binary input sequence is converted to a bipolar representation (-1s and 1s).
2. Determination of Bit Duration and Carrier Frequency:
   * The bit duration is set to 1, representing the time to transmit one bit.
   * The carrier frequency is determined based on the bit duration.
3. Generation of BPSK Modulated Signal:
   * The modulated signal is created by multiplying the bipolar data with a carrier waveform.
4. Data Visualization:
   * The waveforms are visualized to gain insights into the modulation process.

**PROGRAM**

from \_\_future\_\_ import division

import numpy as np

import scipy

import matplotlib.pylab as plt

unipolar\_arr = np.array([1, 0, 1, 1, 0])

bipolar = 2\*unipolar\_arr - 1

bit\_duration = 1

amplitude\_scaling\_factor = bit\_duration/2

freq = 3/bit\_duration

n\_samples = 1000

time = np.linspace(0, 5, n\_samples)

samples\_per\_bit = n\_samples/unipolar\_arr.size

dd = np.repeat(unipolar\_arr, samples\_per\_bit)

bb = np.repeat(bipolar, samples\_per\_bit)

dw = dd

bw = bb

waveform = np.sqrt(2\*amplitude\_scaling\_factor/bit\_duration) \* np.cos(2\*np.pi \* freq \* time)  # no need for np.dot to perform scalar-scalar multiplication or scalar-array multiplication

bpsk\_w = bw\*waveform

f, ax = plt.subplots(4,1, sharex=True, sharey=True, squeeze=True)

ax[0].plot(time, dw)

ax[1].plot(time, bw)

ax[2].plot(time, waveform)

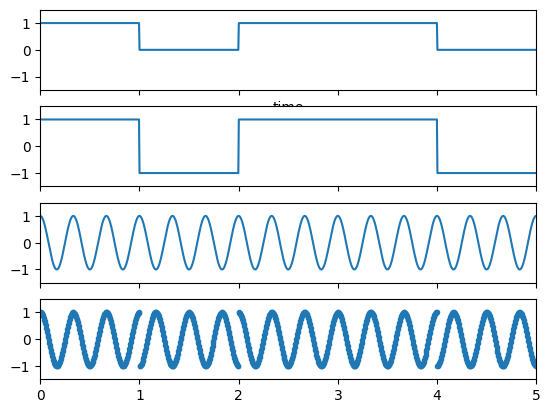
ax[3].plot(time, bpsk\_w, '.')

ax[0].axis([0, 5, -1.5, 1.5])

ax[0].set\_xlabel('time')

plt.show()

**OUTPUT:**

****

**EXPERIMENT-3**

**Aim** : To analyze the Bit Error Rate (BER) performance of a binary communication system under different Signal-to-Noise Ratios (SNR).

**Equipment Required** : Python environment with NumPy and Matplotlib libraries installed.

**Theory** : Bit Error Rate (BER) is a metric used to evaluate the performance of a communication system. It measures the probability of an error occurring in the received data bits compared to the transmitted data bits.

In this practical, we'll simulate a simple binary communication system. We'll transmit a binary signal (0s and 1s) through an additive white Gaussian noise (AWGN) channel, which is characterized by SNR. The transmitted signal is affected by noise, and the received signal is processed to estimate the transmitted bits. BER is calculated as the ratio of the number of bits received incorrectly to the total number of transmitted bits.

**Algorithm**:

* Import necessary libraries, including NumPy and Matplotlib.
* Initialize the block length, a range of SNR values, and an empty list to store BER results.
* Loop through each SNR value:
* Generate a random binary signal sys of length block\_length.
* Calculate SNR from SNR in dB.
* Generate random Gaussian noise.
* Transmit the signal by scaling it with the square root of SNR.
* Add noise to the transmitted signal to simulate the channel.
* Decode the received signal by thresholding it (bits greater than 0 are considered as 1, and others as 0).
* Count the bit errors by comparing the decoded bits to the original sys signal.
* Calculate BER as the ratio of bit errors to the block length.Display SNR and corresponding BER values.
* Create a plot of SNR vs. BER using Matplotlib.

