KIT- KalaignarKarunanidhi Institute of Technology

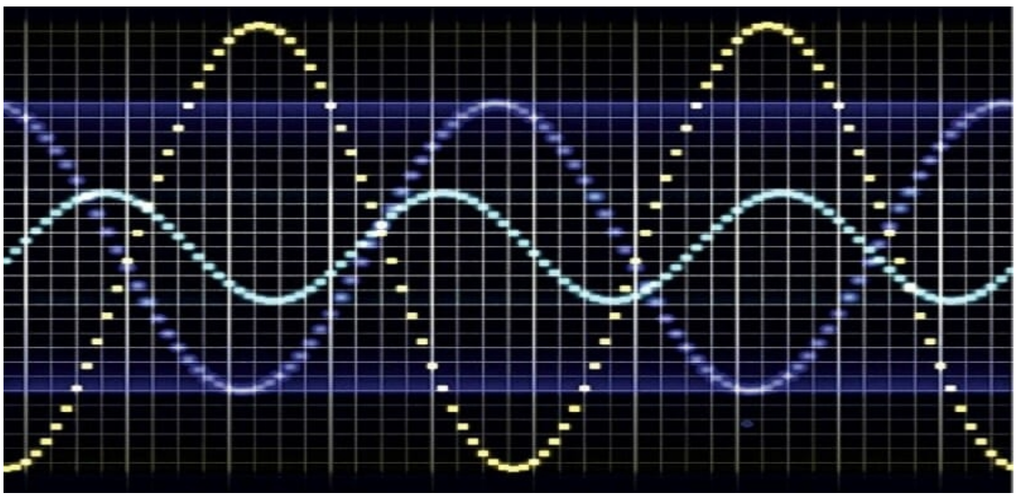
(An Autonomous Institution, Affiliated to Anna University Chennai)

Coimbatore – 641402

Department of Electronics and Communication Engineering

B23ECP303

SIGNALS AND SYSTEMS LABORATORY



LABORATORY RECORD

Academic Year 2024-25 (Odd Semester)

Student's Name :..

Register Number :..

Year / Semester / Sec :..

KIT- KalaignarKarunanidhi Institute of Technology

(An Autonomous Institution, Affiliated to Anna University Chennai)

Coimbatore – 641402

Department of Electronics and Communication Engineering

BONAFIDE CERTIFICATE

Department of

…………………………………………………………………………………………………….

Record Work of ………………………………………………………………………….

Laboratory Certified that this record is the bona fide work done by

Name: ……...……………………………………………………………………………………

Class: ….…………………………………….. Roll No: ………………………………………

Branch…………………………………………………………………………………………..

Place: KIT, CBE **Faculty In-Charge** **HoD / ECE**

University Register No……………………………………………

Submitted for the University Practical Examination held on

……………………………………

INTERNAL EXAMINER **EXTERNAL EXAMINER**

Syllabus

LIST OF EXPERIMENTS

1	Build a program to generate elementary , continuous and discrete signal
2	Generate discrete signal and calculate energy/ Power of a signal.
3	Understand the properties and the different representations of LTI systems
4	Understand the concepts of convolution
5	Analyse the effects of sampling in the time and frequency domains
6	Understand the basics of Uniform Quantization
7	Analysis and Synthesis of Signals using Discrete Fourier Transform
8	Analyse the ECG signal to find heart rate of a patient

Total Instructional hours: 30

Course Outcomes: Students will be able to	
CO1	Examine the mathematical concepts on signals and systems using appropriate tools
CO2	Involve in independent / team work, communicate effectively and engage in lifelong learning

List of Equipment Required; Requirements for a Batch of 30 Students

Sl. No.	Description of the Equipment	Quantity required (Nos.)
1.	Computer	30
2.	Software — Python/ Appropriate tools	-

KIT- KalaignarKarunanidhi Institute of Technology

(An Autonomous Institution, Affiliated to Anna University Chennai)

Coimbatore – 641402

Department of Electronics and Communication Engineering

Rubrics for the Evaluation of Laboratory

Ex. No.	Date	Name of the Experiment	Page Number	Algorithm(20marks)	Output & Conclusion (25Marks)	Inference (10 marks)	Viva Voce (20 marks)	Total (75 marks)	Signature of the Faculty Member
1		Build a program to generate elementary , continuous and discrete signal							
2		Generate discrete signal and calculate energy/ Power of a signal.							
3		Understand the properties and the different representations of LTI systems							
4		Understand the concepts of convolution							
5		Analyse the effects of sampling in the time and frequency domains							
6		Understand the basics of Uniform Quantization							

Ex. No.	Date	Name of the Experiment	Page Number	Algorithm(20marks)	Output & Conclusion (25Marks)	Inference (10 marks)	Viva Voce (20 marks)	Total (75 marks)	Signature of the Faculty Member
7		Analysis and Synthesis of Signals using Discrete Fourier Transform							
8		Analyse the ECG signal to find heart rate of a patient							

Model Exam Marks (25):______ Total (100):______

Signature of the Faculty Member

EX NO : 01	Build a program to generate elementary , continuous and discrete signal
DATE:	

AIM:

To Build a program to generate elementary , continuous and discrete signal

CONTINUOUS TIME - UNIT IMPULSE

ALGORITHM:

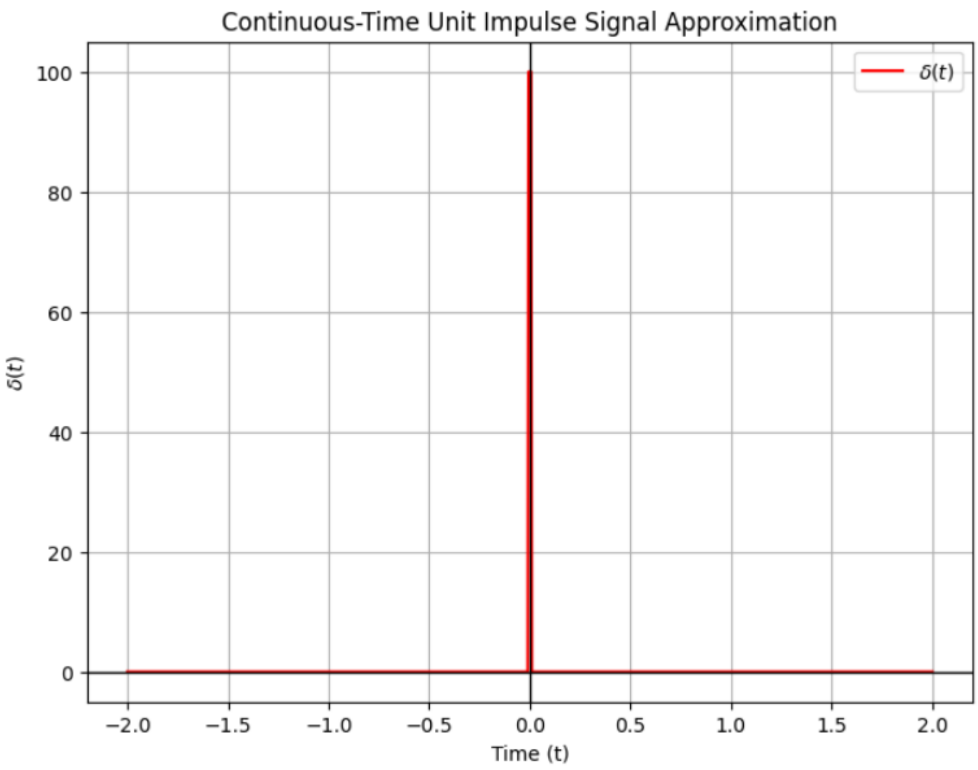
1. Start the program and import the numpy and matplotlib libraries.
2. Define the discrete time range np. arrange
3. Generate a discrete unit impulse signal using signal.unit_impulse(). Specify the length of the impulse based on the length of the time array t and set the index to 'mid' to position the impulse at the center.
4. Plot the signal plt. plot() to create a stem plot of continuous signal x(n) and set x- axis and y-axis.
5. Display the plot.
6. Stop the program

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

# Time range (continuous time)
t = np.linspace(-2, 2, 1000)  # Time from -2 to 2 seconds with 1000 points
epsilon = 0.01  # Small value to simulate the width of the impulse

# Continuous-time unit impulse signal approximation
impulse = np.zeros_like(t)  # Start with zeros
impulse[np.abs(t) < epsilon] = 1 / epsilon  # Set a very narrow spike at t =
0

# Plot the continuous-time unit impulse signal
plt.figure(figsize=(8, 6))
```

```
plt.plot(t, impulse, label=r'$\delta(t)$', color='red')
plt.axhline(0, color='black', linewidth=1)
plt.axvline(0, color='black', linewidth=1)
plt.title("Continuous-Time Unit Impulse Signal Approximation")
plt.xlabel("Time (t)")
plt.ylabel(r"$\delta(t)$")
plt.grid(True)
plt.legend()
plt.show()
```

OUTPUT:



Continuous-Time Unit Impulse Signal Approximation
$\delta(t)$
100
80
60
40
20
0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
Time (t)
$\delta(t)$

DISCRETE TIME - UNIT IMPULSE

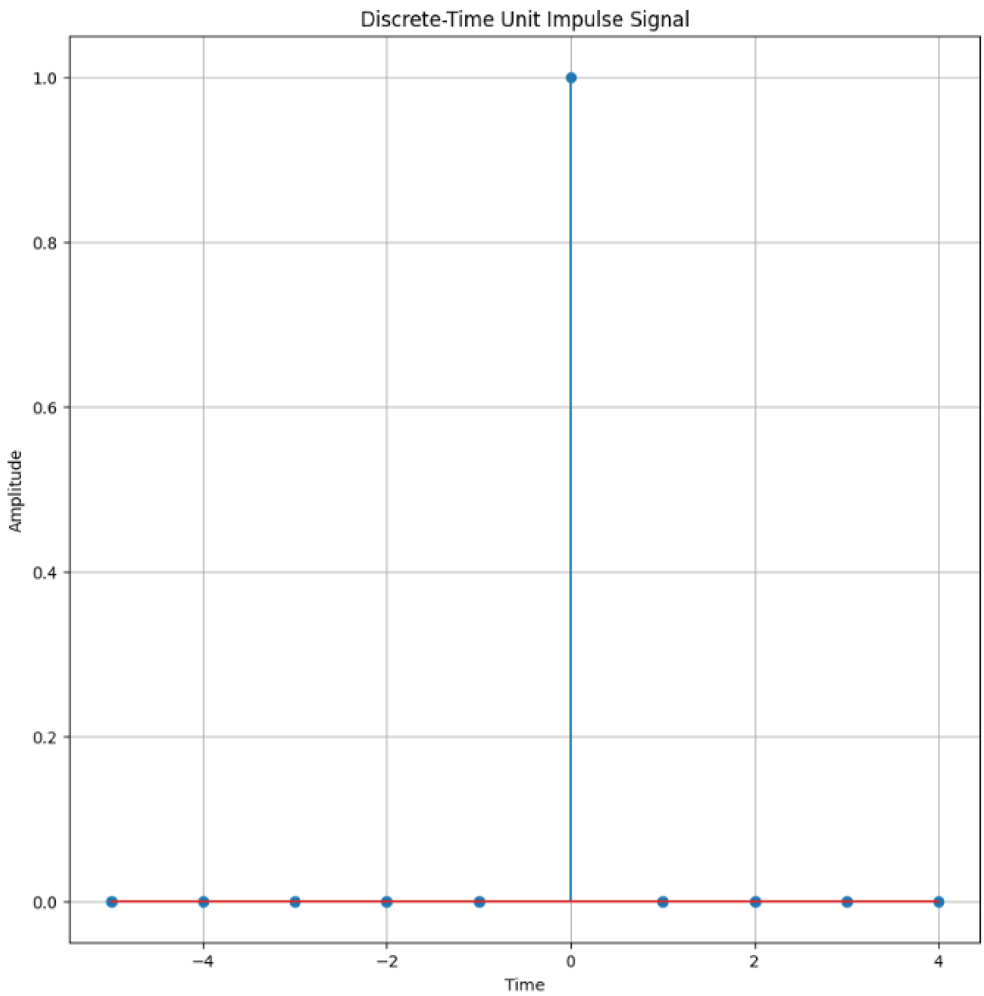
ALGORITHM:

1. Start the program and import the numpy and matplotlib libraries.
2. Define the discrete time range np. arrange
3. Generate a discrete unit impulse signal using signal.unit_impulse().
   Specify the length of the impulse based on the length of the time array t and set the index to 'mid' to position the impulse at the center.
4. Plot the signal plt. stem() to create a stem plot of Discrete signal x(n) and set x- axis and y-axis.
5.Display the plot.
5. Stop the program.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal # Import the signal module from SciPy

# Time values
n = np.arange(-5, 5, 1)  # Adjust range and step as needed

# Unit impulse signal
impulse_signal = signal.unit_impulse(len(n), idx='mid')

# Plot the signal
plt.figure(figsize=(10,10))
plt.stem(n, impulse_signal)
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Discrete-Time Unit Impulse Signal')
plt.grid(True)
plt.show()
```

