**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')

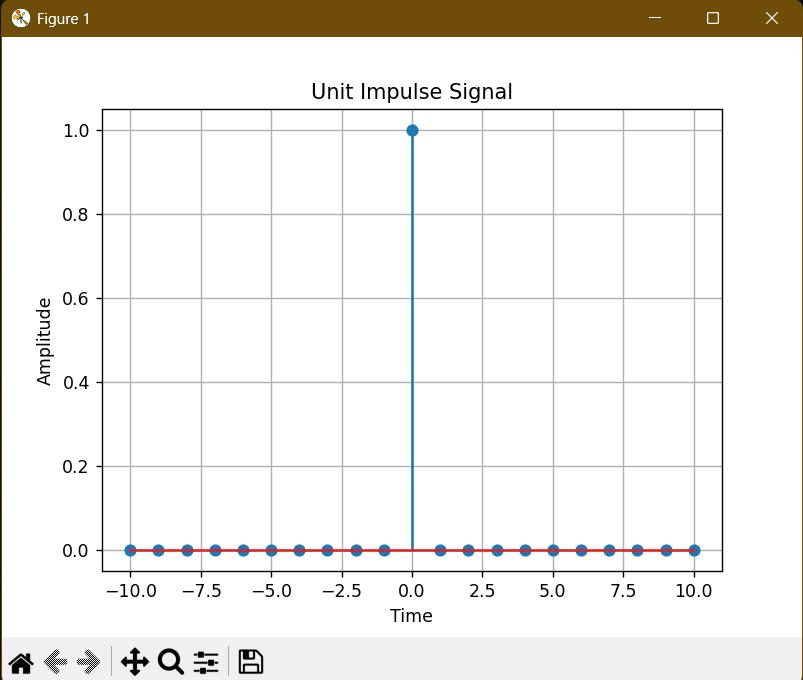
plt.ylabel('Amplitude')

plt.title('Unit Impulse Signal')

plt.grid(True)

plt.show()

**Output:**

****

**AIM**

Simulate impulse train

**Code:**

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')

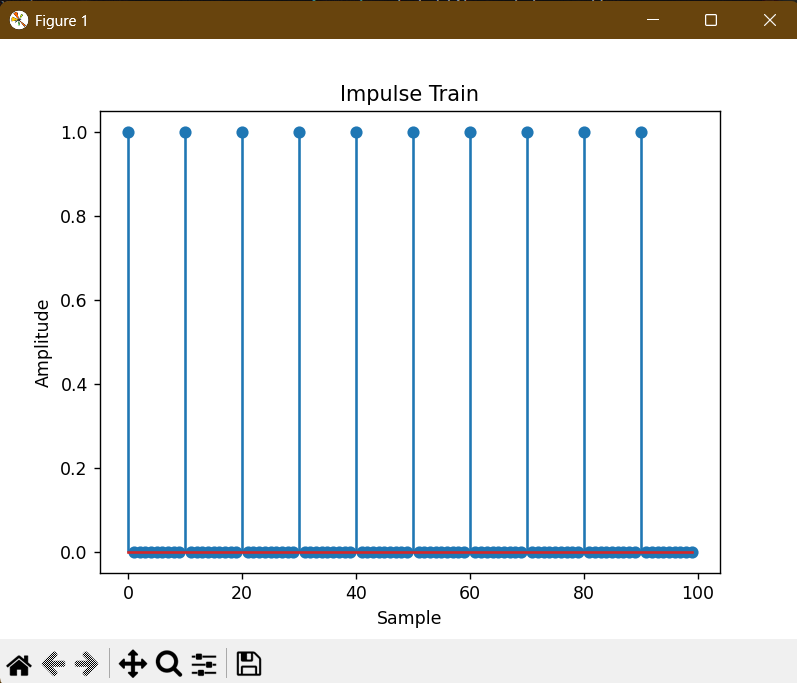
plt.ylabel('Amplitude')

plt.show()

