Experiment No: 01

Experiment Name: Analysis of Poles, Zeros, and Roots of Transfer Functions

Objectives:

1. To determine the poles of the given transfer functions H(s)H(s)H(s).
2. To calculate the zeros of the given transfer functions H(s)H(s)H(s).
3. To plot the poles and zeros on the complex plane and understand their positions concerning system stability.

Theory:

In control systems and signal processing, transfer functions are commonly represented as ratios of polynomials in the complex frequency variable, typically denoted by s in the Laplace domain. A transfer function H(s) is expressed as:

$$H(s) = \frac{N(s)}{D(s)}$$

where N(s) is the numerator polynomial and D(s) is the denominator polynomial.

1. Poles and Zeros of a Transfer Function:

- **Poles** are the values of s that make the denominator D(s)=0. They represent frequencies at which the system's response goes to infinity. Poles are critical for determining system stability.
- **Zeros** are the values of s that make the numerator N(s)=0. Zeros represent the frequencies at which the output of the system becomes zero for non-zero input.

2. Stability Analysis: The location of poles in the complex plane directly affects the system's stability. For a continuous-time system:

- If all poles lie in the left half of the complex plane (i.e., have negative real parts), the system is stable.
- Poles on the right half of the plane indicate an unstable system.
- Poles on the imaginary axis imply a marginally stable system.

3. Roots of the System: The roots of the numerator and denominator polynomials give the zeros and poles, respectively. Identifying the roots provides insights into the system's transient behavior and response.

In this experiment, we will analyse two transfer functions:

a. $H(s) = \frac{S^3 + 1}{S^4 + 2S^2 + 1}$

b. $H(s) = \frac{4S^3 + 8S + 10}{2S^3 + 8S^2 + 18s + 20}$

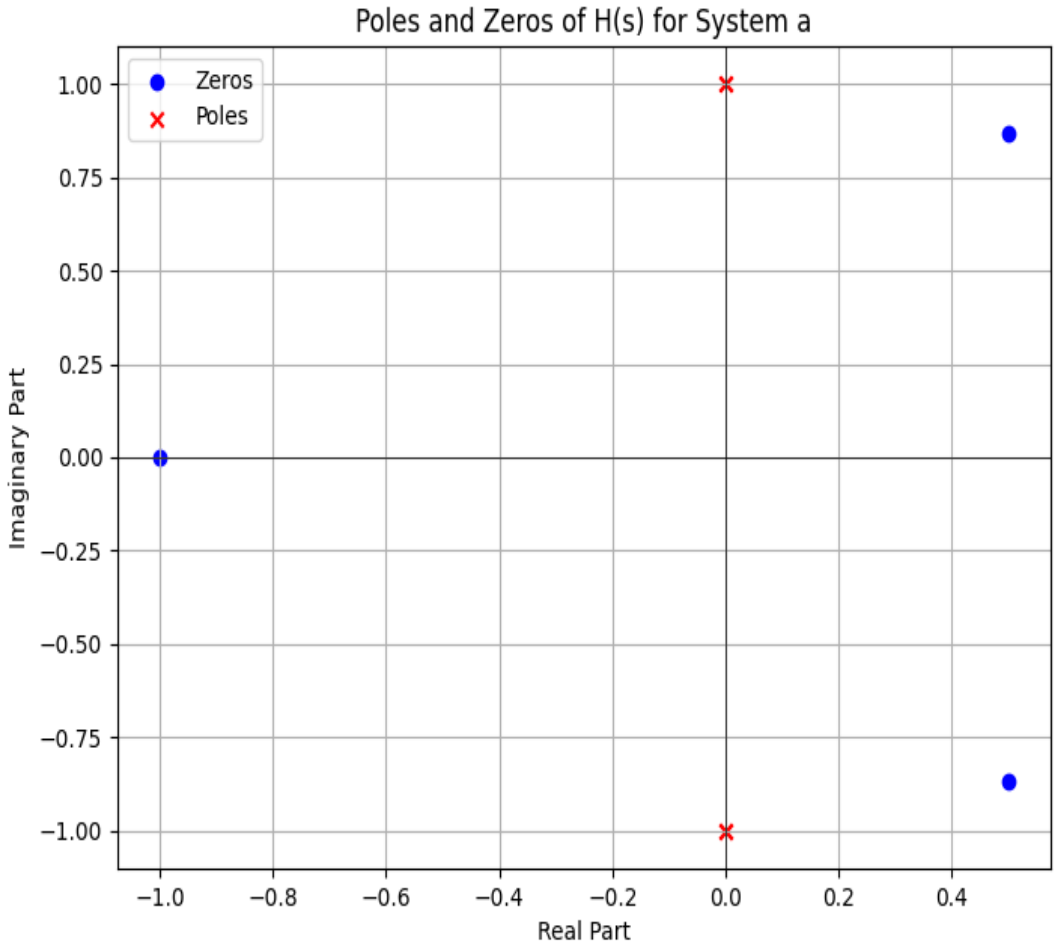
Source Code (Python):