OUTPUT:



Discrete-Time Unit Impulse Signal
1.0
0.8
0.6
0.4
0.2
0.0
Amplitude
−4
−2
0
2
4
Time

CONTINUOUS TIME - UNIT PARABOLIC

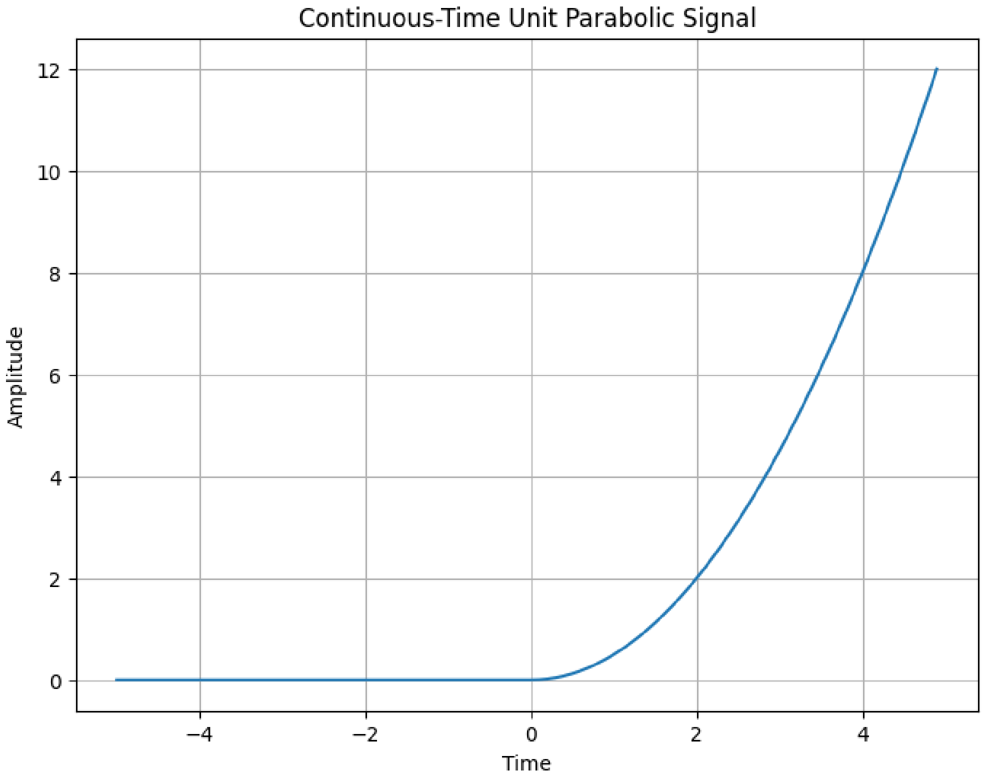
ALGORITHM:

1. Start the program and import the numpy and matplotlib libraries.
2. Define the discrete time range np. arrange
3. Create an array n that spans from -5 to 5 with a step of 0.1.
4.. Generate a continuous unit parabolic signal using the condition for non-negativity.
5. Plot the signal plt. plot() to create a stem plot of continuous signal x(n) and set x- axis and y-axis.
6.Display the plot.
7. Stop the program.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
# Time values
t = np.arange(-5, 5, 0.1)
# Unit parabolic signal
parabolic_signal = 0.5 * t**2 * (t >= 0) # Condition for non-negativity
# Plot
plt.figure(figsize=(8, 6))
plt.plot(t, parabolic_signal)
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Continuous-Time Unit Parabolic Signal')
plt.grid(True)
plt.show()
```

OUTPUT:


Continuous-Time Unit Parabolic Signal
Amplitude
Time
12
10
8
6
4
2
0
−4
−2
0
2
4

DISCRETE TIME - UNIT PARABOLIC

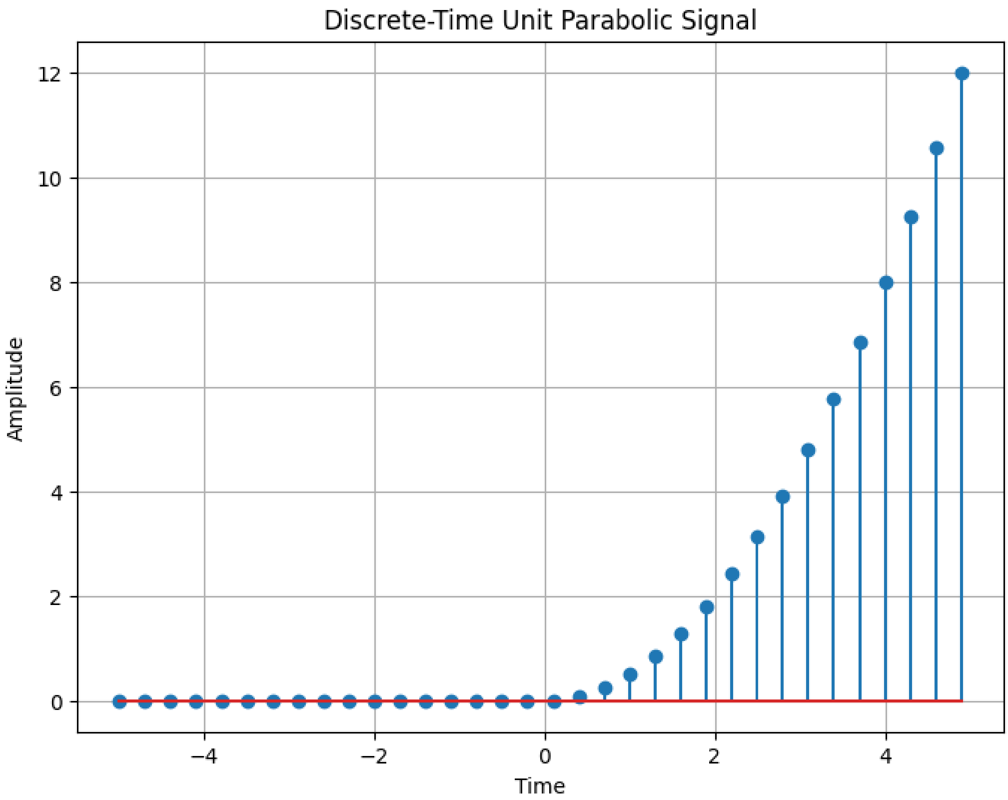
ALGORITHM:

1. Start the program and import the numpy and matplotlib libraries.
2. Define the continuous time range np. arrange
3. Create an array n that spans from -5 to 5 with a step of 0.3.
4.. Generate a continuous unit parabolic signal using the condition for non-negativity.
5. Plot the signal plt. stem() to create a stem plot of discrete signal x(n) and set x- axis and y-axis.
6.Display the plot.
7. Stop the program.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
# Time values
n = np.arange(-5, 5, 0.3)
# Unit parabolic signal
parabolic_signal = 0.5 * n**2 * (n >= 0) # Condition for non-negativity
# Plot
plt.figure(figsize=(8, 6))
plt.stem(n, parabolic_signal)
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Discrete-Time Unit Parabolic Signal')
plt.grid(True)
plt.show()
```

OUTPUT:



Discrete-Time Unit Parabolic Signal
12
10
8
6
4
2
0
Amplitude
−4
−2
0
2
4
Time

CONTINUOUS TIME - UNIT RAMP

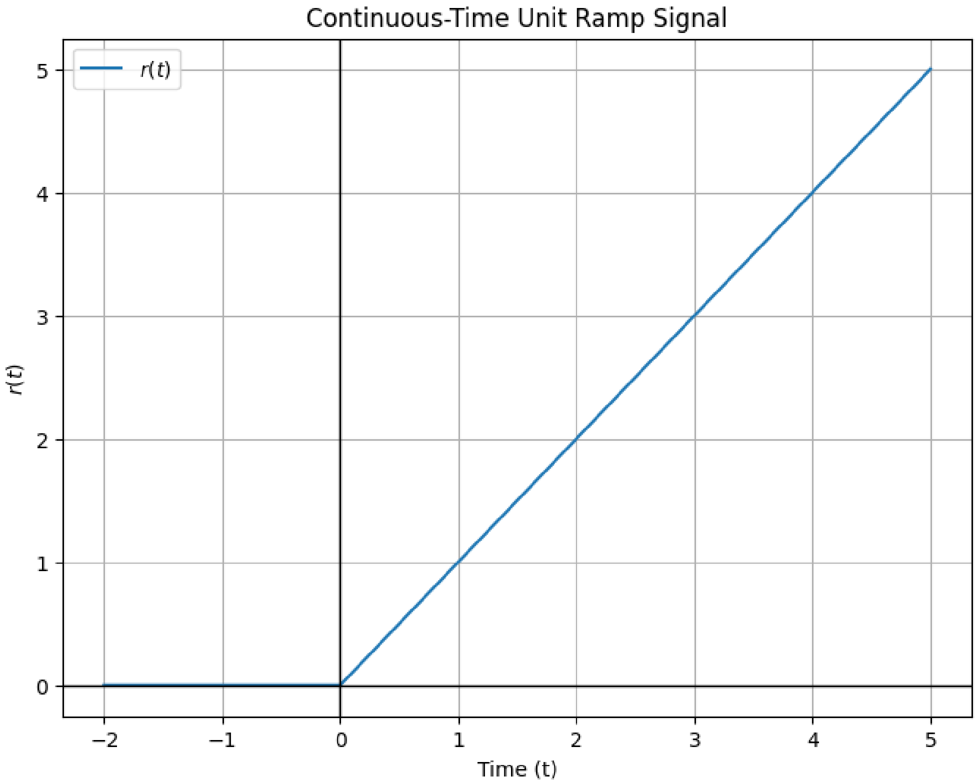
ALGORITHM:

1. Start the program and import the numpy and matplotlib libraries.
2. Define the discrete time range np. arrange
3. Create an array t that spans from -5 to 10 with a step of 0.1.
4. Generate a continuous unit ramp signal using np.where by setting values to 1 when t is greater than or equal to 0, and 0 otherwise.
5. Plot the signal plt. plot() to create a plot of continuous signal x(n) and set x- axis and y-axis.
6.Display the plot.
7. Stop the program.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

# Time range (continuous time)
t = np.linspace(-2, 5, 1000)  # Time from -2 to 5 seconds with 1000 points
# Continuous-time unit ramp signal r(t)
r = np.maximum(0, t)  # r(t) = t for t >= 0, and 0 otherwise
# Plot the continuous-time unit ramp signal
plt.figure(figsize=(8, 6))
plt.plot(t, r, label=r'$r(t)$')
plt.axhline(0, color='black', linewidth=1)
plt.axvline(0, color='black', linewidth=1)
plt.title("Continuous-Time Unit Ramp Signal")
plt.xlabel("Time (t)")
plt.ylabel(r"$r(t)$")
plt.grid(True)
plt.legend()
plt.show()
```

OUTPUT:



DISCRETE TIME - UNIT RAMP

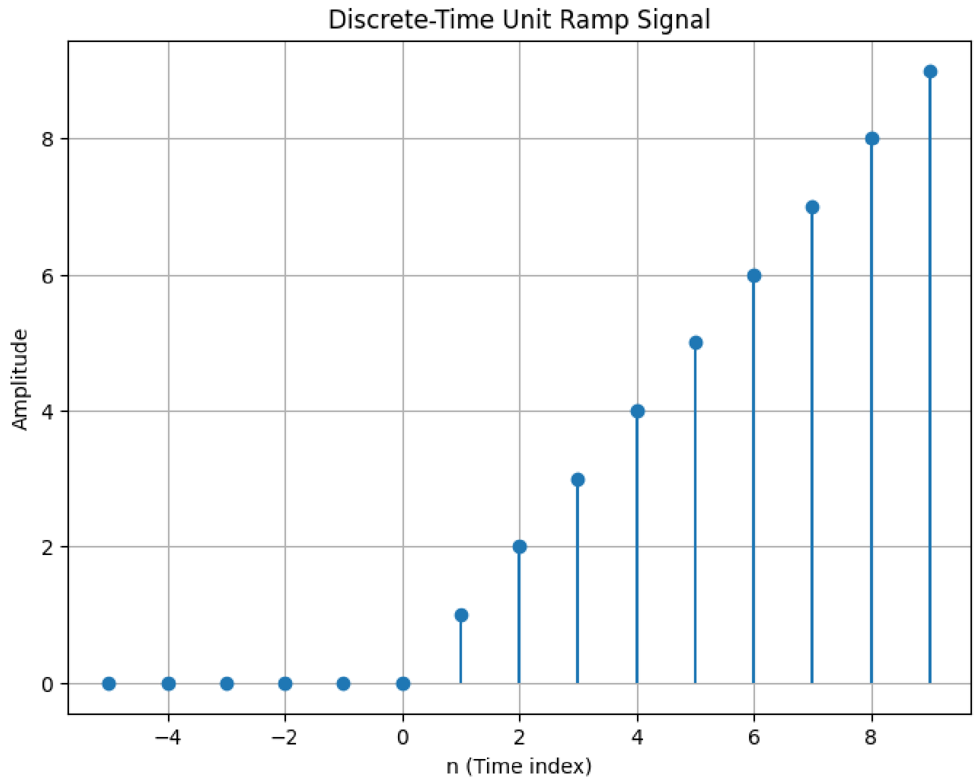
ALGORITHM:

1. Start the program and import the numpy and matplotlib libraries.
2. Define the discrete time range np. arrange
3. Create an array n that spans from -5 to 10.
4. Generate a discrete unit ramp signal using np.where by setting values to 1 when n is greater than or equal to 0, and 0 otherwise.
5. Plot the signal plt. stem() to create a stem plot of digital signal x(n) and set x- axis and y-axis.
6. Display the plot.
7. Stop the program.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

# Time index n (for discrete time signals, it's usually integers)
n = np.arange(-5, 10, 1)  # Time range from -5 to 9 (discrete time indices)

# Unit Ramp signal r[n]
r = np.maximum(0, n)  # r[n] = n for n >= 0, and 0 otherwise

# Plot the discrete-time unit ramp signal
plt.figure(figsize=(8, 6))
plt.stem(n, r, basefmt=" ", use_line_collection=True)
plt.xlabel('n (Time index)')
plt.ylabel('Amplitude')
plt.title('Discrete-Time Unit Ramp Signal')
plt.grid(True)
plt.show()
```

OUTPUT:



Discrete-Time Unit Ramp Signal
Amplitude
n (Time index)
0
2
4
6
8
−4
−2
0
2
4
6
8

CONTINUOUS TIME - UNIT STEP

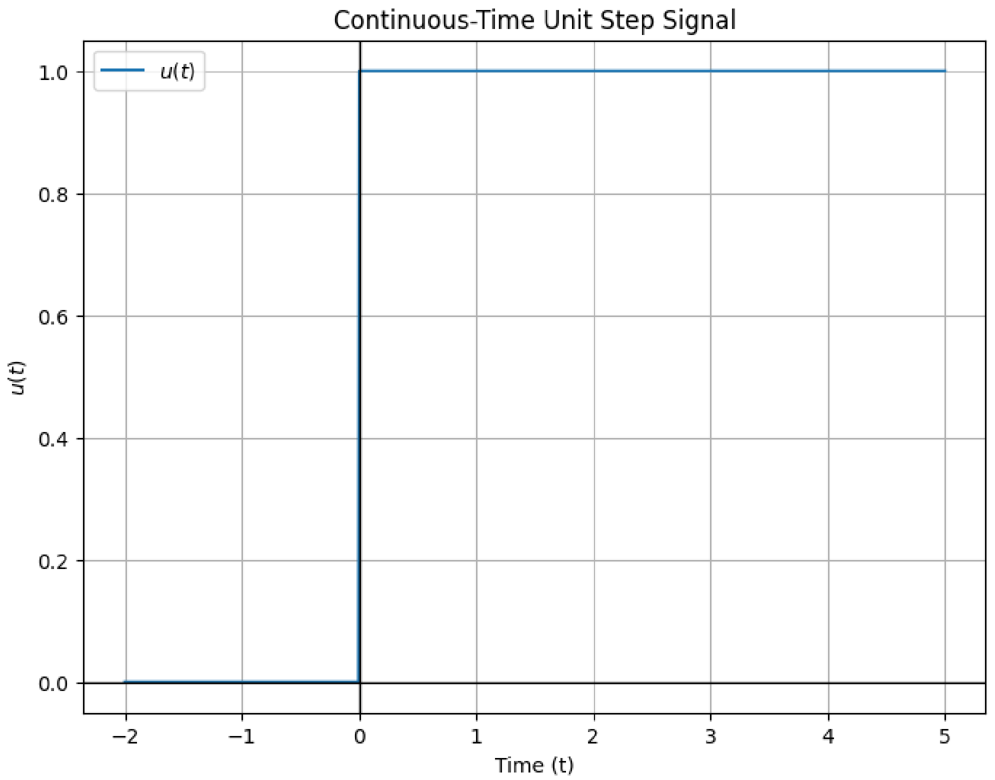
ALGORITHM:

1. Start the program and import the numpy and matplotlib libraries.
2. Define the discrete time range np. arrange
3. Create an array t that ranges from -5 to 10 with increments of 0.1.
4. Generate a continuous unit step signal using np.where by setting values to 1 when t is greater than or equal to 0, and 0 otherwise.
5. Plot the signal plt. plot() to create a plot of continuous signal x(n) and set x- axis and y-axis.
6.Display the plot.
7. Stop the program