# Save the impulse train array (optional)

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

**Output:**



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

**Code:**

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')

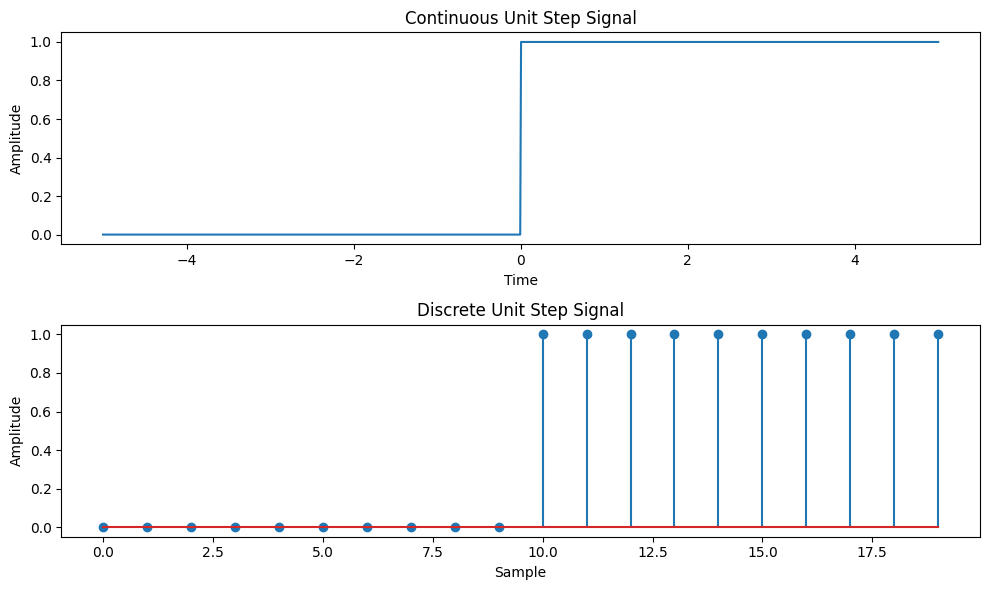
plt.tight\_layout()

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=',')

**Output:**

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

**Code:**

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()

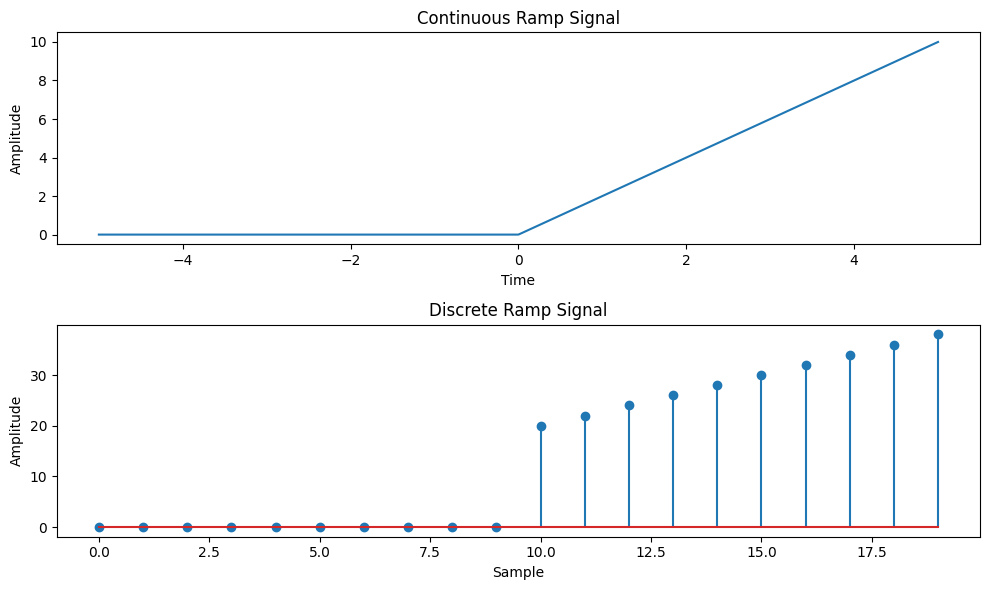
plt.show()

