7. Sampling and Reconstruction of Signals: Analysis of Aliasing Effects and Proper Signal Reconstruction. 8. Coding and Decoding Digital Signals

1 Sampling and Reconstruction of Signals: Analysis of Aliasing Effects and Proper Signal Reconstruction.

1.1 Theoretical Background

This laboratory explores the fundamental principles of sampling and reconstruction in signal processing. The topics covered include the Nyquist-Shannon sampling theorem, aliasing effects, and methods of signal reconstruction.

1.1.1 Nyquist-Shannon Sampling Theorem

The Nyquist-Shannon sampling theorem states that a continuous-time signal $x(t)$ can be completely reconstructed from its samples $x[n]$ if the sampling frequency f_s is greater than twice the maximum frequency component $f_{\max}$ of the signal:

$$f_s > 2f_{\max}.$$

The condition $f_s = 2f_{\max}$ is known as the Nyquist rate.

1.1.2 Aliasing

When a signal is sampled at a frequency lower than the Nyquist rate, the high-frequency components of the signal overlap in the frequency domain. This phenomenon, called *aliasing*, leads to distortion and makes it impossible to accurately reconstruct the original signal.

The frequency f_a of an aliased component is given by:

$$f_a = |f - kf_s|, \quad k \in \mathbb{Z},$$

where f is the original frequency of the component, and k is an integer.

1.1.3 Signal Reconstruction

Reconstruction of a sampled signal is typically performed using a low-pass filter to remove the aliasing effects. The most commonly used reconstruction method is *sinc interpolation*, where the reconstructed signal $x(t)$ is given by:

$$x(t) = \sum_{n=-\infty}^{\infty} x[n]\text{sinc}\left(\frac{t - nT}{T}\right),$$

where $T = 1/f_s$ is the sampling period.

1.2 Practical Examples in Python

This section includes practical examples to demonstrate sampling, aliasing, and reconstruction.

1.2.1 Example 1: Demonstration of Aliasing

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

# Original signal parameters
f_signal = 5  # Frequency of the signal (Hz)
t = np.linspace(0, 1, 1000, endpoint=False)  # Time
   vector
signal = np.sin(2 * np.pi * f_signal * t)  # Original
   signal

# Sampling parameters
f_sample_low = 8  # Low sampling frequency (Hz)
f_sample_high = 20  # High sampling frequency (Hz)

# Sampling the signal
t_low = np.arange(0, 1, 1 / f_sample_low)
t_high = np.arange(0, 1, 1 / f_sample_high)
samples_low = np.sin(2 * np.pi * f_signal * t_low)
samples_high = np.sin(2 * np.pi * f_signal * t_high)
```

```
# Plotting
plt.figure(figsize=(10, 6))
plt.plot(t, signal, label='Original Signal')
plt.stem(t_low, samples_low, linefmt='r-', markerfmt='ro
    ', basefmt=" ", label='Low Sampling')
plt.stem(t_high, samples_high, linefmt='g-', markerfmt='
    go', basefmt=" ", label='High Sampling')
plt.title('Aliasing Demonstration')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend()
plt.grid()
plt.show()
```

1.2.2 Example 2: Signal Reconstruction

```
from scipy.signal import resample

# Reconstructing the signal using high sampling rate
num_samples = 1000
reconstructed_signal = resample(samples_high,
    num_samples)

# Plotting the reconstruction
plt.figure(figsize=(10, 6))
plt.plot(t, signal, label='Original Signal')
plt.plot(t, reconstructed_signal, label='Reconstructed
    Signal', linestyle='--')
plt.title('Signal Reconstruction')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend()
plt.grid()
plt.show()
```

1.3 Solving Basic Problems in Sampling and Reconstruction

This section provides step-by-step instructions on solving basic problems related to sampling and reconstruction of signals. It also demonstrates how to construct the various types of waves mentioned in the tasks, including sine, cosine, square, triangular, and sawtooth waves.

1.3.1 Step-by-Step Procedure

1. **Define the Signal:** Start by defining the original signal in terms of its type (sine, cosine, square, triangular, or sawtooth wave) and frequency f. Specify the time range t over which the signal will be evaluated.

2. **Choose the Sampling Frequency:** Select an appropriate sampling frequency f_s. Ensure $f_s > 2f$ to meet the Nyquist criterion, unless deliberately exploring aliasing.

3. **Sample the Signal:** Create a sampled version of the signal by evaluating it at discrete time points determined by the sampling period $T = 1/f_s$.

4. **Analyze Aliasing:** If f_s is below the Nyquist rate, observe the aliasing effect by plotting the sampled signal.

5. **Reconstruct the Signal:** Use interpolation techniques (e.g., sinc interpolation or linear interpolation) to reconstruct the original signal and compare it with the original.

1.3.2 Constructing Different Types of Waves

Below are examples of how to construct the basic signal types using Python:

Sine Wave

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

# Parameters
f = 5  # Frequency (Hz)
t = np.linspace(0, 1, 1000, endpoint=False)  # Time
   vector
sine_wave = np.sin(2 * np.pi * f * t)

# Plotting
plt.plot(t, sine_wave, label="Sine Wave")
plt.title("Sine Wave")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid()
plt.legend()
plt.show()
```