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import TransferFunction, tf2zpk
# Function to calculate and plot poles and zeros
def plot_poles_zeros(num, den, title):
# Calculate poles and zeros
zeros, poles, _ = tf2zpk(num, den)
# Plot
plt.figure(figsize=(8, 6))
plt.scatter(np.real(zeros), np.imag(zeros), color='blue', marker='o', label="Zeros")
plt.scatter(np.real(poles), np.imag(poles), color='red', marker='x', label="Poles")
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.xlabel("Real Part")
plt.ylabel("Imaginary Part")
plt.title(title)
plt.legend()
plt.grid()
plt.show()
# Define systems
# System a: H(s) = (s^3 + 1) / (s^4 + 2s^2 + 1)
num_a = [1, 0, 0, 1]      # Numerator coefficients of H(s) for system a
den_a = [1, 0, 2, 0, 1]   # Denominator coefficients of H(s) for system a
# System b: H(s) = (4s^2 + 8s + 10) / (2s^3 + 8s^2 + 18s + 20)
```

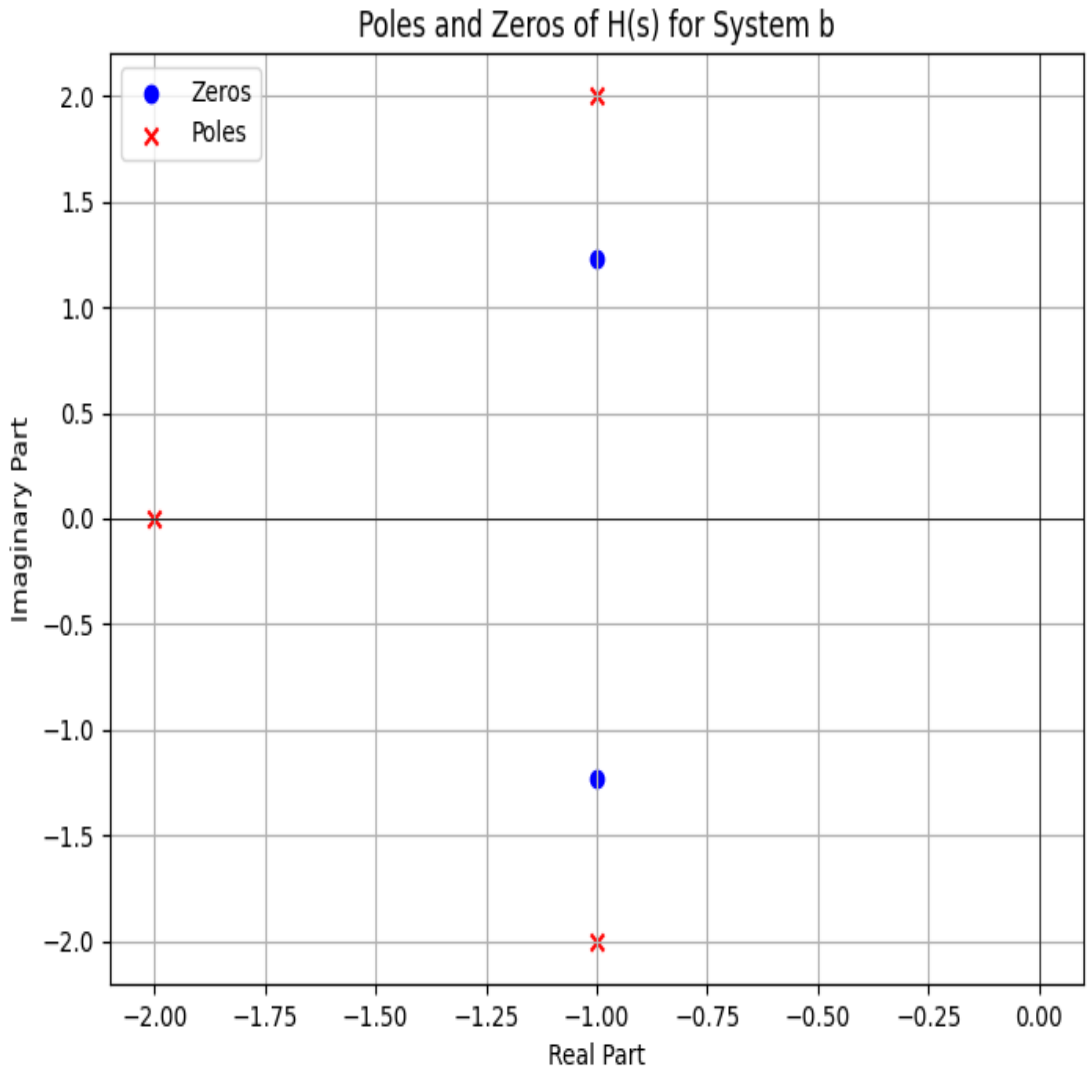
```
num_b = [4, 8, 10]        # Numerator coefficients of H(s) for system b
den_b = [2, 8, 18, 20]    # Denominator coefficients of H(s) for system b
# Plot poles and zeros for each system
plot_poles_zeros(num_a, den_a, "Poles and Zeros of H(s) for System a")
plot_poles_zeros(num_b, den_b, "Poles and Zeros of H(s) for System b")
```

Output:

Pole-Zero Plot for $\boldsymbol{H(s)} = \frac{\boldsymbol{S^3+1}}{\boldsymbol{S^4+2S^2+1}}$



And Pole-Zero Plot for $\boldsymbol{H(s) = \frac{4S^3 + 8S + 10}{2S^3 + 8S^2 + 18s + 20}}$



Experiment No: 02

Experiment Name: Analysis of Discrete-Time System Stability Using Pole-Zero Plots in the Z-Domain.

Objectives:

1. To utilize the "zplane" command in Python to visualize the poles and zeros of discrete-time systems.
2. To determine the stability of discrete-time systems by examining the pole-zero plot.
3. To understand the behavior of system responses through the arrangement of poles and zeros on the z-plane.

Theory:

In discrete-time signal processing, the transfer function H(z) is crucial for representing the relationship between the input and output of a system. H(z) is expressed in terms of z, which is the complex frequency variable in the z-domain. The general form of a transfer function is:

$H(Z)= \frac{N(z)}{D(Z)}$

where N(z) and D(z) are polynomials representing the numerator and denominator, respectively.

1. **Poles and Zeros**:
    - **Poles** are the values of z that make the denominator D(z) equal to zero. Poles indicate frequencies where the system's output could become infinite, contributing to the system's potential instability.
    - **Zeros** are the values of z that make the numerator N(z) equal to zero. Zeros signify frequencies where the system response becomes zero, essentially "canceling out" certain input frequencies.
    - The locations of poles and zeros on the complex z-plane provide insights into the stability and frequency response of the system.
2. **Stability Analysis**:

- In the z-domain, a system is stable if all poles lie within the unit circle on the complex plane. This criterion ensures that the system response decays over time, leading to bounded output for a bounded input.
- Poles outside the unit circle indicate an unstable system, while poles on the unit circle suggest marginal stability.

3. **The zplane Command**:
   - The zplane command in Python/MATLAB plots the poles and zeros of a discrete-time transfer function on the zzz-plane.
   - Poles are marked with an “X” and zeros with an “O.” This visual representation aids in quickly assessing the stability of a system.

In this experiment, we are analyzing two systems represented by the following transfer functions:

System a: H(z) = $\frac{1+z^{-2}}{2+z^{-1}-0.5z^{-2}+0.25z^{-3}}$

System b: H(z) = $\frac{1+z^{-1}+\frac{3}{2}z^{-2}+\frac{1}{2}z^{-3}}{1+\frac{3}{2}z^{-1}+\frac{1}{2}z^{-2}}$

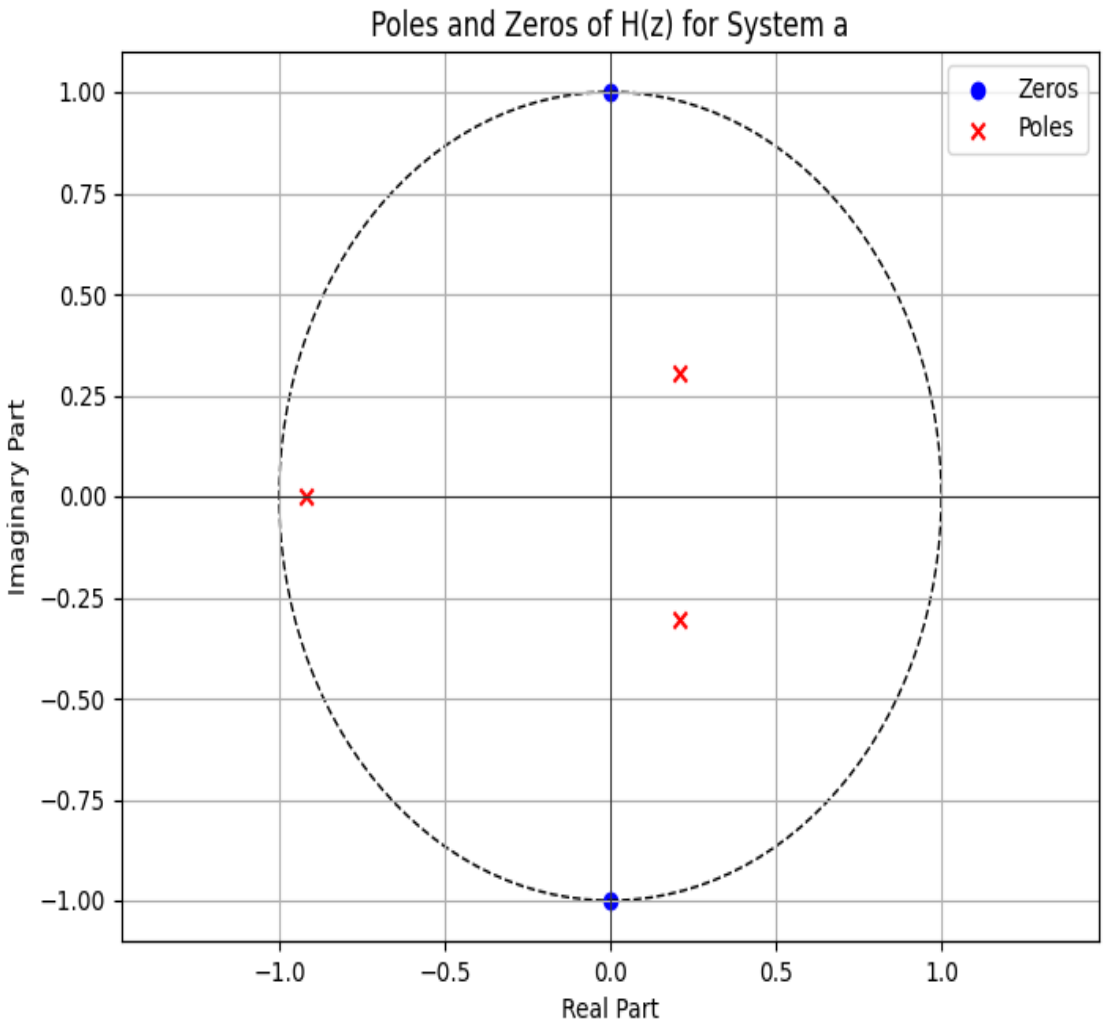
Source Code (Python):

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import tf2zpk

# Define a function for z-plane plot
def zplane(num, den, title):
    # Compute poles and zeros
    zeros, poles, _ = tf2zpk(num, den)

    # Plot
    plt.figure(figsize=(8, 6))
    plt.scatter(np.real(zeros), np.imag(zeros), color='blue', marker='o', label='Zeros')
```