# Save the ramp signal arrays (optional)

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

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

Output:



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

**Code:**

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()

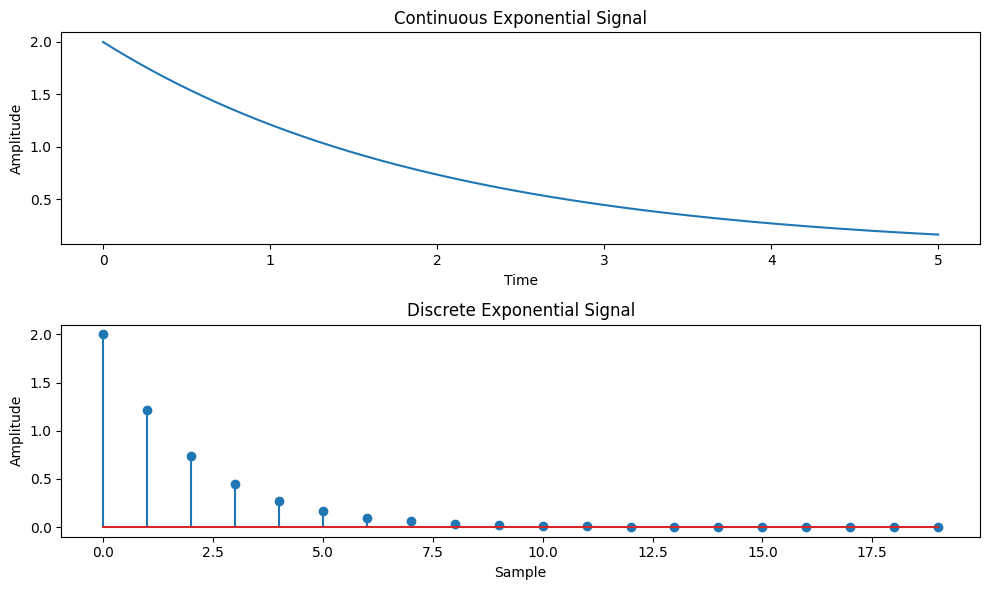
# Save the exponential signal arrays (optional)

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

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

**Explanation:**

**Output:**



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

**Code:**

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()

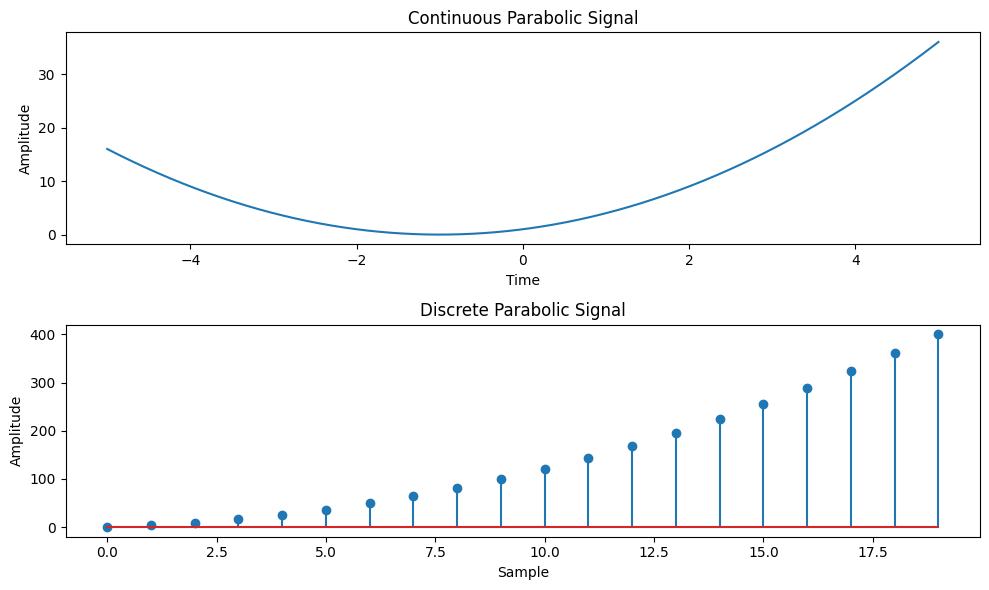
plt.show()