Cosine Wave

```python
# Cosine wave
cosine_wave = np.cos(2 * np.pi * f * t)

# Plotting
plt.plot(t, cosine_wave, label="Cosine Wave")
plt.title("Cosine Wave")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid()
plt.legend()
plt.show()
```

Square Wave

```python
from scipy.signal import square

# Square wave
square_wave = square(2 * np.pi * f * t)

# Plotting
plt.plot(t, square_wave, label="Square Wave")
plt.title("Square Wave")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid()
plt.legend()
plt.show()
```

Triangular Wave

```python
from scipy.signal import sawtooth

# Triangular wave (modification of sawtooth wave)
triangular_wave = sawtooth(2 * np.pi * f * t, width=0.5)

# Plotting
plt.plot(t, triangular_wave, label="Triangular Wave")
plt.title("Triangular Wave")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid()
plt.legend()
plt.show()
```

Sawtooth Wave

```
# Sawtooth wave
sawtooth_wave = sawtooth(2 * np.pi * f * t)

# Plotting
plt.plot(t, sawtooth_wave, label="Sawtooth Wave")
plt.title("Sawtooth Wave")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid()
plt.legend()
plt.show()
```

1.3.3 Combining Multiple Signals

You can combine different types of waves (e.g., sine and cosine) to create a composite signal:

```
# Mixed signal: sum of sine and cosine waves
f1 = 5  # Frequency of sine wave
f2 = 10  # Frequency of cosine wave
mixed_signal = np.sin(2 * np.pi * f1 * t) + np.cos(2 *
   np.pi * f2 * t)

# Plotting
plt.plot(t, mixed_signal, label="Mixed Signal (Sine +
   Cosine)")
plt.title("Mixed Signal")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid()
plt.legend()
plt.show()
```

1.3.4 Guidelines for Analyzing Sampling and Reconstruction

- **Aliasing Demonstration:** Choose $f_s < 2f$ and observe the distortion in the sampled signal.
- **Proper Reconstruction:** Choose $f_s > 2f$ and compare the reconstructed signal with the original.
- **Visualization:** Always plot the original, sampled, and reconstructed signals for comparison.

This foundation prepares to work on the assignments described in the next section.

2 Coding and Decoding Digital Signals

Objective

The aim of this laboratory session is to introduce students to the principles of coding and decoding digital signals, with practical applications of compression algorithms to optimize the representation and transmission of signals. The session focuses on signal coding, decoding, and reconstruction using Python.

2.1 Theoretical Background

2.1.1 Coding and Decoding of Digital Signals

Coding and decoding of digital signals involve converting continuous-time (analog) or discrete-time signals into a form that can be efficiently stored, transmitted, or processed.

Key Definitions:

- **Signal Coding:** The process of representing a signal in a compact or optimized form, often to reduce redundancy or to ensure robust transmission over communication channels.
- **Signal Decoding:** The reverse operation that reconstructs the original signal from the coded version.

Applications:

- Efficient storage of audio, video, or sensor signals.
- Error detection and correction in signal transmission.
- Compression for reduced bandwidth usage in communication systems.

2.1.2 Compression Algorithms for Signals

Compression methods are often used for efficient coding. Key types of compression for digital signals include:

- **Delta Encoding:** Encodes differences between successive signal samples.

- **Predictive Coding:** Estimates future signal samples based on previous ones and encodes only the residuals.
- **Quantization:** Reduces the number of bits required to represent a signal sample, typically at the cost of precision.
- **Transform-Based Compression:** Converts the signal into a different domain (e.g., Discrete Cosine Transform or Wavelet Transform) to identify and discard less significant components.

Compression Metrics:

- **Signal-to-Noise Ratio (SNR):** $\text{SNR} = 10 \log_{10} \left(\frac{\text{Signal Power}}{\text{Noise Power}} \right)$
- **Compression Ratio (CR):** $\text{CR} = \frac{\text{Original Signal Size}}{\text{Compressed Signal Size}}$

2.2 Practical Examples in Python

This section provides Python examples for coding and decoding digital signals.