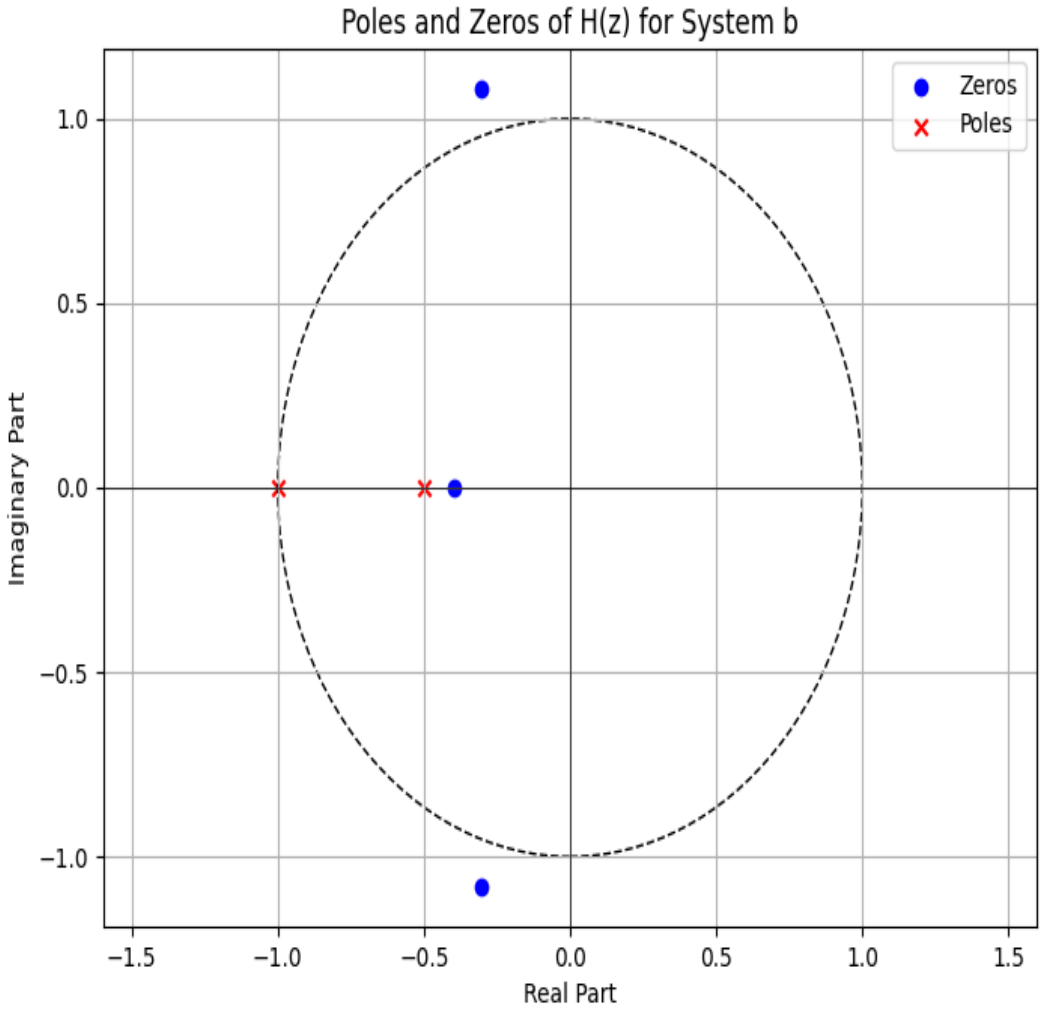
```
    plt.scatter(np.real(poles), np.imag(poles), color='red', marker='x', label='Poles')

    # Draw unit circle for reference
    unit_circle = plt.Circle((0, 0), 1, color='black', fill=False, linestyle='--', linewidth=1)
    plt.gca().add_patch(unit_circle)

    plt.axhline(0, color='black', linewidth=0.5)
    plt.axvline(0, color='black', linewidth=0.5)
    plt.xlabel("Real Part")
    plt.ylabel("Imaginary Part")
    plt.title(title)
    plt.legend()
    plt.grid()
    plt.axis('equal')
    plt.show()

# Define the systems
# System a: H(z) = (1 + z^(-2)) / (2 + z^(-1) - 0.5z^(-2) + 0.25z^(-3))
num_a = [1, 0, 1]           # Numerator coefficients for H(z) in system a
den_a = [2, 1, -0.5, 0.25]    # Denominator coefficients for H(z) in system a

# System b: H(z) = (1 + z^(-1) + 1.5*z^(-2) + 0.5*z^(-3)) / (1 + 1.5*z^(-1) + 0.5*z^(-2))
num_b = [1, 1, 1.5, 0.5]      # Numerator coefficients for H(z) in system b
den_b = [1, 1.5, 0.5]         # Denominator coefficients for H(z) in system b

# Plot poles and zeros for each system
zplane(num_a, den_a, "Poles and Zeros of H(z) for System a")
zplane(num_b, den_b, "Poles and Zeros of H(z) for System b")
```

