Digital Information Processing Lab 2025

Class 1: Final Report

Group 8 : Vision Matrix

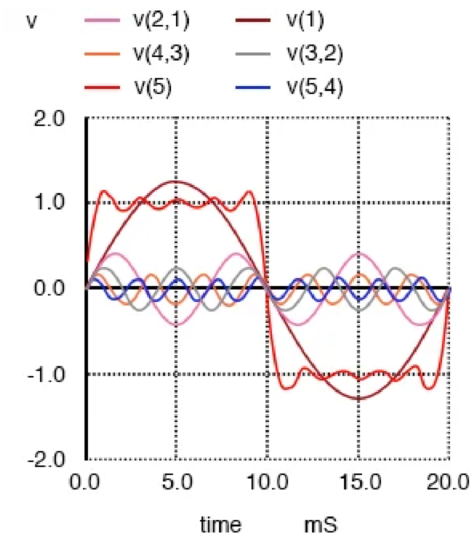
11.05.2025

Name	Mat. Nr.	Email
Navya Sajeev Warrier	252782	navya.sajeev@st.ovgu.de
Sanjay Pulparambil Girish	249360	sanjay.pulparambil@st.ovgu.de
Daniyal Rasheed Khan	248711	daniyal.khan@st.ovgu.de

Preparatory task

Task 1. Explain the relationship between square wave and sine wave signals

The relationship between rectangular and sinusoidal signals can be explained through Fourier series. A rectangular signal can be expressed as an infinite sum of sinusoidal signals, where each term in the sum is a harmonic of the fundamental frequency. The amplitude and phase of each harmonic are determined by the shape of the rectangular signal. In contrast, a sinusoidal signal is itself a single harmonic and can be considered the simplest form of a periodic signal. When a rectangular signal is filtered through a low-pass filter, the resulting output is a sinusoidal signal with the same fundamental frequency as the rectangular signal. This is because the low-pass filter removes all the harmonics of the rectangular signal except for the fundamental frequency, which is preserved in the output. Thus, we can see that rectangular signals and sinusoidal signals are intimately connected, and that the Fourier series provides a powerful tool for analyzing and understanding this relationship [1]

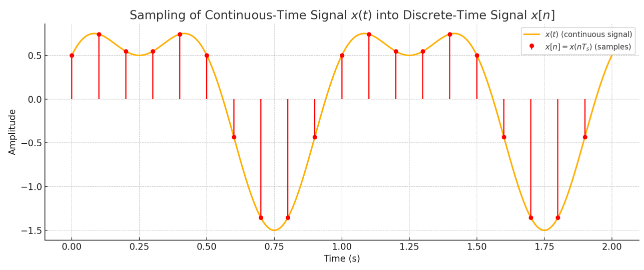


Sum of 1st, 3rd, 5th, 7th and 9th harmonics approximates square wave. [2]

Task 2. Describe the sampling schematically and use it to explain the formula for sampling in the time domain.

Sampling in the Time Domain

1. Schematic Description of Sampling



Sampling is the process of converting a continuous-time signal (x(t)) into a discrete-time signal (x[n]) by taking periodic samples at every (T_s) seconds.

The actual discrete-time sequence used in practice is:

[x[n] = x(nT_s)]

Where:

- x(t) : Continuous-time signal
- T : Sampling period (time between samples)
- x[n] : Discrete-time signal

Sampling is modeled by multiplying $x(t)$ with an impulse train $p(t)$:

$$x_\delta(t) = x(t) \cdot p(t)$$

$$p(t) = \sum_{n=-\infty}^{\infty} \delta(t - nT_s) \quad \text{(periodic impulse train)}$$

$$x_\delta(t) = \sum_{n=-\infty}^{\infty} x(nT_s) \cdot \delta(t - nT_s) \quad \text{(weighted impulses)}[1]$$

Here, $x_\delta(t)$ represents the sampled signal as a train of impulses at intervals T_s, each scaled by the original signal's value at that time. This means the sampled signal consists of Dirac impulses located at the sampling instants t = nTs, each weighted by the value of the signal at that point.

References of Exercise 3b:

[1] https://www.control.utoronto.ca/~jwsimpson/courses/ECE216H-CompleteNotes-1x1.pdf

[2]Sampling figure: GitHub source (generated by chatgpt)

Class 1: Sampling and Reconstruction

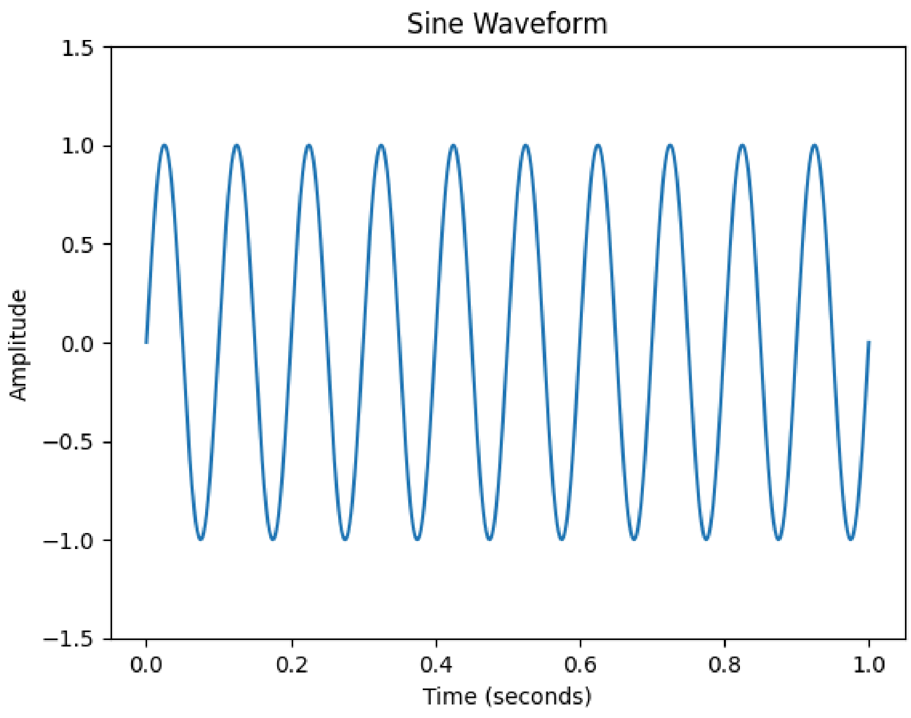
Exercise 1: Ploting

Generate signals and display them:

- sine
- rectangle
- triangle

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

```
def plot_sine_wave(amplitude, frequency, phase, duration):
    """Plots a sine wave with the given parameters."""
    # Generate time samples
    t = np.linspace(0, duration, 1000)

    # Compute the sine wave
    y = amplitude * np.sin(2 * np.pi * frequency * t + phase)

    # Plot the waveform
    plt.plot(t, y)
    plt.xlabel('Time (seconds)')
    plt.ylabel('Amplitude')
    plt.ylim(top=1.5, bottom=-1.5)
    plt.title('Sine Waveform')
    plt.show()

plot_sine_wave(amplitude=1, frequency=10, phase=0, duration=1)
```