2.2.1 Delta Encoding and Decoding

Delta encoding represents differences between consecutive signal samples.

```python
import numpy as np

# Delta Encoding
def delta_encode(signal):
    return np.diff(np.insert(signal, 0, signal[0]))

# Delta Decoding
def delta_decode(encoded_signal):
    return np.cumsum(encoded_signal)

# Example
original_signal = [1, 2, 2, 3, 5, 8]
encoded_signal = delta_encode(original_signal)
decoded_signal = delta_decode(encoded_signal)

print("Original Signal:", original_signal)
print("Encoded Signal:", encoded_signal)
print("Decoded Signal:", decoded_signal)
```

2.2.2 Quantization of Signal Samples

Quantization reduces the precision of signal samples to save storage or bandwidth.

```
def quantize(signal, levels):
    min_val, max_val = min(signal), max(signal)
    step_size = (max_val - min_val) / levels
    quantized_signal = np.round((signal - min_val) /
        step_size) * step_size + min_val
    return quantized_signal

# Example
original_signal = np.array([1.1, 2.3, 2.9, 3.7, 5.1])
quantized_signal = quantize(original_signal, levels=3)

print("Original Signal:", original_signal)
print("Quantized Signal:", quantized_signal)
```

2.2.3 Reconstructing Signals After Compression

This example demonstrates signal reconstruction after compression using a simple DCT.

```
from scipy.fftpack import dct, idct

# Apply Discrete Cosine Transform (DCT)
def apply_dct(signal):
    return dct(signal, norm='ortho')

# Reconstruct signal using inverse DCT
def reconstruct_signal(dct_signal, threshold):
    dct_signal[np.abs(dct_signal) < threshold] = 0
    return idct(dct_signal, norm='ortho')

# Example
original_signal = np.array([1, 2, 3, 4, 5, 6, 7, 8])
dct_signal = apply_dct(original_signal)
reconstructed_signal = reconstruct_signal(dct_signal,
   threshold=5)

print("Original Signal:", original_signal)
print("Reconstructed Signal:", np.round(
   reconstructed_signal, 2))
```

2.2.4 Trade-off Analysis

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import dct, idct

# Original Signal
N = 64  # Number of samples
t = np.linspace(0, 1, N)
signal = np.sin(2 * np.pi * 5 * t)  # Example sine wave

# Function for compression, reconstruction, and analysis
def analyze_tradeoff(signal, thresholds):
    original_size = len(signal)
    results = {"thresholds": [], "compression_ratios":
       [], "distortions": []}

    for threshold in thresholds:
        # Apply DCT
        dct_coeffs = dct(signal, norm='ortho')

        # Apply Thresholding (Compression)
        compressed_coeffs = np.where(abs(dct_coeffs) >
           threshold, dct_coeffs, 0)

        # Calculate Compression Ratio
        compressed_size = np.count_nonzero(
           compressed_coeffs)
        compression_ratio = original_size /
           compressed_size

        # Reconstruct Signal
        reconstructed_signal = idct(compressed_coeffs,
           norm='ortho')

        # Calculate Distortion (MSE)
        mse = np.mean((signal - reconstructed_signal) **
            2)

        # Store Results
        results["thresholds"].append(threshold)
        results["compression_ratios"].append(
           compression_ratio)
        results["distortions"].append(mse)

    return results

# Perform Analysis for a Range of Thresholds
thresholds = np.linspace(0.1, 1.0, 10)  # Threshold
   values
results = analyze_tradeoff(signal, thresholds)