Output:

Pole-Zero Plot of H(z) = $\frac{1+z^{-2}}{2+z^{-1}-0.5z^{-2}+0.25z^{-3}}$ is:



& Pole-Zero Plot of H(Z)) = $\frac{1+z^{-1}+\frac{3}{2}z^{-2}+\frac{1}{2}z^{-3}}{1+\frac{3}{2}z^{-1}+\frac{1}{2}z^{-2}}$ is:


Poles and Zeros of H(z) for System b
Zeros
Poles
Imaginary Part
Real Part
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

Experiment No: 07

Experiment Name: Feature Extract and Noise Reduction of PPG Signal

Objectives:

1. To extract key features from Photoplethysmogram (PPG) signals, which help in analyzing heart rate, blood oxygen saturation, and other cardiovascular indicators.
2. To apply noise reduction techniques to the PPG signal for clearer interpretation and analysis.
3. To understand the importance of preprocessing PPG signals in biomedical signal processing.

Theory:

Photoplethysmogram (PPG) signals are optical signals that measure changes in blood volume in peripheral circulation, typically captured by shining light through the skin and detecting variations in light absorption. These signals are widely used for non-invasive monitoring of heart rate, oxygen saturation (SpO_2), and other cardiovascular parameters.

Key Components in PPG Signal Processing

1. **Feature Extraction**:
    - **Heart Rate**: Heart rate is derived from the frequency of peaks in the PPG waveform, where each peak corresponds to a heartbeat.
    - **Amplitude and Baseline Drift**: Amplitude can give insight into blood volume changes, while baseline drift (slow fluctuation of the waveform baseline) may occur due to motion artifacts, respiration, or sensor placement.
    - **Pulse Interval and Pulse Shape**: These are analyzed to assess cardiovascular health, with pulse intervals indicating variability and pulse shape relating to vascular conditions.
2. **Noise Reduction**:
    - **Types of Noise**: Common noise sources include:

        - **Motion Artifacts**: Caused by subject movement, leading to irregularities in the waveform.
        - **High-frequency Noise**: Electrical or ambient noise that introduces high-frequency components.
    - **Filtering Techniques**: Noise reduction in PPG signals is typically achieved through filtering, which removes unwanted frequency components.
        - **Low-pass Filters**: Remove high-frequency noise, preserving the main PPG waveform.
        - **Bandpass Filters**: Isolate the frequency range specific to physiological signals (usually 0.5 to 5 Hz for heart rate).
        - **Moving Average Filter**: Reduces short-term fluctuations, providing smoother signals.
3. **Feature Extraction Workflow**:
    - **Signal Preprocessing**: First, noise reduction techniques are applied.
    - **Peak Detection**: Detects peaks corresponding to heartbeats.
    - **Parameter Calculation**: Calculates intervals, pulse amplitude, and other features relevant to the cardiovascular analysis.