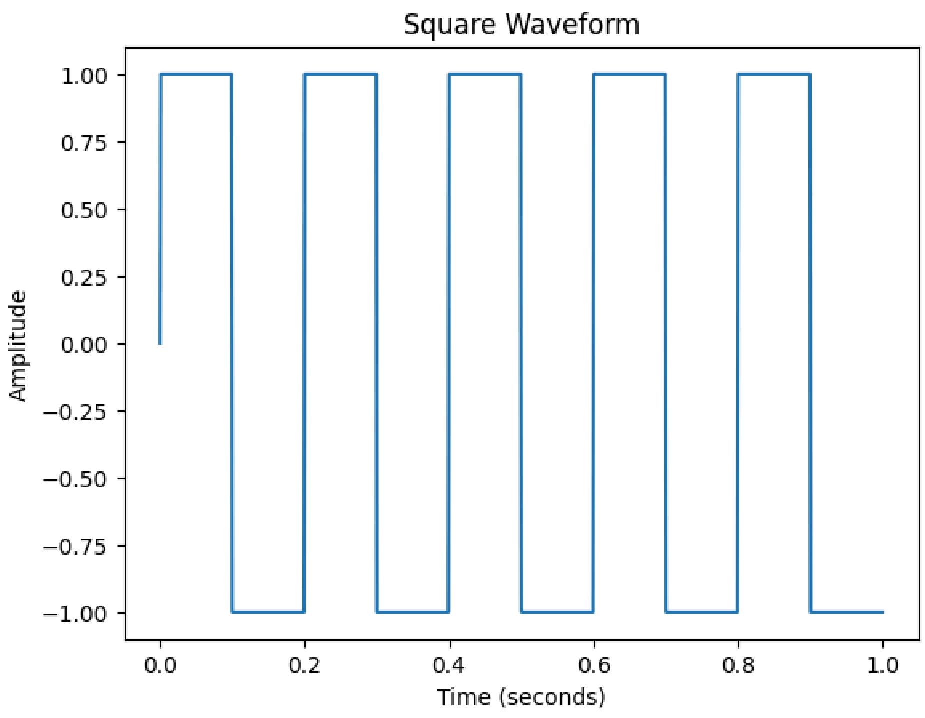


This function takes four arguments: amplitude, frequency, phase, and duration. The amplitude parameter sets the maximum amplitude of the sine wave, the frequency parameter sets the frequency of the sine wave in hertz, the phase parameter sets the phase offset of the sine wave in radians, and the duration parameter sets the length of time in seconds over which the sine wave is plotted.

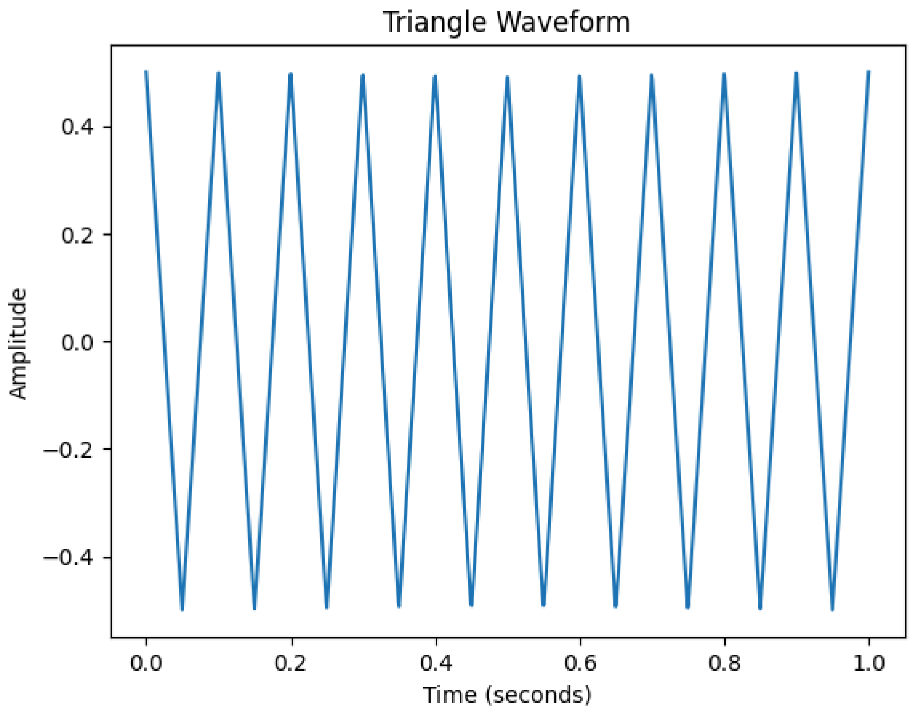
(consider looking into a plotting tutorial)

`plot_sine_wave(amplitude=1, frequency=10, phase=0, duration=1)` displays a sine wave with the frequency of 10 hz in the timespan of 1 s.

```
def plot_square_wave(amplitude, frequency, phase, duration):
    """Plots a square wave with the given parameters."""
    # Generate time samples
    t = np.linspace(0, duration, 1000)

    # Compute the square wave
    y = amplitude * np.sign(np.sin(2 * np.pi * frequency * t + phase))

    # Plot the waveform
    plt.plot(t, y)
    plt.xlabel('Time (seconds)')
    plt.ylabel('Amplitude')
    plt.title('Square Waveform')
    plt.show()


def plot_triangle_wave(amplitude, frequency, phase, duration):
    """Plots a triangle wave with the given parameters."""
    # Generate time samples
    t = np.linspace(0, duration, 1000)

    # Compute the triangle wave
    y = amplitude * np.abs(2 * np.mod(t * frequency + phase, 1.0) - 1) - amplitude/2

    # Plot the waveform
    plt.plot(t, y)
    plt.xlabel('Time (seconds)')
    plt.ylabel('Amplitude')
    plt.title('Triangle Waveform')
    plt.show()


plot_square_wave(amplitude=1, frequency=5, phase=0, duration=1)
```



```
plot_triangle_wave(amplitude=1, frequency=10, phase=0, duration=1)
```



These functions work similarly to the plot_sine_wave function. The plot_square_wave function takes five arguments: amplitude, frequency, duty_cycle, phase, and duration. The amplitude, frequency, phase, and duration parameters work the same as in the plot_sine_wave function. The duty_cycle parameter sets the percentage of time that the square wave is in the high state. This function generates a square wave using the np.sign and np.mod functions.

The plot_triangle_wave function takes four arguments: amplitude, frequency, phase, and duration. This function generates a triangle wave using the np.abs and np.mod functions.

Interactive plots

You can use "widgets" to adjust the parameters for different functions. In this case, we use two "sliders" to adjust the frequency and amplitude. To ensure that the output also changes when a change is made, there is an "Observer" function that runs when a change is made.

```
import ipywidgets as widgets
from IPython.display import display, clear_output

output = widgets.Output()
freq_slider = widgets.FloatSlider(value=2, min=0, max=10, step=0.1, description='Frequenz:')
amp_slider = widgets.FloatSlider(value=1, min=-2, max=2, step=0.1, description='Amplitude:')


# Function to handle UI element change
def on_ui_element_change(change):
    with output:
        clear_output(wait=True)
        plot_sine_wave(amplitude=amp_slider.value, frequency=freq_slider.value, phase=0, duration=1)

freq_slider.observe(on_ui_element_change, names='value')
amp_slider.observe(on_ui_element_change, names='value')

display(freq_slider)
display(amp_slider)
display(output)
```

```
Frequenz:    2.00
Amplitude:   1.00
```

Exercise 2a: interactive plot

- Write a function that displays all three signal types in one plot on top of each other in different colors.
- Use sliders to change settings add a button to reset values

Here is some code for help:

```
# Define a function to be called when the button is clicked
def on_button_click(b):
    print("Button clicked!")

# Create a button widget
button = widgets.Button(description="Click Me!")

# Attach the function to the button's click event
button.on_click(on_button_click)

# Display the button
display(button)
```

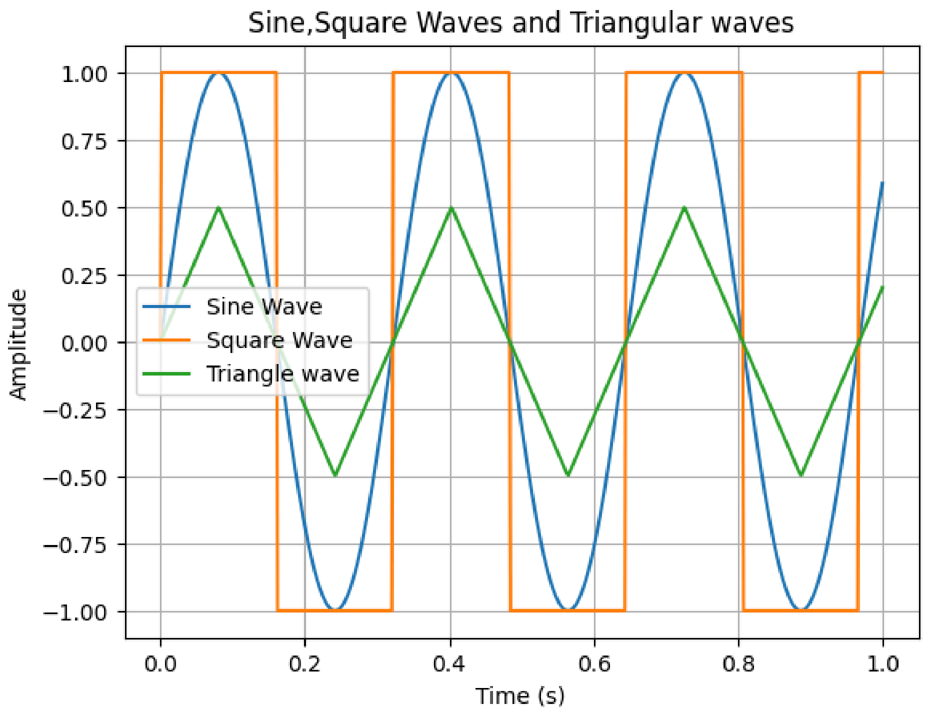
```
Click Me!
```

```
# ToDo
#check both on 1 graph:
import numpy as np
from IPython.display import clear_output
import matplotlib.pyplot as plt
import ipywidgets as widgets
from IPython.display import display



output = widgets.Output()
freq_slider = widgets.FloatSlider(value=2, min=0, max=10, step=0.1, description='Frequenz:')
amp_slider = widgets.FloatSlider(value=1, min=-2, max=2, step=0.1, description='Amplitude:')

# Define a function to be called when the button is clicked
def on_button_click(b):
    freq_slider.value = 2  # Reset frequency slider to its initial value (2)
    print("Button clicked! Frequency reset.")
# Create a button widget
button = widgets.Button(description="Reset!")

# Attach the function to the button's click event
button.on_click(on_button_click)

# Display the button
display(button)

# Function to handle UI element change
def on_ui_element_change(change):
    with output:
        clear_output(wait=True)
       # Combine plotting logic for both waves in a single function
        time = np.linspace(0, 1, 500)
        sine_wave = amp_slider.value * np.sin(2 * np.pi * freq_slider.value * time)
        square_wave = amp_slider.value * np.sign(np.sin(2 * np.pi * freq_slider.value * time))
        triangle_wave = amp_slider.value * np.abs(2 * np.mod(time * freq_slider.value -0.25 , 1.0) - 1) - amp_sl
        # Plot both waves on the same graph
        plt.plot(time, sine_wave, label='Sine Wave')
        plt.plot(time, square_wave, label='Square Wave')
        plt.plot(time, triangle_wave, label='Triangle wave')
        plt.xlabel("Time (s)")
        plt.ylabel("Amplitude")
        plt.title("Sine,Square Waves and Triangular waves")
        plt.grid(True)
        plt.legend()  # Add a legend to distinguish the waves
        plt.show()

freq_slider.observe(on_ui_element_change, names='value')
amp_slider.observe(on_ui_element_change, names='value')

display(freq_slider)
display(amp_slider)
display(output)
```

Reset!

Frequenz: 3.10

Amplitude: 1.00



Exercise 2b: interactive addition of two sine signals

Write a function that adds and displays two sinusoidal signals.

- The signals should be adjustable in frequency and phase
- The individual signals and the resulting signal should be displayed

Here is some code snipets for help:

```
# Create interactive widgets
freq1_slider = widgets.FloatSlider(value=2, min=0.0, max=20, step=0.1, description='Frequenz 1:')
phase1_slider = widgets.FloatSlider(value=0, min=-np.pi, max=np.pi, step=1/8*np.pi, description='Phase 1:')

freq2_slider = widgets.FloatSlider(value=8, min=0.0, max=20, step=0.1, description='Frequenz 2:')
phase2_slider = widgets.FloatSlider(value=0, min=-np.pi, max=np.pi, step=1/8*np.pi, description='Phase 2:')
```

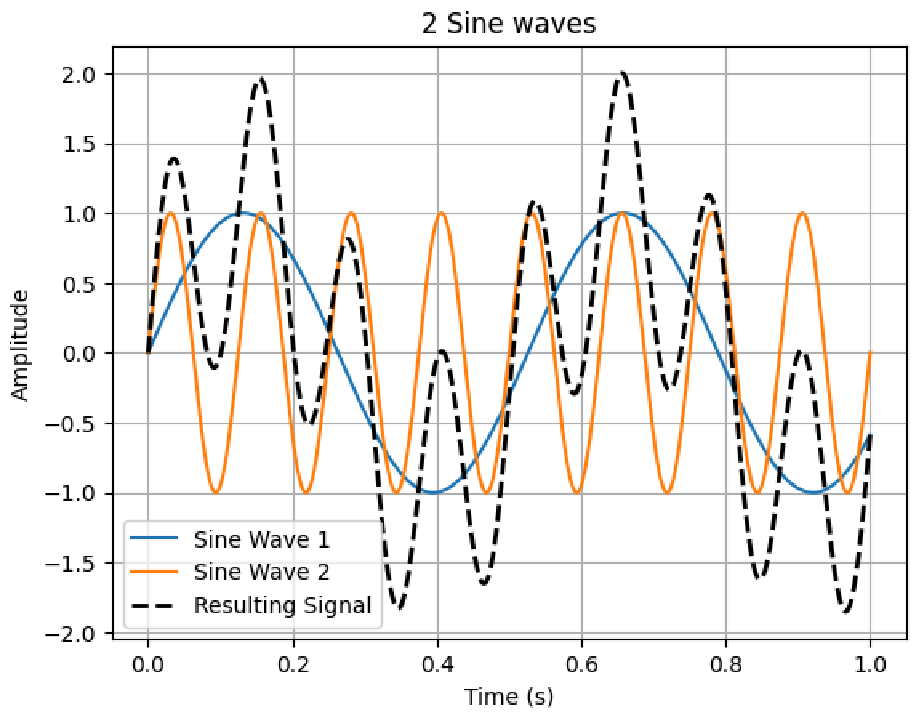
```
# Optional formatting
freq1_slider.style.handle_color = 'lightblue'
phase1_slider.style.handle_color = 'lightblue'
freq2_slider.style.handle_color = 'orange'
phase2_slider.style.handle_color = 'orange'


#check both on 1 graph: Exercise 2b
import numpy as np
from IPython.display import clear_output
import matplotlib.pyplot as plt
import ipywidgets as widgets
from IPython.display import display

output = widgets.Output()
freq1_slider = widgets.FloatSlider(value=2, min=0.0, max=20, step=0.1, description='Frequenz 1:')
phase1_slider = widgets.FloatSlider(value=0, min=-np.pi, max=np.pi, step=1/8*np.pi, description='Phase 1:')

freq2_slider = widgets.FloatSlider(value=8, min=0.0, max=20, step=0.1, description='Frequenz 2:')
phase2_slider = widgets.FloatSlider(value=0, min=-np.pi, max=np.pi, step=1/8*np.pi, description='Phase 2:')

# Fixed amplitude value
amplitude = 1

# Function to handle UI element change
def on_ui_element_change(change):
    with output:
        clear_output(wait=True)
       # Combine plotting logic for both waves in a single function
        time = np.linspace(0, 1, 1000)
        sine_wave1 = amplitude * np.sin(2 * np.pi * freq1_slider.value * time + phase1_slider.value)
        sine_wave2 = amplitude * np.sin(2 * np.pi * freq2_slider.value * time + phase2_slider.value)
        resulting_signal = sine_wave1 + sine_wave2

         # Plot both waves on the same graph
        plt.plot(time, sine_wave1, label='Sine Wave 1')
        plt.plot(time, sine_wave2, label='Sine Wave 2')
        plt.plot(time, resulting_signal, label='Resulting Signal', linestyle='--', linewidth=2, color='black') #
        plt.xlabel("Time (s)")
        plt.ylabel("Amplitude")
        plt.title("2 Sine waves")
        plt.grid(True)
        plt.legend()  # Add a legend to distinguish the waves
        plt.show()

freq1_slider.observe(on_ui_element_change, names='value')
phase1_slider.observe(on_ui_element_change, names='value')
freq2_slider.observe(on_ui_element_change, names='value')
freq2_slider.observe(on_ui_element_change, names='value')


# Optional formatting
freq1_slider.style.handle_color = 'lightblue'
phase1_slider.style.handle_color = 'lightblue'
freq2_slider.style.handle_color = 'orange'
phase2_slider.style.handle_color = 'orange'

display(freq1_slider)
display(phase1_slider)
display(freq2_slider)
display(phase2_slider)
display(output)
```

Frequenz 1: 1.90

Phase 1: 0.00

Frequenz 2: 8.00

Phase 2: 0.00



Excercise 3a: sampling

Try out sampling of the different signals

- Sine signal
- Triangular signal
- Square wave signal

Compare the sampled signal with the original signal, while variating the sampling frequency.

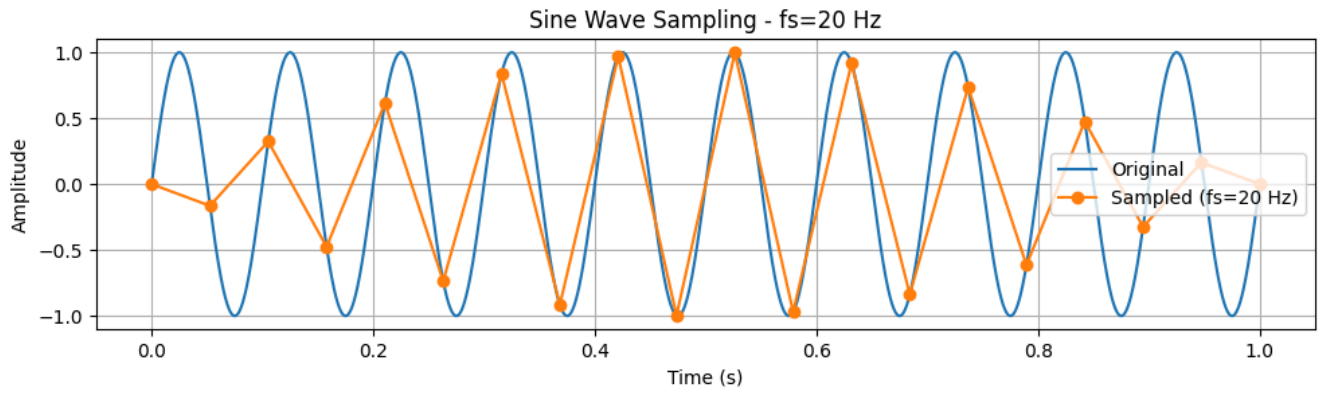
- What observations can be made?
- What frequency is necessary to recognise the signal?

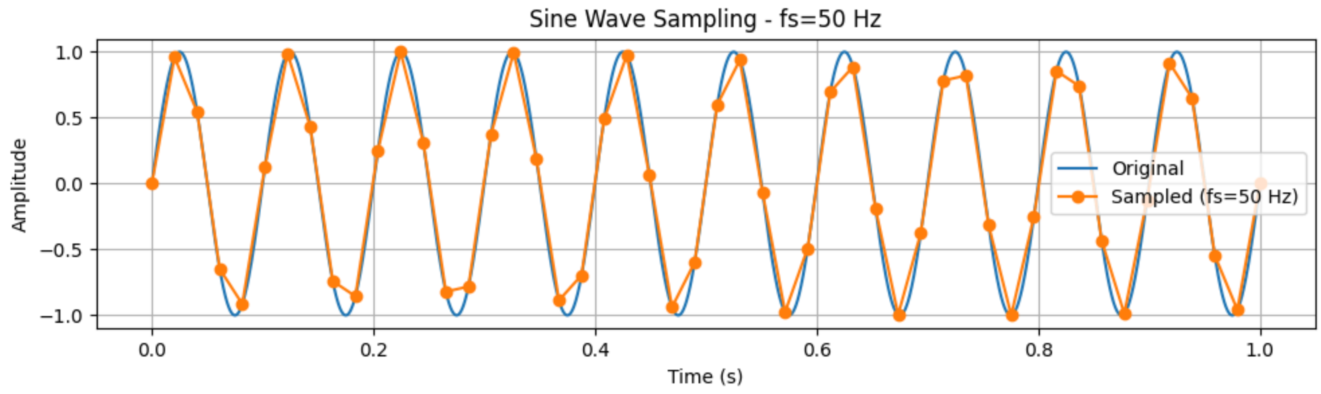
```
def generate_sine_wave(amplitude, frequency, phase, duration):
    """Generates a sine wave with the given parameters."""
    # Generate time samples
    t = np.linspace(0, duration, int(duration*44100))
    # Compute the sine wave
    y = amplitude * np.sin(2 * np.pi * frequency * t + phase)
    return y

def generate_square_wave(amplitude, frequency, duty_cycle, phase, duration):
    """Generates a square wave with the given parameters."""
    # Generate time samples
    t = np.linspace(0, duration, int(duration*44100))
    # Compute the square wave
    y = amplitude * np.sign(np.sin(2 * np.pi * frequency * t + phase))
    # Set the duty cycle
    y[np.mod(t, 1.0/frequency) > duty_cycle/frequency] = -amplitude

    return y

def generate_triangle_wave(amplitude, frequency, phase, duration):
    """Generates a triangle wave with the given parameters."""
    # Generate time samples
    t = np.linspace(0, duration, int(duration*44100))

    # Compute the triangle wave
    y = amplitude * np.abs(2 * np.mod(t * frequency + phase, 1.0) - 1) - amplitude/2

    return y

def sample_waveform(waveform, frequency, duration):
    """Samples the given waveform at the given frequency."""
    # Generate time samples
    t = np.linspace(0, duration, int(duration*frequency))
    # Sample the waveform
    y = np.interp(t, np.linspace(0, duration, len(waveform)), waveform)

    return y
```

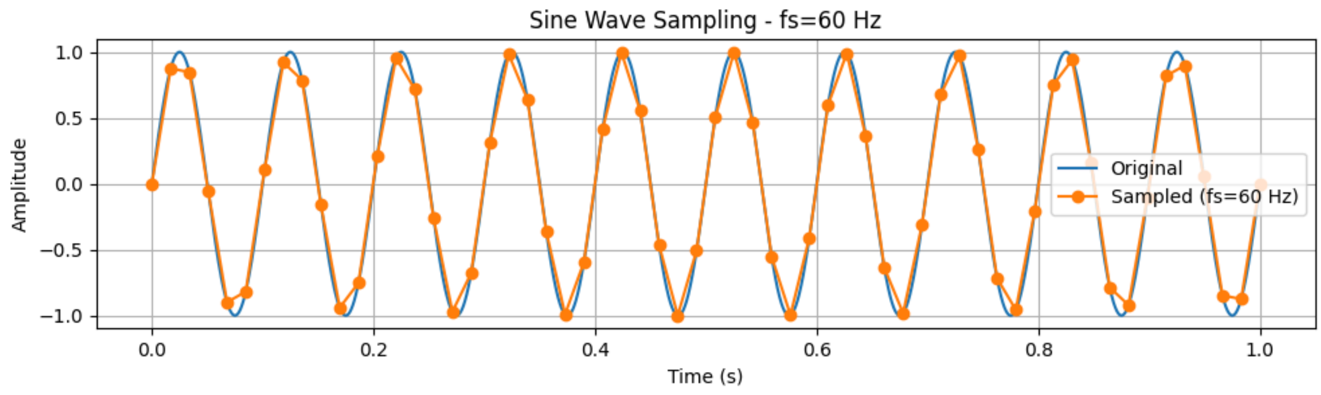
These functions all work the same as in Excercise 1, but now they return the generated waveforms as NumPy arrays instead of plotting them directly. The new sample_waveform function takes three arguments: waveform, which is the waveform to be sampled, frequency, which is the sampling frequency in Hz, and duration, which is the duration of the sampled waveform in seconds. This function uses the np.interp function to resample the waveform at the desired sampling frequency.