**Program**:

import numpy as np

block\_length = 10000

snr\_db = np.arange(1, 15, 1)

ber\_results = []

for snr\_db\_value in snr\_db:

    snr = 10 \*\* (snr\_db\_value \* 0.1)

    sys = 2 \* (np.random.randint(0, 2, block\_length) - 0.5)

    noise = np.random.normal(0, 1, block\_length)

    transmitted\_bits = sys \* np.sqrt(snr)

    received\_bits = transmitted\_bits + noise

    decoded\_bits = 2\*(received\_bits>0) -1

    bit\_errors = np.sum(decoded\_bits != sys)

    ber = bit\_errors / block\_length

    ber\_results.append(ber)

for snr\_db\_value, ber in zip(snr\_db, ber\_results):

    print(f"SNR (dB): {snr\_db\_value}, Bit Error Rate: {ber:.4f}")

import matplotlib.pyplot as plt

plt.figure()

plt.semilogy(snr\_db, ber\_results, marker='o', linestyle='-')

plt.title('SNR vs. Bit Error Rate (BER)')

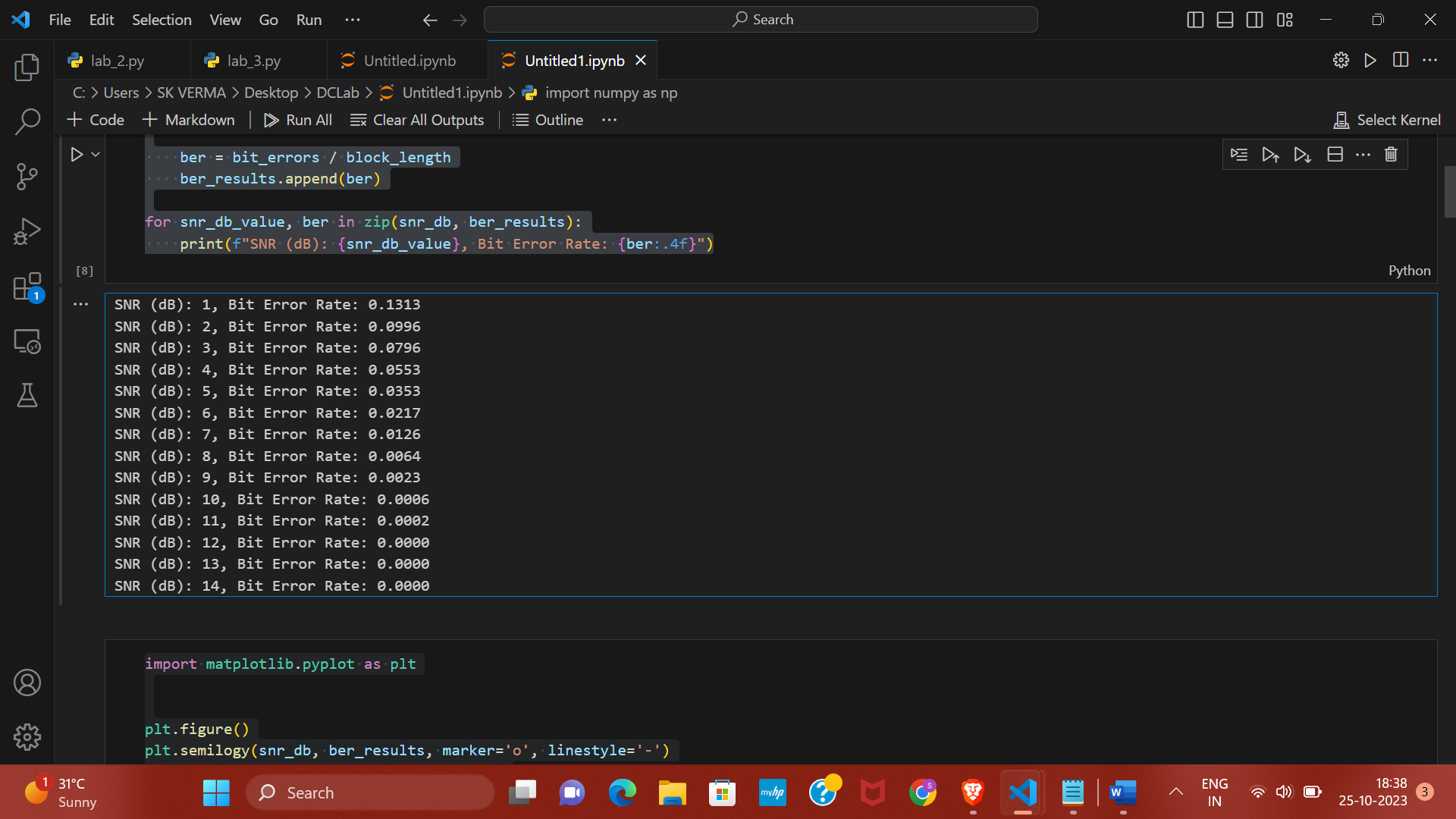
plt.xlabel('SNR (dB)')

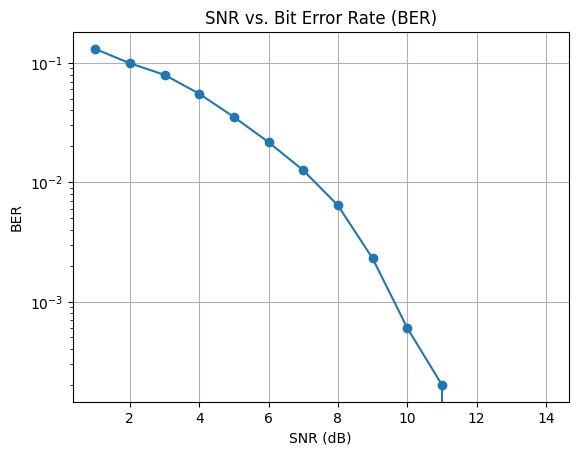
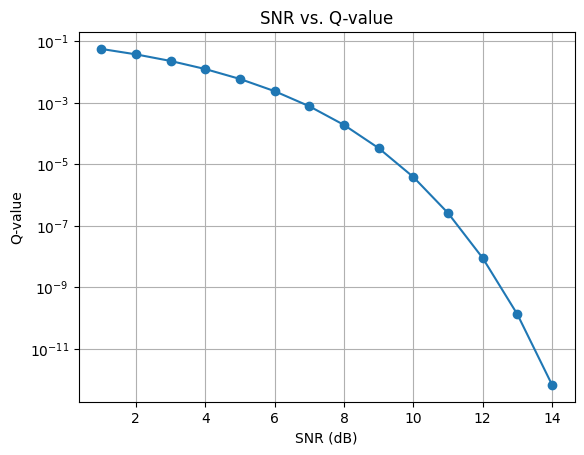
plt.ylabel('BER')

plt.grid(True)

plt.show()

**Output:**



**Result:**

The Bit Error Rate (BER) performance of the binary communication system is analyzed under different Signal-to-Noise Ratios (SNR). As the SNR increases, the BER decreases, indicating a more reliable communication system. The plot visualizes this relationship between SNR and BER.

**EXPERIMENT-4**

**Aim** : To analyze the Bit Error Rate (BER) performance of an Amplitude-Shift Keying (ASK) binary communication system under different Signal-to-Noise Ratios (SNR).

**Equipment Required** : Python environment with NumPy and Matplotlib libraries installed.

**Theory** : Amplitude-Shift Keying (ASK) is a digital modulation technique in which the amplitude of the carrier signal is varied to transmit binary data. In this practical, we'll simulate a binary ASK communication system and evaluate its BER performance under various SNR conditions. The transmitted signal is affected by noise, and the received signal is processed to estimate the transmitted bits.

BER is calculated as the ratio of the number of bits received incorrectly to the total number of transmitted bits.

**Algorithm**:

1. Import the NumPy library.
2. Define the block length, a range of SNR values, and an empty list for storing BER results.
3. Loop through each SNR value:
   * Calculate SNR from SNR in dB.
   * Generate a random binary signal **sys** of length **block\_length**.
   * Generate random Gaussian noise.
   * Transmit the signal by scaling it with the square root of SNR.
   * Add noise to the transmitted signal to simulate the channel.
   * Decode the received signal by thresholding it using the ASK modulation rule.
   * Count the bit errors by comparing the decoded bits to the original **sys** signal.
   * Calculate BER as the ratio of bit errors to the block length.
4. Display SNR and corresponding BER values.

**Program:**

import numpy as np

block\_length = 10000

snr\_db = np.arange(1, 15, 1)

ber\_results = []

for snr\_db\_value in snr\_db:

    snr = 10 \*\* (snr\_db\_value \* 0.1)

    sys = np.random.randint(0, 2, block\_length)

    noise = np.random.normal(0, 1, block\_length)

    transmitted\_bits = sys \* np.sqrt(snr)

    received\_bits = transmitted\_bits + noise

    decoded\_bits = (received\_bits>((snr)\*\*0.5/2))

    bit\_errors = np.sum(decoded\_bits != sys)

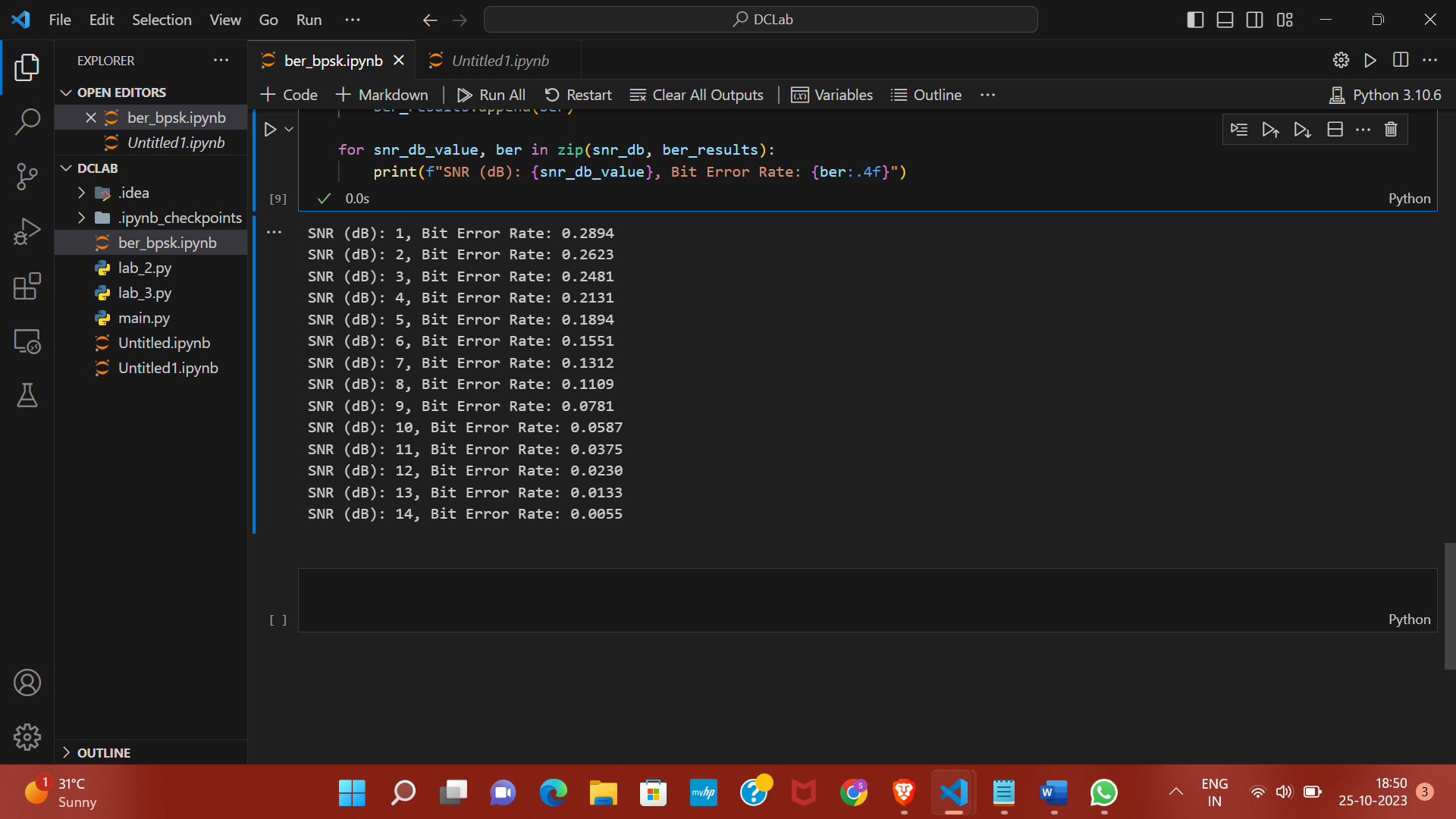
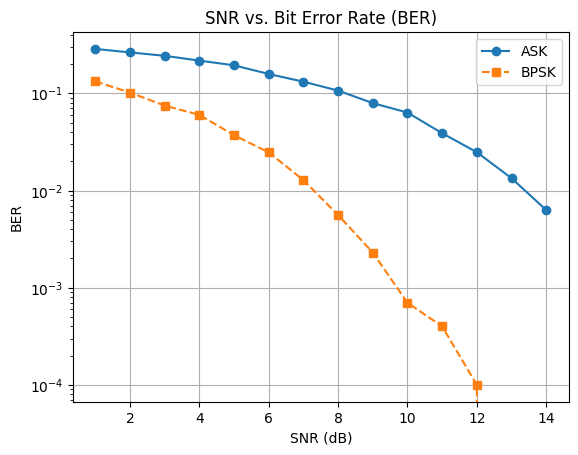
    ber = bit\_errors / block\_length

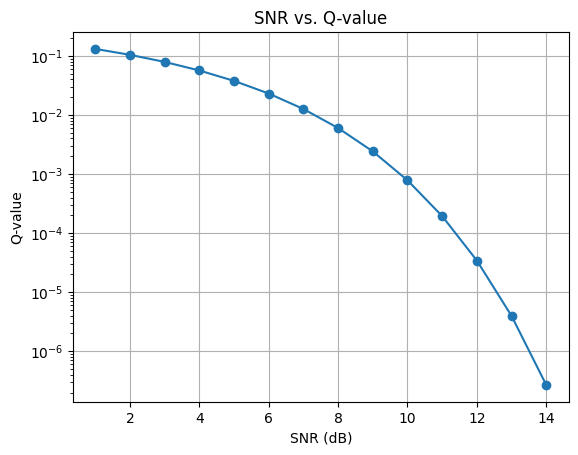
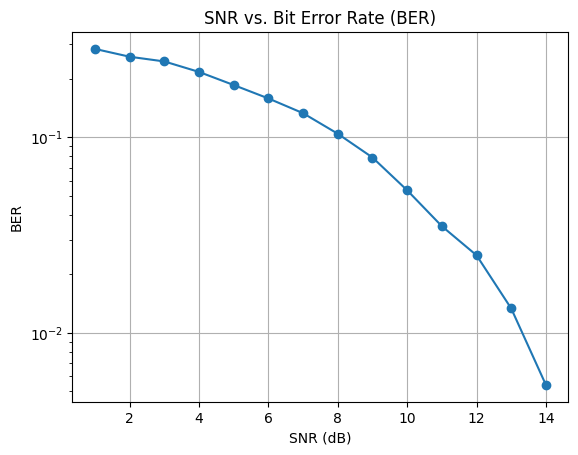
    ber\_results.append(ber)

for snr\_db\_value, ber in zip(snr\_db, ber\_results):

    print(f"SNR (dB): {snr\_db\_value}, Bit Error Rate: {ber:.4f}")

Output:

**Result:**

The Bit Error Rate (BER) performance of the binary communication system is analyzed under different Signal-to-Noise Ratios (SNR). As the SNR increases, the BER decreases, indicating a more reliable communication system. The plot visualizes this relationship between SNR and BER.

**EXPERIMENT- 5**

**Aim:** To analyze the Bit Error Rate (BER) performance of a Quadrature Phase Shift Keying (QPSK) binary communication system under different Signal-to-Noise Ratios (SNR).

**Equipment Required:**

1. Python environment with NumPy library for mathematical operations.
2. Matplotlib library for data visualization.

**Theory:** Quadrature Phase Shift Keying (QPSK) is a digital modulation technique that encodes two bits into each symbol by varying the phase of the carrier signal. In this practical, we'll simulate a binary QPSK communication system and evaluate its BER performance under various SNR conditions. The transmitted signal is affected by noise, and the received signal is processed to estimate the transmitted bits. BER is calculated as the ratio of the number of bits received incorrectly to the total number of transmitted bits.

**Algorithm:**

1. Import the NumPy library.
2. Define the block length, a range of SNR values, and an empty list for storing BER results.
3. Loop through each SNR value:
   * Calculate SNR from SNR in dB.
   * Generate a random binary signal **sys** of length **block\_length**.
   * Modulate the binary signal into QPSK symbols.
   * Generate random complex Gaussian noise.
   * Add noise to the transmitted QPSK symbols to simulate the channel.
   * Demodulate the received QPSK symbols by comparing the real part to zero.
   * Count the bit errors by comparing the decoded bits to the original **sys** signal.
   * Calculate BER as the ratio of bit errors to the block length.
   * Append the BER to the results list.
4. Display SNR and corresponding BER values.

This experiment simulates a binary QPSK communication system, varying the SNR to assess its performance under different noise conditions. The BER is calculated and presented for analysis.

PROGRAM:

ber\_results\_qpsk = []

for snr\_db\_value in snr\_db:

    snr = 10 \*\* (snr\_db\_value \* 0.1)

    sys\_real = 2 \* (np.random.randint(0, 2, block\_length) - 0.5)

    sys\_imag = 2 \* (np.random.randint(0, 2, block\_length) - 0.5)

    transmitted\_real = sys\_real \* np.sqrt(snr)

    transmitted\_imag = sys\_imag \* np.sqrt(snr)

    noise\_real = np.random.normal(0, 1, block\_length)

    noise\_imag = np.random.normal(0, 1, block\_length)

    received\_real = transmitted\_real + noise\_real

    received\_imag = transmitted\_imag + noise\_imag

    decoded\_real = (received\_real > 0).astype(int)

    decoded\_imag = (received\_imag > 0).astype(int)

    bit\_errors\_real = np.sum(decoded\_real != (sys\_real > 0).astype(int))

    ber\_real = bit\_errors\_real / block\_length

    bit\_errors\_imag = np.sum(decoded\_imag != (sys\_imag > 0).astype(int))

    ber\_imag = bit\_errors\_imag / block\_length

    ber\_qpsk = 0.5 \* (ber\_real + ber\_imag)

    ber\_results\_qpsk.append(ber\_qpsk)

SNR VALUES AND COMPARISON OF ALL ABOVE THREE SCHEMES:

from scipy.stats import norm

import numpy as np

import matplotlib.pyplot as plt

snr\_db = np.arange(1, 15, 1)

q\_values\_ask = []

q\_values\_bpsk = []

q\_values\_qpsk = []

for snr\_db\_value in snr\_db:

    snr = 10 \*\* (snr\_db\_value \* 0.1)

    q\_ask = 1 - norm.cdf(np.sqrt(snr))

    q\_bpsk = 1 - norm.cdf(np.sqrt(2 \* snr))

    q\_qpsk = 1 - (2\*norm.cdf(np.sqrt(2\*snr)) - (norm.cdf(np.sqrt(2\*snr)))\*\*2)

    q\_values\_ask.append(q\_ask)

    q\_values\_bpsk.append(q\_bpsk)

    q\_values\_qpsk.append(q\_qpsk)

plt.figure()

plt.semilogy(snr\_db, q\_values\_ask, marker='o', linestyle='-', label='ASK')

plt.semilogy(snr\_db, q\_values\_bpsk, marker='s', linestyle='--', label='BPSK')

plt.semilogy(snr\_db, q\_values\_qpsk, marker='o', linestyle='-.', label='QPSK')

plt.title('SNR vs. Q-value')

plt.xlabel('SNR (dB)')

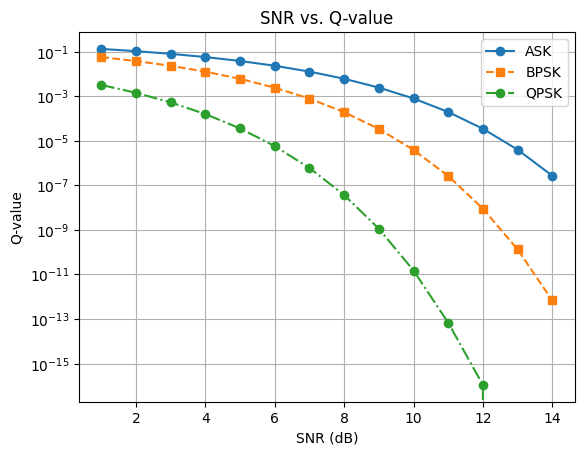
plt.ylabel('Q-value')

plt.grid(True)

plt.legend()

plt.show()

OUTPUT:



**EXPERIMENT-6**

**Aim:** The aim of this experiment is to assess the Bit Error Rate (BER) performance of M-ary Pulse Amplitude Modulation (M-ary PAM) and M-ary Quadrature Amplitude Modulation (M-ary QAM) binary communication systems under varying Signal-to-Noise Ratios (SNR).

**Theory:**  M-ary Pulse Amplitude Modulation (M-ary PAM) and M-ary Quadrature Amplitude Modulation (M-ary QAM) are digital modulation techniques widely used in communication systems. These modulation schemes encode multiple bits into each symbol, making them more bandwidth-efficient than simpler modulation techniques.

**Apparatus:** Numpy, Matplotlib.

**CODE FOR QAM:**

import numpy as np

from scipy.special import erfc

import matplotlib.pyplot as plt

modulation\_orders = [8, 4, 16, 64]

snr\_dB\_range = np.arange(0, 21, 1)

ber\_results = []

for M in modulation\_orders:

    ber\_modulation = []

    for snr\_dB in snr\_dB\_range:

        snr = 10 \*\* (snr\_dB / 10.0)

        pe = 4 \* (1 - 1 / np.sqrt(M)) \* erfc(np.sqrt(3 \* snr \* np.log2(M) / (M - 1)))

        ber\_modulation.append((1 - (1 - pe) \*\* np.log2(M)))

    ber\_results.append(ber\_modulation)

plt.figure()

for i, M in enumerate(modulation\_orders):

    plt.semilogy(snr\_dB\_range, ber\_results[i], marker='o', label=f'{M}-QAM')

plt.xlabel('SNR (dB)')

plt.ylabel('BER')

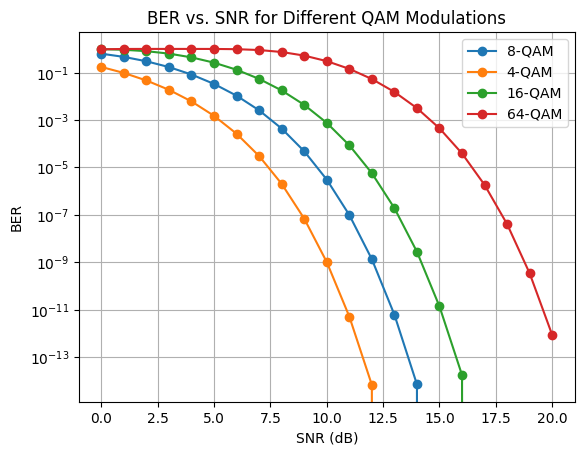
plt.title('BER vs. SNR for Different QAM Modulations')

plt.legend()

plt.grid()

plt.show()

**Output:**

****

**CODE FOR BOTH M-ARY PAM AND QAM**

import numpy as np

from scipy.special import erfc

import matplotlib.pyplot as plt

modulation\_orders = {

    'PAM': [4, 8, 16],

    'QAM': [4, 8, 16, 64]

}

snr\_dB\_range = np.arange(0, 21, 1)

ber\_results = {key: [] for key in modulation\_orders.keys()}

# Loop through each modulation scheme

for modulation\_type, modulation\_order\_list in modulation\_orders.items():

    for M in modulation\_order\_list:

        ber\_modulation = []

        for snr\_dB in snr\_dB\_range:

            snr = 10 \*\* (snr\_dB / 10.0)

            if modulation\_type == 'PAM':

                pe = 2 \* (1 - 1 / M) \* erfc(np.sqrt(6 \* snr \* np.log2(M) / (M \*\* 2 - 1)))

            elif modulation\_type == 'QAM':

                pe = 4 \* (1 - 1 / np.sqrt(M)) \* erfc(np.sqrt(3 \* snr \* np.log2(M) / (M - 1)))

ber\_modulation.append((1 - (1 - pe) \*\* np.log2(M)))

        ber\_results[modulation\_type].append(ber\_modulation)

plt.figure()

for modulation\_type, modulation\_order\_list in modulation\_orders.items():

    for i, M in enumerate(modulation\_order\_list):

        plt.semilogy(snr\_dB\_range, ber\_results[modulation\_type][i], marker='o', label=f'{M}-{modulation\_type}')

plt.xlabel('SNR (dB)')

plt.ylabel('BER')

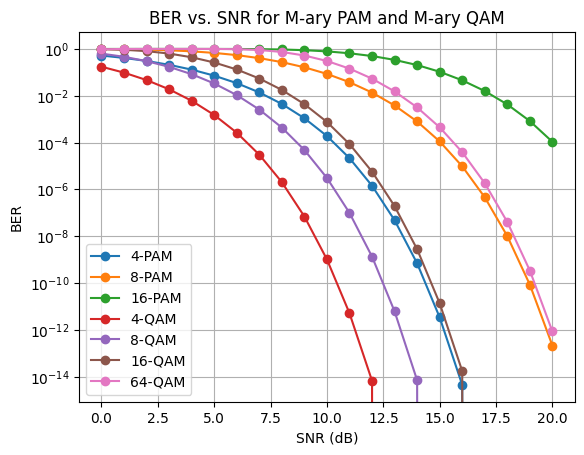
plt.title('BER vs. SNR for M-ary PAM and M-ary QAM')

plt.legend()

plt.grid()

plt.show()

**OUTPUT:**

****