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

# Time range (continuous time)
t = np.linspace(-2, 5, 1000)  # Time from -2 to 5 seconds with 1000 points

# Continuous-time unit step signal u(t)
u = np.heaviside(t, 0)  # u(t) = 1 for t >= 0, and 0 for t < 0

# Plot the continuous-time unit step signal
plt.figure(figsize=(8, 6))
plt.plot(t, u, label=r'$u(t)$')
plt.axhline(0, color='black', linewidth=1)
plt.axvline(0, color='black', linewidth=1)
plt.title("Continuous-Time Unit Step Signal")
plt.xlabel("Time (t)")
plt.ylabel(r"$u(t)$")
plt.grid(True)
plt.legend()
plt.show()
```

OUTPUT:



Continuous-Time Unit Step Signal
u(t)
1.0
0.8
0.6
0.4
0.2
0.0
−2
−1
0
1
2
3
4
5
Time (t)

DISCRETE TIME - UNIT STEP

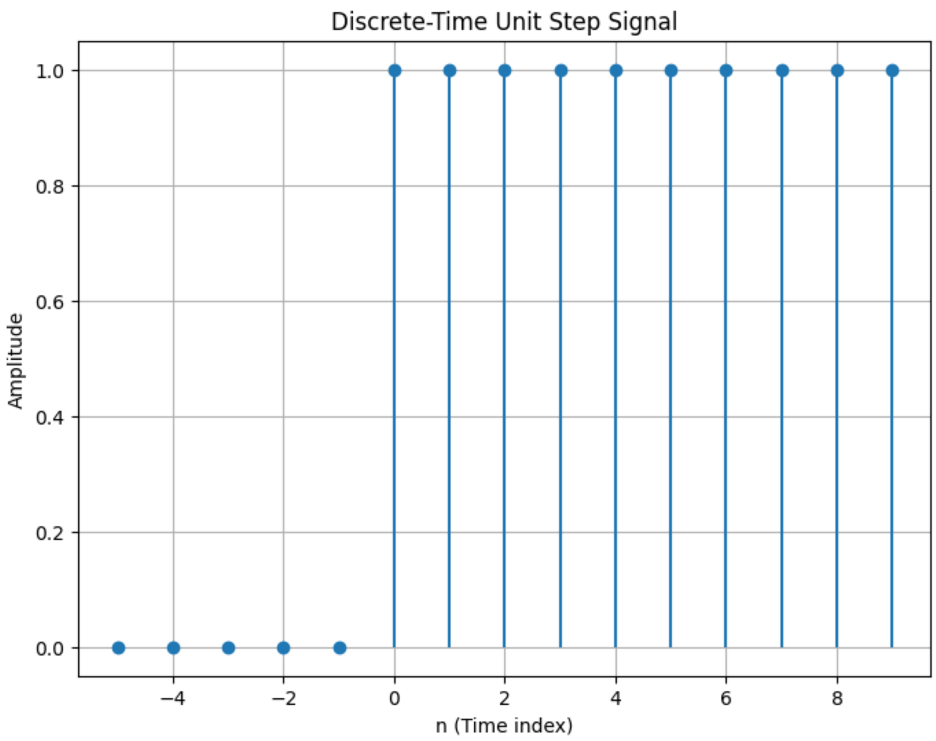
ALGORITHM:

1. Start the program and import the numpy and matplotlib libraries.
2. Define the discrete time range np. arrange
3. Create an array n that ranges from -5 to 10 with increments of 1.
4. Generate a discrete unit step signal using np.where by setting values to 1 when n is greater than or equal to 0, and 0 otherwise.
5. Plot the signal plt. stem() to create a stem plot of discrete signal x(n) and set x- axis and y-axis.
6. Display the plot.
7. Stop the program.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

# Time index n (discrete time)
n = np.arange(-5, 10, 1)  # Time range from -5 to 9 (discrete indices)

# Discrete-time unit step signal u[n]
u = np.where(n >= 0, 1, 0)  # u[n] = 1 for n >= 0, and 0 for n < 0

# Plot the discrete-time unit step signal
plt.figure(figsize=(8, 6))
plt.stem(n, u, basefmt=" ", use_line_collection=True)
plt.xlabel('n (Time index)')
plt.ylabel('Amplitude')
plt.title('Discrete-Time Unit Step Signal')
plt.grid(True)
plt.show()
```

OUTPUT:



INFERENCE:

RESULT:

EX NO : 02	Generate discrete signal and calculate energy/ Power of a signal
DATE:	

AIM:

To Generate discrete signal and calculate energy/ Power of a signal

ALGORITHM:

1. Start the program and import the numpy and matplotlib libraries.
2. Define the discrete time range np. arrange
3. Define the signal as n*u(n).
4. Plot the signal plt. stem() to create a stem plot of discrete signal x(n) and set x- axis and y-axis.
5. Calculate the energy of the signal.
6. Calculate the average power of the signal.
7. Classify the signal,

    7.1 if the energy is finite (energy<np.inf) and the average power is greater than 1, classify it as a power signal.

    7.2 if the energy is finite and the avg power is 0, classify it as an energy signal

    7.3 otherwise, classify it as neither a power signal nor an energy signal.

8. Print the results
9. Stop the program.

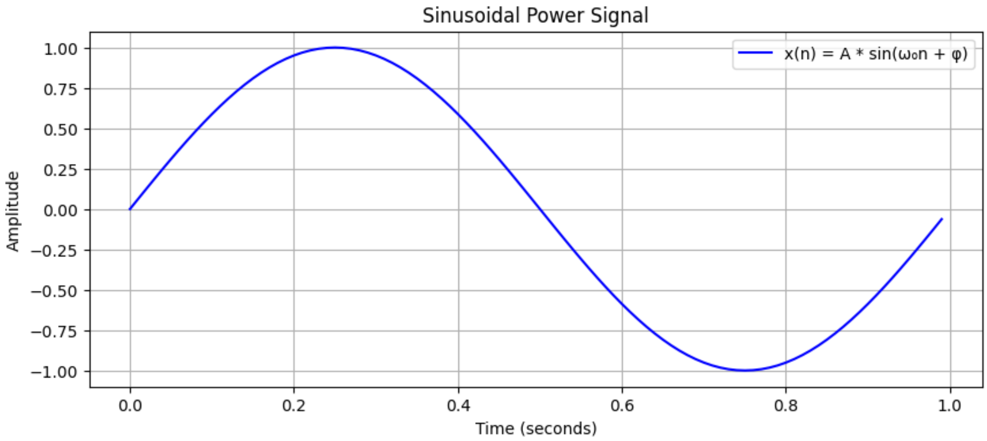
ENERGY SIGNAL

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Parameters
A = 1         # Amplitude
ϕ = 0         # Phase shift
n = np.arange(0, 1, 0.01)  # Time index (discrete)

# Generate the sinusoidal signal
x_n = A * np.sin(2 * np.pi * n + ϕ)

# Plotting
plt.figure(figsize=(10, 4))
plt.plot(n, x_n, label='x(n) = A * sin(ω₀n + ϕ)', color='blue')
plt.title('Sinusoidal Power Signal')
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.grid()
plt.legend()
plt.show()

# Calculate the energy of the signal (finite time interval)
energy = np.sum(x_n**2)  # Energy over the finite time interval

# Calculate the average power of the signal
average_power = np.mean(x_n**2)  # Average power over the time interval

# Classification
if average_power > 0:
    classification = "The signal is a power signal."
else:
    classification = "The signal is an energy signal."

# Print results
print(f"Energy of the signal: {energy}")  # Energy over the finite interval
print(f"Average Power of the signal: {round(average_power, 2)}")  # Print average power
print(classification)
```

OUTPUT:




```
Energy of the signal: 50.0
Average Power of the signal: 0.5
The signal is a power signal.
```

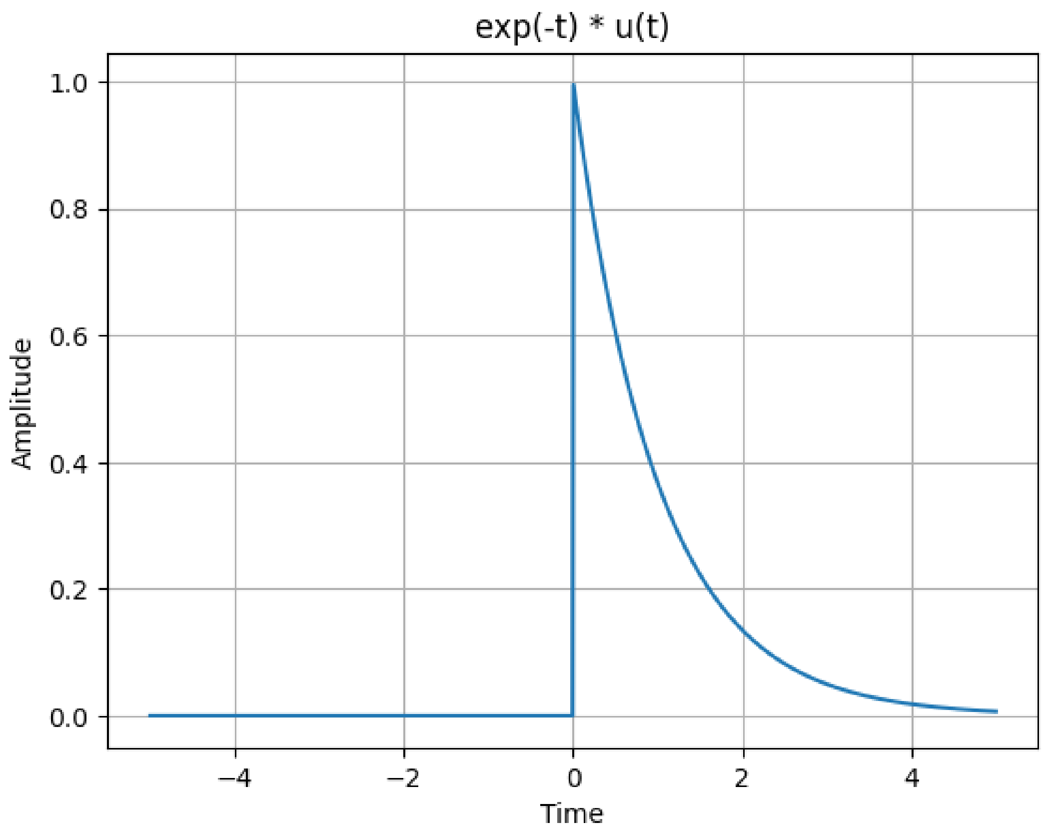
POWER SIGNAL

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

# Define the time range
t = np.linspace(-5, 5, 1000)  # Increased resolution for better visualization

# Define the signal x(t) = e^(-t) * u(t)
signal = np.exp(-t) * np.where(t >= 0, 1, 0)

# Plot the signal
plt.plot(t, signal)
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('exp(-t) * u(t)')
plt.grid(True)
plt.show()

# Calculate the energy of the signal
energy = np.trapz(signal**2, t)
```

```
# Calculate the power of the signal (over a long enough interval)
T = 1000  # Large enough T for power calculation
power = (1 / (2 * T)) * np.trapz(signal**2, t)

# Classification based on energy and average power
if energy < np.inf and power == 0:
    classification = "The signal is a power signal."
elif energy < np.inf and power > 0:
    classification = "The signal is an energy signal."
else:
    classification = "The signal is neither energy nor a power signal."

# Print results
print(f"Energy of the signal: {round(energy, 1)}")  # Energy is conceptually finite
print(f"Average Power of the signal: {round(power, 2)}")  # Print average power
print(classification)
```

OUTPUT:




```
Energy of the signal: 0.5
Average Power of the signal: 0.0
The signal is an energy signal.
```

INFERENCE:

RESULT:

EX NO : 03	**Understand the properties and the different representations of LTI systems**
DATE:	

AIM:

To Understand the properties and the different representations of LTI systems

LINEARITY PROPERTY

ALGORITHM:

1.Start the program
2.Import NumPy for numerical operations and Matplotlib for plotting.
3.Create a function system(x) that returns the square of the input x.
4.Initialize two input arrays, x1 and x2.
5.Calculate the outputs y1 and y2 by passing x1 and x2 through the system function.
6.Compute a new input array by adding x1 and x2, and find its output y3.
7.Verify if the system is linear by checking if y3 equals y1 + y2.
8.Output the sums of y1 and y2, the value of y3, and whether the system is linear.
9.Create a plot to visualize the relationship between inputs and outputs, including labels and a grid.
10.Stop the program.