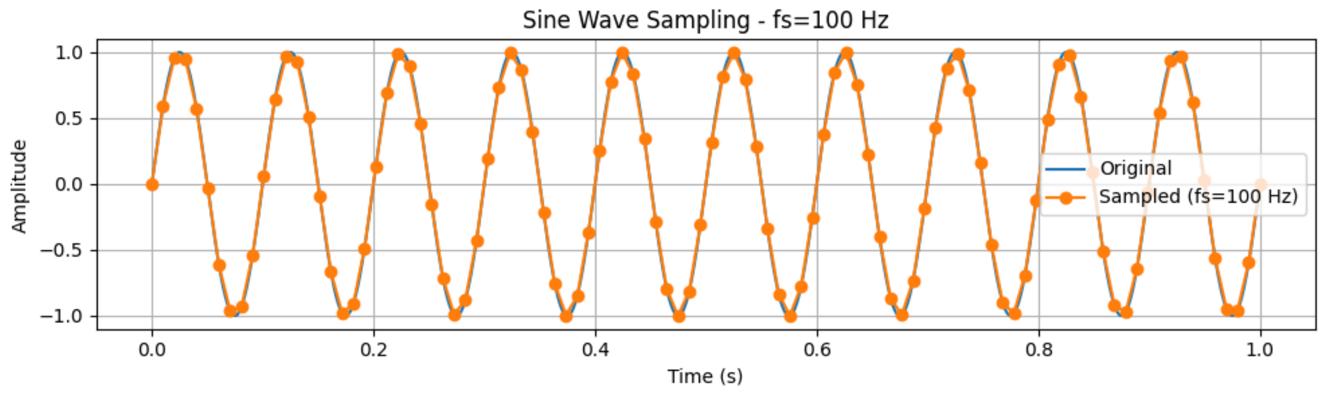
```
# ToDo
import numpy as np
import matplotlib.pyplot as plt

def generate_sine_wave(amplitude, frequency, phase, duration):
    """Generates a sine wave with the given parameters."""
    # Generate time samples
    t = np.linspace(0, duration, int(duration*44100))
    # Compute the sine wave
    y = amplitude * np.sin(2 * np.pi * frequency * t + phase)
    return y

def generate_square_wave(amplitude, frequency, duty_cycle, phase, duration):
    """Generates a square wave with the given parameters."""
    # Generate time samples
    t = np.linspace(0, duration, int(duration*44100))
    # Compute the square wave
    y = amplitude * np.sign(np.sin(2 * np.pi * frequency * t + phase))
    # Set the duty cycle
    y[np.mod(t, 1.0/frequency) > duty_cycle/frequency] = -amplitude

    return y

def generate_triangle_wave(amplitude, frequency, phase, duration):
    """Generates a triangle wave with the given parameters."""
    # Generate time samples
    t = np.linspace(0, duration, int(duration*44100))

    # Compute the triangle wave
    y = amplitude * np.abs(2 * np.mod(t * frequency + phase, 1.0) - 1) - amplitude/2

    return y
def sample_waveform(waveform, frequency, duration):
    """Samples the given waveform at the given frequency."""
    # Generate time samples
    t = np.linspace(0, duration, int(duration*frequency))
    # Sample the waveform
    y = np.interp(t, np.linspace(0, duration, len(waveform)), waveform)

    return y
```

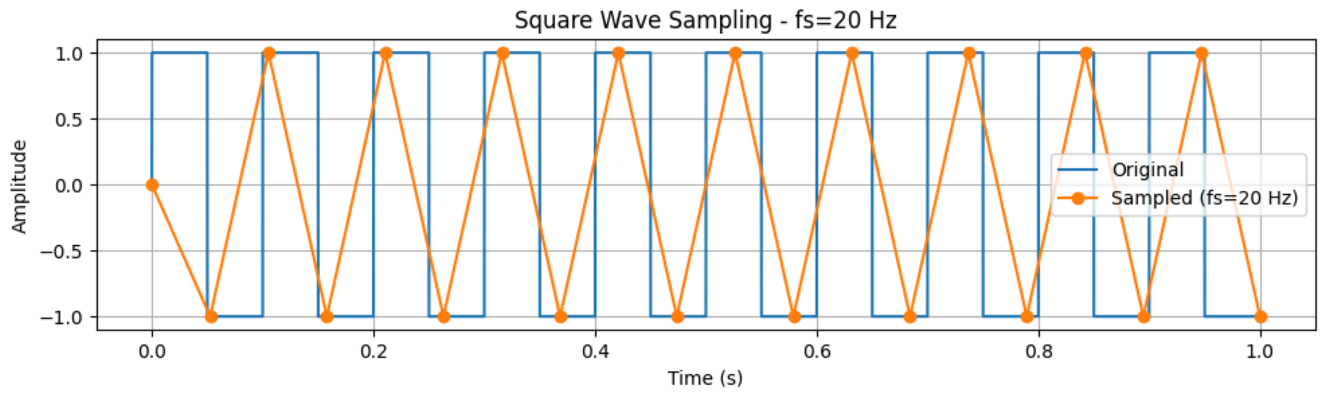
```
# --- Main execution ---
amplitude = 1
frequency = 10  # Frequency of the original signals
phase = 0
duration = 1  # Duration of the signal
duty_cycle = 0.5 #duty cycle for the square wave



# Generate original waveforms
original_sine = generate_sine_wave(amplitude, frequency, phase, duration)
original_square = generate_square_wave(amplitude, frequency, duty_cycle, phase, duration)
original_triangle = generate_triangle_wave(amplitude, frequency, phase, duration)

# Sampling frequencies to test
sampling_frequencies = [ 20, 50, 60, 100]  # Hz

# Plot separate figures for each waveform with 5 rows for sampling frequencies
waveforms = {'Sine': original_sine, 'Square': original_square, 'Triangle': original_triangle}
for waveform_name, original_waveform in waveforms.items():
    print('                                              ')
    plt.figure(figsize=(10,12))  # Create a new figure for each waveform
    for i, fs in enumerate(sampling_frequencies):
        # Sample the waveform
        sampled_waveform = sample_waveform(original_waveform, fs, duration)
        # Create subplot for the current sampling frequency
        plt.subplot(len(sampling_frequencies), 1, i + 1) #(rows,col,selected_index)
        # Plot the original and sampled signal
        plt.plot(np.linspace(0, duration, len(original_waveform)), original_waveform, label='Original')
        plt.plot(np.linspace(0, duration, len(sampled_waveform)), sampled_waveform, label='Sampled (fs={} Hz)'.f
        plt.title(f'{waveform_name} Wave Sampling - fs={fs} Hz')
        plt.xlabel("Time (s)")
        plt.ylabel("Amplitude")
        plt.grid(True)
        plt.tight_layout()  # Add this line to adjust spacing
        plt.legend()

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



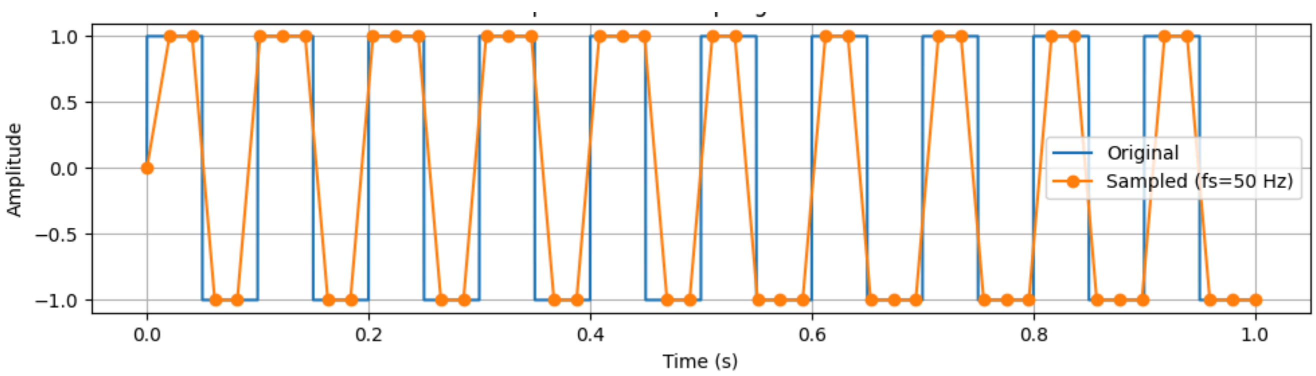


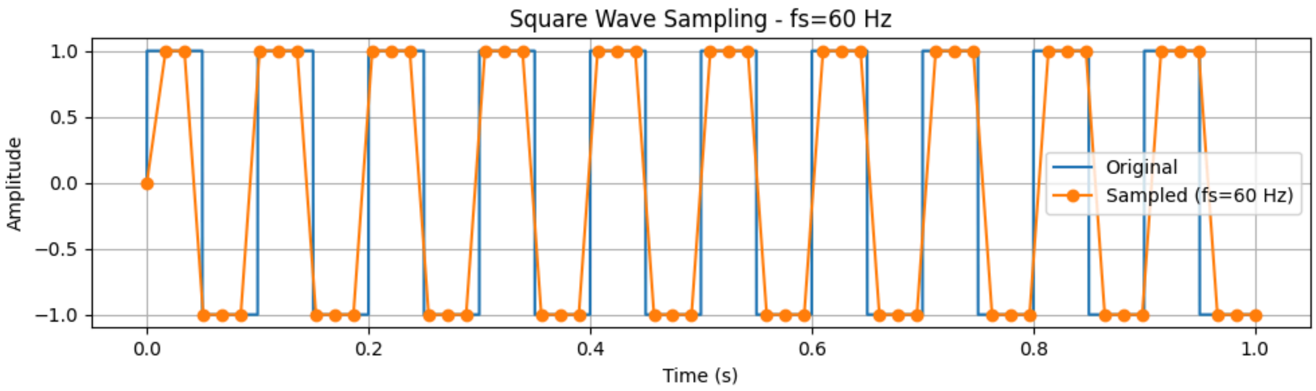


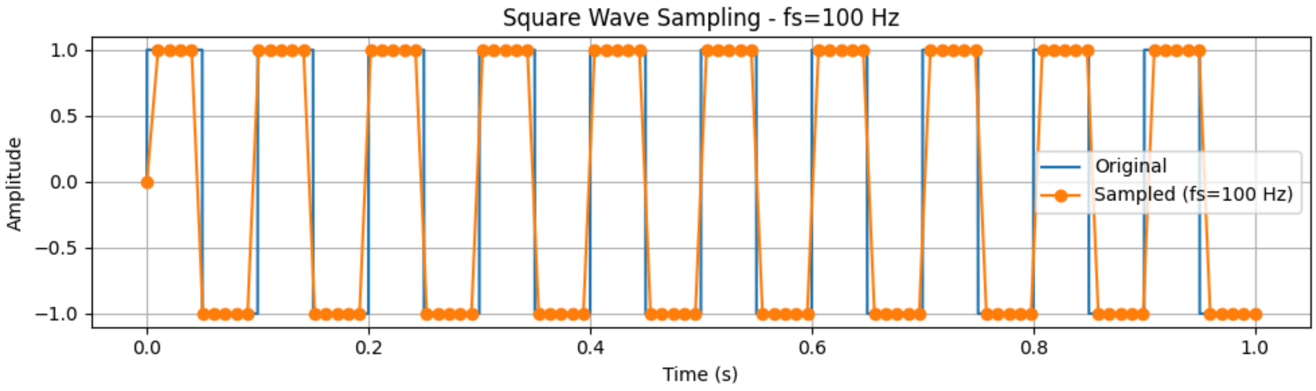


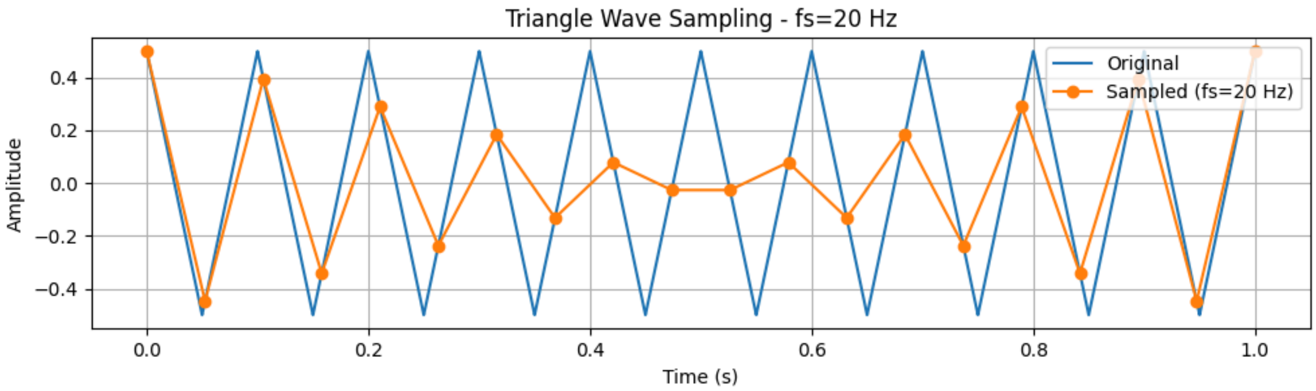


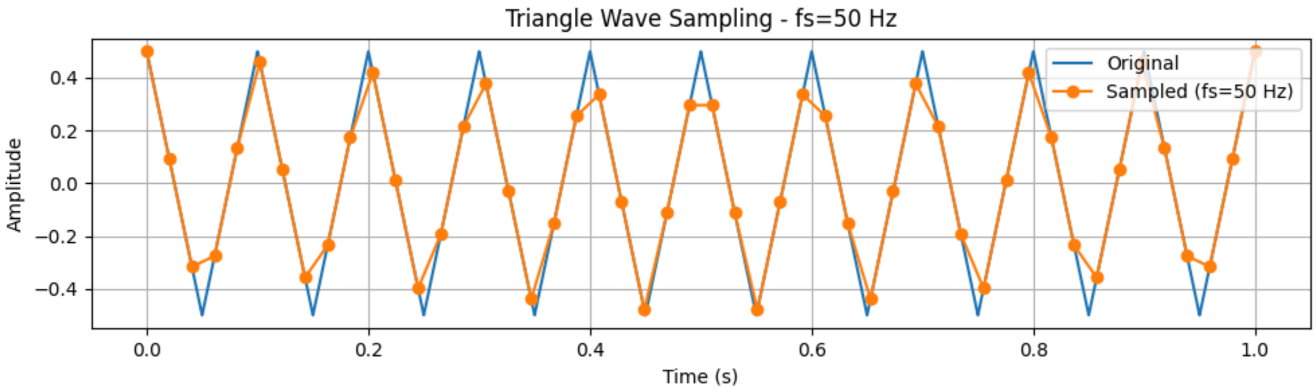
Square Wave Sampling - fs=50 Hz

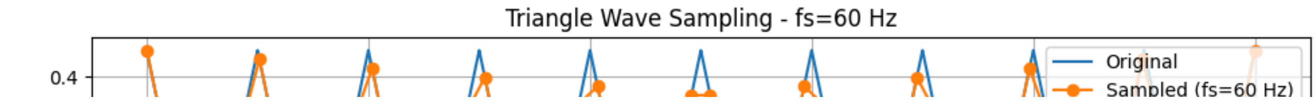

Amplitude
Time (s)
Original
Sampled (fs=50 Hz)

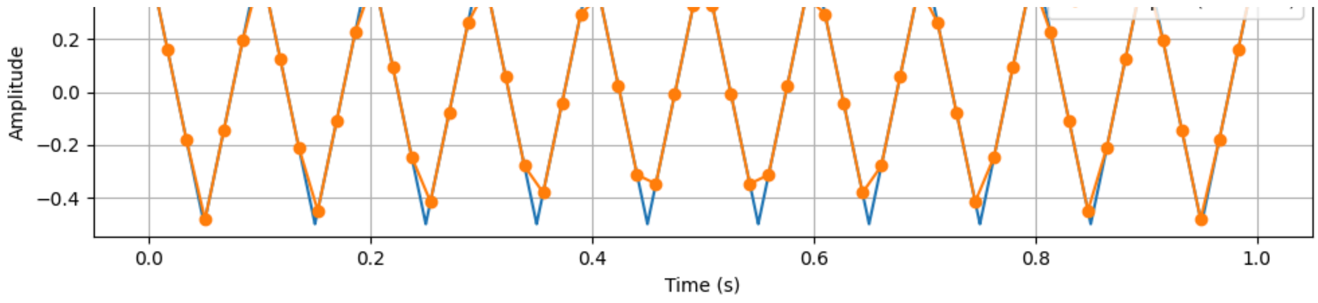

Square Wave Sampling - fs=60 Hz
Amplitude
Time (s)
Original
Sampled (fs=60 Hz)


Square Wave Sampling - fs=100 Hz
Amplitude
Time (s)
Original
Sampled (fs=100 Hz)

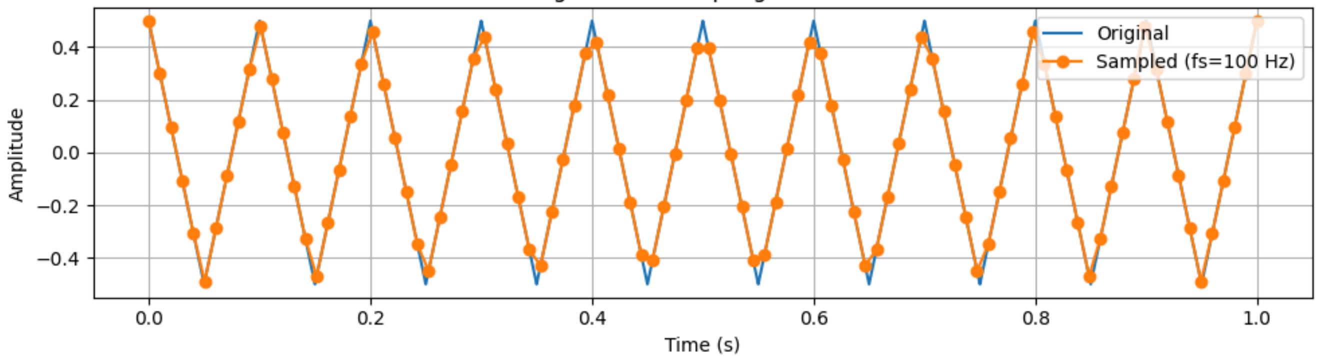

Triangle Wave Sampling - fs=20 Hz
Amplitude
Time (s)
Original
Sampled (fs=20 Hz)


Triangle Wave Sampling - fs=50 Hz
Amplitude
Time (s)
Original
Sampled (fs=50 Hz)


Triangle Wave Sampling - fs=60 Hz
Original
Sampled (fs=60 Hz)



Triangle Wave Sampling - fs=100 Hz



--> observations made:

- the sign wave - satisfies the Nyquist theory
- the square wave - has very sharp edges and needs higher harmonics dominate shape
- the triangular wave -Rich in odd harmonics and requires higher sampling to preserve shape.

  frequency necessary to recognise the signal:

Here, fs represents the sampling frequency (the rate at which samples are taken) and $f$max represents the maximum frequency present in the signal, then according to the Nyquist theorem[1]:

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

- If the sampling frequency is too low compared to the frequency of the signal being sampled, aliasing occurs.
- Aliasing leads to distortion in thesampled signal, making it difficult to reconstruct the original signal accurately.
- Higher sampling frequencies generally result in more accurate representation and better recognition of the original signal's characteristics.

Task1: References

[1] https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem

Exercise 3b: advanced sampling methods

Create a plot using different sampling functions:

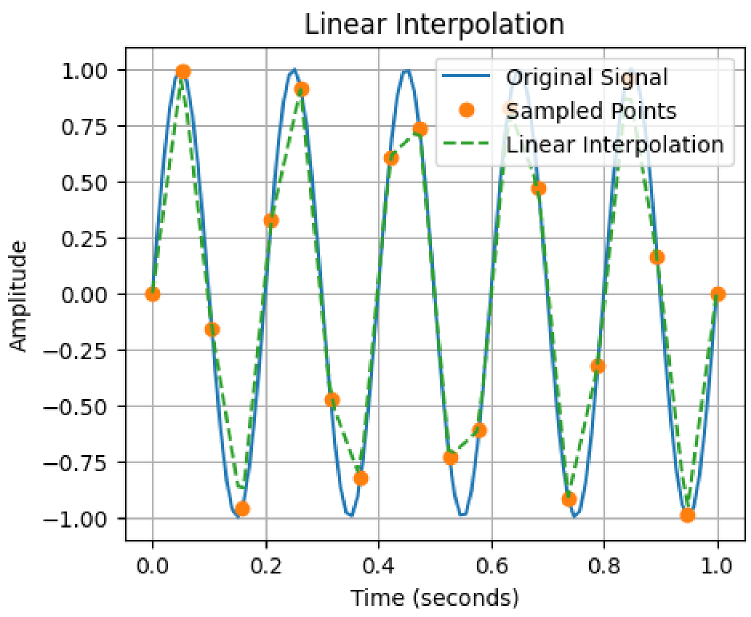
- Zero-order hold
- Sinc Reconsturcion
- Linear interpolation of sampled values

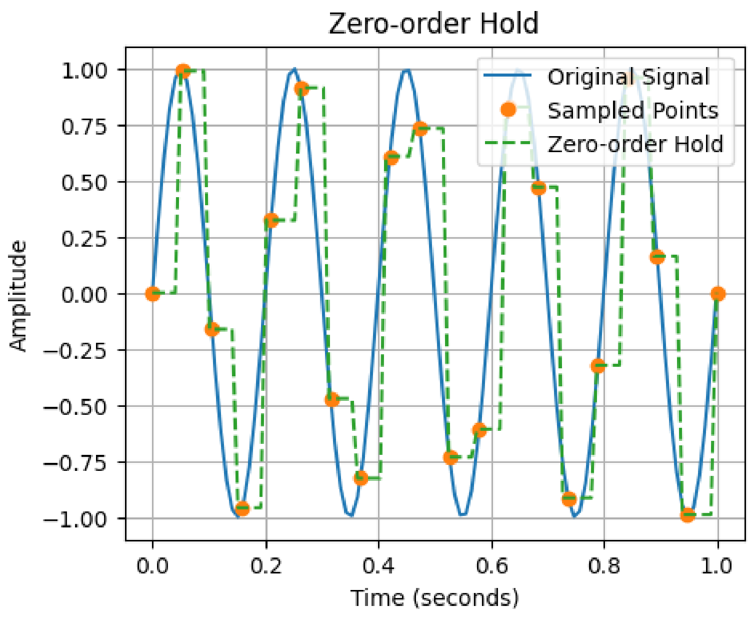
Compare the graphs.

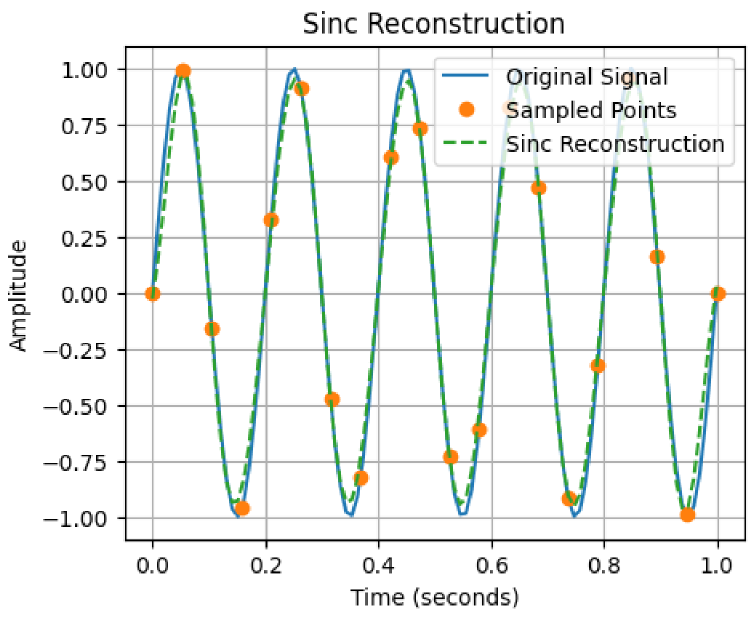
- What differences can be seen ?

```
#"Exercise 3b:"

import numpy as np
import matplotlib.pyplot as plt

# Zero-order hold reconstruction
def zero_order_hold(sampled_signal, sample_times, t):
    reconstructed = np.zeros_like(t) #t: Represents the continuous time points for which the signal needs to be
    sample_indices = np.searchsorted(t, sample_times, side='right') - 1
    sample_indices = np.clip(sample_indices, 0, len(t) - 1)
    for i in range(len(sample_times) - 1):
        reconstructed[sample_indices[i]:sample_indices[i + 1]] = sampled_signal[i]
    reconstructed[sample_indices[-1]:] = sampled_signal[-1]
    return reconstructed

# Sinc reconstruction
def sinc_reconstruct(sampled_signal, sample_times, t, reco_sample_rate):
    reconstructed = np.zeros_like(t)
    for i, ts in enumerate(sample_times):
        reconstructed += sampled_signal[i] * np.sinc((t - ts) * reco_sample_rate)
    return reconstructed

# Generate a sample signal (e.g., a sine wave)
sample_rate = 100  # Hz
duration = 1  # seconds
t = np.linspace(0, duration, sample_rate * duration)
original_signal = np.sin(2 * np.pi * 5 * t)  # 5 Hz sine wave
```

```
# Sample the signal
sampling_frequency = 20  # Hz
sample_times = np.linspace(0, duration, int(duration*sampling_frequency))
sampled_signal = np.interp(sample_times, t, original_signal)

# Reconstruct the signal using different methods
reconstructed_signal_linear = np.interp(t, sample_times, sampled_signal)
reconstructed_signal_zoh = zero_order_hold(sampled_signal, sample_times, t)
reconstructed_signal_sinc = sinc_reconstruct(sampled_signal, sample_times, t, reco_sample_rate=sampling_frequenc

# Plot Linear Interpolation
plt.figure(figsize=(5, 4))
plt.plot(t, original_signal, label='Original Signal')
plt.plot(sample_times, sampled_signal, 'o', label='Sampled Points')
plt.plot(t, reconstructed_signal_linear,label='Linear Interpolation',linestyle='--')
plt.title('Linear Interpolation')
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.legend(loc='upper right')
plt.show()

# Plot Zero-order Hold
plt.figure(figsize=(5,4))
plt.plot(t, original_signal, label='Original Signal')
plt.plot(sample_times, sampled_signal, 'o', label='Sampled Points')
plt.plot(t, reconstructed_signal_zoh, label='Zero-order Hold',linestyle='--')
plt.title('Zero-order Hold')
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.legend(loc='upper right')
plt.show()

# Plot Sinc Reconstruction
plt.figure(figsize=(5,4))
plt.plot(t, original_signal, label='Original Signal')
plt.plot(sample_times, sampled_signal, 'o', label='Sampled Points')
plt.plot(t, reconstructed_signal_sinc, label='Sinc Reconstruction',linestyle='--')
plt.title('Sinc Reconstruction')
plt.xlabel('Time (seconds)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.legend(loc='upper right')
plt.show()
```







```
# Zero-order hold reconstruction
def zero_order_hold(sampled_signal, sample_times, t):
    reconstructed = np.zeros_like(t)
    sample_indices = np.searchsorted(t, sample_times, side='right') - 1
    sample_indices = np.clip(sample_indices, 0, len(t) - 1)
    for i in range(len(sample_times) - 1):
        reconstructed[sample_indices[i]:sample_indices[i + 1]] = sampled_signal[i]
    reconstructed[sample_indices[-1]:] = sampled_signal[-1]
    return reconstructed

# Sinc reconstruction
def sinc_reconstruct(sampled_signal, sample_times, t):
    reconstructed = np.zeros_like(t)
    for i, ts in enumerate(sample_times):
        reconstructed += sampled_signal[i] * np.sinc((t - ts) * reco_sample_slider.value)
    return reconstructed

# on change:
    reconstructed_signal_linear = np.interp(t, sample_times, sampled_signal)
    reconstructed_signal_zoh = zero_order_hold(sampled_signal, sample_times, t)
    reconstructed_signal_sinc = sinc_reconstruct(sampled_signal, sample_times, t)

# for plots:

#plt.grid(True)
#plt.legend()
```

1. Zero-Order Hold (ZOH)

Description: The zero-order hold (ZOH) method keeps the value of each sample constant until the next sample arrives, resulting in a **staircase-like signal**. It is commonly used in digital-to-analog converters (DACs).

Reconstruction Formula:

$$x_{\text{ZOH}}(t) = \sum_{n=-\infty}^{\infty} x[n] \cdot \Pi\left(\frac{t - nT_s - T_s/2}{T_s}\right) [1]$$

Where (\Pi(t)) is the rectangular pulse function.

2. Linear Interpolation (First-Order Hold)

Description
Linear interpolation connects each pair of consecutive samples with straight lines, offering a **piecewise linear approximation** of the original signal.[2]

- Produces a much smoother reconstruction than ZOH.
- Still introduces angular corners where slope changes, especially noticeable at low sampling rates.

3. Sinc Interpolation (Ideal Reconstruction)

Description
Sinc interpolation provides **perfect reconstruction** of a bandlimited signal when sampled above the Nyquist rate.

Reconstruction Formula:

$$x(t) = \sum_{n=-\infty}^{\infty} x[n] \cdot \text{sinc}\left(\frac{t - nT_s}{T_s}\right) [3]$$

- Closely matches the original smooth sine wave.
- Computationally expensive and not realizable in real-time hardware.

--> differences observed:

- Linear interpolation is simple but not smooth; it doesn't match the original waveform exactly, especially at curves.
- Zero-order hold is the least accurate visually and introduces flat segments — useful in DACs but not good for signal analysis.
- Sinc reconstruction provides the most accurate and smooth approximation but can show errors at the boundaries when applied to finite-length signals.

Only sinc reconstruction is capable of perfect recovery if the Nyquist criterion is satisfied and the signal is band-limited.

References of Exercise 3b:

[1] https://en.wikipedia.org/wiki/Zero-order_hold?utm_source=chatgpt.com

[2] https://ocw.mit.edu/courses/res-6-007-signals-and-systems-spring-2011/8d5b8d4542cb5981605e26501d260517_MITRES_6_007S11_lec17.pdf

[3]https://en.wikipedia.org/wiki/Whittaker%E2%80%93Shannon_interpolation_formula