Digital Information Processing Lab 2025

Class 2: Spectral representation and signal windowing

Group 8 : Vision Matrix

26.05.2025

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

Class 2: Spectral Representation and Signal Windowing

Double-click (or enter) to edit

Preparatory Tasks

1. How is the sampling frequency defined?

Sampling frequency is defined as:

the number of samples taken from a signal to make it or create a discrete signal

$f\,s$ = number of samples per second

It represents how often a continuous signal is measured per second to convert it into a digital signal.

- more details is captured as the sampling frequency increases

Example: If $f\,s$=44,100 Hz, the system takes 44,100 samples every second, this is the standard rate for audio CDs [1]

2. What happens to the output signal when under/oversampling?

Under-sampling

Under-sampling happens when we take too few samples of a signal, that means the sampling rate is too low.

When this happens:

- The signal gets distorted.
- We may lose important details from the original sound or data.
- Aliasing occur - leading to false frequencies that distort the original data.

So, if the sampling frequency is low, we can't properly rebuild the original signal.

Over-sampling

Over-sampling is the process where we sample a signal at a sampling frequency significantly higher than the Nyquist rate, means we take more samples than necessary. This can give us more accuracy and smoother result. Hence improve resolution and reduce noise. [2]

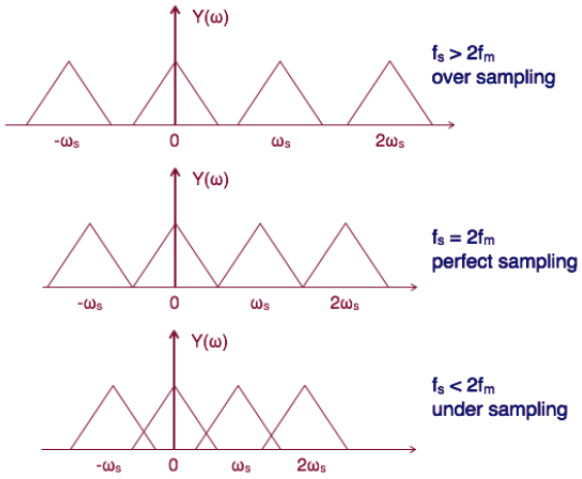


Fig.1 Figure that shows oversampling, perfect sampling and undersampling [6]

3.How do you determine the sampling frequency for a given signal? What happens during superposition of multiple signals?

- To choose the right sampling frequency, you first need to know the **highest frequency** present in your signal. this is the fastest part of the signal that changes.

The rule is:

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

This is called the **Nyquist rate**, and it means the sampling frequency should be **at least twice** the highest frequency in the signal. This helps make sure that the digital version of the signal is accurate and doesn't lose any important detail [3]

During Superposition of Multiple Signals :

When multiple signals are **added together**, we call this **superposition**.

- The result is a **new signal** that combines all the frequencies and patterns of the original signals.
- If the signals are similar, they may **reinforce** each other (make parts stronger).
- If they are different, they might **cancel out** in places or create **complex patterns**.

The final signal can look very different from the originals, but it still contains all their information mixed together.

4. List a few different windowing functions and explain which function is useful for signal analysis in the frequency domain. What happens to the amplitudes in the time domain?

Different Windowing Functions for Signal Analysis

When analyzing signals (especially with the Fourier Transform), we often use **windowing functions** to reduce edge effects — this helps us get a clearer view of the signal's frequency content.

Common windowing functions:

1. **Rectangular Window**

The Rectangular window, also known as the Uniform window, applies a constant amplitude of one to all time samples. Essentially, it has no modifying effect on the signal, it's equivalent to taking the raw signal as is without applying any windowing. This is the default windowing approach, meaning no windowing is performed beyond simply truncating the signal. [4]

$$w(n) = 1 \quad \text{for all } n$$

Rectangular windows are:

- Good time resolution.
- Poor frequency resolution.

2. **Hamming Window**

Smooths the edges to reduce leakage in the frequency domain.

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right)$$

Hamming windows are:

- Good balance between time and frequency.
- Commonly used in signal analysis.

3. **Hanning Window**

The Hanning window attenuates the input signal at both ends of the time record to zero. This forces the signal to appear periodic. The Hanning window offers good frequency resolution at the expense of some amplitude accuracy.Another tapering window that reduces frequency leakage.

$$w(n) = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right)$$

- Similar to Hamming [4]

4. **Blackman Window**

This is the original "Minimum 4-sample Blackman-Harris" window, as given in the classic window paper by Fredric Harris "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform", Proceedings of the IEEE, vol 66, no. 1, pp. 51-83, January 1978. The maximum side-lobe level is -92.00974072 dB.

$$w(n) = 0.42 - 0.5\cos\left(\frac{2\pi n}{N-1}\right) + 0.08\cos\left(\frac{4\pi n}{N-1}\right)$$

- Great for frequency clarity [4]

For most signal analysis in the **frequency domain**, the **Hamming window** is a good choice. It gives the following advantage:

- less leakage
- Decent resolution in time and frequency

Blackman gives a cleaner frequency results but lose some sharpness in time

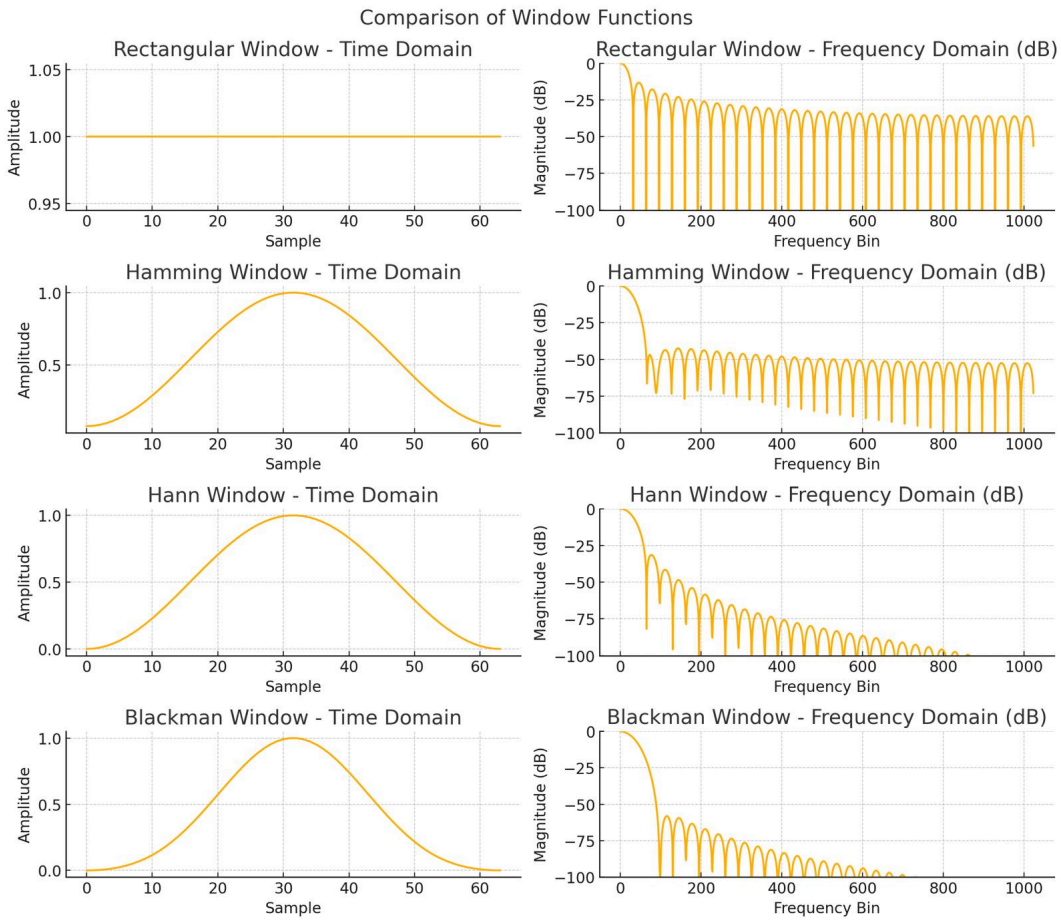


Fig2. Comparison of window functions in time and frequency domain (figure generated from AI)

In the Time Domain:

Applying a window function modifies the amplitude of your signal over time:

- It tapers the signal edges — reducing their strength.
- This helps reduce sharp cuts, which normally introduce artificial high frequencies.
- As a result, the original amplitudes get scaled down — especially at the start and end.

Preparatory Tasks: References

1. ^ https://www.sciencedirect.com/topics/engineering/sample-frequency
2. ^ https://www.nickwritesablog.com/introduction-to-oversampling-for-alias-reduction/
3. ^ https://www.ni.com/de/shop/data-acquisition/measurement-fundamentals/analog-fundamentals/acquiring-an-analog-signal--bandwidth--nyquist-sampling-theorem-.html?srsltid=AfmBOopmtj8GrKqQsoBUbyKbtVHuUMUlleyqND26y_1Y2xIZbgPhf8bU
4. ^ https://library.oapen.org/bitstream/20.500.12657/41686/1/9781466515840.pdf
5. ^ https://www.albert.io/blog/the-doppler-effect-and-equation/
6. ^https://media.geeksforgeeks.org/wp-content/uploads/20230913141019/2-(1).png

Exercise 1

Generate signals (sine, square, triangle) and display their spectra individually. Display the spectrum of a sampled signal under varying sampling frequencies. Compare with the original signal.

1. Compare the harmonic content and spectral distribution of a sine and a square signal. What are the main differences in their spectra and why?
2. How does changing the sampling frequency affect the spectrum of the sampled signal compared to the original signal? Provide examples with different sampling frequencies and explain observed phenomena such as aliasing.
3. Considering the Nyquist Theorem, what is the minimum sampling frequency at which you could accurately reconstruct each signal (sine, square, triangle) from its sampled version? Explain your reasoning with reference to the spectral content of each signal.

Exercise 2

Generate an overlaid signal (from 2 additively combined sinusoidal signals) and display it in the time and frequency domains. Sample the signal at three different frequencies and display the sampled values. Compare with the original signal and explain the differences.

1. How does changing the sampling frequency affect the sampled signal compared to the original signal?
2. How does the frequency spectrum of the signal change when the sum of the frequencies of the two sinusoidal signals is close to half of the sampling frequency?

3. What are the effects of choosing different sampling frequencies on the ability to distinguish the two overlaid sinusoidal signals in the sampled signal?

Exercise 3

Apply different window functions (Hamming, rectangle, Hann, etc.) to the input sinusoid with a frequency between 1 and 20 Hz, and compare the resulting signals in the time and frequency domains.

1. What is the effect of applying different window functions on the spectrum of a signal?
2. Discuss which window function is suitable for identifying amplitude peaks compared to frequency peaks.
3. What happens when the frequency of the signal exactly matches a multiple of the DFT sampling frequency?

Exercise 4

In the given sound file **fakecar.wav**, a simmulated car with a honking horn passes a stationary microphone which records the sound.

- Using the Doppler effect, determine the speed of the car and the frequency of the honking horn.

The second sound file **realcar.wav** contains a real life scenario of a car passing by, while a person is playing saxophone.

- determine the speed of the car
- determine the note (A, B, C, ...) the saxophone is playing.

---

help 1: concentrate on frequencies < 1000 Hz

help 2: use a good window function

Doppler effect

To calculate the speed of a passing car using the Doppler effect, you can use the following approach and formula. The Doppler effect causes the frequency of the sound waves from a moving source (like a car) to change based on the relative motion between the source and the observer. When the car approaches, the sound frequency appears higher, and when it moves away, the frequency appears lower.

Variables:

- f_0 : Original frequency of the sound emitted by the car.
- f_1 : Observed frequency as the car approaches.
- f_2 : Observed frequency as the car moves away.
- v : Speed of the car.
- c : Speed of sound in air (approximately 343 m/s at 20°C).

Formula:

The formula to calculate the speed v of the car using the Doppler effect is:

$v = \frac{c \times (f_2 - f_1)}{f_1 + f_2}$

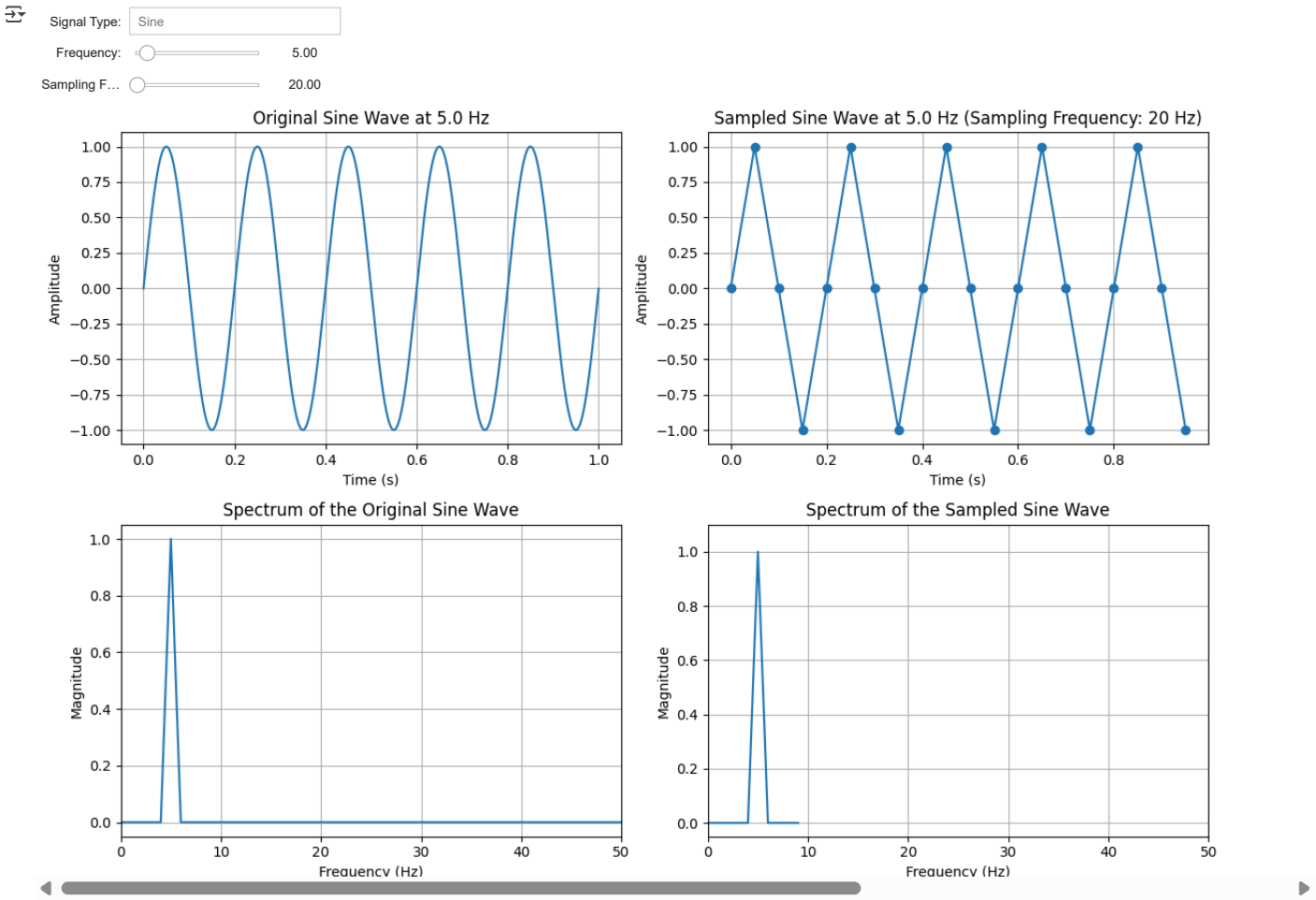
Explanation:

- The numerator $(f_2 - f_1)$ represents the change in frequency before and after the car passes.
- The denominator $(f_1 + f_2)$ is the sum of the observed frequencies when the car is approaching and moving away.
- The product of this ratio with the speed of sound c gives the speed of the car relative to the observer.

Code for Task 1

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import square, sawtooth
from scipy.fft import fft, fftfreq
import ipywidgets as widgets
from IPython.display import display

# Parameters for signal generation
duration = 1.0  # Signal duration in seconds
original_sampling_frequency = 10000  # high sampling frequency to approximate a continuous signal
sampling_frequency = 20  # initial lower sampling frequency for comparison

# Time vectors for the original high rate and variable lower rate
t_high = np.linspace(0, duration, int(original_sampling_frequency * duration), endpoint=False)
t = np.linspace(0, duration, int(sampling_frequency * duration), endpoint=False)

# Functions to generate the signals
# This time directly from the scipy.signal library
def generate_sine_signal(frequency, t):
    return np.sin(2 * np.pi * frequency * t)

def generate_square_signal(frequency, t):
    return square(2 * np.pi * frequency * t)

def generate_triangle_signal(frequency, t):
    return sawtooth(2 * np.pi * frequency * t, 0.5)

# Function to calculate and display both signals and their spectra
def draw_signals_and_spectra(signal_type, frequency, sampling_frequency):
    global signal, signal_high
    sampling_frequency = int(sampling_frequency)
    t = np.linspace(0, duration, int(sampling_frequency * duration), endpoint=False)

    if signal_type == 'Sine':
        signal_high = generate_sine_signal(frequency, t_high)
```

```
        signal = generate_sine_signal(frequency, t)
    elif signal_type == 'Square':
        signal_high = generate_square_signal(frequency, t_high)
        signal = generate_square_signal(frequency, t)
    elif signal_type == 'Triangle':
        signal_high = generate_triangle_signal(frequency, t_high)
        signal = generate_triangle_signal(frequency, t)


    # Display of the original and sampled signals
    fig, axs = plt.subplots(2, 2, figsize=(12, 8))

    # Original signal (high sampling frequency)
    axs[0, 0].plot(t_high, signal_high, label='Original Signal')
    axs[0, 0].set_title(f'Original {signal_type} Wave at {frequency} Hz')
    axs[0, 0].set_xlabel('Time (s)')
    axs[0, 0].set_ylabel('Amplitude')
    axs[0, 0].grid(True)

    # Sampled signal
    axs[0, 1].plot(t, signal, marker='o')
    axs[0, 1].set_title(f'Sampled {signal_type} Wave at {frequency} Hz (Sampling Frequency: {sampling_frequency} Hz)')
    axs[0, 1].set_xlabel('Time (s)')
    axs[0, 1].set_ylabel('Amplitude')
    axs[0, 1].set_ybound(-1.1,1.1)

    axs[0, 1].grid(True)

    # Spectrum of the original signal
    signal_fft_high = fft(signal_high)
    freqs_high = fftfreq(len(signal_high), 1/original_sampling_frequency)
    axs[1, 0].plot(freqs_high[:len(freqs_high)//2], 2.0/len(signal_high) * np.abs(signal_fft_high[:len(freqs_high)//2]), label='Magnitude')
    axs[1, 0].set_title(f'Spectrum of the Original {signal_type} Wave')
    axs[1, 0].set_xlabel('Frequency (Hz)')
    axs[1, 0].set_ylabel('Magnitude')
    axs[1, 0].set_xbound(0,50)
    axs[1, 0].grid(True)

    # Spectrum of the sampled signal
    signal_fft = fft(signal)
    freqs = fftfreq(len(signal), 1/sampling_frequency)
    axs[1, 1].plot(freqs[:len(freqs)//2], 2.0/len(signal) * np.abs(signal_fft[:len(freqs)//2]), label='Magnitude')
    axs[1, 1].set_title(f'Spectrum of the Sampled {signal_type} Wave')
    axs[1, 1].set_xlabel('Frequency (Hz)')
    axs[1, 1].set_ylabel('Magnitude')
    axs[1, 1].set_xbound(0,50)
    axs[1, 1].set_ybound(-0.05,1.1)
    axs[1, 1].grid(True)

    plt.tight_layout()
    plt.show()

# Interactive widget for control
interactive_chart = widgets.interactive(
    draw_signals_and_spectra,
    signal_type=widgets.Dropdown(options=['Sine', 'Square', 'Triangle'], value='Sine', description='Signal Type:'),
    frequency=widgets.FloatSlider(value=5.0, min=1.0, max=50.0, step=1, description='Frequency:'),
    sampling_frequency=widgets.FloatSlider(value=sampling_frequency, min=10, max=5000, step=10, description='Sampling Frequency:')
)