For this experiment, we will use Python to implement noise reduction and feature extraction on a sample PPG signal. Filtering will be applied for noise reduction, and then key features will be extracted.

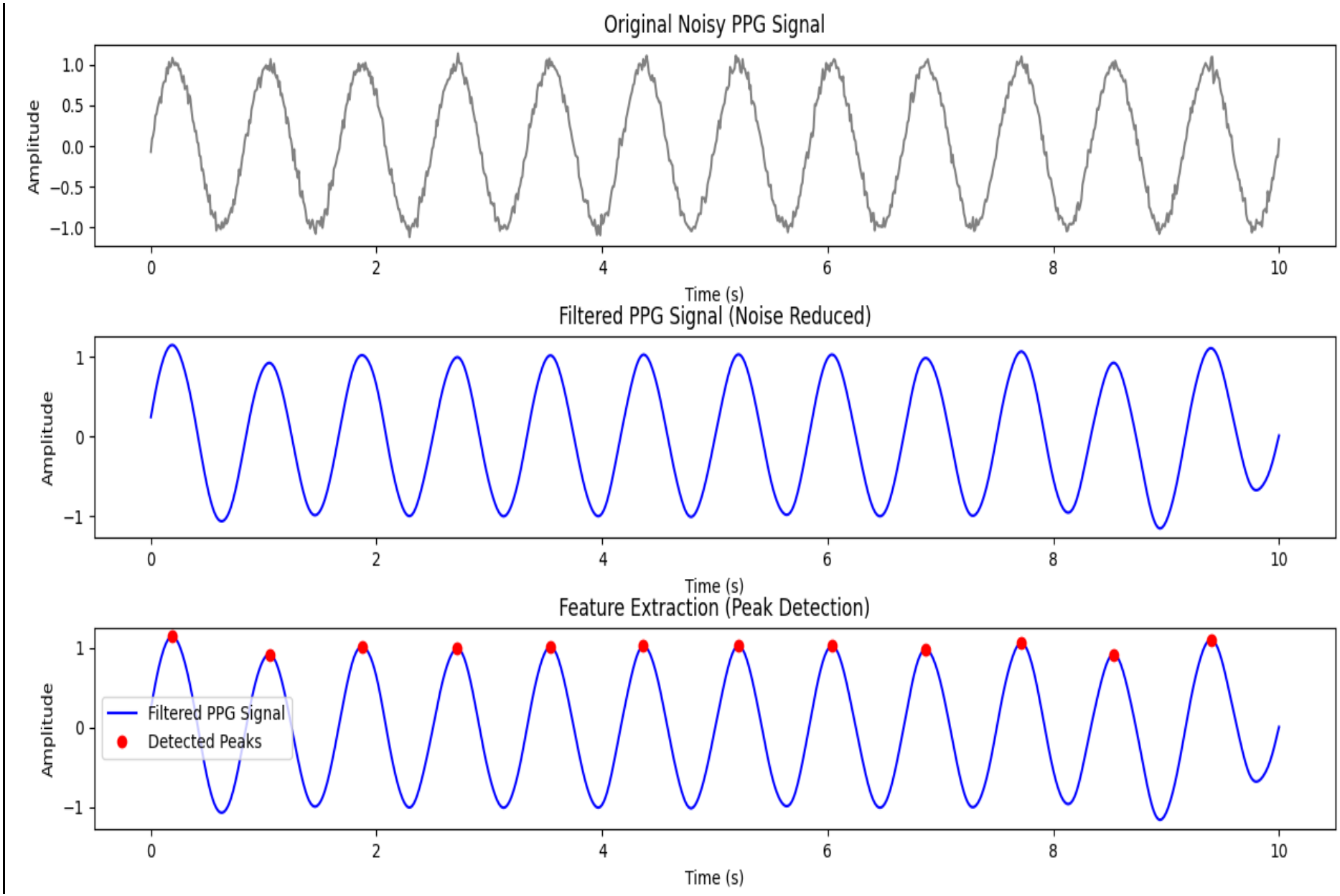
Source Code (Python):

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

# Generate synthetic PPG signal (for demonstration)
# Sampling frequency
fs = 100  # Hz
```

```
t = np.linspace(0, 10, fs * 10)  # 10 seconds of data

# Create a synthetic PPG signal with a frequency of 1.2 Hz (about 72 BPM)

ppg_signal = 1.0 * np.sin(2 * np.pi * 1.2 * t) + 0.05 * np.random.randn(len(t))  # Add small random noise


# Butterworth bandpass filter for noise reduction

def bandpass_filter(signal, lowcut, highcut, fs, order=4):

    nyquist = 0.5 * fs

    low = lowcut / nyquist

    high = highcut / nyquist

    b, a = butter(order, [low, high], btype='band')

    y = filtfilt(b, a, signal)

    return y


# Filter settings for PPG signal (0.5-5 Hz band)

lowcut = 0.5  # Lower frequency cutoff in Hz

highcut = 5.0 # Higher frequency cutoff in Hz


# Apply bandpass filter to reduce noise

filtered_ppg = bandpass_filter(ppg_signal, lowcut, highcut, fs)


# Peak detection for feature extraction

peaks, _ = find_peaks(filtered_ppg, distance=fs/2)  # Distance set to avoid multiple peaks per pulse


# Plotting

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


# Plot original noisy PPG signal

plt.subplot(3, 1, 1)
```

```
plt.plot(t, ppg_signal, color='gray')
plt.title("Original Noisy PPG Signal")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")

# Plot filtered PPG signal
plt.subplot(3, 1, 2)
plt.plot(t, filtered_ppg, color='blue')
plt.title("Filtered PPG Signal (Noise Reduced)")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")