# Save the parabolic signal arrays (optional)

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

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

**Output:**



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

**Code:**

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()

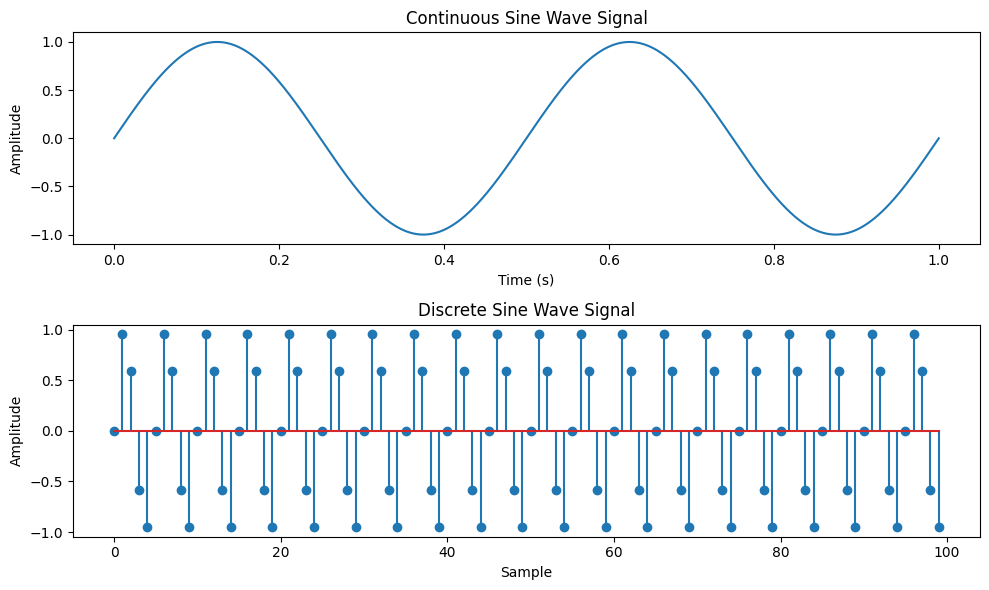
plt.show()

# 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=',')

**Output:**



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

**Code:**

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')

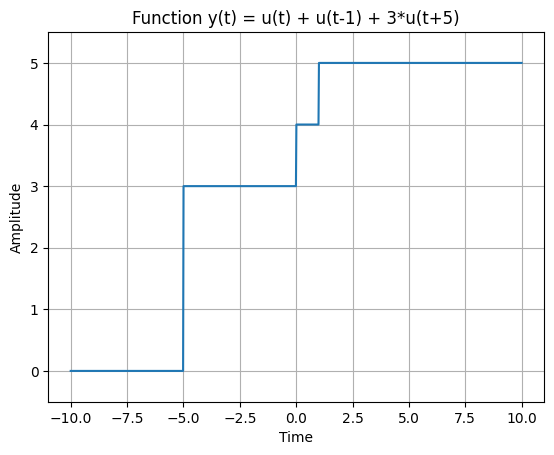
plt.ylabel('Amplitude')

plt.ylim([-0.5, 5.5])

plt.grid(True)

plt.show()

**Output:**



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

**Code:**

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()

**Output:**

