

**INDEX**

|  |  |
| --- | --- |
| Sl. | Name of the experiment |
| 1. | Signal Operation ->adding, shifting, multiplication, folding |
| 2. | Convolution |
| 3. | Correlation |
| 4. | Signal Sequence |
| 5. | PPG signal-filtering, feature extraction, pick detection |
| 6. | DFT |
| 7. | Frequency Bin |

**Experiment No-1. Signal Operations (Addition, Shifting, Folding)**

**Theory:**

1. **Signal Addition**: Adding two signals element-wise.

y(t)=x1(t)+x2(t)y(t) = x\_1(t) + x\_2(t)y(t)=x1​(t)+x2​(t)

1. **Signal Shifting**: Shifting a signal in time by a certain amount.

x(t−t0)x(t - t\_0)x(t−t0​)

1. **Signal Multiplication**: Pointwise multiplication of two signals.

y(t)=x1(t)⋅x2(t)y(t) = x\_1(t) \cdot x\_2(t)y(t)=x1​(t)⋅x2​(t)

1. **Signal Folding**: Time reversal of a signal.

xf(t)=x(−t)x\_f(t) = x(-t)xf​(t)=x(−t)

**Objective:**

The main objective of this lab is to understand and implement fundamental signal processing operations such as **addition**, **shifting**, **multiplication**, and **folding** using Python. These operations are crucial in signal processing, communications, and control systems.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

# Define the range

n = np.arange(-10, 11)

def impulse\_signal(n):

    return np.where(n == 0, 1, 0)

def step\_signal(n):

    return np.where(n >= 0, 1, 0)

def ramp\_signal(n):

    return np.where(n >= 0, n, 0)

# Generate signals

impulse = impulse\_signal(n)

step = step\_signal(n)

ramp = ramp\_signal(n)

# Plot signals

plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)

plt.stem(n, impulse)

plt.title("Impulse Signal")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

plt.subplot(1, 3, 2)

plt.stem(n, step)

plt.title("Step Signal")

plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

plt.subplot(1, 3, 3)

plt.stem(n, ramp)

plt.title("Ramp Signal")

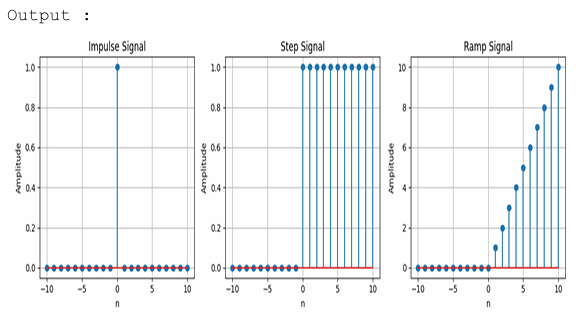
plt.xlabel("n")

plt.ylabel("Amplitude")

plt.grid()

plt.tight\_layout()

plt.show()



**Experiment No-2. Convolution**

**Theory:**

Convolution is a mathematical operation that combines two signals to produce a third. It shows how an input signal is modified by a system’s impulse response to produce an output signal. In discrete time, it's the sum of products of the input and shifted impulse response; in continuous time, it's an integral of their product. Convolution is essential in signal processing for tasks like filtering and system analysis.

**Objective:**

The objective of this lab is to understand and implement the convolution operation, which is a fundamental concept in signal processing, linear systems analysis, and image processing. In this lab, we will convolve two signals (or functions) in both continuous and discrete time using Python.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

def signal\_addition(x1, x2):

return x1 + x2

def signal\_multiplication(x1, x2):

    return x1 \* x2

def signal\_scaling(x, alpha):

    return alpha \* x

def signal\_shifting(n, shift):

    return n + shift

def signal\_folding(x):

    return np.flip(x)

n = np.array([-2, -1, 0, 1, 2])

x1 = np.array([1, 2, 3, 4, 5])

x2 = np.array([5, 4, 3, 2, 1])

added\_signal = signal\_addition(x1, x2)

multiplied\_signal = signal\_multiplication(x1, x2)

scaled\_signal = signal\_scaling(x1, 2)

shifted\_signal1 = signal\_shifting(n, -2)

shifted\_signal2 = signal\_shifting(n, 2)

folded\_signal = signal\_folding(x1)

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

plt.subplot(4, 2, 1)

plt.stem(n, x1)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Original Signal x1")

plt.grid()

plt.subplot(4, 2, 2)

plt.stem(n, x2)

plt.xlabel("Time ")

plt.ylabel("Amplitude")

plt.title("Original Signal x2")

plt.grid()

plt.subplot(4, 2, 3)

plt.stem(n, added\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Signal Addition")

plt.grid()

plt.subplot(4, 2, 4)

plt.stem(n, multiplied\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Signal Multiplication")

plt.grid()

plt.subplot(4, 2, 5)

plt.stem(n, scaled\_signal)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Scaled Signal (x1 \* 2)")

plt.grid()

plt.subplot(4, 2, 6)

plt.stem(shifted\_signal1, x1)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Shifted Signal (Shift = -2)")

plt.grid()

plt.subplot(4, 2, 7)

plt.stem(shifted\_signal2, x1)

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.title("Shifted Signal (Shift = +2)")

plt.grid()

plt.subplot(4, 2, 8)

plt.stem(n, folded\_signal)

plt.xlabel("Time")

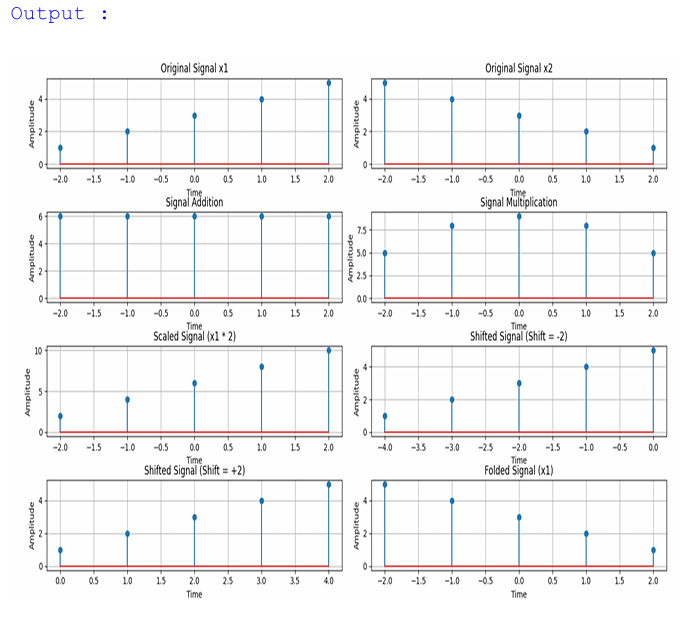
plt.ylabel("Amplitude")

plt.title("Folded Signal (x1)")

plt.grid()

plt.tight\_layout()

plt.show()



**Experiment No-3. Correlation**

**Theory:**

**Correlation** is a statistical measure that describes the relationship or similarity between two signals (or variables). In signal processing, it is used to measure how similar two signals are, and how much one signal resembles or matches another when shifted in time.

**Objective:** The objective of this lab is to understand and implement the **correlation** operation, which measures the similarity between two signals. Correlation is used in signal processing for tasks such as pattern recognition, noise reduction, and system identification.

**Code:**

**import numpy as np**

**import matplotlib.pyplot as plt**

**from scipy.signal import correlate, correlation\_lags**

**def compute\_autocorrelation(signal):**

**auto\_corr = correlate(signal, signal, mode='full', method='auto')**

**lags = correlation\_lags(len(signal), len(signal), mode='full')**

**return auto\_corr, lags**

**def compute\_cross\_correlation(signal1, signal2):**

**cross\_corr = correlate(signal1, signal2, mode='full', method='auto')**

**lags = correlation\_lags(len(signal1), len(signal2), mode='full')**

**return cross\_corr, lags**

**fs = 1000  # Sampling frequency in Hz**

**t = np.linspace(0, 1, fs, endpoint=False)  # Time vector**

**freq = 5  # Frequency of the sine wave**

**sin\_signal = np.sin(2 \* np.pi \* freq \* t)**

**auto\_corr, lags\_auto = compute\_autocorrelation(sin\_signal)**

**signal1 = sin\_signal**

**signal2 = np.roll(signal1, 100)**

**cross\_corr, lags\_cross = compute\_cross\_correlation(signal1, signal2)**

**noise = np.random.normal(0, 0.5, fs)**

**noisy\_signal = signal1 + noise**

**cross\_corr\_noise, lags\_noise = compute\_cross\_correlation(signal1, noisy\_signal)**

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

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

**plt.plot(lags\_auto, auto\_corr)**

**plt.title("Autocorrelation of a Sinusoidal Signal")**

**plt.xlabel("Lag")**

**plt.ylabel("Autocorrelation")**

**plt.grid()**

**plt.subplot(3, 1, 2)**

**plt.plot(lags\_cross, cross\_corr)**

**plt.title("Cross-Correlation between Two Signals")**

**plt.xlabel("Lag")**

**plt.ylabel("Cross-Correlation")**

**plt.grid()**

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

**plt.plot(lags\_noise, cross\_corr\_noise)**

**plt.title("Cross-Correlation with Noisy Signal")**

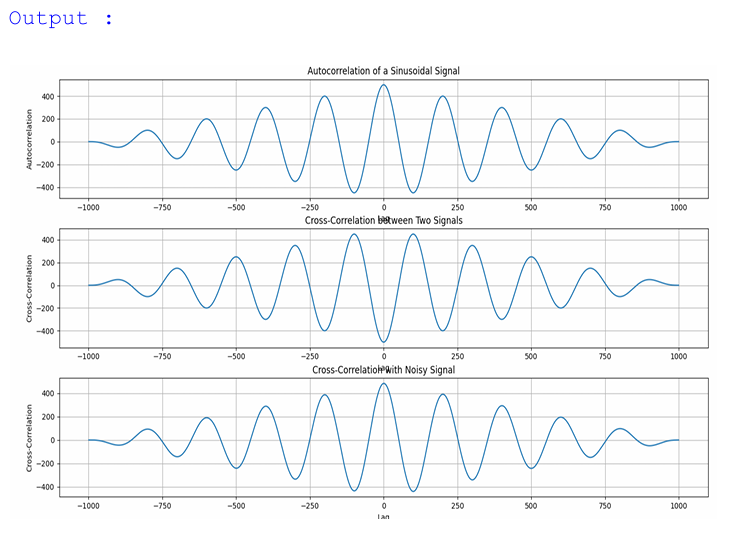
**plt.xlabel("Lag")**

**plt.ylabel("Cross-Correlation")**

**plt.grid()**

**plt.tight\_layout()**

**plt.show()**

****

**Experiment No-4. Signal Sequence**

**Theory:**

A **signal sequence** is a set of values that represents a signal over time or another variable. In signal processing, it typically refers to discrete-time signals (values at specific time points) or continuous-time signals (values at every point in time). Common examples include:

* **Impulse Signal**: A value of 1 at one point, and 0 elsewhere.
* **Step Signal**: 0 for negative time indices and 1 for non-negative indices.
* **Sinusoidal Signal**: A periodic wave, often used to represent oscillations.

Signal sequences are essential for analyzing, processing, and manipulating signals in various applications.

**Objective:**  
To generate, analyze, and visualize basic signal sequences (impulse, step, and sinusoidal) using Python.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.signal import convolve

def compute\_convolution(signal1, signal2):

conv\_result = convolve(signal1, signal2, mode='full', method='auto')

return conv\_result

fs = 1000  # Sampling frequency in Hz

t = np.linspace(0, 1, fs, endpoint=False)  # Time vector

freq = 5  # Frequency of the sine wave

sin\_signal = np.sin(2 \* np.pi \* freq \* t)

conv\_auto = compute\_convolution(sin\_signal, sin\_signal)

signal1 = sin\_signal

signal2 = np.roll(signal1, 100)

conv\_shifted = compute\_convolution(signal1, signal2)

noise = np.random.normal(0, 0.5, fs)

noisy\_signal = signal1 + noise

conv\_noisy = compute\_convolution(signal1, noisy\_signal)

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

plt.subplot(3, 1, 1)

plt.plot(conv\_auto)

plt.title("Autoconvolution of a Sinusoidal Signal")

plt.xlabel("Samples")

plt.ylabel("Convolution Output")

plt.grid()

plt.subplot(3, 1, 2)

plt.plot(conv\_shifted)

plt.title("Convolution between Signal and Shifted Version")

plt.xlabel("Samples")

plt.ylabel("Convolution Output")

plt.grid()

plt.subplot(3, 1, 3)

plt.plot(conv\_noisy)

plt.title("Convolution with Noisy Signal")

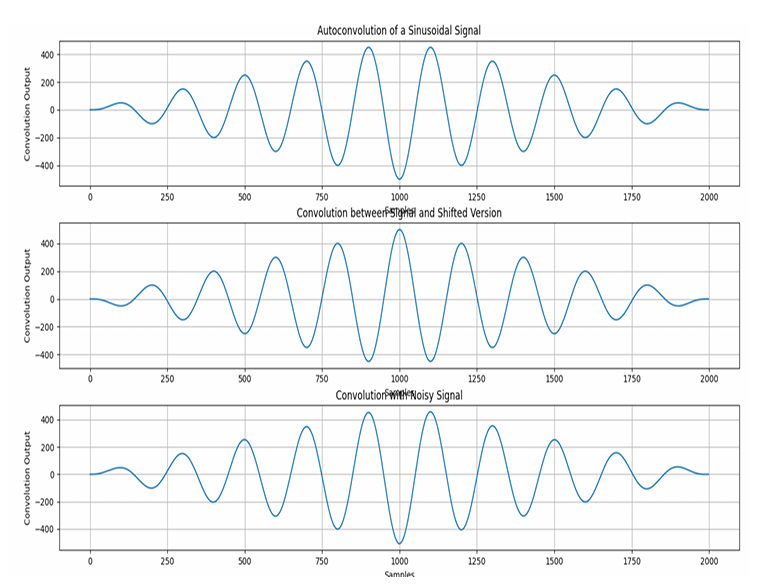
plt.xlabel("Samples")

plt.ylabel("Convolution Output")

plt.grid()

plt.tight\_layout()

plt.show()



**Experiment No-5. PPG Signal Processing (Filtering, Feature Extraction, Peak Detection, Heart Rate (HR), Systolic & Diastolic Peaks, Pulse Transit Time (PTT))**

**Theory:**

* **PPG Signal**: A non-invasive technique to measure blood volume changes.
* **Filtering**: To remove noise (using a bandpass filter, e.g., 0.5-5 Hz).
* **Peak Detection**: Identify systolic peaks (heartbeats) and diastolic peaks.
* **Heart Rate (HR)**: Measured by the time interval between systolic peaks, expressed in BPM.
* **Pulse Transit Time (PTT)**: Time difference between systolic peaks in PPG and other signals (e.g., ECG).

**Objective:**  
To process a **Photoplethysmogram (PPG)** signal by applying filtering, peak detection, feature extraction, and calculating heart rate (HR), systolic and diastolic peaks, and Pulse Transit Time (PTT).

**Code:**

import numpy as np

import scipy.signal as signal

import matplotlib.pyplot as plt

def bandpass\_filter(data, fs=100):

b, a = signal.butter(4, [0.5 / (0.5 \* fs), 5.0 / (0.5 \* fs)], btype='band')

return signal.filtfilt(b, a, data)

def detect\_peaks(signal\_data):

return signal.find\_peaks(signal\_data, distance=50)[0]

def extract\_heart\_rate(peaks, fs=100):

if len(peaks) < 2:

return 0

rr\_intervals = np.diff(peaks) / fs

return 60 / np.mean(rr\_intervals)

# Generate synthetic PPG signal

fs = 100

t = np.linspace(0, 10, fs \* 10)

sine\_signal = np.sin(2 \* np.pi \* 1.2 \* t)

noise\_signal = 0.1 \* np.random.normal(0, 1, len(t))

ppg\_signal = sine\_signal + noise\_signal

# Process PPG signal

filtered\_signal = bandpass\_filter(ppg\_signal, fs)

normalized\_signal = (filtered\_signal - np.min(filtered\_signal)) / (np.max(filtered\_signal) -

np.min(filtered\_signal))

peaks = detect\_peaks(normalized\_signal)

heart\_rate = extract\_heart\_rate(peaks, fs)

# Print results

print("Filtered Signal (first 10 values):", filtered\_signal[:10])

print("Detected Peaks (first 10 indices):", peaks[:10])

print(f"Estimated Heart Rate: {heart\_rate:.2f} BPM")

# Plot results

plt.figure(figsize=(12, 9))

plt.subplot(3, 2, 1)

plt.plot(t, sine\_signal, label='Raw Sine Signal')

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 2)

plt.plot(t, noise\_signal, label='Raw Noise Signal')

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 3)

plt.plot(t, ppg\_signal, label='Raw PPG Signal')

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 4)

plt.plot(t, filtered\_signal, label='Filtered PPG Signal')

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 5)

plt.plot(t, normalized\_signal, label='Normalized PPG Signal')

plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.subplot(3, 2, 6)

plt.plot(t, normalized\_signal,label=f'PPG with Detected Peaks')

plt.plot(t[peaks], normalized\_signal[peaks],'ro', label='Detected Peaks')

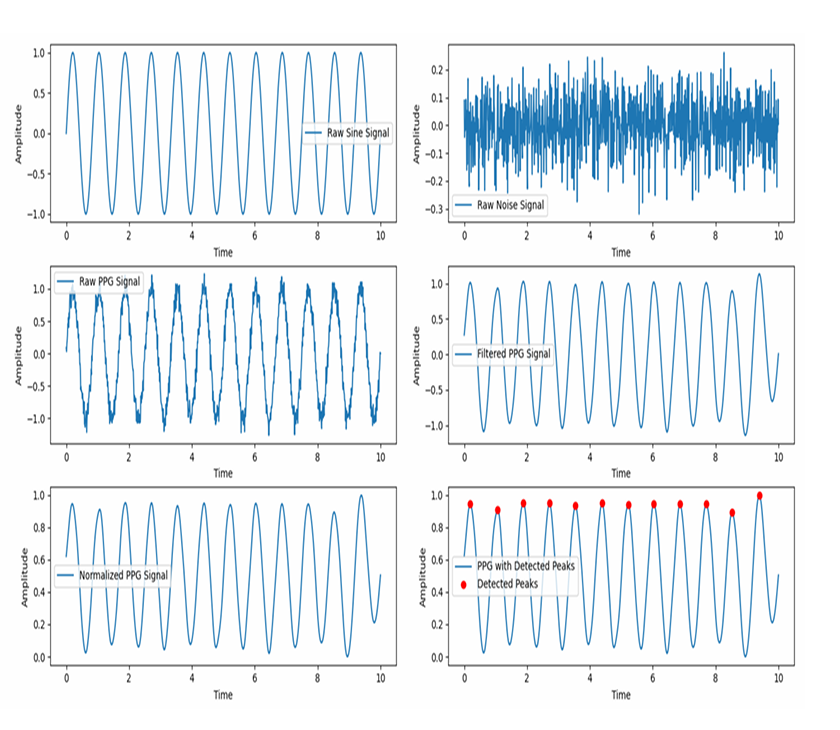
plt.xlabel("Time")

plt.ylabel("Amplitude")

plt.legend()

plt.tight\_layout()

plt.show()



**Experiment No-6. DFT**

**Theory:**

The Discrete Fourier Transform (DFT) converts a discrete-time signal from the time domain to the frequency domain. It decomposes the signal into sinusoidal components of different frequencies**.**

**Objective:**  
To compute the **Discrete Fourier Transform (DFT)** of a signal and analyze its frequency components.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

# Input sequence and N

x = [1,1,1,1]

N= 4

x = np.pad(x, (0, N - len(x)), mode='constant')

# DFT computation

X = np.fft.fft(x, N)

# IDFT computation (Inverse DFT)

x\_reconstructed = np.fft.ifft(X)

# Print the DFT and IDFT values

print("DFT values:", X)

print("Reconstructed IDFT values:", x\_reconstructed.real)

# Plot the input signal

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

plt.subplot(3, 1, 1)

plt.stem(range(len(x)), x)

plt.title('Input Signal x(n)')

plt.xlabel('n')

plt.ylabel('x(n)')

plt.grid()

# Plot the magnitude of DFT

plt.subplot(3, 1, 2)

plt.stem(range(N), np.abs(X))

plt.title('DFT Magnitude |X(k)|')

plt.xlabel('k')

plt.ylabel('|X(k)|')

plt.grid()

# Plot the IDFT signal

plt.subplot(3, 1, 3)

plt.stem(range(N), x\_reconstructed.real)

plt.title('Reconstructed Signal x(n) from IDFT')

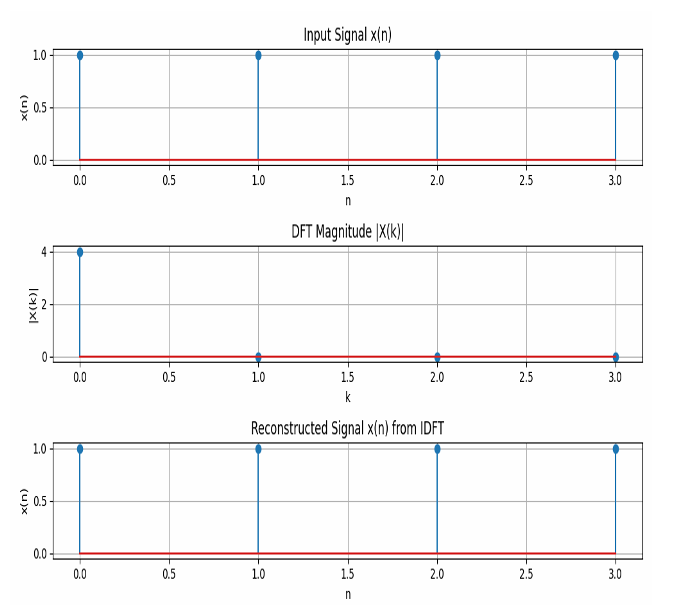
plt.xlabel('n')

plt.ylabel('x(n)')

plt.grid()

plt.tight\_layout()

plt.show()

****

**Experiment No-7. Frequency Bin**

**Theory:**

In DFT, a **frequency bin** corresponds to a specific frequency range that each DFT output represents. Each bin in the DFT contains information about the amplitude of a particular frequency component of the signal.

**Objective:**  
To understand **frequency bins** in the **Discrete Fourier Transform (DFT)** and analyze how a signal is represented in different frequency bins.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

# Signal parameters

fs = 1000 # Sampling frequency (Hz)

T = 1 / fs # Sampling period

duration = 1 # Signal duration in seconds

t = np.arange(0, duration, T) # Time vector

# Define a signal (sum of two sine waves)

f1 = 50 # Frequency of the first sine wave (Hz)

f2 = 120 # Frequency of the second sine wave (Hz)

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

# Compute the DFT of the signal

N = len(signal) # Number of samples

X = np.fft.fft(signal) # Compute the FFT (Discrete Fourier Transform)

f = np.fft.fftfreq(N, T) # Frequency vector for DFT bins

# Plot the time-domain signal

plt.figure(figsize=(12, 6))

plt.subplot(2, 1, 1)

plt.plot(t, signal, label='Time Domain Signal')

plt.title('Time Domain Signal')

plt.xlabel('Time [s]')

plt.ylabel('Amplitude')

plt.grid(True)

# Plot the frequency-domain (DFT) representation showing frequency bins

plt.subplot(2, 1, 2)

plt.plot(f[:N // 2], np.abs(X)[:N // 2], label='Magnitude of DFT')

plt.title('Frequency Domain (DFT) with Frequency Bins')

plt.xlabel('Frequency [Hz]')

plt.ylabel('Magnitude')

plt.grid(True)

plt.tight\_layout()

plt.show()