# Plot filtered signal with detected peaks
plt.subplot(3, 1, 3)
plt.plot(t, filtered_ppg, color='blue', label="Filtered PPG Signal")
plt.plot(t[peaks], filtered_ppg[peaks], "ro", label="Detected Peaks")
plt.title("Feature Extraction (Peak Detection)")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.legend()

plt.tight_layout()
plt.show()
```

Output:


Original Noisy PPG Signal
Amplitude
1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10
Time (s)
Filtered PPG Signal (Noise Reduced)
Amplitude
1
0
−1
0
2
4
6
8
10
Time (s)
Feature Extraction (Peak Detection)
Amplitude
Filtered PPG Signal
Detected Peaks
1
0
−1
0
2
4
6
8
10
Time (s)

Experiment No: 08

Experiment Name: Analog to Digital and Digital to Analog Conversion.

Objectives:

1. To understand the concepts of Analog-to-Digital Conversion (ADC) and Digital-to-Analog Conversion (DAC).
2. To implement ADC and DAC using sampling and reconstruction techniques in Python.
3. To apply the Fast Fourier Transform (FFT) to analyze the frequency components of a digital signal.

Theory:

In signal processing, the conversion between analog and digital signals is fundamental, as it allows real-world continuous signals (analog) to be processed by digital systems like computers.

1. Analog-to-Digital Conversion (ADC)

Analog-to-Digital Conversion involves sampling and quantizing a continuous-time analog signal into a discrete-time digital signal.

- **Sampling**: Sampling captures the amplitude of an analog signal at regular intervals. The sampling rate, or frequency (usually measured in Hz), is critical. According to the **Nyquist Theorem**, the sampling rate must be at least twice the maximum frequency of the signal to avoid aliasing (distortion caused by undersampling).

  $$f_s \geq 2f_{max}$$

  Where f_s is the sampling rate and f_{max} is the maximum frequency of the signal.

- **Quantization**: After sampling, each sample's amplitude is rounded to the nearest level within a fixed range. This step approximates the continuous signal into discrete steps, introducing quantization error, which affects accuracy.

2. Digital-to-Analog Conversion (DAC)

Digital-to-Analog Conversion is the process of reconstructing an analog signal from a digital signal. This process often uses interpolation to smooth out the steps between sampled points.

- **Reconstruction**: Reconstruction uses a low-pass filter to remove high-frequency artifacts introduced by sampling. An ideal reconstruction reconstructs the original continuous signal exactly.

3. Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the Discrete Fourier Transform (DFT), which decomposes a signal into its frequency components. FFT is particularly useful in analyzing digital signals to observe the underlying frequencies.

- **Frequency Spectrum**: The FFT transforms the time-domain signal into the frequency domain, allowing analysis of dominant frequencies within the signal.
- **Application**: FFT is widely used in audio processing, signal analysis, and communications.

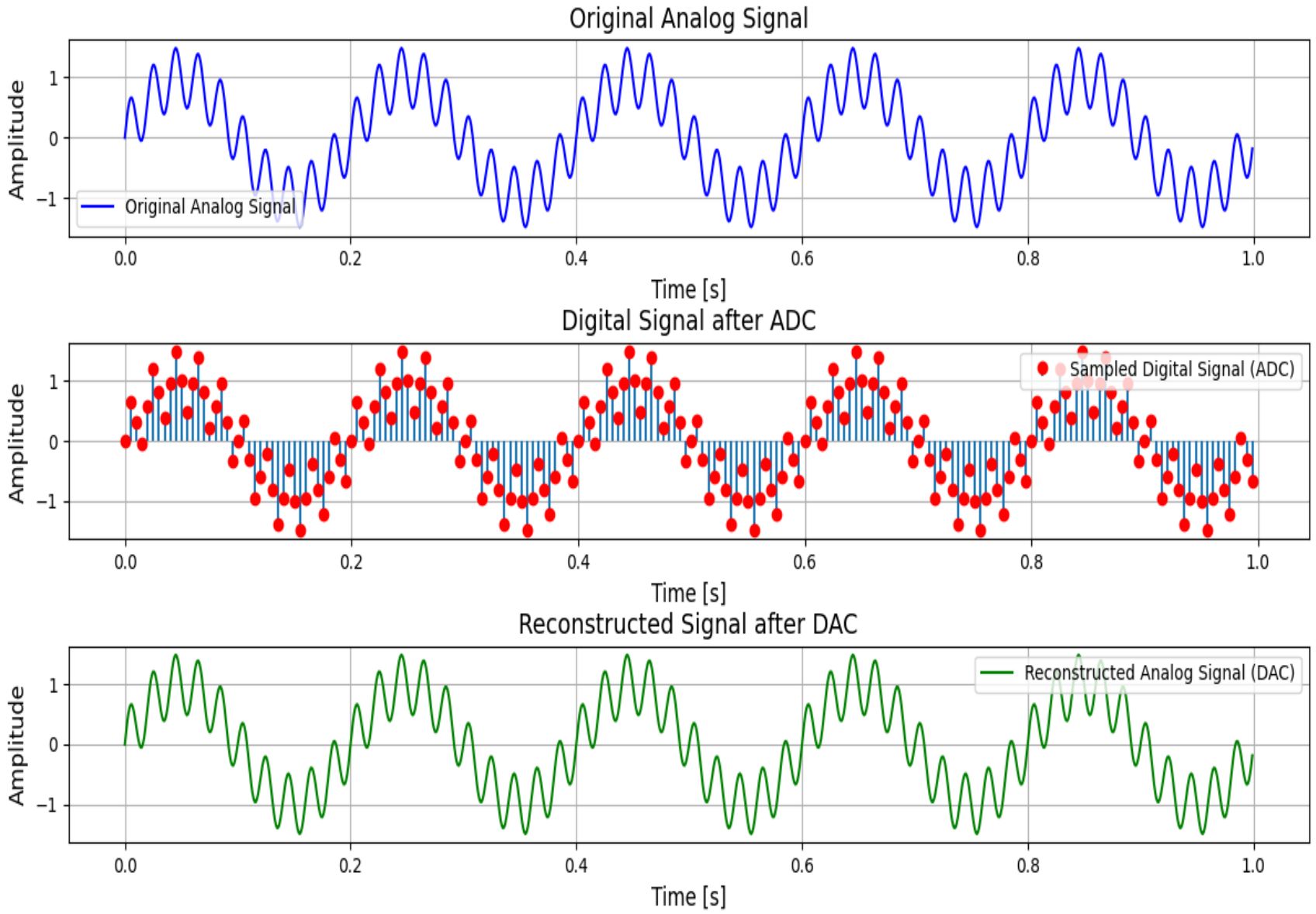
Source Code (Python):

```
import numpy as np

import matplotlib.pyplot as plt

from scipy.fft import fft, fftfreq

from scipy.signal import resample


# Parameters

fs = 1000  # Sampling rate (Hz)

T = 1.0    # Duration of the signal in seconds

t = np.linspace(0, T, int(fs * T), endpoint=False)  # Time vector


# Generate an analog signal (combination of 5 Hz and 50 Hz sine waves)

f1, f2 = 5, 50

analog_signal = np.sin(2 * np.pi * f1 * t) + 0.5 * np.sin(2 * np.pi * f2 * t)
```

```
# Perform ADC - Sampling the analog signal
sampling_rate = 200  # Sampling rate for ADC (Hz)
t_sampled = np.linspace(0, T, int(sampling_rate * T), endpoint=False)
digital_signal = np.sin(2 * np.pi * f1 * t_sampled) + 0.5 * np.sin(2 * np.pi * f2 * t_sampled)

# DAC - Reconstruction by resampling to the original rate
reconstructed_signal = resample(digital_signal, len(t))

# FFT of the sampled signal
fft_values = fft(digital_signal)
fft_freqs = fftfreq(len(digital_signal), 1 / sampling_rate)

# Plotting
plt.figure(figsize=(12, 10))

# Original Analog Signal
plt.subplot(3, 1, 1)
plt.plot(t, analog_signal, label="Original Analog Signal", color="blue")
plt.xlabel("Time [s]", fontsize=12)
plt.ylabel("Amplitude", fontsize=12)
plt.title("Original Analog Signal", fontsize=14)
plt.legend()
plt.grid()


# Digital Signal after ADC (Sampled)
plt.subplot(3, 1, 2)
plt.stem(t_sampled, digital_signal, basefmt=" ", markerfmt="ro", label="Sampled Digital
Signal (ADC)")
```

```
plt.xlabel("Time [s]", fontsize=12)
plt.ylabel("Amplitude", fontsize=12)
plt.title("Digital Signal after ADC", fontsize=14)
plt.legend()
plt.grid()


# Reconstructed Signal (DAC)
plt.subplot(3, 1, 3)
plt.plot(t, reconstructed_signal, label="Reconstructed Analog Signal (DAC)", color="green")
plt.xlabel("Time [s]", fontsize=12)
plt.ylabel("Amplitude", fontsize=12)
plt.title("Reconstructed Signal after DAC", fontsize=14)
plt.legend()
plt.grid()


# Adjust layout to prevent overlap
plt.tight_layout(pad=3.0)

plt.show()
```

Output:


Original Analog Signal
Amplitude
Original Analog Signal
Time [s]
Digital Signal after ADC
Amplitude
Sampled Digital Signal (ADC)
Time [s]
Reconstructed Signal after DAC
Amplitude
Reconstructed Analog Signal (DAC)
Time [s]

Experiment No: 05

Experiment Name: Determination of FFT of a Given Signal

Objectives:

1. To compute the Fast Fourier Transform (FFT) of a given time-domain signal.
2. To analyze the frequency components of the signal using FFT.
3. To visualize the amplitude spectrum of the signal and understand its frequency characteristics.

Theory:

The **Fast Fourier Transform (FFT)** is a powerful algorithm for converting a time-domain signal into its frequency-domain representation. The FFT is an efficient implementation of the **Discrete Fourier Transform (DFT)**, which allows us to analyze the frequency components of a signal.

Key Concepts in FFT

1. **Fourier Transform**:
    - The Fourier Transform decomposes a signal into a sum of sinusoidal components, each characterized by a specific frequency, amplitude, and phase. This helps in understanding the frequency content of signals, which is essential for signal processing, communications, and spectral analysis.
2. **Discrete Fourier Transform (DFT)**:
    - When dealing with discrete signals, we use the DFT, which provides a finite sum of sinusoidal terms. The DFT is defined as:

      $$X[k]=\sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi kn/N}$$

      where N is the number of samples, x[n] is the signal in the time domain, and X[k] is the frequency-domain representation.
3. **Fast Fourier Transform (FFT)**:
    - The FFT is a computationally efficient algorithm to calculate the DFT, reducing the time complexity from O(N^2) to O(NlogN). It is widely used due to its speed and efficiency.

Properties of FFT

1. **Magnitude Spectrum**:

- The magnitude of the FFT output indicates the amplitude of each frequency component present in the signal.

2. **Frequency Resolution**:
    - The frequency resolution of an FFT depends on the sampling rate fs and the number of points N used in the FFT. The resolution is given by

$$\Delta f=\frac{f_s}{N}$$

3. **Symmetry**:
    - For real-valued signals, the FFT result is symmetric around the center frequency, meaning that only half of the FFT output is unique.

Practical Application

The FFT is essential for analyzing the spectral characteristics of signals in audio processing, image analysis, and communications. By examining the FFT, we can determine the dominant frequencies, filter specific frequency bands, or compress signal information.

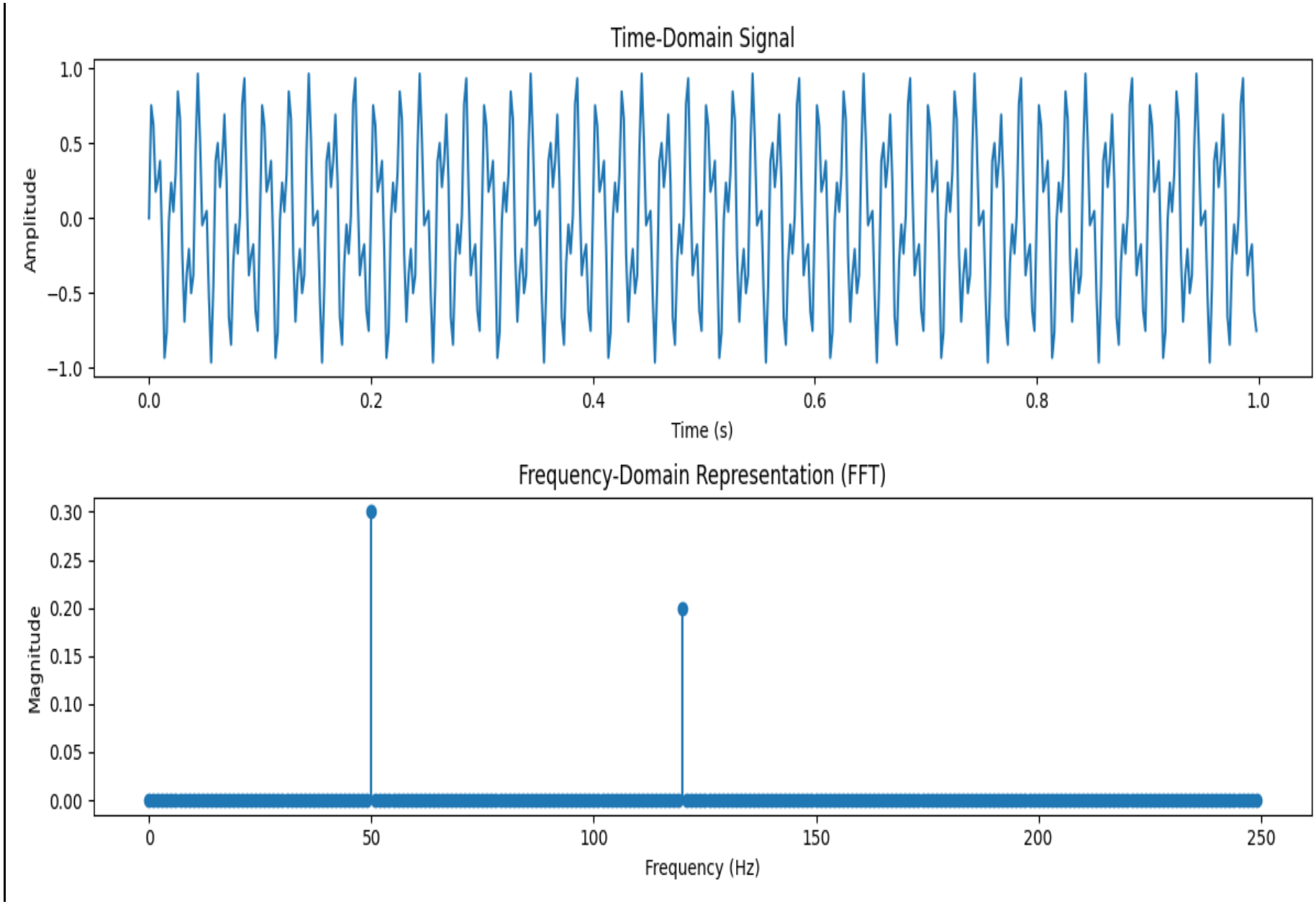
Source Code (Python):