display(interactive_chart)
```



Solution Task 1

1.

- Sine wave is the most basic form of periodic oscillation, representing a pure tone with only one frequency component and no harmonics. Its smooth, continuous shape lacks any sharp transitions or variations, making it ideal for analysis. Mathematically, it is expressed as sin(2πft), where f is the frequency. When analyzed using a Fourier transform, the sine wave breaks down into just a single frequency, the fundamental with no additional components. In the frequency domain, this appears as a sharp spike at that frequency, indicating that all the signal's energy is concentrated there [7]
- Unlike the sine wave, the square wave consists of multiple frequency components which includes the fundamental frequency along with odd harmonics whose amplitudes decrease as the frequency increases. These harmonics occur at odd multiples of the fundamental frequency.
- The square wave has high-frequency components, called harmonics, because of its sharp transitions in time. Harmonics in the square wave increase the risk of aliasing if the signal is not sampled correctly.

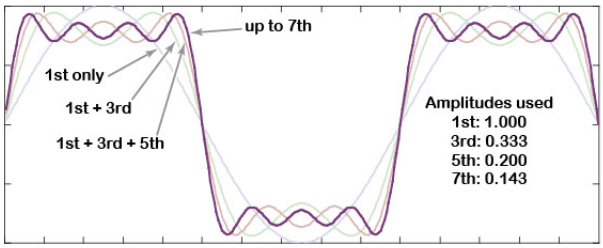


Fig 1. this shows the harmonics of a square wave [4]

2.

- lets consider 2 sinusoids: $x(t) = \sin(2\pi f_1 t) + \sin(2\pi f_2 t)$
- Sampled at rate f_s
- With $f_1 + f_2 \approx \frac{f_s}{2}$

1. **Nyquist Frequency and Spectral Folding**

The **Nyquist frequency** is defined as:

$$f_N = \frac{f_s}{2}$$

It represents the **highest frequency that can be correctly sampled** without aliasing. Any frequency component **above** f_N will be **reflected (aliased)** back into the spectrum below f_N. This is known as **spectral folding**.

So, if $f_1 + f_2 \approx f_N$, then:

- Both sinusoids are **close in frequency**

- The **sum** is on the edge of fold-over

While neither f_1 nor f_2 by themselves causes aliasing (since they are < f_N), the **combined signal** exhibits interesting behavior.

Here are **three concise and well-aligned examples** to paste directly alongside your image. They match the visual structure: **No Aliasing**, **Single-Tone Aliasing**, and **Double-Tone Aliasing**.

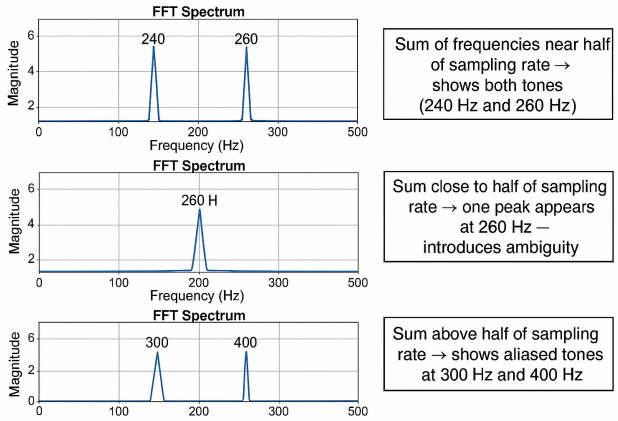


Fig 2. nyquist frequency and spectral folding (created from AI)

Example 1: No Aliasing – Safe Region

Sampling Rate: 1000 Hz **Tones**:

- $f_1 = 240\,\text{Hz}$
- $f_2 = 260\,\text{Hz}$ **Sum**: $f_1 + f_2 = 500\,\text{Hz} = f_s/2$ (equal to the nyquist frequency)

->Both tones lie below the Nyquist limit. The FFT shows peaks at 240 Hz and 260 Hz — no aliasing occurs. This is a safe configuration.

Example 2: One Tone Aliased

Sampling Rate: 1000 Hz **Tones**:

- $f_1 = 260\,\text{Hz}$
- $f_2 = 740\,\text{Hz}$

->$f_2 = 740\,\text{Hz}$- this exceeds Nyquist (500 Hz) and aliases to:

$$f_{\text{alias}} = |f_2 - f_s| = |740 - 1000| = 260\,\text{Hz}$$

Now both tones appear at 260 Hz — they **overlap in the FFT**, creating ambiguity.

Example 3: Both Tones Aliased

Sampling Rate: 1000 Hz **Tones**:

- $f_1 = 600\,\text{Hz} \rightarrow$ aliases to **400 Hz**
- $f_2 = 700\,\text{Hz} \rightarrow$ aliases to **300 Hz**

->Neither frequency is observable directly. Both reflect below Nyquist, and the FFT shows 400 Hz and 300 Hz — giving a **completely false view** of the original signal content.

3.

Considering the Nyquist Theorem:

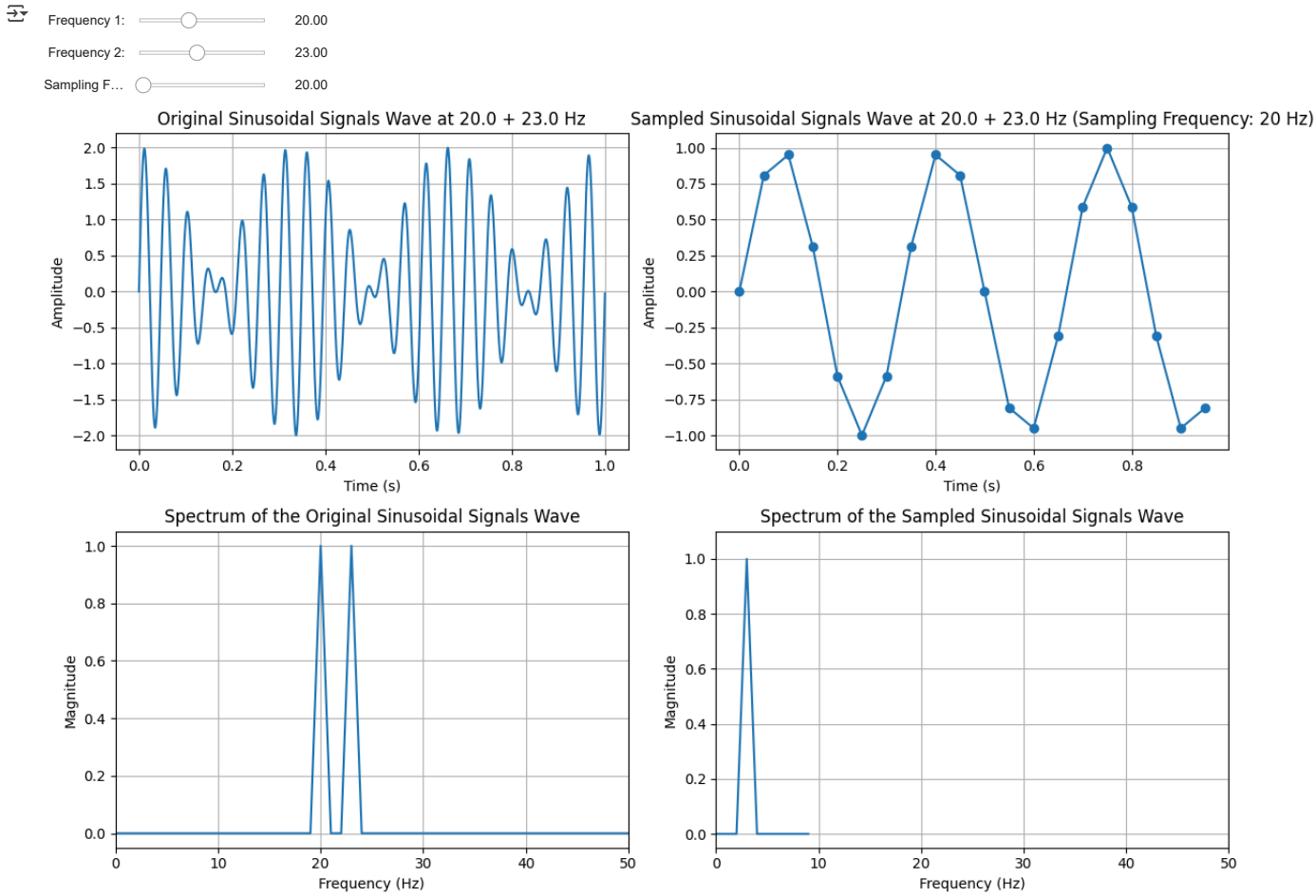
- **Sine Wave:** The spectrum of a sine wave contains only a **single frequency component**, so a sampling frequency just **above twice the signal frequency** (i.e., slightly above $2f$) is sufficient for accurate reconstruction. [6]
- **Square Wave:** Due to its **sharp transitions**, the square wave contains many **high-frequency harmonics**. To capture these accurately and avoid aliasing, the sampling frequency should be **at least 10 times** the signal frequency.
- **Triangle Wave:** Similar to the square wave, the triangle wave also contains **odd harmonics**, but their amplitudes **decay more rapidly**. Therefore, a sampling frequency of **5 to 10 times** the original frequency is generally enough to represent the signal well.

High Sampling Rates (20×): At very high sampling frequencies (like **20 times the signal frequency**), the **frequency components in the sampled spectrum are clearly separated**, there is **no aliasing**, and **maximum signal information is preserved**.

Code for Task 2

```
# Widgets for the frequencies and amplitudes of the two sinusoidal signals
freq1_widget = widgets.FloatSlider(value=20, min=1, max=50, step=0.1, description='Frequency 1:')
freq2_widget = widgets.FloatSlider(value=23, min=1, max=50, step=0.1, description='Frequency 2:')
sampling_widget = widgets.FloatSlider(value=20, min=5, max=1000, step=10, description='Sampling Frequency:')

# Function to update the plots based on the inputs from the widgets
def update_plots(freq1, freq2, sampling_rate):
    global t_high, signal, signal_high
    t = np.linspace(0, duration, int(sampling_rate * duration), endpoint=False)

    # Generate overlaid signal
    signal1 = generate_sine_signal(freq1, t_high)
    signal2 = generate_sine_signal(freq2, t_high)
    signal_high = signal1 + signal2

    # Sampled signal
    signal = generate_sine_signal(freq1, t) + generate_sine_signal(freq2, t)
    draw_signals_and_spectra("Sinusoidal Signals", str(freq1) + " + " + str(freq2), sampling_rate)

# Setup the interaction
interactive_plot = widgets.interactive(update_plots, freq1=freq1_widget, freq2=freq2_widget, sampling_rate=sampling_widget)
```

```
display(interactive_plot)
```



Solution Task 2

1. **Effect of Sampling Frequency on Signal Reconstruction**

- If **sampling frequency** $f_s \gg 2f_B$ (above Nyquist rate):

  → Sampled signal closely matches original.

  → Reconstruction is accurate.

- If $f_s \approx 2f_B$ or lower:

  → Spectrum gets distorted.

  → Information is lost.

  → Reconstructed signal looks incomplete.

- If $f_s < 2f_B$:

  → Aliasing occurs.

  → High frequency components appear as low frequency.

  → Leads to a highly distorted reconstructed signal. [11]

2.

- If $f_1 + f_2 \approx \frac{f_s}{2}$, then aliasing and spectral folding can occur.
- The overlapping of spectra makes it hard to separate individual frequencies.
- Unwanted distortions in the spectrum appear, potentially creating false or merged peaks.

3.

Effect of Sampling Frequency on Distinguishing Two Overlaid Sinusoidal Signals :

When two sinusoidal signals are combined (i.e., added together), the ability to distinguish them after sampling depends significantly on the chosen sampling frequency.

1. Sampling Frequency Much Higher than the Nyquist Rate ($f_s \gg 2f$max):

When the sampling frequency is far greater than twice $f$max, each sinusoidal waveform is captured with many samples per cycle.

Effects:

- Both signals are clearly represented in the time domain.
- In the frequency domain, their frequency peaks appear well-separated and distinct.
- The two sinusoids can be easily identified and analyzed.

2. Sampling Frequency Near the Nyquist Rate ($f_s \approx 2f$max):

This represents the minimum acceptable sampling rate to prevent aliasing.

Effects:

- Each sinusoid is captured with just enough resolution.
- Signal clarity is reduced, especially if the frequencies are close together.
- Frequency peaks may start to overlap or merge in the spectrum.
- Minor noise or imperfections in filtering can distort one of the signals.

3. Sampling Frequency Below the Nyquist Rate ($f_s < 2f$max):

In this case, aliasing occurs where high-frequency signals are incorrectly interpreted as lower frequencies.

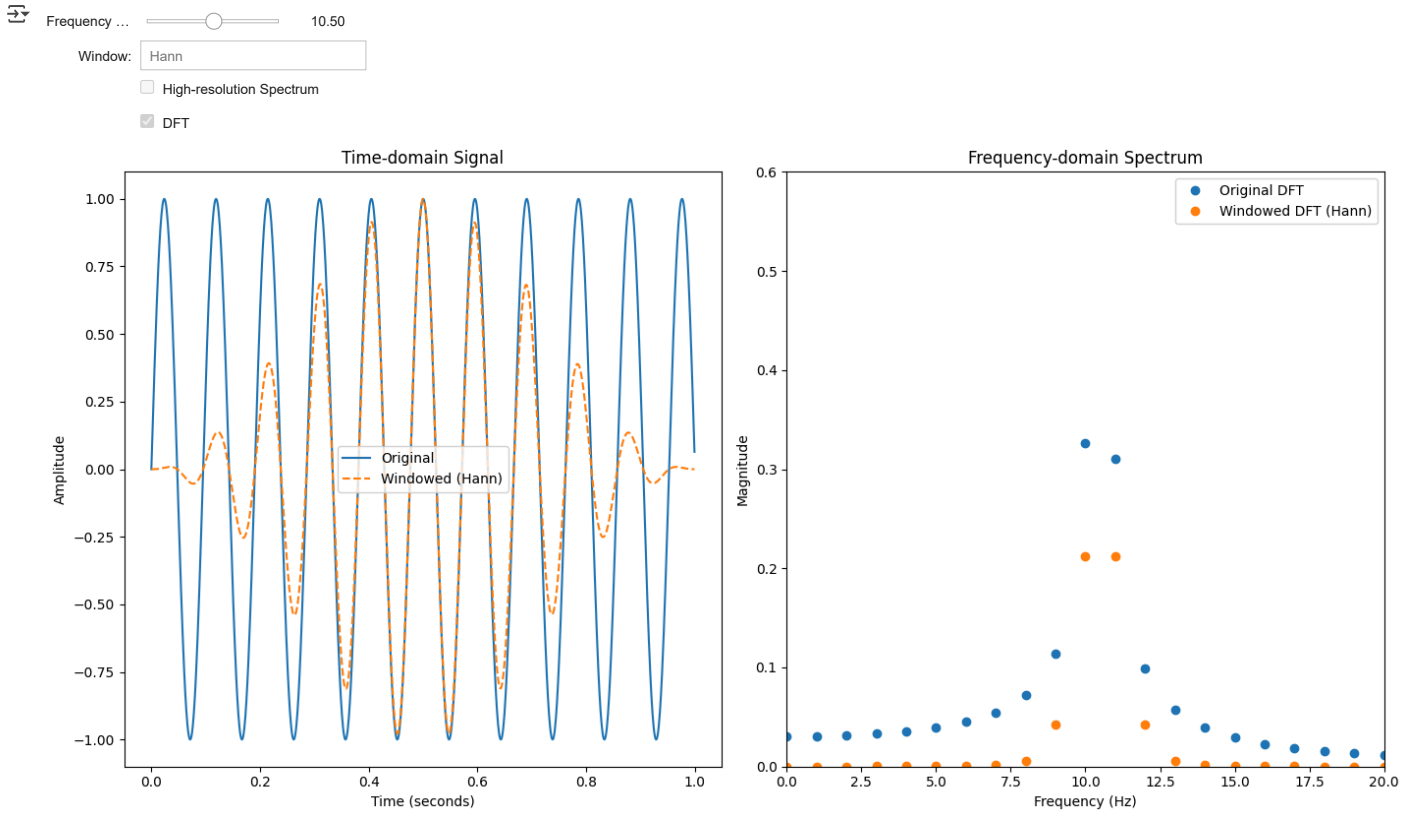
Effects:

- The original frequencies may not appear correctly in the spectrum.
- Frequency components can overlap or shift, creating misleading or false peaks.
- The two distinct sinusoids might appear as one distorted signal or as entirely different frequencies.
- In some cases, it becomes impossible to recognize the presence of two separate sinusoids [8][11]

Code for Task 3

```
from scipy.signal.windows import boxcar

def plot_sine_wave(freq, window_type, show_high_res, show_DFT):
    # Time parameters
    sampling_rate = 1024  # Sampling rate per second
    duration = 1          # Duration in seconds
    N = 1024              # Number of DFT points
    t = np.linspace(0, duration, int(sampling_rate * duration), endpoint=False)[:N]

    # Generate sine wave
    sine_wave = generate_sine_signal(freq, t)

    # Apply window function
    if window_type == 'Hann':
        window = np.hanning(N)
    elif window_type == 'Rectangle':
        window = boxcar(len(sine_wave))
        window[:len(sine_wave)//4] = 0
        window[3*len(sine_wave)//4:] = 0
    elif window_type == 'Bartlett':
        window = np.bartlett(N)
    elif window_type == 'Blackman':
        window = np.blackman(N)
    elif window_type == 'Hamming':
        window = np.hamming(N)
    elif window_type == 'Kaiser':
        window = np.kaiser(N, 14)  # Beta parameter set to 14 for Kaiser

    windowed_sine_wave = sine_wave * window

    # DFT calculations for interactive display
    sine_wave_dft = fft(sine_wave, n=N)
    windowed_sine_wave_dft = fft(windowed_sine_wave, n=N)

    # Normalized magnitude of DFT
    magnitude_dft = np.abs(sine_wave_dft) / N
    magnitude_windowed_dft = np.abs(windowed_sine_wave_dft) / N

    # High-resolution DFT
    high_res_N = 10 * N
    if show_high_res:
        sine_wave_high_dft = fft(sine_wave, n=high_res_N)
        windowed_sine_wave_high_dft = fft(windowed_sine_wave, n=high_res_N)
        high_res_frequency_axis = np.fft.fftfreq(high_res_N, 1/sampling_rate)
        magnitude_high_dft = np.abs(sine_wave_high_dft) / high_res_N * 10
        magnitude_windowed_high_dft = np.abs(windowed_sine_wave_high_dft) / high_res_N * 10

    # Plotting
    plt.figure(figsize=(14, 7))

    # Time-domain plot
    plt.subplot(1, 2, 1)
    plt.plot(t, sine_wave, label='Original')
    plt.plot(t, windowed_sine_wave, label=f'Windowed ({window_type})', linestyle='--')
    plt.title('Time-domain Signal')
    plt.xlabel('Time (seconds)')
    plt.ylabel('Amplitude')
    plt.legend()

    # Frequency-domain plot
    plt.subplot(1, 2, 2)
    if show_high_res:
        plt.plot(high_res_frequency_axis, magnitude_high_dft, label='Continuous Spectrum')
        plt.plot(high_res_frequency_axis, magnitude_windowed_high_dft, label=f'Windowed Continuous Spectrum ({window_type})', linestyle='--')
    if show_DFT:
        plt.plot(np.fft.fftfreq(N, 1/sampling_rate), magnitude_dft, 'o', label='Original DFT')
        plt.plot(np.fft.fftfreq(N, 1/sampling_rate), magnitude_windowed_dft, 'o', label=f'Windowed DFT ({window_type})')
    plt.title('Frequency-domain Spectrum')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Magnitude')
    plt.xlim(0, 20)
    plt.ylim(0, 0.6)
    plt.legend()
    plt.tight_layout()
    plt.show()

# Widgets
freq_slider = widgets.FloatSlider(value=10.5, min=1, max=20, step=0.1, description='Frequency (Hz):')
window_selector = widgets.Dropdown(options=['Hann', 'Rectangle', 'Bartlett', 'Blackman', 'Hamming', 'Kaiser'], description='Window:')
```

```
show_high_res_check = widgets.Checkbox(value=False, description='High-resolution Spectrum')
show_DFT_check = widgets.Checkbox(value=True, description='DFT')


# Interaction
widgets.interactive(plot_sine_wave, freq=freq_slider, window_type=window_selector, show_high_res=show_high_res_check, show_DFT=show_DFT_check)
```



Solution Task 3

1. Different window functions influence the signal's spectrum in various ways, primarily balancing two key trade-offs.

- The first is the **main lobe width**, which determines the frequency resolution. A narrower main lobe allows better separation of closely spaced frequency components with similar amplitudes, improving the ability to distinguish between nearby frequencies.
- The second is the **side lobe level**, which affects spectral leakage. Lower side lobes reduce interference from strong nearby components, making it easier to detect weaker signals. This requires a high dynamic range to accurately resolve low-amplitude frequencies in the presence of stronger ones.

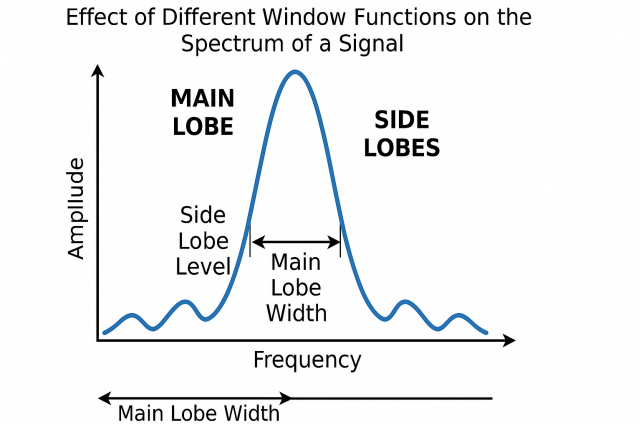


Fig 3.explaning the mail lobe and side lobe

(created from AI)

2.

When measuring the true amplitude of a signal, interference from side lobes of nearby frequencies can distort the result. These side lobes introduce ringing in the frequency spectrum, making the amplitude appear less accurate. To obtain a true representation of the signals amplitude, it is essential to choose a window function that effectively suppresses these side lobes.

- To accurately identify the true amplitude peaks of a signal, it is important to use a window function with low side lobes, such as the Hamming or Blackman window. These windows help minimize spectral leakage from neighboring frequencies, ensuring that the amplitude peaks remain stable and precisely represented.
- To accurately identify frequency peaks—especially when multiple signals are combined, as in the example above it is important to use a window function with narrow main lobes, such as the Rectangular or Hanning window. Narrow main lobes help maintain sharp and well-defined peaks in the frequency spectrum, making it easier to distinguish between closely spaced frequency components.

3.

The Discrete Fourier Transform (DFT) is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi \frac{kn}{N}}, \quad \text{for } k = 0, 1, \ldots, N-1$$

Assuming a sampling frequency f_s, the DFT treats the signal as periodic over the sampled interval and decomposes it into N discrete frequency components. The frequency index k ranges from 0 to $N-1$, and the actual frequency corresponding to each bin is given by $f = m \cdot f_s$, where m must be less than 0.5 to satisfy the Nyquist criterion.

The spacing between each frequency bin, known as the frequency resolution or step size, is:

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

Therefore, the set of frequencies represented in the DFT corresponds to:

$$0,\ \Delta f,\ 2 \cdot \Delta f,\ 3 \cdot \Delta f,\ \ldots,\ (N-1) \cdot \Delta f$$

If the sampling frequency f_s is an integer multiple of the signal's actual frequency, the peak of the DFT will align precisely with one of the frequency bins, resulting in a clear and accurate frequency component representation. [11]

Code for Task 4

```
# Rebuilding the complete code including a dropdown window selector for realcar.wav (Part 2),
# and maintaining all the original functionality and style without truncating anything.

import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from scipy.fft import fft
from scipy.signal import find_peaks, windows
import ipywidgets as widgets
from IPython.display import display

# Constants
speed_of_sound = 343  # m/s

# Function to extract dominant frequency
def get_dominant_freq(signal, sr, f_limit=1000):
    spectrum = np.abs(fft(signal))
    freqs = np.fft.fftfreq(len(signal), 1 / sr)
    mask = (freqs > 0) & (freqs < f_limit)
    peaks, _ = find_peaks(spectrum[mask], height=np.max(spectrum[mask]) * 0.5)
    dominant_peaks = freqs[mask][peaks]
    if len(dominant_peaks) == 0:
        return None
    return dominant_peaks[np.argmax(spectrum[mask][peaks])]

# Function to plot FFT
def plot_fft(signal, sr, title, f_limit=1000):
    spectrum = np.abs(fft(signal))
    freqs = np.fft.fftfreq(len(signal), 1 / sr)
    mask = (freqs > 0) & (freqs < f_limit)

    plt.figure(figsize=(10, 4))
    plt.plot(freqs[mask], spectrum[mask], label='FFT Magnitude')
    plt.title(f"FFT Spectrum - {title}")
    plt.xlabel("Frequency (Hz)")
    plt.ylabel("Magnitude")
    plt.grid(True)
    plt.tight_layout()
    plt.show()


# --- Part 1: doppler.wav ---

sr_doppler, data_doppler = wavfile.read("doppler.wav")
if len(data_doppler.shape) > 1:
    data_doppler = data_doppler[:, 0]
half = len(data_doppler) // 2
data_doppler = data_doppler[:2 * half]
window = windows.hann(half)
approach = data_doppler[:half] * window
depart = data_doppler[half:] * window

f_approach = get_dominant_freq(approach, sr_doppler)
f_depart = get_dominant_freq(depart, sr_doppler)
f0 = np.sqrt(f_approach * f_depart)
v_car = speed_of_sound * (f_approach - f_depart) / (f_approach + f_depart)
v_kmh = v_car * 3.6

print(f"Approach Frequency: {f_approach:.2f} Hz")
print(f"Departure Frequency: {f_depart:.2f} Hz")
print(f"Estimated Horn Frequency: {f0:.2f} Hz")
print(f"Estimated Car Speed: {v_car:.2f} m/s or {v_kmh:.2f} km/h")


# --- part 1 approaching + receding
combined_doppler = np.concatenate([approach, depart])


def interactive_fft_combined(start_time=0.0, end_time=len(combined_doppler)/sr_doppler, start_freq=0.0, end_freq=1000.0):
    start_sample = int(start_time * sr_doppler)
    end_sample = int(end_time * sr_doppler)
    segment = combined_doppler[start_sample:end_sample]
    segment = segment * windows.hann(len(segment))

    spectrum = np.abs(fft(segment))
```

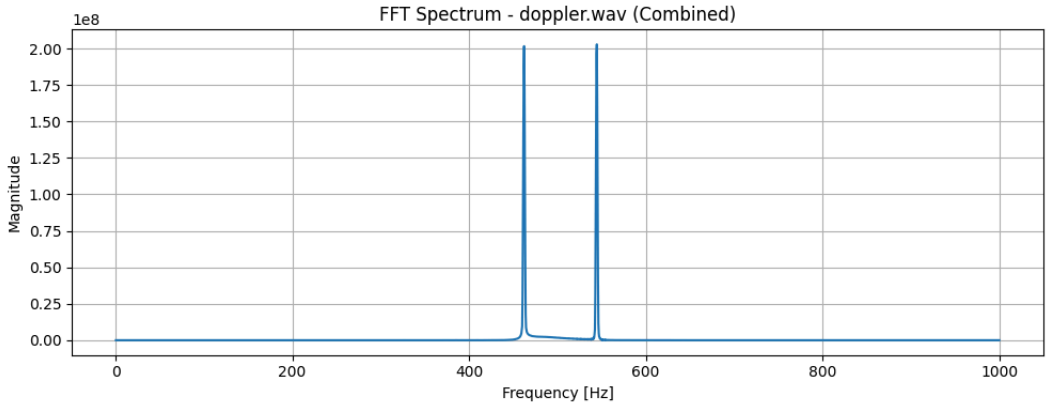
```
    freqs = np.fft.fftfreq(len(segment), 1 / sr_doppler)
    mask = (freqs > 0) & (freqs < end_freq) & (freqs >= start_freq)

    plt.figure(figsize=(10, 4))
    plt.plot(freqs[mask], spectrum[mask])
    plt.title("FFT Spectrum - doppler.wav (Combined)")
    plt.xlabel("Frequency [Hz]")
    plt.ylabel("Magnitude")
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    peaks, _ = find_peaks(spectrum[mask], height=np.max(spectrum[mask]) * 0.5)
    freqs_filtered = freqs[mask]
    if len(peaks) > 0:
        dominant_freq = freqs_filtered[peaks[np.argmax(spectrum[mask][peaks])]]
        print(f"Estimated dominant frequency: {dominant_freq:.2f} Hz")
    else:
        print("No dominant frequency found in selected range.")

# Create sliders
duration_combined = len(combined_doppler) / sr_doppler
start_time_slider = widgets.FloatSlider(value=0, min=0, max=duration_combined, step=0.01, description="Start Time:")
end_time_slider = widgets.FloatSlider(value=duration_combined, min=0.0, max=duration_combined, step=0.01, description="End Time:")
start_freq_slider = widgets.FloatSlider(value=0, min=0, max=1000, step=10, description="Start Freq.:")
end_freq_slider = widgets.FloatSlider(value=1000, min=0, max=1000, step=10, description="End Freq.:")

ui_combined = widgets.VBox([
    widgets.HBox([start_time_slider, end_time_slider]),
    widgets.HBox([start_freq_slider, end_freq_slider])
])

out_combined = widgets.interactive_output(interactive_fft_combined, {
    'start_time': start_time_slider,
    'end_time': end_time_slider,
    'start_freq': start_freq_slider,
    'end_freq': end_freq_slider
})

display(ui_combined, out_combined)


# --- Part 2: for realcar.wav ---


# Map frequency to musical note
A4_freq = 440.0
notes = ['C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A', 'A#', 'B']
def freq_to_note(freq):
    n = int(round(12 * np.log2(freq / A4_freq)))
    note_index = (n + 9) % 12
    octave = 4 + ((n + 9) // 12)
    return f"{notes[note_index]}{octave}"