# Plot Compression Ratio vs. Distortion
plt.figure(figsize=(8, 6))
plt.plot(results["compression_ratios"], results["
   distortions"], marker='o')
plt.title("Trade-off Between Compression Ratio and
   Signal Distortion")
plt.xlabel("Compression Ratio")
plt.ylabel("Mean Squared Error (Distortion)")
plt.grid()
plt.show()
```

2.3 Solving Basic Problems in Signal Coding

2.4 Problem 1: Delta Encoding and Decoding

Problem: Encode and decode the signal $[10, 12, 15, 15, 16]$ using delta coding.

2.4.1 Problem 2: Quantization Levels

Problem: Quantize the signal $[0.2, 0.5, 0.8, 1.2, 1.5]$ to 4 levels and calculate the quantization error.

2.4.2 Problem 3: Transform-Based Compression

Problem: Apply DCT to the signal $[5, 10, 15, 20, 25, 30]$ and reconstruct it with a threshold of 10.

2.4.3 Problem 4: Trade-off Analysis

Problem: Compare signal distortion and compression ratio for various thresholds in DCT compression.

3 Task Assignments for sampling and reconstruction

Each student will be assigned a unique variant of the task. The variants involve analyzing a signal under different conditions of sampling and reconstruction.

Variant 1. Analyze a sine wave with frequency $f = 10\,\text{Hz}$, sampled at $f_s = 15\,\text{Hz}$.

Variant 2. Analyze a cosine wave with frequency $f = 20\,\text{Hz}$, sampled at $f_s = 25\,\text{Hz}$.

Variant 3. Analyze a square wave with frequency $f = 5\,\text{Hz}$, sampled at $f_s = 8\,\text{Hz}$.

Variant 4. Analyze a triangular wave with frequency $f = 8\,\text{Hz}$, sampled at $f_s = 12\,\text{Hz}$.

Variant 5. Analyze a mixed signal (sum of sine and cosine waves) with $f_1 = 5\,\text{Hz}$, $f_2 = 15\,\text{Hz}$, sampled at $f_s = 20\,\text{Hz}$.

Variant 6. Investigate aliasing for a sine wave with $f = 30\,\text{Hz}$, sampled at $f_s = 40\,\text{Hz}$.

Variant 7. Reconstruct a sine wave with $f = 10\,\text{Hz}$, sampled at $f_s = 30\,\text{Hz}$.

Variant 8. Demonstrate aliasing for a sine wave with $f = 50\,\text{Hz}$, sampled at $f_s = 45\,\text{Hz}$.

Variant 9. Reconstruct a cosine wave with $f = 15\,\text{Hz}$, sampled at $f_s = 25\,\text{Hz}$.

Variant 10. Analyze a sawtooth wave with $f = 7\,\text{Hz}$, sampled at $f_s = 10\,\text{Hz}$.

Variant 11. Investigate reconstruction for a sine wave with $f = 10\,\text{Hz}$, sampled at $f_s = 50\,\text{Hz}$.

Variant 12. Demonstrate aliasing for a square wave with $f = 20\,\text{Hz}$, sampled at $f_s = 18\,\text{Hz}$.

Variant 13. Reconstruct a mixed signal with $f_1 = 5\,\text{Hz}$, $f_2 = 10\,\text{Hz}$, sampled at $f_s = 20\,\text{Hz}$.

Variant 14. Analyze aliasing for a sine wave with $f = 25\,\text{Hz}$, sampled at $f_s = 30\,\text{Hz}$.

Variant 15. Demonstrate reconstruction of a sine wave with $f = 8\,\text{Hz}$, sampled at $f_s = 16\,\text{Hz}$.

4 Task Assignments on Coding/Decoding

Below are 15 task variants to assign to students, each based on the four problems described earlier. Each student will work on a unique variant.

Variant 1. Solve Problem 1: Encode and decode the signal $[15, 20, 25, 25, 30]$ using delta coding.

Variant 2. Solve Problem 1: Encode and decode the signal $[8, 10, 12, 12, 14]$ using delta coding.

Variant 3. Solve Problem 1: Encode and decode the signal $[3, 6, 9, 9, 12]$ using delta coding.

Variant 4. Solve Problem 2: Quantize the signal $[1.2, 2.3, 3.1, 4.5, 5.7]$ to 3 levels and calculate the quantization error.

Variant 5. Solve Problem 2: Quantize the signal $[0.1, 0.5, 1.0, 1.5, 2.0]$ to 4 levels and calculate the quantization error.

Variant 6. Solve Problem 2: Quantize the signal $[0.3, 0.8, 1.2, 1.9, 2.7]$ to 5 levels and calculate the quantization error.

Variant 7. Solve Problem 3: Apply DCT to the signal $[10, 20, 30, 40, 50, 60]$ and reconstruct it with a threshold of 15.

Variant 8. Solve Problem 3: Apply DCT to the signal $[5, 10, 15, 20, 25, 30]$ and reconstruct it with a threshold of 8.

Variant 9. Solve Problem 3: Apply DCT to the signal $[12, 24, 36, 48, 60, 72]$ and reconstruct it with a threshold of 10.

Variant 10. Solve Problem 4: Compare signal distortion and compression ratio for thresholds of 5, 10, and 15 in DCT compression for the signal $[8, 16, 24, 32, 40, 48]$.

Variant 11. Solve Problem 4: Compare signal distortion and compression ratio for thresholds of 10, 20, and 30 in DCT compression for the signal $[10, 20, 30, 40, 50, 60]$.

Variant 12. Solve Problem 4: Compare signal distortion and compression ratio for thresholds of 2, 4, and 6 in DCT compression for the signal $[4, 8, 12, 16, 20, 24]$.

Variant 13. Solve Problem 4: Compare signal distortion and compression ratio for thresholds of 1, 5, and 10 in DCT compression for the signal $[2, 4, 6, 8, 10, 12]$.

Variant 14. Solve Problem 4: Compare signal distortion and compression ratio for thresholds of 5, 10, and 15 in DCT compression for the signal $[3, 6, 9, 12, 15, 18]$.

Variant 15. Solve Problem 4: Compare signal distortion and compression ratio for thresholds of 4, 8, and 12 in DCT compression for the signal $[3, 6, 9, 12, 15, 18]$.