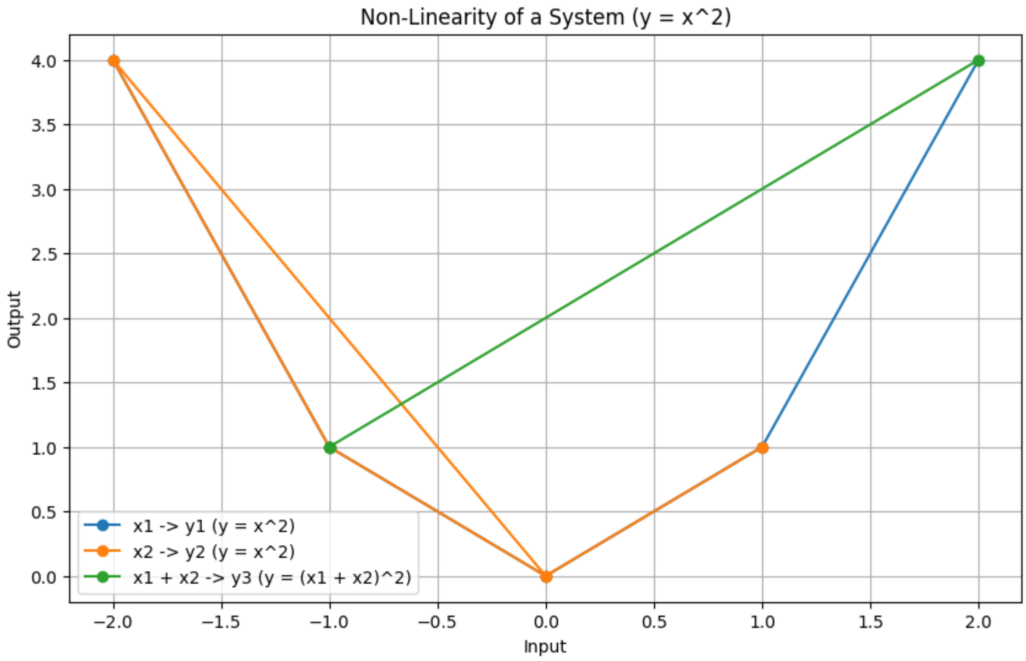
PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

# Define a system (example: squaring the input)
def system(x):
    return x**2

# Input signals
x1 = np.array([-2, -1, 0, 1, 2])
x2 = np.array([1, 0, -1, -2, 0])

# Outputs for individual inputs
```

```
y1 = system(x1)
y2 = system(x2)

# Combined input and output
y3 = system(x1 + x2)  # Applying system to the sum of x1 and x2

# Check for linearity: y3 should be equal to y1 + y2 for linearity
print("y1 + y2:", y1 + y2)
print("y3:", y3)
print("Is the system linear?", np.array_equal(y3, y1 + y2))

# Plotting for visualization
plt.figure(figsize=(10, 6))
plt.plot(x1, y1, 'o-', label='x1 -> y1 (y = x^2)')
plt.plot(x2, y2, 'o-', label='x2 -> y2 (y = x^2)')
plt.plot(x1 + x2, y3, 'o-', label='x1 + x2 -> y3 (y = (x1 + x2)^2)')
plt.xlabel('Input')
plt.ylabel('Output')
plt.title('Non-Linearity of a System (y = x^2)')
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT:

```
y1 + y2: [5 1 1 5 4]
y3: [1 1 1 1 4]
Is the system linear? False
```



TIME SHIFTING PROPERTY

ALGORITHM:

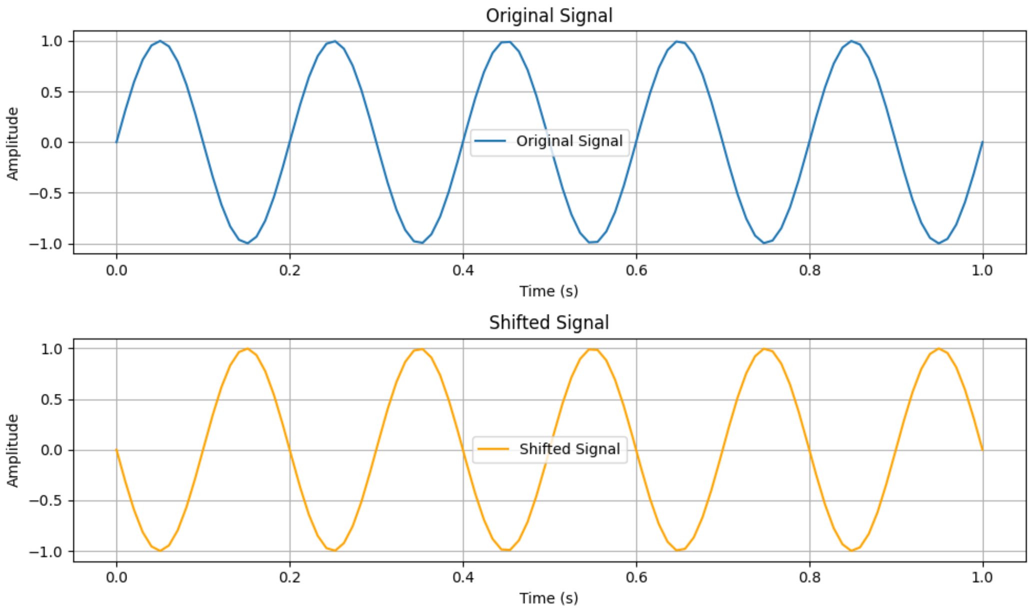
1.Start the program.
2.Import NumPy for numerical operations and Matplotlib for plotting.
3.Create an array t using np.linspace to represent time from 0 to 1 second, divided into 100 points.
4.Create a sine wave signal x with a frequency of 5 Hz using the formula $x = \sin(2\pi ft)$.
5.Set a variable t0 to define the time delay.
6.Calculate the time-shifted signal x_shifted by adjusting the time array in the sine function.
7.Initialize a plot with a specified figure size.
8.Create the first subplot for the original sine wave and the second subplot for the time-shifted sine wave.
9.Stop the program.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

# Generate a time domain signal (example: sine wave)
t = np.linspace(0, 1, 100)  # Time from 0 to 1 second
x = np.sin(2 * np.pi * 5 * t)  # Sine wave with frequency 5 Hz

# Time shift the signal
t0 = 0.10  # Time delay in seconds (0.1 seconds for a visible shift)
t_shifted = t - t0  # Shift time axis by t0
x_shifted = np.sin(2 * np.pi * 5 * t_shifted)  # Apply the sine wave with
shifted time

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

# Original signal plot
plt.subplot(2, 1, 1)
plt.plot(t, x, label='Original Signal')
plt.xlabel('Time (s)')
```

```
plt.ylabel('Amplitude')
plt.title('Original Signal')
plt.legend()
plt.grid(True)

# Shifted signal plot
plt.subplot(2, 1, 2)
plt.plot(t, x_shifted, label='Shifted Signal', color='orange')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Shifted Signal')
plt.legend()
plt.grid(True)

plt.tight_layout()  # Ensure that the subplots don't overlap
plt.show()
```

OUTPUT:



Original Signal
Original Signal
Amplitude
1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0
Time (s)
Shifted Signal
Shifted Signal
Amplitude
1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0
Time (s)

TIME REVERSAL PROPERTY

ALGORITHM:

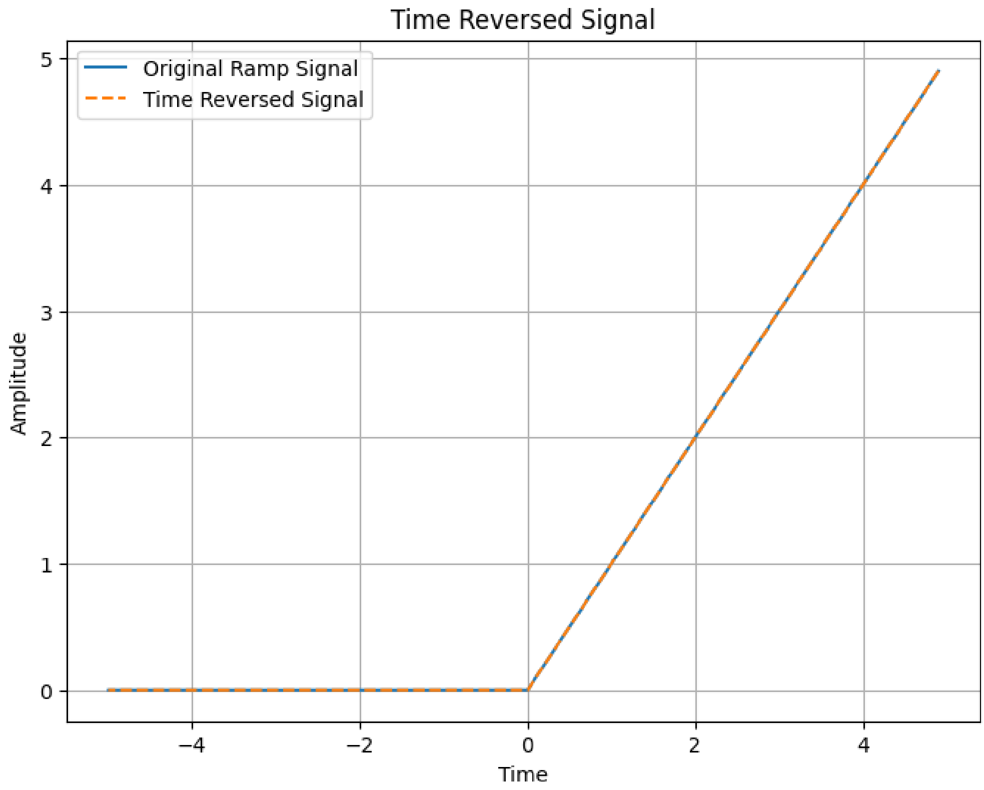
1.Start the program
2. Import NumPy for numerical calculations and Matplotlib for plotting.
3.Create an array t that spans from -5 to 5 with a step of 0.1.
4.Create ramp_signal using np.where, setting values to t when t is greater than or equal to 0, and 0 otherwise.
5.Use np.flip() to reverse the ramp_signal, storing the result in time_reversed_signal.
6.Initialize a plot with a specified figure size.
7. Plot the ramp_signal against the time array t and overlay the time_reversed_signal on the same plot.
9.Add labels for the x-axis and y-axis, as well as a title and grid lines.
10.Stop the program.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

# Time values
t = np.arange(-5, 5, 0.1)  # Adjust range and step as needed

# Unit ramp signal
ramp_signal = np.where(t >= 0, t, 0)

# Time reversal using np.flip()
time_reversed_signal = np.flip(ramp_signal)  # Use np.flip() to reverse the
signal

# Create reversed time vector to match the flipped signal
t_reversed = np.flip(t)  # Reverse the time vector

# Plot the signals
plt.figure(figsize=(8, 6))

# Plot original ramp signal
plt.plot(t, ramp_signal, label='Original Ramp Signal')
```

```
# Plot time-reversed signal
plt.plot(t_reversed, time_reversed_signal, label='Time Reversed Signal',
linestyle='--')

plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Time Reversed Signal')
plt.grid(True)
plt.legend()
plt.show()
```

