**AIM: To plot unit impulse signal using Python.**

In signal processing, a unit impulse signal, also known as a Dirac delta function or impulse function, is a mathematical function that has the value of 1 at time zero and zero elsewhere. Generating a unit impulse signal in Python can be done using the NumPy library.

**Steps:**

* First, we import the necessary libraries: numpy for numerical computations and matplotlib.pyplot for plotting.
* Next, we define a function called unit\_impulse that takes two arguments: length (the length of the signal) and position (the position of the impulse within the signal).
* Inside the unit\_impulse function, we create an array of zeros with the specified length using np.zeros(length).
* We set the value at the specified position to 1, representing the impulse.
* We define the parameters **start**, **stop**, and **step** to specify the x-axis range and step size.
* We use np.arange to generate the x-axis values. The np.arange function creates an array of numbers from start to stop (inclusive) with a step size of step.
* We calculate the length of the impulse signal using len(x), which represents the number of x-axis values.
* We modify the unit\_impulse function call to calculate the position of the impulse based on the x-axis range. Since we want the impulse at n=0, we set the position to abs(start)//step, which calculates the index of 0 in the x-axis array.
* abs(start): The abs() function returns the absolute value of the start variable. This is done to ensure that the result is always positive, regardless of whether start is positive or negative.
* abs(start)//step: The // operator performs integer division, which discards the decimal part of the division result and returns the integer quotient. In this case, abs(start)//step calculates the number of steps from the start value to reach the position of 0 in the x-axis array.
* The unit\_impulse function returns the generated signal.
* By using abs(start)//step as the position argument in the unit\_impulse function call, we ensure that the impulse is correctly positioned at n=0 within the generated unit impulse signal.
* For example, if start = -10 and step = 1, the expression abs(start)//step evaluates to (10)//(1), which equals 10. This means that the impulse will be placed at index 10 in the signal, aligning it with n=0.
* Using abs(start)//step allows the code to handle both positive and negative start values correctly, ensuring that the unit impulse is positioned accurately regardless of the specified x-axis range and step size.
* We then define the desired length and position for the unit impulse signal.
* We generate the unit impulse signal by calling the unit\_impulse function with the specified parameters.
* Finally, we plot the unit impulse signal using plt.stem to create a stem plot, and then display the plot using plt.show().
* When you run this code, it will generate a stem plot showing the unit impulse signal with a spike at the specified position (in this case, position 10).
* plt.grid(True) in the code helps to improve the visual representation of the plot by adding gridlines, making it easier to analyze.
* Note: To run this code, you'll need to have the NumPy and Matplotlib libraries installed. You can install them using pip install numpy matplotlib.

**Program:**

**import numpy as np**

**import matplotlib.pyplot as plt**

**def unit\_impulse(length, position):**

**signal = np.zeros(length)**

**signal[position] = 1**

**return signal**

**# Parameters**

**start = -10 # Start value of the x-axis range**

**stop = 10 # Stop value of the x-axis range**

**step = 1 # Step size**

**# Generate x-axis values**

**x = np.arange(start, stop+step, step)**

**# Generate unit impulse signal**

**impulse\_signal = unit\_impulse(len(x), abs(start)//step)**

**# Plot the signal**

**plt.stem(x, impulse\_signal)**

**plt.xlabel('Time')**

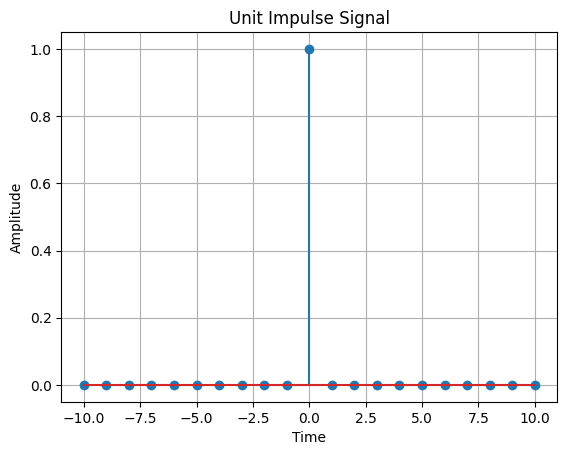
**plt.ylabel('Amplitude')**

**plt.title('Unit Impulse Signal')**

**plt.grid(True)**

**plt.show()**

**Output:**

****

**AIM**

Simulate impulse train

**Theory:**

An impulse train, also known as a Dirac comb, is a periodic sequence of impulses. Each impulse has an amplitude of 1 and occurs at regular intervals. The impulse train is often used in signal processing to model periodic phenomena or to analyze the frequency response of systems.

The impulse train is defined as follows:

impulse\_train(n, period) = 1, if n % period == 0

= 0, otherwise

where n is the sample index and period is the period of the impulse train.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Define the parameters for the impulse train: signal length and period.

Create an empty array to store the impulse train.

Iterate over each sample index.

Check if the sample index is a multiple of the period. If yes, assign the value 1 to the impulse train array at that index; otherwise, assign the value 0.

Plot and display the impulse train.

Save the impulse train array (optional).

Now, let's see the Python program that simulates an impulse train:

**Program**

import numpy as np

import matplotlib.pyplot as plt

def simulate\_impulse\_train(signal\_length, period):

impulse\_train = np.zeros(signal\_length)

for n in range(signal\_length):

if n % period == 0:

impulse\_train[n] = 1

return impulse\_train

# Define the parameters for the impulse train

signal\_length = 100 # Length of the impulse train

period = 10 # Period of the impulse train

# Simulate the impulse train

impulse\_train = simulate\_impulse\_train(signal\_length, period)

# Plot and display the impulse train

plt.stem(impulse\_train)

plt.title('Impulse Train')

plt.xlabel('Sample')

plt.ylabel('Amplitude')

plt.show()

# Save the impulse train array (optional)

# np.savetxt('impulse\_train.txt', impulse\_train, delimiter=',')

**Explanation:**

We import the required modules, including numpy for array operations and matplotlib.pyplot for plotting.

The simulate\_impulse\_train function takes the signal length and period as input and returns an array representing the impulse train.

In the main part of the program, we define the parameters for the impulse train: signal\_length (the length of the impulse train) and period (the period of the impulse train).

We simulate the impulse train by calling the simulate\_impulse\_train function with the specified parameters.

We plot and display the impulse train using matplotlib.pyplot.stem.

The impulse train array can be optionally saved to a file using numpy.savetxt.

You can modify the signal\_length and period variables to change the length and period of the impulse train, respectively.

Uncomment the np.savetxt line if you want to save the impulse train array to a text file.

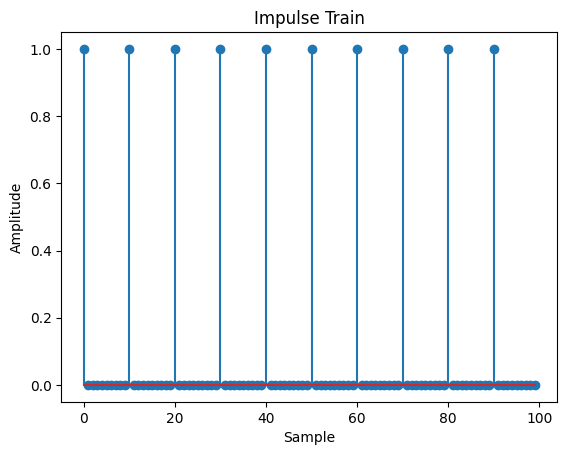
Make sure you have numpy and matplotlib installed (pip install numpy matplotlib) before running this program.

**Expected Output:**

A stem plot will be displayed showing the impulse train.

The impulse train array will be saved to a text file (optional).

Feel free to experiment with different parameters for the impulse train and observe the output.



**AIM:** Write a python program to Simulate continuous and discrete unit step signal,

**Theory:**

A unit step signal, also known as a Heaviside step function, is a signal that is 0 for negative time or indices and 1 for non-negative time or indices. It is commonly used to represent a sudden change or transition in a system.

The continuous unit step function is defined as:

u(t) = 0, t < 0

= 1, t >= 0

The discrete unit step function is defined as:

u[n] = 0, n < 0

= 1, n >= 0

**Flowchart:**

The flowchart for the program will consist of the following steps:

Define the time range (for the continuous signal) or the number of samples (for the discrete signal).

Create an empty array to store the unit step signal.

Iterate over each time or sample index.

Set the value of the unit step signal to 0 for negative time or indices, and 1 for non-negative time or indices.

Plot and display the unit step signal.

Save the unit step signal array (optional).

Now, let's see the Python program that simulates continuous and discrete unit step signals:

**Program:**

import numpy as np

import matplotlib.pyplot as plt

def simulate\_continuous\_unit\_step(time):

unit\_step = np.zeros\_like(time)

unit\_step[time >= 0] = 1

return unit\_step

def simulate\_discrete\_unit\_step(num\_samples):

unit\_step = np.zeros(num\_samples)

unit\_step[num\_samples // 2:] = 1

return unit\_step

# Define the time range for the continuous unit step signal

time = np.linspace(-5, 5, 1000) # Time range from -5 to 5

# Simulate the continuous unit step signal

continuous\_unit\_step = simulate\_continuous\_unit\_step(time)

# Define the number of samples for the discrete unit step signal

num\_samples = 20 # Number of samples

# Simulate the discrete unit step signal

discrete\_unit\_step = simulate\_discrete\_unit\_step(num\_samples)

# Plot and display the continuous and discrete unit step signals

plt.figure(figsize=(10, 6))

plt.subplot(2, 1, 1)

plt.plot(time, continuous\_unit\_step)

plt.title('Continuous Unit Step Signal')

plt.xlabel('Time')

plt.ylabel('Amplitude')

plt.subplot(2, 1, 2)

plt.stem(discrete\_unit\_step)

plt.title('Discrete Unit Step Signal')

plt.xlabel('Sample')

plt.ylabel('Amplitude')

plt.tight\_layout()

plt.show()

# Save the unit step signal arrays (optional)

# np.savetxt('continuous\_unit\_step.txt', continuous\_unit\_step, delimiter=',')

# np.savetxt('discrete\_unit\_step.txt', discrete\_unit\_step, delimiter=',')

**Explanation:**

We import the required modules, including numpy for array operations and matplotlib.pyplot for plotting.

The simulate\_continuous\_unit\_step function takes a time array as input and returns an array representing the continuous unit step signal. It creates an array of zeros with the same shape as the input time array and sets the values to 1 for non-negative time values.

The simulate\_discrete\_unit\_step function takes the number of samples as input and returns an array representing the discrete unit step signal. It creates an array of zeros with the specified number of samples and sets the values to 1 starting from the middle index.

In the main part of the program, we define the time range for the continuous unit step signal using numpy.linspace.

We simulate the continuous unit step signal by calling the simulate\_continuous\_unit\_step function with the time array.

We define the number of samples for the discrete unit step signal.

We simulate the discrete unit step signal by calling the simulate\_discrete\_unit\_step function with the number of samples.

We plot and display the continuous and discrete unit step signals using matplotlib.pyplot.plot and matplotlib.pyplot.stem, respectively.

The unit step signal arrays can be optionally saved to text files using numpy.savetxt.

You can modify the time range for the continuous unit step signal and the num\_samples for the discrete unit step signal according to your requirements.

Uncomment the np.savetxt lines if you want to save the unit step signal arrays to text files.

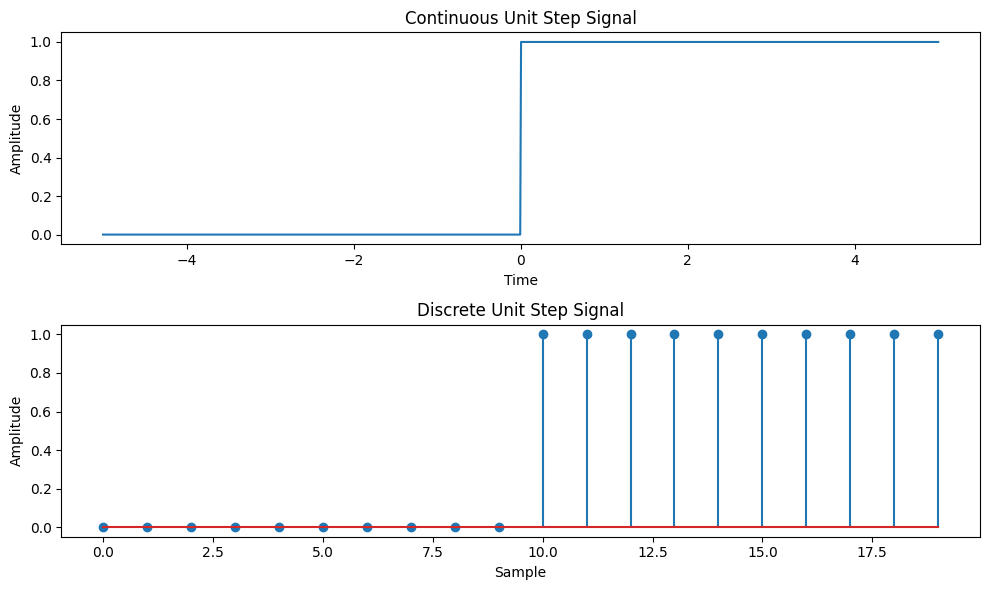
Make sure you have numpy and matplotlib installed (pip install numpy matplotlib) before running this program.

**Expected Output:**

Two plots will be displayed: the continuous unit step signal and the discrete unit step signal.

The unit step signal arrays will be saved to text files (optional).

Feel free to experiment with different time ranges and numbers of samples to generate unit step signals of various lengths.



**AIM:** Write a python program to Simulate continuous and discrete ramp signal,

**Theory:**

A ramp signal is a signal that varies linearly with time. It starts at an initial value and increases or decreases at a constant rate. Ramp signals are commonly used in various applications, such as signal processing, control systems, and digital communications.

The continuous ramp signal is defined as:

ramp(t) = a \* t, for t >= 0

= 0, for t < 0

where a is the slope of the ramp.

The discrete ramp signal is defined as:

ramp[n] = a \* n, for n >= 0

= 0, for n < 0

where a is the slope of the ramp.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Define the time range (for the continuous signal) or the number of samples (for the discrete signal).

Create an empty array to store the ramp signal.

Iterate over each time or sample index.

Set the value of the ramp signal to a \* t for continuous or a \* n for discrete, where a is the slope and t or n is the time or sample index.

Plot and display the ramp signal.

Save the ramp signal array (optional).

Now, let's see the Python program that simulates continuous and discrete ramp signals:

**Program**

import numpy as np

import matplotlib.pyplot as plt

def simulate\_continuous\_ramp(time, slope):

ramp = np.zeros\_like(time)

ramp[time >= 0] = slope \* time[time >= 0]

return ramp

def simulate\_discrete\_ramp(num\_samples, slope):

ramp = np.zeros(num\_samples)

ramp[num\_samples // 2:] = slope \* np.arange(num\_samples // 2, num\_samples)

return ramp

# Define the time range for the continuous ramp signal

time = np.linspace(-5, 5, 1000) # Time range from -5 to 5

# Define the number of samples and slope for the discrete ramp signal

num\_samples = 20 # Number of samples

slope = 2 # Slope of the ramp

# Simulate the continuous ramp signal

continuous\_ramp = simulate\_continuous\_ramp(time, slope)

# Simulate the discrete ramp signal

discrete\_ramp = simulate\_discrete\_ramp(num\_samples, slope)

# Plot and display the continuous and discrete ramp signals

plt.figure(figsize=(10, 6))

plt.subplot(2, 1, 1)

plt.plot(time, continuous\_ramp)

plt.title('Continuous Ramp Signal')

plt.xlabel('Time')

plt.ylabel('Amplitude')

plt.subplot(2, 1, 2)

plt.stem(discrete\_ramp)

plt.title('Discrete Ramp Signal')

plt.xlabel('Sample')

plt.ylabel('Amplitude')

plt.tight\_layout()

plt.show()

# Save the ramp signal arrays (optional)

# np.savetxt('continuous\_ramp.txt', continuous\_ramp, delimiter=',')

# np.savetxt('discrete\_ramp.txt', discrete\_ramp, delimiter=',')

**Explanation:**

We import the required modules, including numpy for array operations and matplotlib.pyplot for plotting.

The simulate\_continuous\_ramp function takes a time array and a slope as input and returns an array representing the continuous ramp signal. It creates an array of zeros with the same shape as the input time array and sets the values to slope \* time for non-negative time values.

The simulate\_discrete\_ramp function takes the number of samples and a slope as input and returns an array representing the discrete ramp signal. It creates an array of zeros with the specified number of samples and sets the values to slope \* n starting from the middle index.

In the main part of the program, we define the time range for the continuous ramp signal using numpy.linspace.

We define the number of samples and slope for the discrete ramp signal.

We simulate the continuous ramp signal by calling the simulate\_continuous\_ramp function with the time array and slope.

We simulate the discrete ramp signal by calling the simulate\_discrete\_ramp function with the number of samples and slope.

We plot and display the continuous and discrete ramp signals using matplotlib.pyplot.plot and matplotlib.pyplot.stem, respectively.

The ramp signal arrays can be optionally saved to text files using numpy.savetxt.

You can modify the time range for the continuous ramp signal, the num\_samples for the discrete ramp signal, and the slope of the ramps according to your requirements.

Uncomment the np.savetxt lines if you want to save the ramp signal arrays to text files.

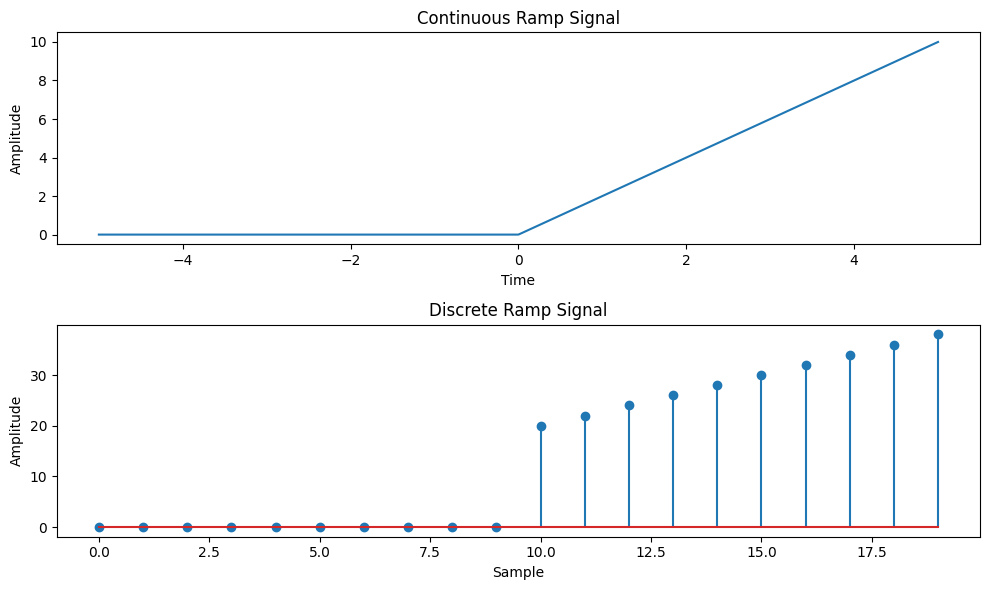
Make sure you have numpy and matplotlib installed (pip install numpy matplotlib) before running this program.

**Expected Output:**

Two plots will be displayed: the continuous ramp signal and the discrete ramp signal.

The ramp signal arrays will be saved to text files (optional).

Feel free to experiment with different time ranges, numbers of samples, and slopes to generate ramp signals of various lengths and slopes.



**AIM:** Write a python program to Simulate continuous and discrete exponential signal,

**Theory:**

An exponential signal is a signal whose amplitude changes exponentially with time or index. Exponential signals can exhibit either growth or decay behavior, depending on the sign of the exponential term.

The continuous exponential signal is defined as:

x(t) = A \* exp(b \* t)

where A is the initial amplitude, b is the exponential coefficient, and t is the time.

The discrete exponential signal is defined as:

x[n] = A \* exp(b \* n)

where A is the initial amplitude, b is the exponential coefficient, and n is the sample index.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Define the time range (for the continuous signal) or the number of samples (for the discrete signal).

Create an empty array to store the exponential signal.

Iterate over each time or sample index.

Set the value of the exponential signal to A \* exp(b \* t) for continuous or A \* exp(b \* n) for discrete, where A is the initial amplitude, b is the exponential coefficient, and t or n is the time or sample index.

Plot and display the exponential signal.

Save the exponential signal array (optional).

Now, let's see the Python program that simulates continuous and discrete exponential signals:

**Program**

import numpy as np

import matplotlib.pyplot as plt

def simulate\_continuous\_exponential(time, amplitude, coefficient):

exponential\_signal = amplitude \* np.exp(coefficient \* time)

return exponential\_signal

def simulate\_discrete\_exponential(num\_samples, amplitude, coefficient):

exponential\_signal = amplitude \* np.exp(coefficient \* np.arange(num\_samples))

return exponential\_signal

# Define the time range for the continuous exponential signal

time = np.linspace(0, 5, 1000) # Time range from 0 to 5

# Define the number of samples, initial amplitude, and coefficient for the discrete exponential signal

num\_samples = 20 # Number of samples

amplitude = 2 # Initial amplitude

coefficient = -0.5 # Exponential coefficient

# Simulate the continuous exponential signal

continuous\_exponential = simulate\_continuous\_exponential(time, amplitude, coefficient)

# Simulate the discrete exponential signal

discrete\_exponential = simulate\_discrete\_exponential(num\_samples, amplitude, coefficient)

# Plot and display the continuous and discrete exponential signals

plt.figure(figsize=(10, 6))

plt.subplot(2, 1, 1)

plt.plot(time, continuous\_exponential)

plt.title('Continuous Exponential Signal')

plt.xlabel('Time')

plt.ylabel('Amplitude')

plt.subplot(2, 1, 2)

plt.stem(discrete\_exponential)

plt.title('Discrete Exponential Signal')

plt.xlabel('Sample')

plt.ylabel('Amplitude')

plt.tight\_layout()

plt.show()

# Save the exponential signal arrays (optional)

# np.savetxt('continuous\_exponential.txt', continuous\_exponential, delimiter=',')

# np.savetxt('discrete\_exponential.txt', discrete\_exponential, delimiter=',')

**Explanation:**

We import the required modules, including numpy for array operations and matplotlib.pyplot for plotting.

The simulate\_continuous\_exponential function takes a time array, initial amplitude, and exponential coefficient as input and returns an array representing the continuous exponential signal. It calculates the exponential signal using numpy.exp with the specified amplitude and coefficient.

The simulate\_discrete\_exponential function takes the number of samples, initial amplitude, and exponential coefficient as input and returns an array representing the discrete exponential signal. It calculates the exponential signal using numpy.exp with the specified amplitude and coefficient, applied element-wise to the range of sample indices.

In the main part of the program, we define the time range for the continuous exponential signal using numpy.linspace.

We define the number of samples, initial amplitude, and exponential coefficient for the discrete exponential signal.

We simulate the continuous exponential signal by calling the simulate\_continuous\_exponential function with the time array, amplitude, and coefficient.

We simulate the discrete exponential signal by calling the simulate\_discrete\_exponential function with the number of samples, amplitude, and coefficient.

We plot and display the continuous and discrete exponential signals using matplotlib.pyplot.plot and matplotlib.pyplot.stem, respectively.

The exponential signal arrays can be optionally saved to text files using numpy.savetxt.

You can modify the time range for the continuous exponential signal, the number of samples, initial amplitude, and exponential coefficient for the discrete exponential signal according to your requirements.

Uncomment the np.savetxt lines if you want to save the exponential signal arrays to text files.

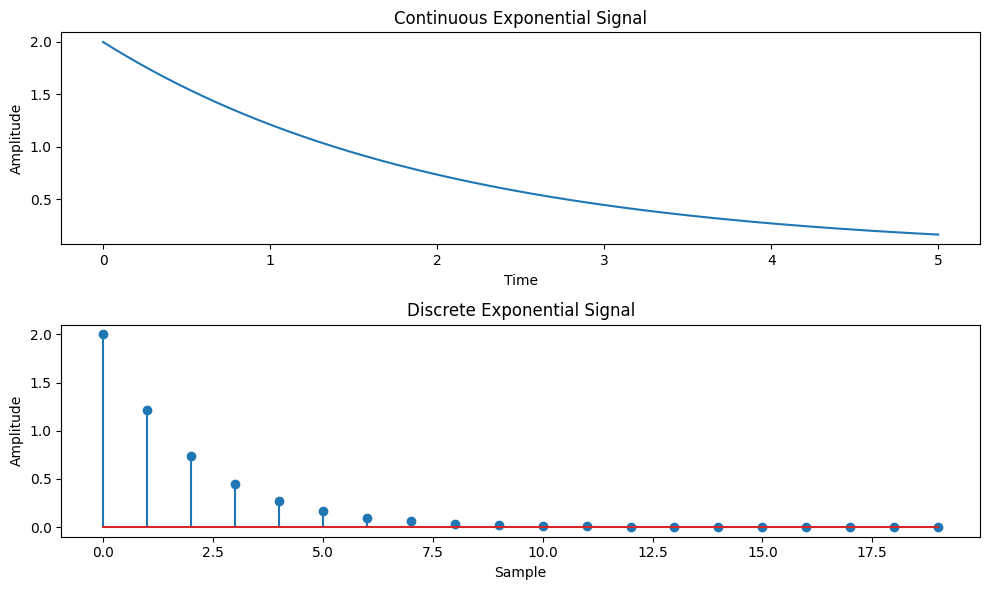
Make sure you have numpy and matplotlib installed (pip install numpy matplotlib) before running this program.

**Expected Output:**

Two plots will be displayed: the continuous exponential signal and the discrete exponential signal.

The exponential signal arrays will be saved to text files (optional).

Feel free to experiment with different time ranges, numbers of samples, initial amplitudes, and exponential coefficients to generate exponential signals of various shapes and characteristics.



**AIM**: Write a python program to Simulate continuous and discrete parabolic signal

**Theory:**

A parabolic signal is a signal whose amplitude changes quadratically with time or index. It follows a parabolic curve and is commonly used to model various phenomena, such as projectile motion, signal processing, and data fitting.

The continuous parabolic signal is defined as:

parabolic(t) = a \* t^2 + b \* t + c

where a, b, and c are coefficients and t is the time.

The discrete parabolic signal is defined as:

parabolic[n] = a \* n^2 + b \* n + c

where a, b, and c are coefficients and n is the sample index.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Define the time range (for the continuous signal) or the number of samples (for the discrete signal).

Create an empty array to store the parabolic signal.

Iterate over each time or sample index.

Set the value of the parabolic signal to a \* t^2 + b \* t + c for continuous or a \* n^2 + b \* n + c for discrete, where a, b, and c are the coefficients and t or n is the time or sample index.

Plot and display the parabolic signal.

Save the parabolic signal array (optional).

Now, let's see the Python program that simulates continuous and discrete parabolic signals:

**Program:**

import numpy as np

import matplotlib.pyplot as plt

def simulate\_continuous\_parabolic(time, coefficients):

parabolic\_signal = np.polyval(coefficients, time)

return parabolic\_signal

def simulate\_discrete\_parabolic(num\_samples, coefficients):

parabolic\_signal = np.polyval(coefficients, np.arange(num\_samples))

return parabolic\_signal

# Define the time range for the continuous parabolic signal

time = np.linspace(-5, 5, 1000) # Time range from -5 to 5

# Define the number of samples and coefficients for the discrete parabolic signal

num\_samples = 20 # Number of samples

coefficients = [1, 2, 1] # Coefficients of the parabolic signal

# Simulate the continuous parabolic signal

continuous\_parabolic = simulate\_continuous\_parabolic(time, coefficients)

# Simulate the discrete parabolic signal

discrete\_parabolic = simulate\_discrete\_parabolic(num\_samples, coefficients)

# Plot and display the continuous and discrete parabolic signals

plt.figure(figsize=(10, 6))

plt.subplot(2, 1, 1)

plt.plot(time, continuous\_parabolic)

plt.title('Continuous Parabolic Signal')

plt.xlabel('Time')

plt.ylabel('Amplitude')

plt.subplot(2, 1, 2)

plt.stem(discrete\_parabolic)

plt.title('Discrete Parabolic Signal')

plt.xlabel('Sample')

plt.ylabel('Amplitude')

plt.tight\_layout()

plt.show()

# Save the parabolic signal arrays (optional)

# np.savetxt('continuous\_parabolic.txt', continuous\_parabolic, delimiter=',')

# np.savetxt('discrete\_parabolic.txt', discrete\_parabolic, delimiter=',')

**Explanation:**

We import the required modules, including numpy for array operations and matplotlib.pyplot for plotting.

The simulate\_continuous\_parabolic function takes a time array and a list of coefficients as input and returns an array representing the continuous parabolic signal. It calculates the parabolic signal using numpy.polyval with the specified coefficients and time array.

The simulate\_discrete\_parabolic function takes the number of samples and a list of coefficients as input and returns an array representing the discrete parabolic signal. It calculates the parabolic signal using numpy.polyval with the specified coefficients and a range of sample indices.

In the main part of the program, we define the time range for the continuous parabolic signal using numpy.linspace.

We define the number of samples and coefficients for the discrete parabolic signal.

We simulate the continuous parabolic signal by calling the simulate\_continuous\_parabolic function with the time array and coefficients.

We simulate the discrete parabolic signal by calling the simulate\_discrete\_parabolic function with the number of samples and coefficients.

We plot and display the continuous and discrete parabolic signals using matplotlib.pyplot.plot and matplotlib.pyplot.stem, respectively.

The parabolic signal arrays can be optionally saved to text files using numpy.savetxt.

You can modify the time range for the continuous parabolic signal, the number of samples, and the coefficients for the discrete parabolic signal according to your requirements.

Uncomment the np.savetxt lines if you want to save the parabolic signal arrays to text files.

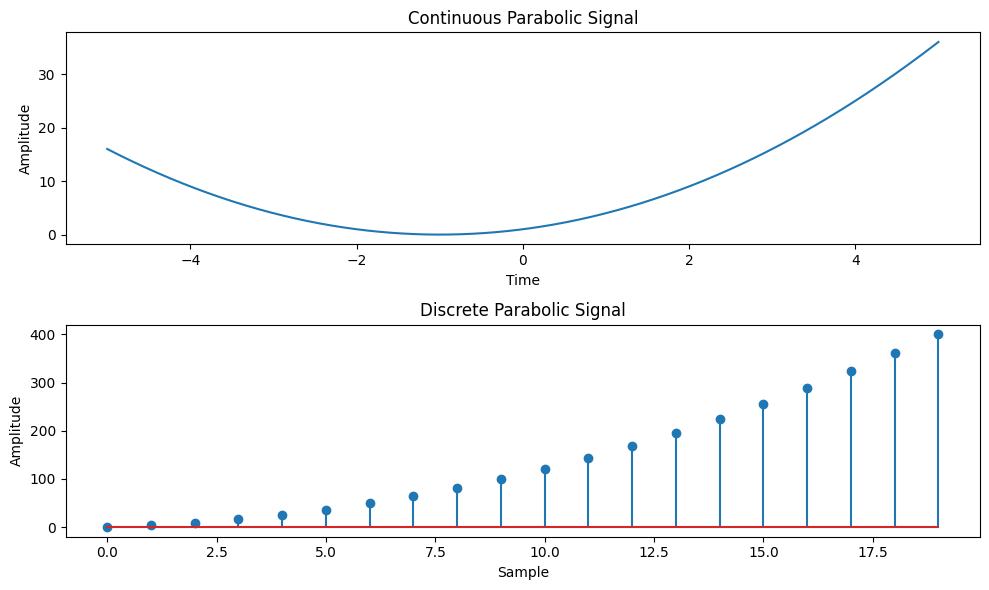
Make sure you have numpy and matplotlib installed (pip install numpy matplotlib) before running this program.

**Expected Output:**

Two plots will be displayed: the continuous parabolic signal and the discrete parabolic signal.

The parabolic signal arrays will be saved to text files (optional).

Feel free to experiment with different time ranges, numbers of samples, and coefficients to generate parabolic signals of various shapes and characteristics.



**AIM:** **Write a python program to Simulate continuous and discrete sine wave signal**

**Theory:**

A sine wave signal is a continuous-time or discrete-time signal that follows a periodic oscillation described by the sine function. It is widely used in various applications, such as signal processing, communications, and audio synthesis.

The continuous sine wave signal is defined as:

sine(t) = A \* sin(2 \* pi \* f \* t + phi)

where A is the amplitude, f is the frequency, t is the time, and phi is the phase.

The discrete sine wave signal is defined as:

sine[n] = A \* sin(2 \* pi \* f \* n / Fs + phi)

where A is the amplitude, f is the frequency, n is the sample index, Fs is the sampling frequency, and phi is the phase.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Define the time range (for the continuous signal) or the number of samples and the sampling frequency (for the discrete signal).

Create an empty array to store the sine wave signal.

Iterate over each time or sample index.

Set the value of the sine wave signal to A \* sin(2 \* pi \* f \* t + phi) for continuous or A \* sin(2 \* pi \* f \* n / Fs + phi) for discrete, where A, f, t, n, Fs, and phi are the corresponding parameters.

Plot and display the sine wave signal.

Save the sine wave signal array (optional).

Now, let's see the Python program that simulates continuous and discrete sine wave signals:

**Program:**

import numpy as np

import matplotlib.pyplot as plt

def simulate\_continuous\_sine\_wave(time, amplitude, frequency, phase):

sine\_wave = amplitude \* np.sin(2 \* np.pi \* frequency \* time + phase)

return sine\_wave

def simulate\_discrete\_sine\_wave(num\_samples, sampling\_frequency, amplitude, frequency, phase):

time = np.arange(num\_samples) / sampling\_frequency

sine\_wave = amplitude \* np.sin(2 \* np.pi \* frequency \* time + phase)

return sine\_wave

# Define the time range for the continuous sine wave signal

time = np.linspace(0, 1, 1000) # Time range from 0 to 1 second

# Define the number of samples, sampling frequency, and parameters for the discrete sine wave signal

num\_samples = 100 # Number of samples

sampling\_frequency = 10 # Sampling frequency in Hz

amplitude = 1 # Amplitude of the sine wave

frequency = 2 # Frequency of the sine wave in Hz

phase = 0 # Phase angle of the sine wave in radians

# Simulate the continuous sine wave signal

continuous\_sine\_wave = simulate\_continuous\_sine\_wave(time, amplitude, frequency, phase)

# Simulate the discrete sine wave signal

discrete\_sine\_wave = simulate\_discrete\_sine\_wave(num\_samples, sampling\_frequency, amplitude, frequency, phase)

# Plot and display the continuous and discrete sine wave signals

plt.figure(figsize=(10, 6))

plt.subplot(2, 1, 1)

plt.plot(time, continuous\_sine\_wave)

plt.title('Continuous Sine Wave Signal')

plt.xlabel('Time (s)')

plt.ylabel('Amplitude')

plt.subplot(2, 1, 2)

plt.stem(discrete\_sine\_wave)

plt.title('Discrete Sine Wave Signal')

plt.xlabel('Sample')

plt.ylabel('Amplitude')

plt.tight\_layout()

plt.show()

# Save the sine wave signal arrays (optional)

# np.savetxt('continuous\_sine\_wave.txt', continuous\_sine\_wave, delimiter=',')

# np.savetxt('discrete\_sine\_wave.txt', discrete\_sine\_wave, delimiter=',')

**Explanation:**

We import the required modules, including numpy for array operations and matplotlib.pyplot for plotting.

The simulate\_continuous\_sine\_wave function takes a time array, amplitude, frequency, and phase as input and returns an array representing the continuous sine wave signal. It calculates the sine wave signal using numpy.sin with the appropriate formula.

The simulate\_discrete\_sine\_wave function takes the number of samples, sampling frequency, amplitude, frequency, and phase as input and returns an array representing the discrete sine wave signal. It first calculates the time array using the number of samples and sampling frequency and then calculates the sine wave signal using numpy.sin with the appropriate formula.

In the main part of the program, we define the time range for the continuous sine wave signal using numpy.linspace.

We define the number of samples, sampling frequency, amplitude, frequency, and phase for the discrete sine wave signal.

We simulate the continuous sine wave signal by calling the simulate\_continuous\_sine\_wave function with the time array and parameters.

We simulate the discrete sine wave signal by calling the simulate\_discrete\_sine\_wave function with the number of samples, sampling frequency, amplitude, frequency, and phase.

We plot and display the continuous and discrete sine wave signals using matplotlib.pyplot.plot and matplotlib.pyplot.stem, respectively.

The sine wave signal arrays can be optionally saved to text files using numpy.savetxt.

You can modify the time range for the continuous sine wave signal, the number of samples, sampling frequency, amplitude, frequency, and phase for the discrete sine wave signal according to your requirements.

Uncomment the np.savetxt lines if you want to save the sine wave signal arrays to text files.

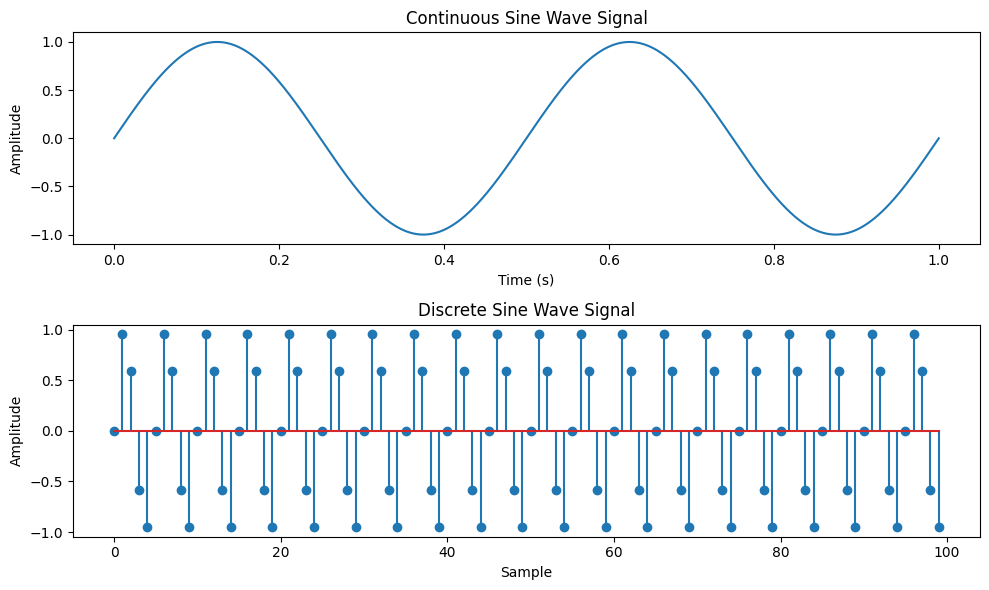
Make sure you have numpy and matplotlib installed (pip install numpy matplotlib) before running this program.

**Expected Output:**

Two plots will be displayed: the continuous sine wave signal and the discrete sine wave signal.

The sine wave signal arrays will be saved to text files (optional).

Feel free to experiment with different time ranges, numbers of samples, sampling



**AIM: Write a python program to simulate y(t)=u(t)+u(t-1)+3u(t+5).**

**Theory:**

The unit step function, denoted as u(t), is a function that is 0 for negative time or indices and 1 for non-negative time or indices. It is commonly used to represent a sudden change or transition in a system.

The function y(t) = u(t) + u(t-1) + 3\*u(t+5) is a combination of three unit step functions. It has a value of 1 at t=0, t=1, and t=-5, and a value of 0 elsewhere.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Define the time range.

Create an empty array to store the function values.

Iterate over each time index.

Calculate the value of y(t) at each time index using the given function formula.

Plot and display the function values.

**Program**

Now, let's see the Python program that simulates y(t) = u(t) + u(t-1) + 3\*u(t+5):

import numpy as np

import matplotlib.pyplot as plt

def simulate\_function(time):

y = np.zeros\_like(time)

y[time >= 0] = 1

y[time >= 1] += 1

y[time >= -5] += 3

return y

# Define the time range

time = np.linspace(-10, 10, 1000)

# Simulate the function

function\_values = simulate\_function(time)

# Plot and display the function

plt.plot(time, function\_values)

plt.title('Function y(t) = u(t) + u(t-1) + 3\*u(t+5)')

plt.xlabel('Time')

plt.ylabel('Amplitude')

plt.ylim([-0.5, 5.5])

plt.grid(True)

plt.show()

**Explanation:**

We import the required modules, including numpy for array operations and matplotlib.pyplot for plotting.

The simulate\_function function takes a time array as input and returns an array representing the function values. It creates an array of zeros with the same shape as the input time array and sets the values according to the function formula.

In the main part of the program, we define the time range using numpy.linspace.

We simulate the function by calling the simulate\_function function with the time array.

We plot and display the function values using matplotlib.pyplot.plot.

We set the title, x-axis label, y-axis label, y-axis limits, and enable the grid.

Finally, we show the plot using matplotlib.pyplot.show.

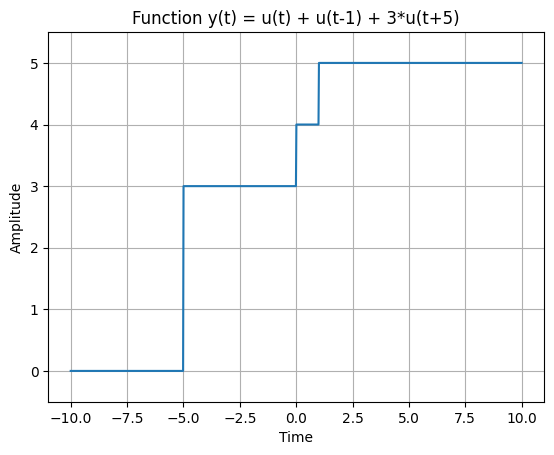
You can modify the time range according to your requirements.

Make sure you have numpy and matplotlib installed (pip install numpy matplotlib) before running this program.

**Expected Output:**

A plot will be displayed showing the function y(t) = u(t) + u(t-1) + 3\*u(t+5).

The function will have a value of 1 at t=0, t=1, and t=-5, and a value of 0 elsewhere.



**AIM: Write a python program to simulate y(t)=Delta(t)+delta(t-1)+3\*delta(t+5).**

**Theory:**

The impulse function, denoted as Delta(t), is a function that is infinite at t=0 and zero elsewhere. It is often used to model instantaneous events.

The shifted impulse function, denoted as delta(t), is similar to the impulse function but is shifted by a certain amount. It is non-zero only at the shifted time.

The function y(t) = Delta(t) + delta(t-1) + 3\*delta(t+5) is a combination of three impulse functions. It has an impulse at t=0, t=1, and t=-5, with different magnitudes.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Define the time range.

Create an empty array to store the function values.

Iterate over each time index.

Calculate the value of y(t) at each time index using the given function formula.

Plot and display the function values.

**Now, let's see the Python program that simulates y(t) = Delta(t) + delta(t-1) + 3\*delta(t+5):**

**import numpy as np**

**import matplotlib.pyplot as plt**

**def simulate\_function(time):**

**y = np.zeros\_like(time)**

**y[time == 0] = 1**

**y[time == 1] += 1**

**y[time == -5] += 3**

**return y**

**# Define the time range**

**time = np.arange(-10, 11)**

**# Simulate the function**

**function\_values = simulate\_function(time)**

**# Plot and display the function**

**plt.stem(time, function\_values)**

**plt.title('Function y(t) = Delta(t) + delta(t-1) + 3\*delta(t+5)')**

**plt.xlabel('Time')**

**plt.ylabel('Amplitude')**

**plt.ylim([-0.5, 4.5])**

**plt.grid(True)**

**plt.show()**

**Explanation:**

We import the required modules, including numpy for array operations and matplotlib.pyplot for plotting.

The simulate\_function function takes a time array as input and returns an array representing the function values. It creates an array of zeros with the same shape as the input time array and sets the values according to the function formula.

In the main part of the program, we define the time range using numpy.arange.

We simulate the function by calling the simulate\_function function with the time array.

We plot and display the function values using matplotlib.pyplot.stem.

We set the title, x-axis label, y-axis label, y-axis limits, and enable the grid.

Finally, we show the plot using matplotlib.pyplot.show.

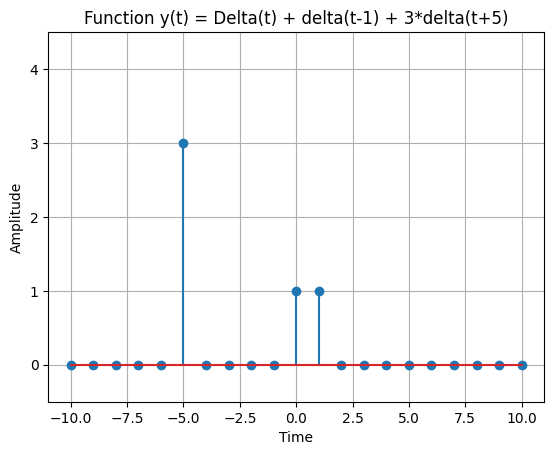
You can modify the time range according to your requirements.

Make sure you have numpy and matplotlib installed (pip install numpy matplotlib) before running this program.

**Expected Output:**

A stem plot will be displayed showing the function y(t) = Delta(t) + delta(t-1) + 3\*delta(t+5).

The function will have impulses at t=0, t=1, and t=-5, with magnitudes of 1, 1, and 3 respectively, and zero elsewhere.



**Experiment-1**

**AIM:** Simulate linear convolution and circular convolution on discrete time signals.

**Theory:**

Linear convolution and circular convolution are mathematical operations used to combine two signals to obtain a third signal. They are widely used in various applications, such as signal processing, image processing, and audio processing.

Linear convolution calculates the sum of element-wise products of two signals, considering the full range of valid indices. It is typically used for finite-length signals and can produce an output signal that is longer than the input signals.

Circular convolution, on the other hand, calculates the sum of element-wise products of two signals, considering a periodic extension of the input signals. It is commonly used for periodic or infinite-length signals and produces an output signal with the same length as the input signals.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Define two discrete-time signals (signal 1 and signal 2).

Compute the linear convolution of the two signals using numpy.convolve.

Compute the circular convolution of the two signals using numpy.fft.ifft.

Plot and display the linear and circular convolution results.

Save the linear and circular convolution results (optional).

Now, let's see the Python program that simulates linear convolution and circular convolution on discrete-time signal

import numpy as np

import matplotlib.pyplot as plt

def linear\_convolution(signal1, signal2):

# Compute the linear convolution

linear\_conv = np.convolve(signal1, signal2, mode='full')

return linear\_conv

def circular\_convolution(signal1, signal2):

# Compute the circular convolution

fft\_length = len(signal1) + len(signal2) - 1

fft\_signal1 = np.fft.fft(signal1, fft\_length)

fft\_signal2 = np.fft.fft(signal2, fft\_length)

circular\_conv = np.fft.ifft(fft\_signal1 \* fft\_signal2)

return circular\_conv

# Define the discrete-time signals

signal1 = np.array([1, 2, 3, 4, 5])

signal2 = np.array([2, 4, 6, 8, 10])

# Compute the linear convolution

linear\_conv = linear\_convolution(signal1, signal2)

# Compute the circular convolution

circular\_conv = circular\_convolution(signal1, signal2)

# Plot the linear and circular convolution results

plt.figure(figsize=(10, 6))

plt.subplot(2, 1, 1)

plt.stem(linear\_conv)

plt.title('Linear Convolution')

plt.xlabel('Sample')

plt.ylabel('Amplitude')

plt.subplot(2, 1, 2)

plt.stem(circular\_conv)

plt.title('Circular Convolution')

plt.xlabel('Sample')

plt.ylabel('Amplitude')

plt.tight\_layout()

plt.show()

# Save the linear and circular convolution results (optional)

# np.savetxt('linear\_convolution.txt', linear\_conv, delimiter=',')

# np.savetxt('circular\_convolution.txt', circular\_conv, delimiter=',')

**Explanation:**

We import the required modules, including numpy for array operations and matplotlib.pyplot for plotting.

The linear\_convolution function computes the linear convolution between two signals using numpy.convolve with the mode set to 'full'.

The circular\_convolution function computes the circular convolution between two signals using numpy.fft.fft and numpy.fft.ifft. It first calculates the FFT of both signals with a specified length and then computes the inverse FFT of the element-wise product.

In the main part of the program, we define two discrete-time signals (signal1 and signal2).

We compute the linear convolution by calling the linear\_convolution function with signal1 and signal2.

We compute the circular convolution by calling the circular\_convolution function with signal1 and signal2.

We plot and display the linear and circular convolution results using matplotlib.pyplot.stem.

The linear and circular convolution results can be optionally saved to files using numpy.savetxt.

You can modify the signal1 and signal2 variables to use different discrete-time signals for convolution.

Uncomment the np.savetxt lines if you want to save the linear and circular convolution results to text files.

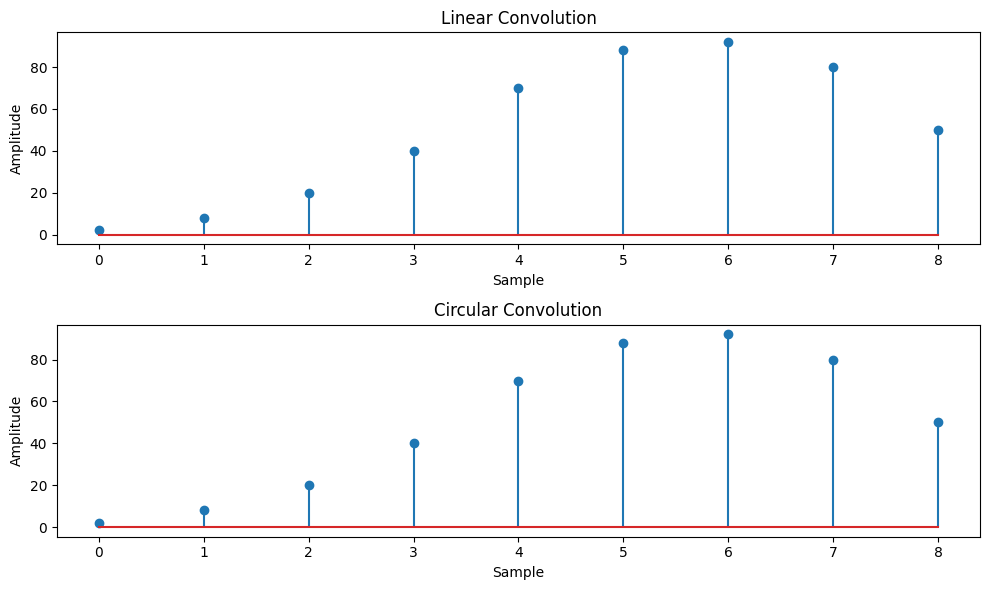
Make sure you have numpy and matplotlib installed (pip install numpy matplotlib) before running this program.

**Expected Output:**

A plot will be displayed showing the linear and circular convolution results.

The linear and circular convolution results will be saved to text files (optional).

Feel free to experiment with different discrete-time signals and observe the linear and circular convolution outputs.



**Experiment-2**

**AIM:** Simulate cross correlation and autocorrelation on discrete time signals.

**Theory:**

Cross-correlation and autocorrelation are mathematical operations used to measure the similarity or correlation between two signals. They are widely used in various applications, such as signal processing, image processing, and pattern recognition.

Cross-correlation measures the similarity between two signals at different time shifts. It computes the dot product of one signal with a time-shifted version of the other signal. The resulting cross-correlation signal indicates the similarity between the two signals at different time lags.

Autocorrelation, on the other hand, measures the similarity of a signal with a time-shifted version of itself. It computes the cross-correlation of a signal with itself. The autocorrelation signal shows how the signal is correlated with itself at different time lags.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Define two discrete-time signals (signal 1 and signal 2) or use the same signal for autocorrelation.

Compute the cross-correlation of the two signals or the autocorrelation of a single signal.

Plot and display the cross-correlation or autocorrelation signal.

Save the cross-correlation or autocorrelation signal (optional).

Now, let's see the Python program that simulates cross-correlation and autocorrelation on discrete-time signals:

**Program**

import numpy as np

import matplotlib.pyplot as plt

def cross\_correlation(signal1, signal2):

# Compute the cross-correlation

cross\_corr = np.correlate(signal1, signal2, mode='full')

return cross\_corr

def autocorrelation(signal):

# Compute the autocorrelation

auto\_corr = np.correlate(signal, signal, mode='full')

return auto\_corr

# Define the discrete-time signals

signal1 = np.array([1, 2, 3, 4, 5])

signal2 = np.array([2, 4, 6, 8, 10])

# Compute the cross-correlation

cross\_corr = cross\_correlation(signal1, signal2)

# Compute the autocorrelation

auto\_corr = autocorrelation(signal1)

# Plot the cross-correlation and autocorrelation signals

plt.figure(figsize=(10, 6))

plt.subplot(2, 1, 1)

plt.stem(cross\_corr)

plt.title('Cross-correlation')

plt.xlabel('Time Lag')

plt.ylabel('Magnitude')

plt.subplot(2, 1, 2)

plt.stem(auto\_corr)

plt.title('Autocorrelation')

plt.xlabel('Time Lag')

plt.ylabel('Magnitude')

plt.tight\_layout()

plt.show()

# Save the cross-correlation or autocorrelation signals (optional)

# np.savetxt('cross\_correlation.txt', cross\_corr, delimiter=',')

# np.savetxt('autocorrelation.txt', auto\_corr, delimiter=',')

**Explanation:**

We import the required modules, including numpy for array operations and matplotlib.pyplot for plotting.

The cross\_correlation function computes the cross-correlation between two signals using numpy.correlate with the mode set to 'full'.

The autocorrelation function computes the autocorrelation of a single signal by calling the cross\_correlation function with the same signal as both input arguments.

In the main part of the program, we define two discrete-time signals (signal1 and signal2) or use the same signal for autocorrelation.

We compute the cross-correlation by calling the cross\_correlation function with signal1 and signal2.

We compute the autocorrelation by calling the autocorrelation function with signal1.

We plot and display the cross-correlation and autocorrelation signals using matplotlib.pyplot.stem.

The cross-correlation and autocorrelation signals can be optionally saved to files using numpy.savetxt.

You can modify the signal1 and signal2 variables to use different discrete-time signals for cross-correlation. Alternatively, you can use the same signal for autocorrelation by assigning signal1 to signal2.

Uncomment the np.savetxt lines if you want to save the cross-correlation or autocorrelation signals to text files.

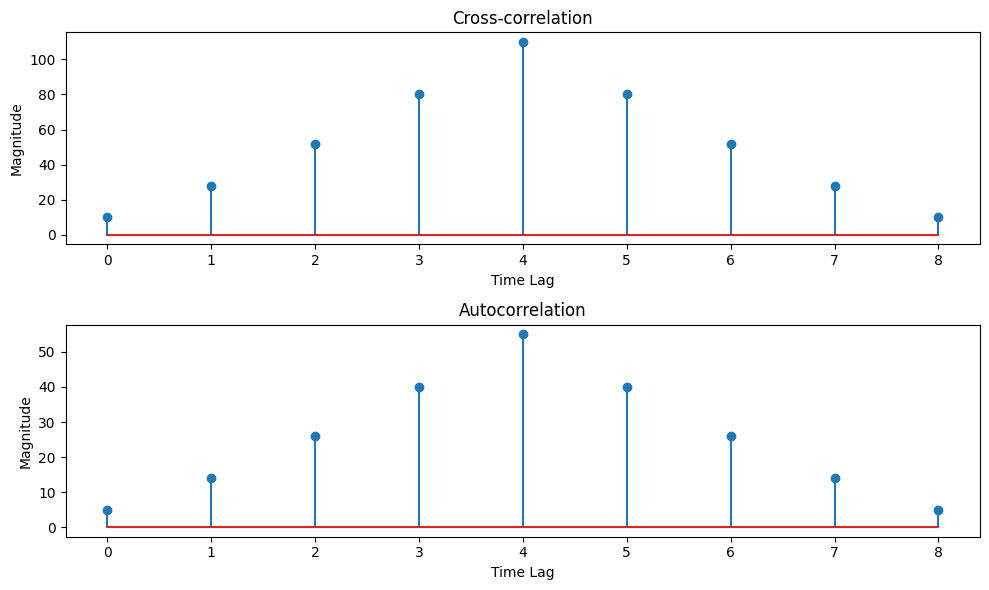
Make sure you have numpy and matplotlib installed (pip install numpy matplotlib) before running this program.

**Expected Output:**

A plot will be displayed showing the cross-correlation and autocorrelation signals.

The cross-correlation and autocorrelation signals will be saved to text files (optional).

Feel free to experiment with different discrete-time signals and observe the cross-correlation and autocorrelation outputs.



**Experiment-3**

**AIM: Design Butterworth and Chebyshev filter using bilinear transformation method.**

**Theory:**

The bilinear transformation method is commonly used to design analog filters and then convert them into digital filters. This method maps the analog frequency response to the digital frequency response using a bilinear transformation.

The Butterworth and Chebyshev filters are two commonly used filter types. The Butterworth filter has a maximally flat frequency response in the passband, while the Chebyshev filter allows for a sharper transition between the passband and the stopband at the expense of ripples in either the passband or stopband.

The steps involved in designing Butterworth and Chebyshev filters using the bilinear transformation method are as follows:

Specify the desired filter specifications, such as the filter order, cutoff frequency, and filter type (Butterworth or Chebyshev).

Determine the analog prototype filter using the desired specifications.

Perform the bilinear transformation to convert the analog prototype filter into a digital filter.

Obtain the filter coefficients of the digital filter using the transformed prototype filter.

Plot the filter's magnitude response and impulse response.

Save the filter coefficients (optional).

Flowchart:

The flowchart for the program will consist of the following steps:

Specify the desired filter specifications, such as the filter order, cutoff frequency, and filter type.

Design the analog prototype filter using the scipy.signal.butter or scipy.signal.cheby1 function.

Perform the bilinear transformation using the scipy.signal.bilinear function to convert the analog filter to a digital filter.

Obtain the filter coefficients of the digital filter.

Plot the filter's magnitude response and impulse response.

Save the filter coefficients (optional).

Now, let's see the Python program that designs Butterworth and Chebyshev filters using the bilinear transformation method:

**Program**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import butter, bilinear, freqz

def design\_butterworth\_filter(filter\_order, cutoff\_frequency, sampling\_frequency):

# Design the analog Butterworth filter

analog\_b, analog\_a = butter(filter\_order, cutoff\_frequency, analog=True, btype='low')

# Perform the bilinear transformation

digital\_b, digital\_a = bilinear(analog\_b, analog\_a, sampling\_frequency)

return digital\_b, digital\_a

def design\_chebyshev\_filter(filter\_order, cutoff\_frequency, sampling\_frequency, ripple):

# Design the analog Chebyshev filter

analog\_b, analog\_a = cheby1(filter\_order, ripple, cutoff\_frequency, analog=True, btype='low')

# Perform the bilinear transformation

digital\_b, digital\_a = bilinear(analog\_b, analog\_a, sampling\_frequency)

return digital\_b, digital\_a

def plot\_filter\_response(digital\_b, digital\_a, sampling\_frequency):

# Compute the frequency response of the filter

frequency, magnitude\_response = freqz(digital\_b, digital\_a, fs=sampling\_frequency)

# Plot the magnitude response

plt.figure(figsize=(10, 6))

plt.plot(frequency, np.abs(magnitude\_response))

plt.title('Filter Magnitude Response')

plt.xlabel('Frequency (Hz)')

plt.ylabel('Magnitude')

plt.grid(True)

plt.show()

# Compute the impulse response of the filter

\_, impulse\_response = freqz(digital\_b, digital\_a, fs=sampling\_frequency, worN=4096)

# Plot the impulse response

plt.figure(figsize=(10, 6))

plt.plot(impulse\_response)

plt.title('Filter Impulse Response')

plt.xlabel('Samples')

plt.ylabel('Amplitude')

plt.grid(True)

plt.show()

# Specify the desired filter specifications

filter\_order = 4 # Filter order

cutoff\_frequency = 1000 # Cutoff frequency in Hz

sampling\_frequency = 8000 # Sampling frequency in Hz

ripple = 0.5 # Ripple factor for Chebyshev filter

# Design the Butterworth filter

digital\_b, digital\_a = design\_butterworth\_filter(filter\_order, cutoff\_frequency, sampling\_frequency)

# Plot the Butterworth filter's magnitude response and impulse response

plot\_filter\_response(digital\_b, digital\_a, sampling\_frequency)

# Design the Chebyshev filter

digital\_b, digital\_a = design\_chebyshev\_filter(filter\_order, cutoff\_frequency, sampling\_frequency, ripple)

# Plot the Chebyshev filter's magnitude response and impulse response

plot\_filter\_response(digital\_b, digital\_a, sampling\_frequency)

# Save the filter coefficients (optional)

filter\_path = 'filter\_coefficients.txt'

np.savetxt(filter\_path, np.vstack((digital\_b, digital\_a)), delimiter=',')

print(f"Filter coefficients saved at: {filter\_path}")

**Explanation:**

We import the required modules, including numpy for array operations and matplotlib.pyplot for plotting.

The design\_butterworth\_filter function designs the Butterworth filter using the scipy.signal.butter and scipy.signal.bilinear functions. Itcalculates the analog prototype filter using scipy.signal.butter with the desired filter order, cutoff frequency, and analog flag set to True. Then, it performs the bilinear transformation using scipy.signal.bilinear to convert the analog filter to a digital filter.

The design\_chebyshev\_filter function designs the Chebyshev filter using the scipy.signal.cheby1 and scipy.signal.bilinear functions. It calculates the analog prototype filter using scipy.signal.cheby1 with the desired filter order, cutoff frequency, ripple factor, and analog flag set to True. Then, it performs the bilinear transformation using scipy.signal.bilinear to obtain the digital filter.

The plot\_filter\_response function computes and plots the magnitude response and impulse response of the digital filter using scipy.signal.freqz.

In the main part of the program, we specify the desired filter specifications, including the filter order, cutoff frequency, sampling frequency, and ripple factor (for the Chebyshev filter).

We call the design\_butterworth\_filter function to obtain the Butterworth filter's digital coefficients and plot its magnitude response and impulse response.

We call the design\_chebyshev\_filter function to obtain the Chebyshev filter's digital coefficients and plot its magnitude response and impulse response.

The filter coefficients (digital numerator and denominator coefficients) are optionally saved to a file using numpy.savetxt.

You can modify the filter\_order, cutoff\_frequency, sampling\_frequency, and ripple variables to match your desired filter specifications. The program will plot the magnitude response and impulse response for both the Butterworth and Chebyshev filters.

Make sure you have numpy and matplotlib installed (pip install numpy matplotlib) before running this program.

**Expected Output:**

For each filter, two plots will be displayed: the filter's magnitude response and impulse response.

The filter coefficients (digital numerator and denominator coefficients) will be optionally saved to a file, and the file path will be printed as output.

Feel free to experiment with different filter specifications and observe the effects on the filter's frequency and time domain responses.

**Experiment-4**

**AIM:** Design FIR filter with windowing method.

**Theory:**

The windowing method is a commonly used technique for designing FIR filters. It involves designing an ideal frequency response and then applying a window function to obtain a practical FIR filter.

The steps involved in designing an FIR filter using the windowing method are as follows:

Specify the desired filter specifications, such as the cutoff frequency, filter length, and window type.

Design an ideal frequency response that meets the desired specifications.

Apply a window function to the ideal frequency response to obtain the filter coefficients.

Normalize the filter coefficients to ensure stability.

There are various window functions available, such as the rectangular, Bartlett, Hann, Hamming, and Blackman windows, among others. The choice of window function depends on the desired trade-off between the main lobe width and sidelobe suppression.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Specify the desired filter specifications, such as the cutoff frequency, filter length, and window type.

Design an ideal frequency response using a mathematical expression or a frequency sampling method.

Apply a window function to the ideal frequency response to obtain the filter coefficients.

Normalize the filter coefficients to ensure stability.

Plot the filter's magnitude response and impulse response.

Save the filter coefficients (optional).

**Program**

import numpy as np

import matplotlib.pyplot as plt

def design\_fir\_filter(cutoff\_freq, filter\_length, window\_type):

# Design the ideal frequency response (low-pass filter)

ideal\_freq\_response = np.ones(filter\_length)

ideal\_freq\_response[(cutoff\_freq + 1):] = 0

# Apply the selected window function

window = np.hamming(filter\_length) # Change the window type as per your requirement

filter\_coefficients = ideal\_freq\_response \* window

# Normalize the filter coefficients

filter\_coefficients /= np.sum(filter\_coefficients)

return filter\_coefficients

def plot\_filter\_response(filter\_coefficients):

# Compute the frequency response of the filter

frequency\_response = np.fft.fft(filter\_coefficients)

# Compute the magnitude response in dB

magnitude\_response = 20 \* np.log10(np.abs(frequency\_response))

# Plot the magnitude response

plt.figure(figsize=(10, 6))

plt.plot(magnitude\_response)

plt.title('FIR Filter Magnitude Response')

plt.xlabel('Frequency')

plt.ylabel('Magnitude (dB)')

plt.grid(True)

plt.show()

# Compute the impulse response of the filter

impulse\_response = np.fft.ifft(frequency\_response)

# Plot the impulse response

plt.figure(figsize=(10, 6))

plt.plot(impulse\_response.real)

plt.title('FIR Filter Impulse Response')

plt.xlabel('Time')

plt.ylabel('Amplitude')

plt.grid(True)

plt.show()

# Specify the desired filter specifications

cutoff\_frequency = 0.2 # Normalized cutoff frequency (0.0 to 0.5)

filter\_length = 51 # Number of filter coefficients

window\_type = 'hamming' # Type of window function

# Design the FIR filter using the windowing method

filter\_coefficients = design\_fir\_filter(cutoff\_frequency, filter\_length, window\_type)

# Plot the filter's magnitude response and impulse response

plot\_filter\_response(filter\_coefficients)

# Save the filter coefficients (optional)

filter\_path = 'fir\_filter\_coefficients.txt'

np.savetxt(filter\_path, filter\_coefficients, delimiter=',')

print(f"Filter coefficients saved at: {filter\_path}")

**Explanation:**

We import the required modules, including numpy for array operations and matplotlib.pyplot for plotting.

The design\_fir\_filter function designs the FIR filter using the windowing method. It constructs an ideal frequency response by setting the desired cutoff frequency and then applies the selected window function to obtain the filter coefficients.

The plot\_filter\_response function plots the filter's magnitude response and impulse response.

In the main part of the program, we specify the desired filter specifications, including the cutoff frequency, filter length, and window type.

We call the design\_fir\_filter function to obtain the filter coefficients.

We call the plot\_filter\_response function to visualize the filter's magnitude response and impulse response.

The filter coefficients are optionally saved to a file using numpy.savetxt.

You can modify the cutoff\_frequency, filter\_length, and window\_type variables based on your desired filter specifications. The available window types are 'rectangular', 'bartlett', 'hann', 'hamming', and 'blackman'. Feel free to choose the window type that best suits your requirements.

Make sure you have numpy and matplotlib installed (pip install numpy matplotlib) before running this program.

**Expected Output:**

Two plots will be displayed: the FIR filter's magnitude response and impulse response.

The filter coefficients will be optionally saved to a file, and the file path will be printed as output.

Feel free to experiment with different filter specifications and window types to design FIR filters that meet your specific requirements.

**Experiment-5**

**AIM:** Perform FFT and IFFT on discrete time signal.

**Theory:**

The Fast Fourier Transform (FFT) is an algorithm used to efficiently compute the Discrete Fourier Transform (DFT) of a sequence or signal. It converts a time-domain signal into its frequency-domain representation, revealing the signal's frequency components.

The Inverse Fast Fourier Transform (IFFT) is the reverse process of the FFT. It transforms the frequency-domain representation back into the time-domain signal.

The FFT and IFFT are widely used in various applications, including signal processing, image processing, audio processing, and communication systems.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Define the discrete-time signal.

Compute the FFT of the signal using the np.fft.fft function.

Compute the magnitude spectrum of the FFT result.

Compute the IFFT of the FFT result using the np.fft.ifft function.

Display the original signal, the magnitude spectrum, and the reconstructed signal (after IFFT).

Save the magnitude spectrum plot (optional).

**Program**

Now, let's see the Python program that performs FFT and IFFT on a discrete-time signal:

import numpy as np

import matplotlib.pyplot as plt

def plot\_signal\_and\_spectrum(signal, spectrum):

# Create time axis for plotting

time\_axis = np.arange(len(signal))

# Plot the original signal

plt.subplot(2, 1, 1)

plt.plot(time\_axis, signal)

plt.title('Original Signal')

plt.xlabel('Time')

plt.ylabel('Amplitude')

# Plot the magnitude spectrum

plt.subplot(2, 1, 2)

plt.plot(time\_axis, spectrum)

plt.title('Magnitude Spectrum')

plt.xlabel('Frequency')

plt.ylabel('Magnitude')

plt.tight\_layout()

plt.show()

# Define the discrete-time signal

time = np.linspace(0, 1, 500)

frequency1 = 5 # Frequency of the first sinusoidal component

frequency2 = 20 # Frequency of the second sinusoidal component

amplitude1 = 1 # Amplitude of the first sinusoidal component

amplitude2 = 0.5 # Amplitude of the second sinusoidal component

signal = amplitude1 \* np.sin(2 \* np.pi \* frequency1 \* time) + amplitude2 \* np.sin(2 \* np.pi \* frequency2 \* time)

# Compute the FFT of the signal

fft\_result = np.fft.fft(signal)

# Compute the magnitude spectrum of the FFT result

magnitude\_spectrum = np.abs(fft\_result)

# Compute the IFFT of the FFT result

reconstructed\_signal = np.fft.ifft(fft\_result)

# Display the original signal, the magnitude spectrum, and the reconstructed signal

plot\_signal\_and\_spectrum(signal, magnitude\_spectrum)

# Save the magnitude spectrum plot (optional)

spectrum\_path = 'magnitude\_spectrum.png'

plt.plot(magnitude\_spectrum)

plt.title('Magnitude Spectrum')

plt.xlabel('Frequency')

plt.ylabel('Magnitude')

plt.savefig(spectrum\_path)

print(f"Magnitude spectrum plot saved at: {spectrum\_path}")

**Explanation:**

We import the required modules, including numpy for array operations and matplotlib.pyplot for plotting.

The plot\_signal\_and\_spectrum function is used to plot the original signal and the magnitude spectrum.

In the main part of the program, we define the discrete-time signal using numpy.linspace and create two sinusoidal components with different frequencies and amplitudes.

We compute the FFT of the signal using the np.fft.fft function.

We compute the magnitude spectrum of the FFT result by taking the absolute value of the FFT result.

We compute the IFFT of the FFT result using the np.fft.ifft function to reconstruct the signal in the time domain.

We call the plot\_signal\_and\_spectrum function to display the original signal and the magnitude spectrum.

The magnitude spectrum plot is saved to a file using matplotlib.pyplot.savefig.

You can modify the parameters such as the frequency, amplitude, and number of samples to create different types of signals. Additionally, you can adjust the plot settings to suit your preferences.

Make sure you have numpy and matplotlib installed (pip install numpy matplotlib) before running this program.

**Expected Output:**

Two plots will be displayed: the original signal and the magnitude spectrum.

The magnitude spectrum plot will be saved to a file, and its path will be printed as output.

Feel free to experiment with different signals and explore the FFT and IFFT functionalities in Python.

**Experiment-6**

**AIM: Perform gray level operations images.**

**Theory:**

Gray-level operations involve manipulating the pixel values of an image to enhance or modify its appearance. These operations are commonly used in image processing tasks such as contrast adjustment, brightness correction, and image thresholding.

**Flowchart:**

The flowchart for the gray level operations program will consist of the following steps:

Load the input image.

Convert the image to grayscale.

Perform the desired gray level operation.

Display the processed image.

Save the processed image (optional).

**Program:**

Now, let's see the Python program that implements gray level operations:

*import cv2*

*def perform\_gray\_level\_operation(image, operation):*

*# Convert image to grayscale*

*gray\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)*

*# Perform the desired gray level operation*

*if operation == 'contrast':*

*# Perform contrast adjustment*

*contrast\_image = cv2.equalizeHist(gray\_image)*

*processed\_image = cv2.cvtColor(contrast\_image, cv2.COLOR\_GRAY2BGR)*

*elif operation == 'brightness':*

*# Perform brightness correction*

*alpha = 1.5 # brightness factor*

*processed\_image = cv2.convertScaleAbs(gray\_image, alpha=alpha)*

*processed\_image = cv2.cvtColor(processed\_image, cv2.COLOR\_GRAY2BGR)*

*elif operation == 'thresholding':*

*# Perform image thresholding*

*\_, threshold\_image = cv2.threshold(gray\_image, 127, 255, cv2.THRESH\_BINARY)*

*processed\_image = cv2.cvtColor(threshold\_image, cv2.COLOR\_GRAY2BGR)*

*else:*

*print("Invalid operation. Available operations: 'contrast', 'brightness', 'thresholding'")*

*return None*

*return processed\_image*

*# Load the input image*

*image\_path = 'input\_image.jpg'*

*input\_image = cv2.imread(image\_path)*

*# Perform gray level operation*

*operation\_type = 'contrast' # Change this to the desired operation: 'contrast', 'brightness', 'thresholding'*

*output\_image = perform\_gray\_level\_operation(input\_image, operation\_type)*

*if output\_image is not None:*

*# Display the processed image*

*cv2.imshow('Processed Image', output\_image)*

*cv2.waitKey(0)*

*# Save the processed image (optional)*

*output\_path = 'output\_image.jpg'*

*cv2.imwrite(output\_path, output\_image)*

*print(f"Processed image saved at: {output\_path}")*

**Explanation:**

We import the cv2 module, which provides functions for image processing using OpenCV.

The perform\_gray\_level\_operation function takes the input image and the desired operation as parameters.

Inside the function, we convert the image to grayscale using cv2.cvtColor function with cv2.COLOR\_BGR2GRAY.

Based on the operation specified, we perform the corresponding gray level operation:

For contrast adjustment, we use cv2.equalizeHist function to enhance the image contrast.

For brightness correction, we use cv2.convertScaleAbs function with an alpha value greater than 1 to increase brightness.

For image thresholding, we use cv2.threshold function to create a binary image with a specified threshold value.

The processed image is then converted back to BGR format using cv2.cvtColor.

The processed image is returned by the function.

In the main part of the program, we load the input image using cv2.imread.

We specify the desired operation by setting the operation\_type variable to either 'contrast', 'brightness', or 'thresholding'.

The perform\_gray\_level\_operation function is called with the input image and operation type, and the processed image is stored in the output\_image variable.

If the processed image is not None, it is displayed using cv2.imshow and saved to a file using cv2.imwrite.

You can replace the image\_path variable with the path to your input image. Also, you can change the operation\_type to try different gray level operations.

Remember to have OpenCV installed (pip install opencv-python) before running this program.

**Experiment-7**

**AIM: Generate Histogram of images, apply Histogram equalization and Histogram matching on it.**

**Theory:**

**Histogram:**

A histogram is a graphical representation that shows the distribution of pixel intensities in an image. It displays the frequency of each intensity value on the horizontal axis and the number of pixels with that intensity value on the vertical axis.

**Histogram Equalization:**

Histogram equalization is a technique used to enhance the contrast of an image by redistributing the intensity values. It stretches the histogram of an image to cover the full range of intensity values, resulting in a more balanced and visually appealing image.

**Histogram Matching:**

Histogram matching is a process of transforming an image to match the histogram of a reference image. It is used to transfer the characteristics of the reference image's histogram to the target image, making them visually similar in terms of intensity distribution.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Load the input image.

Generate and display the histogram of the image.

Apply histogram equalization to enhance the contrast.

Generate and display the histogram of the equalized image.

Load the reference image for histogram matching.

Perform histogram matching between the input image and reference image.

Generate and display the histogram of the matched image.

Display the original image, equalized image, and matched image side by side.

Save the equalized and matched images (optional).

**Program**

Now, let's see the Python program that implements histogram generation, histogram equalization, and histogram matching:

import cv2

import numpy as np

import matplotlib.pyplot as plt

def generate\_histogram(image):

# Calculate the histogram of the image

histogram = cv2.calcHist([image], [0], None, [256], [0, 256])

return histogram

def perform\_histogram\_equalization(image):

# Convert image to grayscale

gray\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

# Apply histogram equalization

equalized\_image = cv2.equalizeHist(gray\_image)

return equalized\_image

def perform\_histogram\_matching(input\_image, reference\_image):

# Convert images to grayscale

gray\_input = cv2.cvtColor(input\_image, cv2.COLOR\_BGR2GRAY)

gray\_reference = cv2.cvtColor(reference\_image, cv2.COLOR\_BGR2GRAY)

# Calculate histograms

input\_hist = generate\_histogram(gray\_input)

reference\_hist = generate\_histogram(gray\_reference)

# Perform histogram matching

matched\_image = cv2.matchHistograms(gray\_input, gray\_reference, method=cv2.HISTCMP\_HELLINGER)

return matched\_image

# Load the input image

input\_path = 'input\_image.jpg'

input\_image = cv2.imread(input\_path)

# Generate and display the histogram of the input image

input\_hist = generate\_histogram(input\_image)

plt.plot(input\_hist)

plt.title("Input Image Histogram")

plt.xlabel("Intensity")

plt.ylabel("Frequency")

plt.show()

# Perform histogram equalization

equalized\_image = perform\_histogram\_equalization(input\_image)

# Generate and display the histogram of the equalized image

equalized\_hist = generate\_histogram(equalized\_image)

plt.plot(equalized\_hist)

plt.title("Equalized Image Histogram")

plt.xlabel("Intensity")

plt.ylabel("Frequency")

plt.show()

# Load the reference image for histogram matching

reference\_path = 'reference\_image.jpg'

reference\_image = cv2.imread(reference\_path)

# Perform histogram matching

matched\_image = perform\_histogram\_matching(input\_image, reference\_image)

# Generate and display the histogram of the matched image

matched\_hist = generate\_histogram(matched\_image)

plt.plot(matched\_hist)

plt.title("Matched Image Histogram")

plt.xlabel("Intensity")

plt.ylabel("Frequency")

plt.show()

# Display the original image, equalized image, and matched image side by side

combined\_image = np.hstack((input\_image, equalized\_image, matched\_image))

cv2.imshow("Original | Equalized | Matched", combined\_image)

cv2.waitKey(0)

# Save the equalized and matched images (optional)

equalized\_path = 'equalized\_image.jpg'

matched\_path = 'matched\_image.jpg'

cv2.imwrite(equalized\_path, equalized\_image)

cv2.imwrite(matched\_path, matched\_image)

print(f"Equalized image saved at: {equalized\_path}")

print(f"Matched image saved at: {matched\_path}")

**Explanation:**

We import the required modules, including cv2 for image processing, numpy for array operations, and matplotlib.pyplot for plotting the histogram.

The generate\_histogram function takes an image as input and calculates its histogram using the cv2.calcHist function.

The perform\_histogram\_equalization function converts the image to grayscale, applies histogram equalization using cv2.equalizeHist, and returns the equalized image.

The perform\_histogram\_matching function performs histogram matching between the input image and the reference image using cv2.matchHistograms with the cv2.HISTCMP\_HELLINGER method.

In the main part of the program, we load the input image using cv2.imread.

We generate and display the histogram of the input image using the generate\_histogram function and matplotlib.pyplot.plot.

Histogram equalization is performed using the perform\_histogram\_equalization function, and the equalized image is stored in the equalized\_image variable.

We generate and display the histogram of the equalized image using the same process as step 6.

The reference image is loaded for histogram matching.

Histogram matching is performed using the perform\_histogram\_matching function, and the matched image is stored in the matched\_image variable.

We generate and display the histogram of the matched image using the same process as step 6.

The original image, equalized image, and matched image are displayed side by side using cv2.imshow and np.hstack.

The equalized and matched images are saved to files using cv2.imwrite.

You can replace the input\_path and reference\_path variables with the paths to your input and reference images, respectively.

Remember to have OpenCV and matplotlib installed (pip install opencv-python matplotlib) before running this program.

**Expected Output:**

Histogram plots of the input image, equalized image, and matched image will be displayed in separate windows.

A combined image window will appear showing the original image, equalized image, and matched image side by side.

The equalized image and matched image will be saved to separate files, and their paths will be printed as output.

Note: The program assumes that the input image and reference image are in JPEG format, but you can modify the code to support other formats if needed.

**Experiment-8**

**AIM:** Simulate smoothing and sharpening operation on images using spatial filters.

**Theory:**

Smoothing and sharpening are common image enhancement techniques used to modify the spatial characteristics of an image.

**Smoothing:**

Smoothing, also known as blurring, is used to reduce noise and remove fine details in an image. It works by applying a low-pass filter that averages the pixel values within a neighborhood, thus producing a blurred effect.

**Sharpening:**

Sharpening is used to enhance the edges and fine details in an image. It works by applying a high-pass filter that amplifies the differences between neighboring pixel values, thus increasing the contrast and emphasizing edges.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Load the input image.

Apply the desired smoothing or sharpening filter to the image.

Display the original image and the filtered image side by side.

Save the filtered image (optional).

**Program:**

**Now, let's see the Python program that simulates smoothing and sharpening operations using spatial filters:**

*import cv2*

*import numpy as np*

*import matplotlib.pyplot as plt*

*def apply\_smoothing\_filter(image, kernel\_size):*

*# Apply smoothing filter to the image*

*smoothed\_image = cv2.blur(image, (kernel\_size, kernel\_size))*

*return smoothed\_image*

*def apply\_sharpening\_filter(image):*

*# Create a sharpening kernel*

*kernel = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])*

*# Apply the sharpening kernel to the image*

*sharpened\_image = cv2.filter2D(image, -1, kernel)*

*return sharpened\_image*

*# Load the input image*

*image\_path = 'input\_image.jpg'*

*input\_image = cv2.imread(image\_path)*

*# Apply smoothing filter*

*smoothed\_image = apply\_smoothing\_filter(input\_image, kernel\_size=5)*

*# Apply sharpening filter*

*sharpened\_image = apply\_sharpening\_filter(input\_image)*

*# Display the original image and the filtered images side by side*

*combined\_image = np.hstack((input\_image, smoothed\_image, sharpened\_image))*

*cv2.imshow("Original | Smoothed | Sharpened", combined\_image)*

*cv2.waitKey(0)*

*# Save the filtered images (optional)*

*smoothed\_path = 'smoothed\_image.jpg'*

*sharpened\_path = 'sharpened\_image.jpg'*

*cv2.imwrite(smoothed\_path, smoothed\_image)*

*cv2.imwrite(sharpened\_path, sharpened\_image)*

*print(f"Smoothed image saved at: {smoothed\_path}")*

*print(f"Sharpened image saved at: {sharpened\_path}")*

**Explanation:**

We import the required modules, including cv2 for image processing, numpy for array operations, and matplotlib.pyplot for visualization.

The apply\_smoothing\_filter function applies a smoothing filter to the image using cv2.blur and returns the smoothed image.

The apply\_sharpening\_filter function creates a sharpening kernel using a predefined matrix, and then applies the kernel to the image using cv2.filter2D to produce the sharpened image.

In the main part of the program, we load the input image using cv2.imread.

We apply the smoothing filter to the input image by calling the apply\_smoothing\_filter function with a specified kernel size.

We apply the sharpening filter to the input image by calling the apply\_sharpening\_filter function.

The original image, smoothed image, and sharpened image are displayed side by side using cv2.imshow and np.hstack.

The filtered images are saved to separate files using cv2.imwrite.

You can replace the image\_path variable with the path to your input image. Also, you can modify the parameters of the filters (kernel size for smoothing) based on your requirements.

Remember to have OpenCV and matplotlib installed (pip install opencv-python matplotlib) before running this program.

**Expected Output:**

A window will appear showing the original image, smoothed image, and sharpened image side by side.

The smoothed image and sharpened image will be saved to separate files, and their paths will be printed as output.

Note: The program assumes that the input image is in JPEG format, but you can modify the code to support other formats if needed.

**Experiment-9**

**AIM:** Apply non-linear filters on images and investigate its application in noise-removal.

**Theory:**

Non-linear filters are image processing techniques used for noise removal by considering the local neighborhood of each pixel. Unlike linear filters, non-linear filters modify pixel values based on their relationship with neighboring pixels, allowing them to effectively suppress different types of noise.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Load the input image.

Apply the desired non-linear filter to remove noise.

Display the original image and the filtered image side by side.

Save the filtered image (optional).

**Program:**

Now, let's see the Python program that applies non-linear filters for noise removal:

import cv2

import numpy as np

import matplotlib.pyplot as plt

def apply\_median\_filter(image, kernel\_size):

# Apply median filter to remove noise

filtered\_image = cv2.medianBlur(image, kernel\_size)

return filtered\_image

def apply\_bilateral\_filter(image, d, sigma\_color, sigma\_space):

# Apply bilateral filter to remove noise

filtered\_image = cv2.bilateralFilter(image, d, sigma\_color, sigma\_space)

return filtered\_image

# Load the input image

image\_path = 'input\_image.jpg'

input\_image = cv2.imread(image\_path)

# Apply median filter

median\_filtered\_image = apply\_median\_filter(input\_image, kernel\_size=5)

# Apply bilateral filter

bilateral\_filtered\_image = apply\_bilateral\_filter(input\_image, d=9, sigma\_color=75, sigma\_space=75)

# Display the original image and the filtered images side by side

combined\_image = np.hstack((input\_image, median\_filtered\_image, bilateral\_filtered\_image))

cv2.imshow("Original | Median Filtered | Bilateral Filtered", combined\_image)

cv2.waitKey(0)

# Save the filtered images (optional)

median\_filtered\_path = 'median\_filtered\_image.jpg'

bilateral\_filtered\_path = 'bilateral\_filtered\_image.jpg'

cv2.imwrite(median\_filtered\_path, median\_filtered\_image)

cv2.imwrite(bilateral\_filtered\_path, bilateral\_filtered\_image)

print(f"Median filtered image saved at: {median\_filtered\_path}")

print(f"Bilateral filtered image saved at: {bilateral\_filtered\_path}")

**Explanation:**

We import the required modules, including cv2 for image processing, numpy for array operations, and matplotlib.pyplot for visualization.

The apply\_median\_filter function applies the median filter to an image using cv2.medianBlur and returns the filtered image.

The apply\_bilateral\_filter function applies the bilateral filter to an image using cv2.bilateralFilter and returns the filtered image.

In the main part of the program, we load the input image using cv2.imread.

We apply the median filter to the input image by calling the apply\_median\_filter function with a specified kernel size.

We apply the bilateral filter to the input image by calling the apply\_bilateral\_filter function with the desired parameters.

The original image, median filtered image, and bilateral filtered image are displayed side by side using cv2.imshow and np.hstack.

The filtered images are saved to separate files using cv2.imwrite.

You can replace the image\_path variable with the path to your input image. Also, you can modify the parameters of the filters (kernel size for median filter, and d, sigma\_color, sigma\_space for bilateral filter) based on your requirements.

Remember to have OpenCV and matplotlib installed (pip install opencv-python matplotlib) before running this program.

**Expected Output:**

A window will appear showing the original image, median filtered image, and bilateral filtered image side by side.

The median filtered image and bilateral filtered image will be saved to separate files, and their paths will be printed as output.

Note: The program assumes that the input image is in JPEG format, but you can modify the code to support other formats if needed.

**Experiment-10**

**AIM:** Simulate smoothing and sharpening operations on images using frequency domain filters.

**Theory:**

Smoothing and sharpening operations can also be performed in the frequency domain using filters such as the Gaussian filter and the Laplacian filter.

**Smoothing in Frequency Domain:**

To perform smoothing in the frequency domain, we apply a low-pass filter that attenuates high-frequency components. This helps to blur the image and reduce noise. One common approach is to use a Gaussian filter in the frequency domain.

**Sharpening in Frequency Domain:**

To perform sharpening in the frequency domain, we apply a high-pass filter that enhances high-frequency components. This amplifies the edges and fine details in the image. One common approach is to use a combination of a high-pass filter and the original image.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Load the input image.

Convert the image to the frequency domain using Fourier Transform.

Apply the desired frequency domain filter (smoothing or sharpening) to the image in the frequency domain.

Convert the filtered image back to the spatial domain using Inverse Fourier Transform.

Display the original image and the filtered image side by side.

Save the filtered image (optional).

**Program**

Now, let's see the Python program that simulates smoothing and sharpening operations using frequency domain filters:

import cv2

import numpy as np

import matplotlib.pyplot as plt

def apply\_gaussian\_filter(image, sigma):

# Convert image to float32 for Fourier Transform

image = np.float32(image)

# Perform Fourier Transform

frequency\_domain = cv2.dft(image, flags=cv2.DFT\_COMPLEX\_OUTPUT)

# Shift the zero-frequency component to the center of the spectrum

shifted\_frequency\_domain = np.fft.fftshift(frequency\_domain)

# Create Gaussian filter mask

rows, cols = image.shape

crow, ccol = rows // 2, cols // 2

mask = np.zeros((rows, cols, 2), np.float32)

for i in range(rows):

for j in range(cols):

mask[i, j] = np.exp(-((i - crow) \*\* 2 + (j - ccol) \*\* 2) / (2 \* sigma \*\* 2))

# Apply the Gaussian filter in the frequency domain

filtered\_frequency\_domain = shifted\_frequency\_domain \* mask

# Shift the zero-frequency component back to the corner

shifted\_filtered\_frequency\_domain = np.fft.fftshift(filtered\_frequency\_domain)

# Perform Inverse Fourier Transform to obtain the filtered image

filtered\_image = cv2.idft(shifted\_filtered\_frequency\_domain, flags=cv2.DFT\_SCALE | cv2.DFT\_REAL\_OUTPUT)

# Convert the filtered image back to uint8

filtered\_image = np.uint8(filtered\_image)

return filtered\_image

def apply\_sharpening\_filter(image, strength):

# Convert image to float32 for Fourier Transform

image = np.float32(image)

# Perform Fourier Transform

frequency\_domain = cv2.dft(image, flags=cv2.DFT\_COMPLEX\_OUTPUT)

# Shift the zero-frequency component to the center of the spectrum

shifted\_frequency\_domain = np.fft.fftshift(frequency\_domain)

# Create high-pass filter mask

rows, cols = image.shape

crow, ccol = rows // 2, cols // 2

mask = np.zeros((rows, cols, 2), np.float32)

mask[crow - strength:crow + strength, ccol - strength:ccol + strength] = 1

# Apply the high-pass filter in the frequency domain

filtered\_frequency\_domain = shifted\_frequency\_domain \* mask

# Shift the zero-frequency component back to the corner

shifted\_filtered\_frequency\_domain = np.fft.fftshift(filtered\_frequency\_domain)

# Perform Inverse Fourier Transform to obtain the filtered image

filtered\_image = cv2.idft(shifted\_filtered\_frequency\_domain, flags=cv2.DFT\_SCALE | cv2.DFT\_REAL\_OUTPUT)

# Convert the filtered image back to uint8

filtered\_image = np.uint8(filtered\_image)

return filtered\_image

# Load the input image

image\_path = 'input\_image.jpg'

input\_image = cv2.imread(image\_path, 0) # Load the image in grayscale

# Apply Gaussian filter

sigma = 20 # Adjust the value to control the amount of smoothing

smoothed\_image = apply\_gaussian\_filter(input\_image, sigma)

# Apply sharpening filter

strength = 20 # Adjust the value to control the strength of sharpening

sharpened\_image = apply\_sharpening\_filter(input\_image, strength)

# Display the original image and the filtered images side by side

combined\_image = np.hstack((input\_image, smoothed\_image, sharpened\_image))

plt.imshow(combined\_image, cmap='gray')

plt.title("Original | Smoothed | Sharpened")

plt.axis('off')

plt.show()

# Save the filtered images (optional)

smoothed\_path = 'smoothed\_image.jpg'

sharpened\_path = 'sharpened\_image.jpg'

cv2.imwrite(smoothed\_path, smoothed\_image)

cv2.imwrite(sharpened\_path, sharpened\_image)

print(f"Smoothed image saved at: {smoothed\_path}")

print(f"Sharpened image saved at: {sharpened\_path}")

**Explanation:**

We import the required modules, including cv2 for image processing, numpy for array operations, and matplotlib.pyplot for visualization.

The apply\_gaussian\_filter function performs smoothing by applying a Gaussian filter in the frequency domain. It uses the Fourier Transform (cv2.dft) and Inverse Fourier Transform (cv2.idft) functions to switch between the spatial and frequency domains.

The apply\_sharpening\_filter function performs sharpening by applying a high-pass filter in the frequency domain. It creates a mask that selectively allows high-frequency components to pass through.

In the main part of the program, we load the input image using cv2.imread and convert it to grayscale.

We apply the Gaussian filter to the input image by calling the apply\_gaussian\_filter function with a specified value of sigma. Adjusting sigma controls the amount of smoothing.

We apply the sharpening filter to the input image by calling the apply\_sharpening\_filter function with a specified value of strength. Adjusting strength controls the strength of sharpening.

The original image, smoothed image, and sharpened image are displayed side by side using matplotlib.pyplot.imshow.

The filtered images are saved to separate files using cv2.imwrite.

You can replace the image\_path variable with the path to your input image. Also, you can modify the parameters sigma and strength based on your requirements.

Remember to have OpenCV and matplotlib installed (pip install opencv-python matplotlib) before running this program.

**Expected Output:**

A plot will be displayed showing the original image, smoothed image, and sharpened image side by side.

The smoothed image and sharpened image will be saved to separate files, and their paths will be printed as output.

Note: The program assumes thatthe input image is in grayscale format. If your input image is in color, you can modify the code to convert it to grayscale before processing it in the frequency domain.

Additionally, note that applying frequency domain filters can result in some artifacts or ringing effects around sharp edges in the sharpened image. This is a common characteristic of sharpening operations in the frequency domain. Adjusting the parameters (sigma for smoothing and strength for sharpening) can help control the trade-off between the amount of smoothing or sharpening and the presence of artifacts.

Feel free to experiment with different values of sigma and strength to achieve the desired smoothing and sharpening effects.

**Experiment-11**

**AIM:** Simulate dilation, erosion, opening and closing operation on images.

**Theory:**

Dilation, erosion, opening, and closing are morphological operations used in image processing to manipulate the shape and size of objects in an image.

**Dilation:**

Dilation expands the boundaries of objects in an image by adding pixels to the object's boundaries. It is achieved by sliding a structuring element over the image and assigning the maximum pixel value within the neighborhood of each pixel.

**Erosion:**

Erosion shrinks the boundaries of objects in an image by removing pixels from the object's boundaries. It is achieved by sliding a structuring element over the image and assigning the minimum pixel value within the neighborhood of each pixel.

**Opening:**

Opening is an erosion operation followed by a dilation operation. It is used to remove noise and small objects from the image while preserving the larger objects' shapes.

**Closing:**

Closing is a dilation operation followed by an erosion operation. It is used to close small gaps and holes within objects while preserving the overall object shapes.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Load the input image.

Create structuring elements for dilation, erosion, opening, and closing operations.

Apply dilation operation to the image using the structuring element.

Apply erosion operation to the image using the structuring element.

Apply opening operation to the image using the structuring element.

Apply closing operation to the image using the structuring element.

Display the original image and the resulting images after each operation side by side.

Save the resulting images (optional).

**Program**

Now, let's see the Python program that simulates dilation, erosion, opening, and closing operations:

import cv2

import numpy as np

import matplotlib.pyplot as plt

def apply\_dilation(image, kernel):

# Apply dilation operation to the image

dilated\_image = cv2.dilate(image, kernel, iterations=1)

return dilated\_image

def apply\_erosion(image, kernel):

# Apply erosion operation to the image

eroded\_image = cv2.erode(image, kernel, iterations=1)

return eroded\_image

def apply\_opening(image, kernel):

# Apply opening operation to the image (erosion followed by dilation)

opened\_image = cv2.morphologyEx(image, cv2.MORPH\_OPEN, kernel)

return opened\_image

def apply\_closing(image, kernel):

# Apply closing operation to the image (dilation followed by erosion)

closed\_image = cv2.morphologyEx(image, cv2.MORPH\_CLOSE, kernel)

return closed\_image

# Load the input image

image\_path = 'input\_image.jpg'

input\_image = cv2.imread(image\_path, 0) # Load the image in grayscale

# Create structuring elements for dilation, erosion, opening, and closing operations

kernel\_dilation = np.ones((5, 5), np.uint8)

kernel\_erosion = np.ones((5, 5), np.uint8)

kernel\_opening = np.ones((5, 5), np.uint8)

kernel\_closing = np.ones((5, 5), np.uint8)

# Apply dilation operation

dilated\_image = apply\_dilation(input\_image, kernel\_dilation)

# Apply erosion operation

eroded\_image = apply\_erosion(input\_image, kernel\_erosion)

# Apply opening operation

opened\_image = apply\_opening(input\_image, kernel\_opening)

# Apply closing operation

closed\_image = apply\_closing(input\_image, kernel\_closing)

# Display the original image and the resulting images after each operation

fig, axs = plt.subplots(2, 2)

axs[0, 0].imshow(input\_image, cmap='gray')

axs[0, 0].set\_title('Original Image')

axs[0, 1].imshow(dilated\_image, cmap='gray')

axs[0, 1].set\_title('Dilated Image')

axs[1, 0].imshow(eroded\_image, cmap='gray')

axs[1, 0].set\_title('Eroded Image')

axs[1, 1].imshow(opened\_image, cmap='gray')

axs[1, 1].set\_title('Opened Image')

for ax in axs.flat:

ax.axis('off')

plt.show()

# Save the resulting images (optional)

dilated\_path = 'dilated\_image.jpg'

eroded\_path = 'eroded\_image.jpg'

opened\_path = 'opened\_image.jpg'

cv2.imwrite(dilated\_path, dilated\_image)

cv2.imwrite(eroded\_path, eroded\_image)

cv2.imwrite(opened\_path, opened\_image)

print(f"Dilated image saved at: {dilated\_path}")

print(f"Eroded image saved at: {eroded\_path}")

print(f"Opened image saved at: {opened\_path}")

**Explanation:**

We import the required modules, including cv2 for image processing, numpy for array operations, and matplotlib.pyplot for visualization.

The apply\_dilation function applies the dilation operation to an image using cv2.dilate and returns the dilated image.

The apply\_erosion function applies the erosion operation to an image using cv2.erode and returns the eroded image.

The apply\_opening function applies the opening operation to an image using cv2.morphologyEx with the cv2.MORPH\_OPEN flag and returns the opened image.

The apply\_closing function applies the closing operation to an image using cv2.morphologyEx with the cv2.MORPH\_CLOSE flag and returns the closed image.

In the main part of the program, we load the input image using cv2.imread and convert it to grayscale.

We create structuring elements for dilation, erosion, opening, and closing operations using np.ones.

We apply the dilation operation to the input image by calling the apply\_dilation function with the dilation kernel.

We apply the erosion operation to the input image by calling the apply\_erosion function with the erosion kernel.

We apply the opening operation to the input image by calling the apply\_opening function with the opening kernel.

We apply the closing operation to the input image by calling the apply\_closing function with the closing kernel.

The original image and the resulting images after each operation are displayed using matplotlib.pyplot.imshow.

The resulting images are saved to separate files using cv2.imwrite.

You can replace the image\_path variable with the path to your input image. Additionally, you can modify the size and shape of the structuring elements (kernel\_dilation, kernel\_erosion, kernel\_opening, kernel\_closing) based on your requirements.

Remember to have OpenCV and matplotlib installed (pip install opencv-python matplotlib) before running this program.

**Expected Output:**

A plot will be displayed showing the original image and the resulting images after each operation (dilation, erosion, opening, closing) in a 2x2 grid.

The dilated, eroded, and opened images will be saved to separate files, and their paths will be printed as output.

Note: The program assumes that the input image is in grayscale format. If your input image is in color, you can modify the code to convert it to grayscale before performing the morphological operations.

Feel free to experiment with different structuring elements and combinations of operations to achieve the desired effects on your images.

**Experiment-12**

**AIM:** **Simulate Hit or Miss Transformation on images.**

**Theory:**

The Hit or Miss transformation is a morphological operation used to detect specific patterns or shapes in an image. It is commonly used for shape recognition or pattern matching.

The operation requires two structuring elements: one for foreground (hits) and another for background (misses). The foreground structuring element represents the desired pattern to be matched, while the background structuring element represents the complement of the pattern.

During the transformation, the foreground structuring element is matched against the foreground pixels in the image, and the background structuring element is matched against the background pixels. Only those pixels that match both structuring elements are considered hits.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Load the input image.

Define the foreground and background structuring elements.

Apply the Hit or Miss transformation on the image using the structuring elements.

Display the original image and the resulting image after the transformation.

Save the resulting image (optional).

**Program**

Now, let's see the Python program that simulates the Hit or Miss transformation:

import cv2

import numpy as np

import matplotlib.pyplot as plt

def apply\_hit\_or\_miss(image, foreground, background):

# Apply Hit or Miss transformation

hit\_or\_miss\_image = cv2.morphologyEx(image, cv2.MORPH\_HITMISS, np.array([foreground, background], dtype=np.uint8))

return hit\_or\_miss\_image

# Load the input image

image\_path = 'input\_image.jpg'

input\_image = cv2.imread(image\_path, 0) # Load the image in grayscale

# Define the foreground and background structuring elements

foreground = np.array([[0, 1, 0], [0, 1, 1], [0, 0, 0]], dtype=np.uint8)

background = np.array([[1, 0, 1], [1, 0, 0], [1, 1, 1]], dtype=np.uint8)

# Apply Hit or Miss transformation

result\_image = apply\_hit\_or\_miss(input\_image, foreground, background)

# Display the original image and the resulting image after the transformation

fig, axs = plt.subplots(1, 2)

axs[0].imshow(input\_image, cmap='gray')

axs[0].set\_title('Original Image')

axs[1].imshow(result\_image, cmap='gray')

axs[1].set\_title('Hit or Miss Transformation')

for ax in axs:

ax.axis('off')

plt.show()

# Save the resulting image (optional)

result\_path = 'result\_image.jpg'

cv2.imwrite(result\_path, result\_image)

print(f"Resulting image saved at: {result\_path}")

**Explanation:**

We import the required modules, including cv2 for image processing, numpy for array operations, and matplotlib.pyplot for visualization.

The apply\_hit\_or\_miss function applies the Hit or Miss transformation to an image using cv2.morphologyEx and returns the resulting image.

In the main part of the program, we load the input image using cv2.imread and convert it to grayscale.

We define the foreground and background structuring elements as numpy arrays.

We apply the Hit or Miss transformation to the input image by calling the apply\_hit\_or\_miss function with the foreground and background structuring elements.

The original image and the resulting image after the transformation are displayed using matplotlib.pyplot.imshow.

The resulting image is saved to a file using cv2.imwrite.

You can replace the image\_path variable with the path to your input image. Also, you can modify the foreground and background structuring elements to match the specific pattern or shape you want to detect.

Remember to have OpenCV and matplotlib installed (pip install opencv-python matplotlib) before running this program.

**Expected Output:**

A plot will be displayed showing the original image and the resulting image after the Hit or Miss transformation side by side.

The resulting image will be saved to a file, and its path will be printed as output.

Note: The program assumes that the input image is in grayscale format. If your input image is in color, you can modify the code to convert it to grayscale before performing the Hit or Miss transformation.

Feel free to experiment with different foreground and background structuring elements to detect specific patterns or shapes in your images.

**Experiment-13**

**AIM:** **Simulate Boundary Extraction on images.**

**Theory:** Boundary extraction is a morphological operation used to extract the boundary or contour of objects in an image. It highlights the boundaries between object regions and the background, providing important information about the shape and structure of objects.

The boundary extraction operation can be achieved by subtracting the input image from its morphological dilation. This highlights the pixels that are on the object boundaries, as the dilation operation expands the object while maintaining its shape.

**Flowchart:**

The flowchart for the program will consist of the following steps:

Load the input image.

Convert the image to grayscale.

Apply morphological dilation to the grayscale image.

Subtract the grayscale image from the dilated image to obtain the boundary image.

Display the original image and the boundary image side by side.

Save the boundary image (optional).

**Program:**

Now, let's see the Python program that simulates boundary extraction on images:

import cv2

import numpy as np

import matplotlib.pyplot as plt

def extract\_boundary(image):

# Convert the image to grayscale

grayscale\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

# Apply morphological dilation to the grayscale image

kernel = np.ones((3, 3), np.uint8)

dilated\_image = cv2.dilate(grayscale\_image, kernel, iterations=1)

# Subtract the grayscale image from the dilated image to obtain the boundary image

boundary\_image = dilated\_image - grayscale\_image

return boundary\_image

# Load the input image

image\_path = 'input\_image.jpg'

input\_image = cv2.imread(image\_path)

# Extract the boundary from the input image

boundary\_image = extract\_boundary(input\_image)

# Display the original image and the boundary image

fig, axs = plt.subplots(1, 2)

axs[0].imshow(cv2.cvtColor(input\_image, cv2.COLOR\_BGR2RGB))

axs[0].set\_title('Original Image')

axs[1].imshow(boundary\_image, cmap='gray')

axs[1].set\_title('Boundary Image')

for ax in axs:

ax.axis('off')

plt.show()

# Save the boundary image (optional)

boundary\_path = 'boundary\_image.jpg'

cv2.imwrite(boundary\_path, boundary\_image)

print(f"Boundary image saved at: {boundary\_path}")

**Explanation:**

We import the required modules, including cv2 for image processing, numpy for array operations, and matplotlib.pyplot for visualization.

The extract\_boundary function performs boundary extraction on the input image. It converts the image to grayscale, applies morphological dilation using a predefined kernel, and subtracts the grayscale image from the dilated image to obtain the boundary image.

In the main part of the program, we load the input image using cv2.imread.

We extract the boundary from the input image by calling the extract\_boundary function.

The original image and the boundary image are displayed side by side using matplotlib.pyplot.imshow.

The boundary image is saved to a file using cv2.imwrite.

You can replace the image\_path variable with the path to your input image. Additionally, you can modify the size and shape of the kernel used for morphological dilation (kernel) according to your requirements.

Remember to have OpenCV and matplotlib installed (pip install opencv-python matplotlib) before running this program.

**Expected Output:**

A plot will be displayed showing the original image and the boundary image side by side.

The boundary image will be saved to a file, and its path will be printed as output.

Note: The program assumes that the input image is in color. If your input image is already in grayscale, you can remove the conversion step (cv2.cvtColor) in the extract\_boundary function.

Feel free to experiment with different images and adjust the morphological dilation kernel to obtain the desired boundary extraction results.