```
import numpy as np

import matplotlib.pyplot as plt

from scipy.fft import fft, fftfreq

from scipy.signal import resample


# Parameters

fs = 1000  # Sampling rate (Hz)

T = 1.0    # Duration of the signal in seconds

t = np.linspace(0, T, int(fs * T), endpoint=False)  # Time vector


# Generate an analog signal (combination of 5 Hz and 50 Hz sine waves)

f1, f2 = 5, 50

analog_signal = np.sin(2 * np.pi * f1 * t) + 0.5 * np.sin(2 * np.pi * f2 * t)


# Perform ADC - Sampling the analog signal
```

```
sampling_rate = 200  # Sampling rate for ADC (Hz)
t_sampled = np.linspace(0, T, int(sampling_rate * T), endpoint=False)
digital_signal = np.sin(2 * np.pi * f1 * t_sampled) + 0.5 * np.sin(2 * np.pi * f2 * t_sampled)

# DAC - Reconstruction by resampling to the original rate
reconstructed_signal = resample(digital_signal, len(t))

# FFT of the sampled signal
fft_values = fft(digital_signal)
fft_freqs = fftfreq(len(digital_signal), 1 / sampling_rate)

# Plotting
plt.figure(figsize=(12, 10))
# Original Analog Signal
plt.subplot(3, 1, 1)
plt.plot(t, analog_signal, label="Original Analog Signal", color="blue")
plt.xlabel("Time [s]", fontsize=12)
plt.ylabel("Amplitude", fontsize=12)
plt.title("Original Analog Signal", fontsize=14)
plt.legend()
plt.grid()
# Digital Signal after ADC (Sampled)
plt.subplot(3, 1, 2)
plt.stem(t_sampled, digital_signal, basefmt=" ", markerfmt="ro", label="Sampled Digital Signal (ADC)")
plt.xlabel("Time [s]", fontsize=12)
plt.ylabel("Amplitude", fontsize=12)
plt.title("Digital Signal after ADC", fontsize=14)
plt.legend()
plt.grid()
```

```
# Reconstructed Signal (DAC)
plt.subplot(3, 1, 3)
plt.plot(t, reconstructed_signal, label="Reconstructed Analog Signal (DAC)", color="green")
plt.xlabel("Time [s]", fontsize=12)
plt.ylabel("Amplitude", fontsize=12)
plt.title("Reconstructed Signal after DAC", fontsize=14)
plt.legend()
plt.grid()

# Adjust layout to prevent overlap
plt.tight_layout(pad=3.0)
plt.show()

# Plotting the FFT Spectrum
plt.figure(figsize=(12, 5))
plt.stem(fft_freqs[:len(fft_freqs) // 2], np.abs(fft_values[:len(fft_values) // 2]),
markerfmt="bo")
plt.xlabel("Frequency [Hz]", fontsize=12)
plt.ylabel("Amplitude", fontsize=12)
plt.title("Frequency Spectrum of Sampled Signal (FFT)", fontsize=14)
plt.grid()
plt.tight_layout(pad=3.0)  # Adjust layout again to prevent overlap
plt.show()
```

Output:


Time-Domain 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)
Frequency-Domain Representation (FFT)
Magnitude
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
50
100
150
200
250
Frequency (Hz)