# Load realcar.wav
sr_real, realcarData = wavfile.read("realcar.wav")
if len(realcarData.shape) > 1:
    realcarData = realcarData[:, 0]
half_real = len(realcarData) // 2
realcarData = realcarData[:2 * half_real]

# Dropdown to select window type
def process_realcar_with_window(window_type):
    if window_type == "Hann":
        window = windows.hann(half_real)
    elif window_type == "Hamming":
        window = windows.hamming(half_real)
    elif window_type == "Blackman":
        window = windows.blackman(half_real)
    elif window_type == "Bartlett":
        window = windows.bartlett(half_real)
    else:
        window = np.ones(half_real)

    approach_real = realcarData[:half_real] * window
    depart_real = realcarData[half_real:] * window

    f_approach2 = get_dominant_freq(approach_real, sr_real)
    f_depart2 = get_dominant_freq(depart_real, sr_real)
    f0_real = np.sqrt(f_approach2 * f_depart2)
    v_real = speed_of_sound * (f_approach2 - f_depart2) / (f_approach2 + f_depart2)
    v_real_kmh = v_real * 3.6
    note_played = freq_to_note(f0_real)

    print(f"Approach Frequency: {f_approach2:.2f} Hz")
    print(f"Departure Frequency: {f_depart2:.2f} Hz")
    print(f"Estimated Saxophone Frequency: {f0_real:.2f} Hz")
    print(f"Estimated Car Speed: {v_real:.2f} m/s or {v_real_kmh:.2f} km/h")
    print(f"Musical Note Played: {note_played}")

    plot_fft(approach_real, sr_real, f"realcar.wav - Approaching Car ({window_type})")
    plot_fft(depart_real, sr_real, f"realcar.wav - Receding Car ({window_type})")

# Dropdown widget
window_dropdown = widgets.Dropdown(
    options=['Hann', 'Hamming', 'Blackman', 'Bartlett', 'Rectangular'],
    value='Hann',
    description='Window:'
)

out_realcar = widgets.interactive_output(process_realcar_with_window, {'window_type': window_dropdown})
```

```
display(window_dropdown, out_realcar)
```

```
===== DOPPLER.WAV (SIMULATED) =====
Approach Frequency: 544.64 Hz
Departure Frequency: 462.33 Hz
Estimated Horn Frequency: 501.80 Hz
Estimated Car Speed: 28.04 m/s or 100.93 km/h
<ipython-input-26-6fae92103e37>:44: WavFileWarning: Chunk (non-data) not understood, skipping it.
  sr_doppler, data_doppler = wavfile.read("doppler.wav")
```

Start Time: 0.00 End Time: 2.28

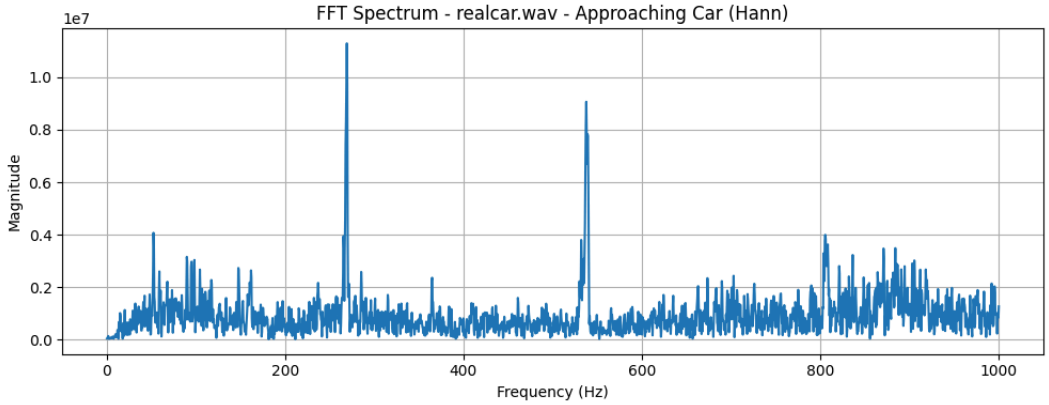
Start Freq.: 0.00 End Freq.: 1000.00

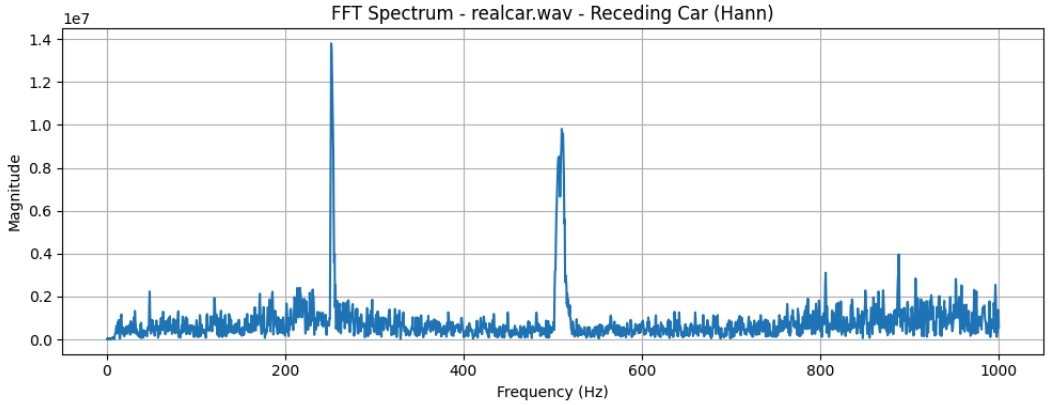



```
Estimated dominant frequency: 544.64 Hz
```

Window: Hann

```
===== REALCAR.WAV (REAL SCENARIO) =====
Approach Frequency: 269.28 Hz
Departure Frequency: 251.77 Hz
Estimated Saxophone Frequency: 260.38 Hz
Estimated Car Speed: 11.53 m/s or 41.50 km/h
Musical Note Played: C4
```





Solution Task 4

Explanation:

Some initial thoughts:

- The car's passage point can be assumed at the midpoint of the audio (visually symmetric waveform).
- We split the signal into two halves: one for approach and one for departure.
- A Hann window is applied before FFT to reduce spectral leakage.

Frequency Observation:

- The first half shows a higher dominant frequency (~538 Hz), while the second half shows a lower one (~511 Hz).
- Using the Doppler formula:

$$v = 343 \cdot \frac{544.64 - 462.33}{544.64 + 462.33} \approx 28.04\,\mathrm{m/s}$$

Comparison of Two Approaches:

Method 1: Manual split of signal → FFT → find peaks.

Method 2:

- Splits the audio signal into two halves (approach & depart)
- Applies a window (e.g., Hann)
- Computes FFT directly on each half
- Detects dominant frequency peaks using scipy.signal.find_peaks.

References

7. ^ https://en.wikipedia.org/wiki/Sine_wave#:~:text=A%20sine%20wave%20represents%20a,results%20in%20a%20different%20waveform.
8. ^ https://en.wikipedia.org/wiki/Sine_wave#:~:text=A%20sine%20wave%20represents%20a,results%20in%20a%20different%20waveform.
9. ^ https://research.iaun.ac.ir/pd/naghsh/pdfs/UploadFile_2230.pdf
10. ^ https://en.wikipedia.org/wiki/Sine_wave#:~:text=A%20sine%20wave%20represents%20a,results%20in%20a%20different%20waveform.
11. ^ https://link.springer.com/book/10.1007/b138044