OUTPUT:



Time Reversed Signal
Original Ramp Signal
Time Reversed Signal
Amplitude
Time
5
4
3
2
1
0
−4
−2
0
2
4

FREQUENCY SHIFTING PROPERTY

ALGORITHM:

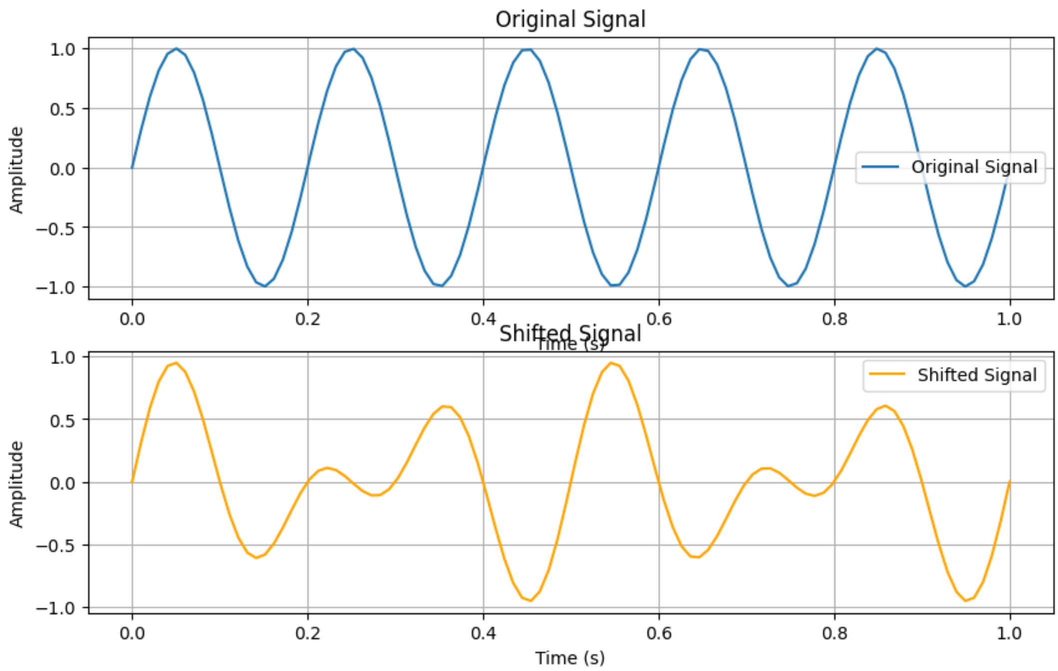
1. Start the program.
2. Import NumPy for numerical operations and Matplotlib for plotting.
3. Create an array t using np.linspace to represent time from 0 to 1 second, divided into 100 points.
4. Create a sine wave signal x with a frequency of 5 Hz using the formula x= sin(2πft).
5. Set a variable f_shift to define the frequency shift.
6. Apply frequency shifting by multiplying the original signal x with a complex exponential using the formula x_shifted=x* e^sin(2πf_shift t)
7. Initialize a plot with a specified figure size.
8. Create the first subplot for the original sine wave and Create the second subplot for the frequency-shifted sine wave.
9. Stop the program.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

# Generate a time domain signal (example: sine wave)
t = np.linspace(0, 1, 100)
x = np.sin(2 * np.pi * 5 * t)

# Frequency to shift by
f_shift = 1 # Shift by 1 Hz

# Frequency shifting using complex exponential multiplication
x_shifted = x * np.exp(2j * np.pi * f_shift * t)

# Plotting
plt.figure(figsize=(10, 6))
```

```
plt.subplot(2, 1, 1)
plt.plot(t, x, label='Original Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Original Signal')
plt.legend()
plt.grid(True)

plt.subplot(2, 1, 2)
plt.plot(t, x_shifted, label='Shifted Signal', color='orange')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Shifted Signal')
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT:


Original Signal
Amplitude
1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0
Time (s)
Original Signal
Shifted Signal
Amplitude
1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0
Time (s)
Shifted Signal

CONJUGATE PROPERTY

ALGORITHM:

1. Start the program.
2. Import the NumPy library for numerical operations.
3. Create a complex number z using a real part and an imaginary part (e.g., 3 + 4j).
4. Use np.conjugate(z) to calculate the conjugate of the complex number, storing it in z_conjugate.
5. Output the original complex number and its conjugate using print statements.
6. Stop the program

PROGRAM:

```
import numpy as np

# Create a complex number
z = 3 + 4j

# Calculate the conjugate
z_conjugate = np.conjugate(z)

# Print the results
print("Original complex number:", z)
print("Conjugate:", z_conjugate)
```

OUTPUT:

Original complex number: **(3+4j)**
Conjugate: **(3-4j)**

INFERENCE:

RESULT:

EX NO : 04	Understand the concepts of convolution
DATE:	

AIM:

To Understand the concepts of convolution

ALGORITHM:

1: Take the two sequences you want to convolve (the input sequence and the filter or impulse response).
2: Find out how long each sequence is.
3: Calculate the length of the final result by adding the lengths of the two sequences and subtracting 1.
4: Create a new matrix where each row will hold a shifted version of the second sequence, and the matrix will have enough rows to accommodate all shifts.
5: Fill the matrix by placing values from the second sequence in each row, making sure the values are aligned properly as you move down the rows.
6: Convert the first sequence into a column (like a vertical list).
7: Multiply the matrix you created by the column from the first sequence to calculate the convolution.
8: Take the result and flatten it back into a single list, which will be your final answer.

PROGRAM:

```
import numpy as np

def linear_convolution_matrix_method(x, h):
    # Length of the input sequences
    M = len(x)
    L = len(h)

    # Length of the result (convolution output)
```

```
    N = M + L - 1

    # Create a Toeplitz matrix for h[n]
    H = np.zeros((N, M))

    # Fill the Toeplitz matrix with appropriate values from h[n]
    for i in range(N):
        for j in range(M):
            if i - j >= 0 and i - j < L:
                H[i, j] = h[i - j]

    # Convert x[n] to a column vector
    x_col = np.reshape(x, (M, 1))

    # Perform the matrix multiplication H * x to get the convolution result
    y = np.dot(H, x_col)

    # Flatten the result to a 1D array
    return y.flatten()

# Example usage
x = [1, 2, 3]  # Input sequence
h = [4, 5]     # Impulse response

# Perform convolution
y = linear_convolution_matrix_method(x, h)

# Print the result
print("Convolution Result:", y)
```

OUTPUT:

Convolution Result: [4. 13. 22. 15.]

INFERENCE:

RESULT:

EX NO : 05	Analyse the effects of sampling in the time and frequency domains
DATE:	

AIM:

To Analyse the effects of sampling in the time and frequency domains

ALGORITHM:

1: Define signal parameters (frequencies, amplitudes, duration).
2: Select different sampling rates to analyze.
3: For each sampling rate:
Generate time vector based on sampling rate.
Sum sine waves with given frequencies and amplitudes to form the signal.
4: Plot the signal in the time domain for each sampling rate.
5: Compute and plot the frequency spectrum using FFT:
Take the positive frequencies and plot their magnitudes.
6: Adjust plot layout and display the results.

PROGRAM:

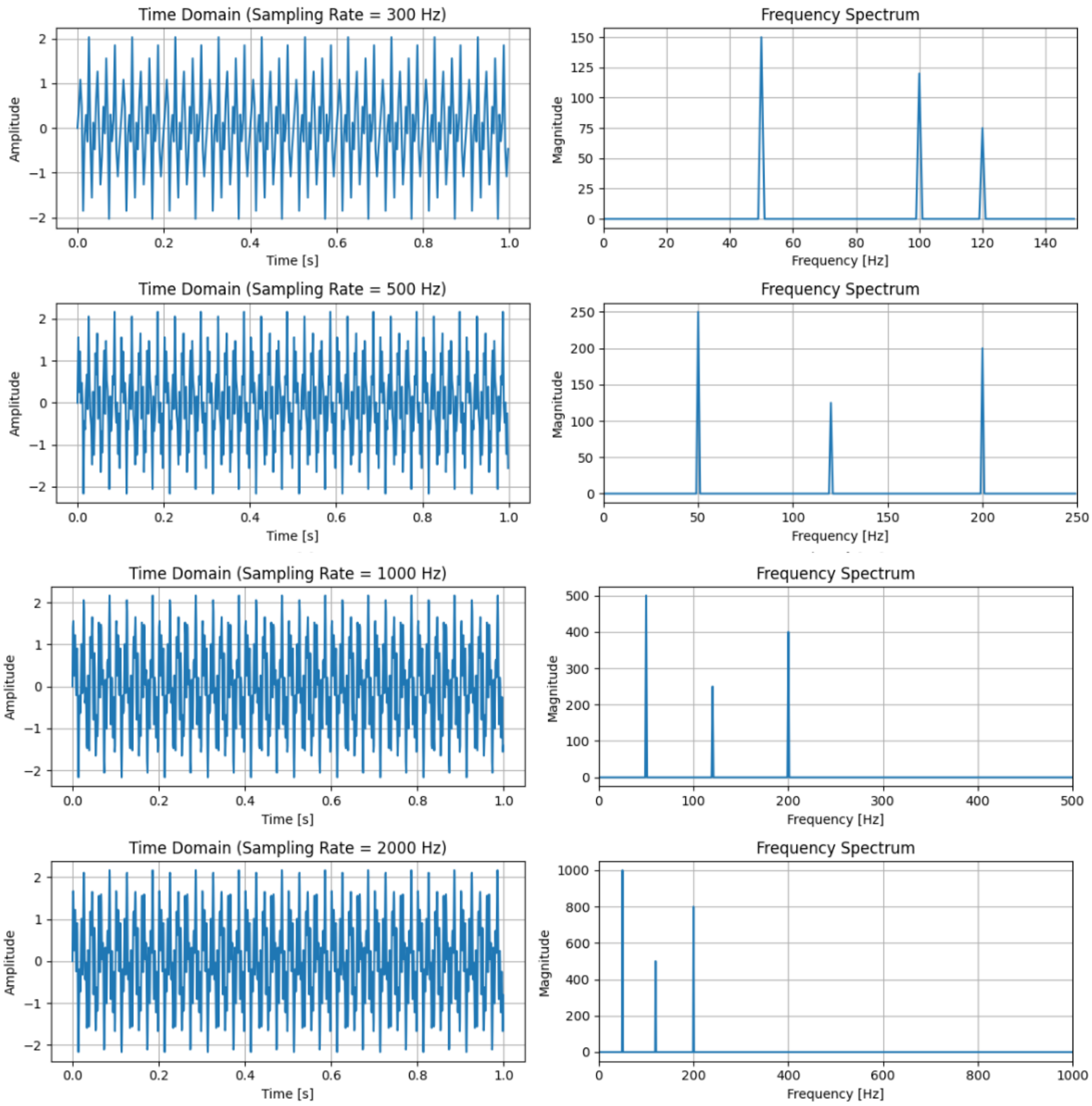
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, fftfreq

# Function to generate a continuous signal (sum of sine waves)
def generate_continuous_signal(frequencies, amplitudes, duration,
sampling_rate=1000):
    """
    Generates a continuous signal (sum of sine waves) sampled at a given
sampling rate.

    :param frequencies: List of frequencies of the sine waves (Hz).
```

```
    :param amplitudes: List of amplitudes corresponding to the sine waves.
    :param duration: Duration of the signal (seconds).
    :param sampling_rate: Sampling rate (samples per second).

    :return: time (discrete), signal (discrete)
    """
    time = np.linspace(0, duration, int(sampling_rate * duration),
endpoint=False)  # Continuous time
    signal = np.zeros_like(time)  # Initialize signal

    # Sum of sine waves with different frequencies and amplitudes
    for freq, amp in zip(frequencies, amplitudes):
        signal += amp * np.sin(2 * np.pi * freq * time)

    return time, signal

# Function to compute the FFT and plot the frequency spectrum
def plot_frequency_spectrum(signal, sampling_rate):
    """
    Compute and plot the frequency spectrum of the signal using FFT.

    :param signal: The discrete signal.
    :param sampling_rate: Sampling rate (samples per second).
    """
    # Compute FFT
    N = len(signal)
    freqs = fftfreq(N, 1/sampling_rate)  # Frequency axis
    fft_values = fft(signal)  # FFT of the signal

    # Only take the positive half of the spectrum (real signals are
symmetric)
    half_N = N // 2
    freqs = freqs[:half_N]
    fft_values = fft_values[:half_N]

    # Plot the magnitude of the frequency spectrum
    plt.plot(freqs, np.abs(fft_values))
    plt.title("Frequency Spectrum")
    plt.xlabel("Frequency [Hz]")
    plt.ylabel("Magnitude")
```

```
    plt.grid(True)
    plt.xlim(0, sampling_rate / 2)  # Nyquist frequency limit

# Function to plot the signal in the time domain
def plot_time_domain(time, signal, title="Signal in Time Domain"):
    """
    Plot the signal in the time domain.

    :param time: Time vector for the signal.
    :param signal: The signal in the time domain.
    :param title: Title for the plot.
    """
    plt.plot(time, signal)
    plt.title(title)
    plt.xlabel("Time [s]")
    plt.ylabel("Amplitude")
    plt.grid(True)

# Main Program
if __name__ == "__main__":
    # Signal parameters
    frequencies = [50, 120, 200]  # Frequencies of the sine waves (Hz)
    amplitudes = [1, 0.5, 0.8]  # Amplitudes of the sine waves
    duration = 1  # Duration of the signal (seconds)

    # Different sampling rates to demonstrate the effect of sampling
    sampling_rates = [300, 500, 1000, 2000]  # Different sampling rates

    # Create the figure for plotting
    plt.figure(figsize=(12, 12))

    # Analyze the effect of different sampling rates
    for i, sampling_rate in enumerate(sampling_rates, start=1):
        # Generate the continuous signal with the given sampling rate
        time, signal = generate_continuous_signal(frequencies, amplitudes,
duration, sampling_rate)

        # Plot the signal in the time domain
        plt.subplot(4, 2, 2*i-1)
```

```
        plot_time_domain(time, signal, title=f"Time Domain (Sampling Rate
= {sampling_rate} Hz)")

        # Plot the frequency spectrum of the signal
        plt.subplot(4, 2, 2*i)
        plot_frequency_spectrum(signal, sampling_rate)

    # Adjust the layout
    plt.tight_layout()
plt.show()
```

OUTPUT:



INFERENCE:

RESULT:

EX NO : 06	Understand the basics of Uniform Quantization
DATE:	

AIM:

To Understand the basics of Uniform Quantization

ALGORITHM:

1: Define the input signal (e.g., a sine wave) and the number of quantization levels.
2: Find the minimum and maximum values of the signal.
3: Calculate the quantization step size using the formula:
4: Quantize the signal:
  Subtract the min value from the signal.
  Divide by the step size and round the result.
  Multiply by the step size and add the min value back.
5: Plot the original signal and the quantized signal.
6: Display the quantization step size.

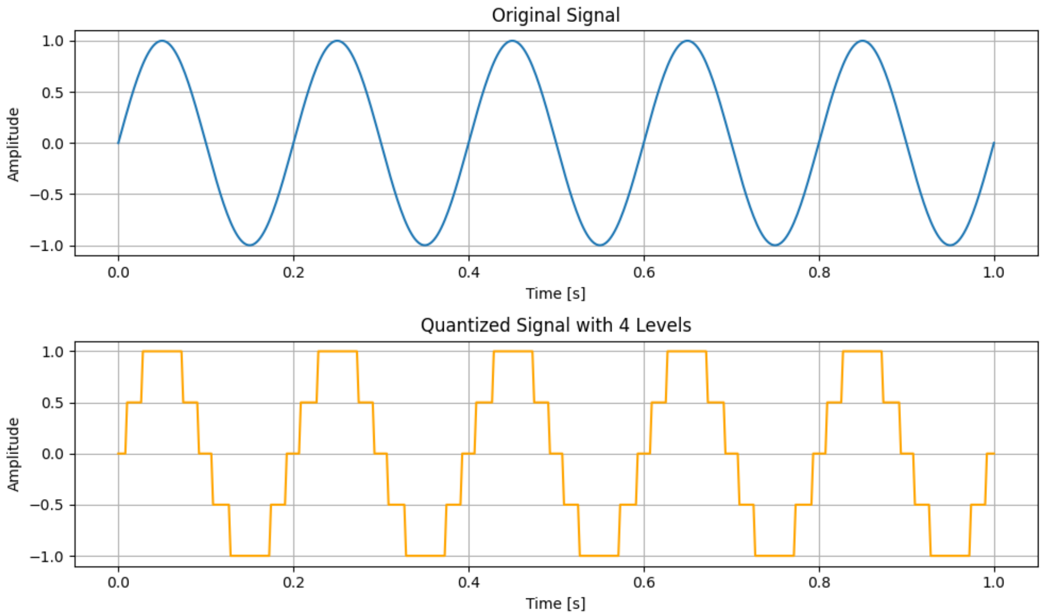
PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt

def uniform_quantization(signal, num_levels):
    """
    Perform uniform quantization on a signal.

    :param signal: The input signal (numpy array).
    :param num_levels: Number of quantization levels.
    :return: Quantized signal (numpy array).
```

```
    """
    # Find the min and max values of the signal
    min_val = np.min(signal)
    max_val = np.max(signal)

    # Calculate the quantization step size
    step_size = (max_val - min_val) / num_levels

    # Quantize the signal
    quantized_signal = np.round((signal - min_val) / step_size) * step_size
+ min_val

    return quantized_signal, step_size

# Example usage:
if __name__ == "__main__":
    # Generate a continuous signal (e.g., sine wave)
    time = np.linspace(0, 1, 500)  # Time vector
    original_signal = np.sin(2 * np.pi * 5 * time)  # A sine wave with
frequency of 5 Hz

    # Set the number of quantization levels (e.g., 4 levels, 8 levels, etc.)
    num_levels = 4

    # Perform uniform quantization
    quantized_signal, step_size = uniform_quantization(original_signal,
num_levels)

    # Plot the original and quantized signals
    plt.figure(figsize=(10, 6))

    # Plot the original signal
    plt.subplot(2, 1, 1)
    plt.plot(time, original_signal, label='Original Signal')
    plt.title('Original Signal')
    plt.xlabel('Time [s]')
    plt.ylabel('Amplitude')
    plt.grid(True)

    # Plot the quantized signal
```

```
    plt.subplot(2, 1, 2)
    plt.plot(time, quantized_signal, label=f'Quantized Signal (Levels =
{num_levels})', color='orange')
    plt.title(f'Quantized Signal with {num_levels} Levels')
    plt.xlabel('Time [s]')
    plt.ylabel('Amplitude')
    plt.grid(True)

    # Show the plots
    plt.tight_layout()
    plt.show()

    print(f"Quantization step size: {step_size}")
```

OUTPUT:



INFERENCE:

RESULT:

EX NO : 07	Analysis and Synthesis of Signals using Discrete Fourier Transform
DATE:	

AIM:

To Analysis and Synthesis of Signals using Discrete Fourier Transform

ALGORITHM:

1: Define signal parameters (sampling rate, duration, and frequencies).
2: Generate the time vector and create the signal (sum of sine waves).
3: Perform DFT on the signal using FFT to get the frequency-domain coefficients.
4: Plot the original signal, magnitude spectrum, and phase spectrum.
5: Perform IDFT using IFFT to reconstruct the signal from DFT coefficients.
6: Plot the original and reconstructed signals for comparison.

PROGRAM:

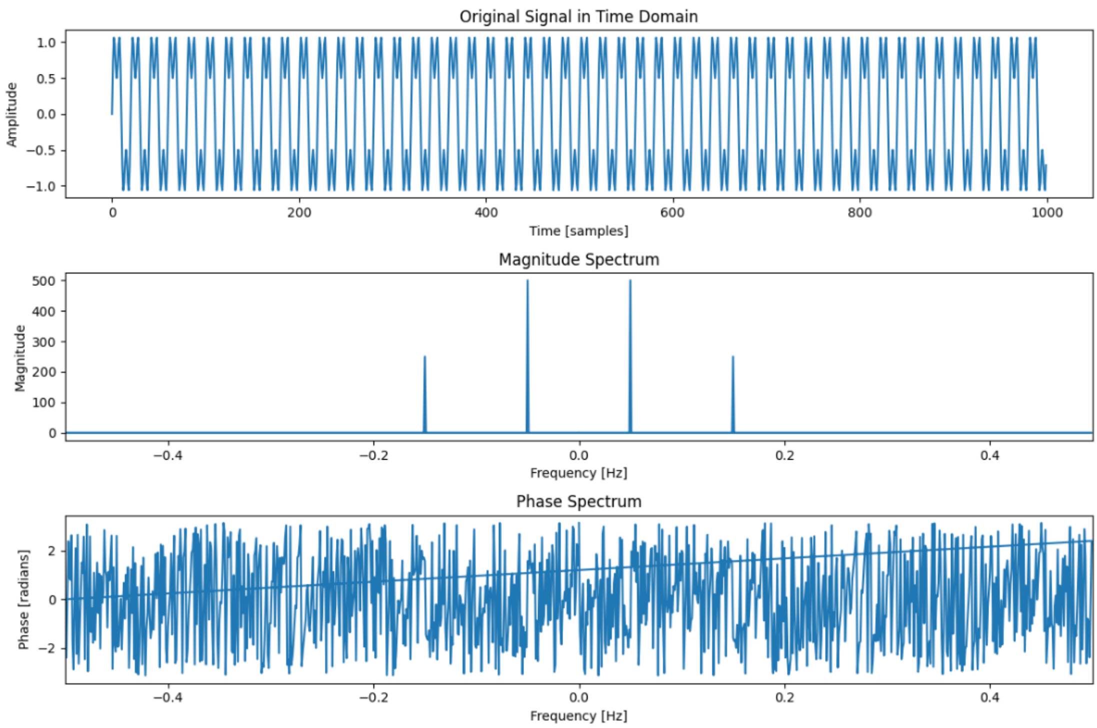
```
import numpy as np
import matplotlib.pyplot as plt

# Function to perform Discrete Fourier Transform (DFT)
def dft_analysis(signal):
    """
    Perform Discrete Fourier Transform (DFT) on the input signal.

    :param signal: The input signal (numpy array).
    :return: The DFT coefficients (numpy array).
```

```
    """
    return np.fft.fft(signal)

# Function to perform Inverse Discrete Fourier Transform (IDFT)
def dft_synthesis(dft_coefficients):
    """
    Perform Inverse Discrete Fourier Transform (IDFT) to reconstruct the
signal.

    :param dft_coefficients: The DFT coefficients (numpy array).
    :return: The reconstructed signal (numpy array).
    """
    return np.fft.ifft(dft_coefficients)

# Function to plot the signal, its magnitude, and phase spectrum
def plot_spectra(signal, dft_coefficients):
    """
    Plot the time-domain signal, magnitude spectrum, and phase
spectrum.

    :param signal: The original signal (numpy array).
    :param dft_coefficients: The DFT coefficients (numpy array).
    """
    # Time-domain signal
    time = np.arange(len(signal))

    # Frequency domain (for plotting)
    freq = np.fft.fftfreq(len(signal))

    # Magnitude and Phase of DFT coefficients
    magnitude = np.abs(dft_coefficients)
    phase = np.angle(dft_coefficients)

    plt.figure(figsize=(12, 8))

    # Plot original signal in time domain
    plt.subplot(3, 1, 1)
    plt.plot(time, signal)
    plt.title("Original Signal in Time Domain")
    plt.xlabel("Time [samples]")
```

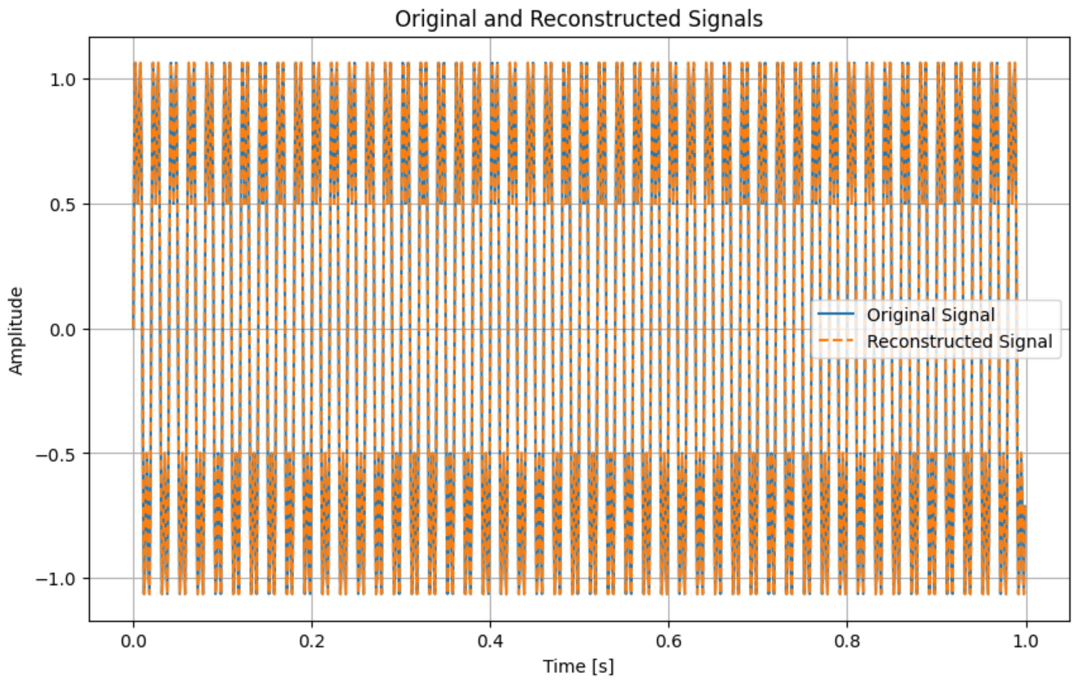
```
    plt.ylabel("Amplitude")

    # Plot Magnitude Spectrum
    plt.subplot(3, 1, 2)
    plt.plot(freq, magnitude)
    plt.title("Magnitude Spectrum")
    plt.xlabel("Frequency [Hz]")
    plt.ylabel("Magnitude")
    plt.xlim(-0.5, 0.5)

    # Plot Phase Spectrum
    plt.subplot(3, 1, 3)
    plt.plot(freq, phase)
    plt.title("Phase Spectrum")
    plt.xlabel("Frequency [Hz]")
    plt.ylabel("Phase [radians]")
    plt.xlim(-0.5, 0.5)

    plt.tight_layout()
    plt.show()

# Example usage
if __name__ == "__main__":
    # Parameters
    sampling_rate = 1000  # Sampling rate (samples per second)
    duration = 1          # Duration of the signal in seconds
    freq1 = 50            # Frequency of the first sine wave (Hz)
    freq2 = 150           # Frequency of the second sine wave (Hz)

    # Generate a signal composed of two sine waves
    time = np.linspace(0, duration, int(sampling_rate * duration),
endpoint=False)
    signal = np.sin(2 * np.pi * freq1 * time) + 0.5 * np.sin(2 * np.pi * freq2
* time)

    # Perform DFT Analysis
    dft_coefficients = dft_analysis(signal)

    # Plot the signal and its spectrum
    plot_spectra(signal, dft_coefficients)
```

```
    # Perform DFT Synthesis (Inverse DFT)
    reconstructed_signal = dft_synthesis(dft_coefficients)

    # Plot the original and reconstructed signals
    plt.figure(figsize=(10, 6))
    plt.plot(time, signal, label='Original Signal')
    plt.plot(time, reconstructed_signal, label='Reconstructed Signal',
linestyle='--')
    plt.title("Original and Reconstructed Signals")
    plt.xlabel("Time [s]")
    plt.ylabel("Amplitude")
    plt.legend()
    plt.grid(True)
plt.show()
```

OUTPUT:



Original Signal in Time Domain
Amplitude
Time [samples]
Magnitude Spectrum
Magnitude
Frequency [Hz]
Phase Spectrum
Phase [radians]
Frequency [Hz]



Original and Reconstructed Signals
Amplitude
Time [s]
Original Signal
Reconstructed Signal

INFERENCE:

RESULT:

EX NO : 08	Analyse the ECG signal to find heart rate of a patient
DATE:	

AIM:

To Analyse the ECG signal to find heart rate of a patient

ALGORITHM:

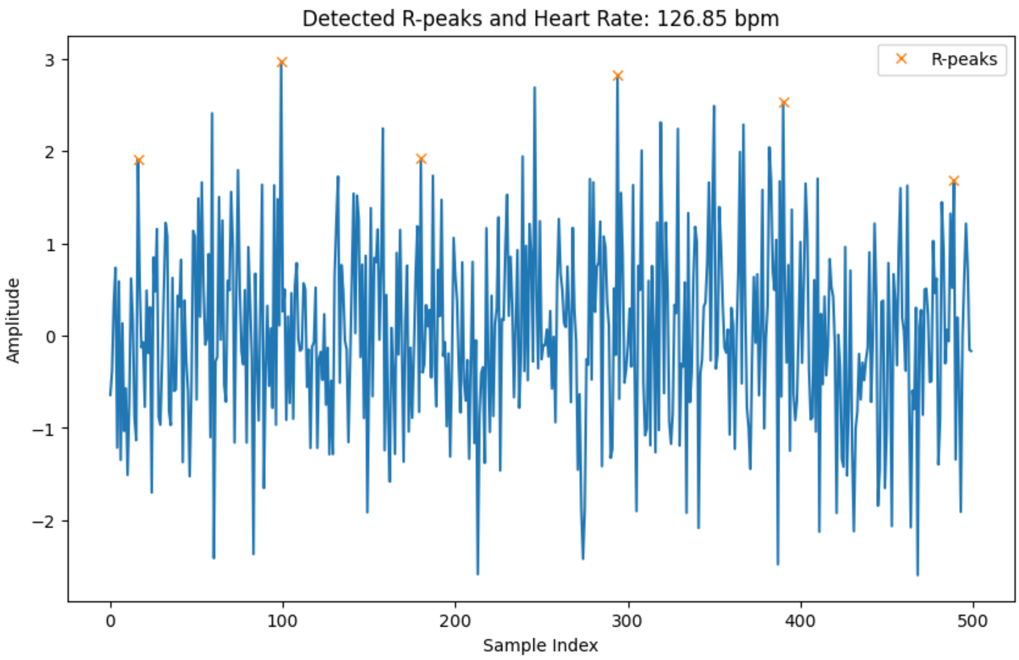
1: Define the ECG signal and set the sampling rate.
2: Detect R-peaks using peak detection (find_peaks), with height and distance thresholds.
3: Calculate R-R intervals (time differences between consecutive R-peaks).
4: Calculate heart rate using the average R-R interval:
5: Plot the ECG signal and detected R-peaks, displaying the heart rate.
6: Print the detected heart rate.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import find_peaks

# Assuming ecg is a 1D numpy array containing the ECG signal data
ecg = np.random.randn(500)  # Replace this with your actual ECG signal

# Sampling rate
fs = 200  # Hz

# Step 1: Identify the R-peaks using a peak detection method
# Use find_peaks from scipy to find the R-peaks (usually the local maxima)
```

```
# We will look for peaks that have a minimum height to avoid noise
peaks, _ = find_peaks(ecg, height=0.5, distance=fs/2.5)  # Adjust distance as per the signal characteristics

# Step 2: Calculate R-R intervals (time difference between consecutive R-peaks)
rr_intervals = np.diff(peaks) / fs  # In seconds

# Step 3: Calculate the heart rate (in beats per minute)
heart_rate = 60 / np.mean(rr_intervals)

# Step 4: Plot the ECG signal and the detected R-peaks
plt.figure(figsize=(10, 6))
plt.plot(ecg)
plt.plot(peaks, ecg[peaks], "x", label="R-peaks")
plt.title(f"Detected R-peaks and Heart Rate: {heart_rate:.2f} bpm")
plt.xlabel("Sample Index")
plt.ylabel("Amplitude")
plt.legend()
plt.show()
print(f"Detected Heart Rate: {heart_rate:.2f} bpm")
```

OUTPUT:



INFERENCE:

RESULT: