Heaven’s Light is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Assignment

Course Code: CSE 3209

Course Title: Digital Signal Processing

<u>Submitted By</u>-	**<u>Submitted To</u>-**
Name : Sajidur Rahman Tarafder	**Md. Farukuzzaman Faruk**
Department : CSE	**Assistant Professor**
Roll No. : 2003154	**Department of CSE, RUET**
Section : C	
Session:2020-21	

Problem Statement:

Generate a discrete time sine wave and perform the following operations:

1. Time Shifting, Reversal, and Scaling
2. Compute the DFT and compare it with `numpy.fft.fft()`
3. Compute the DTFT
4. Compare execution times for manual DFT vs. FFT
5. Apply a low-pass filter in the frequency domain

Objective:

The objective of this lab is to perform various operations on a discrete-time sine wave, compute its Discrete Fourier Transform (DFT) and Discrete-Time Fourier Transform (DTFT), compare the efficiency of DFT and Fast Fourier Transform (FFT), and apply a low-pass filter in the frequency domain.

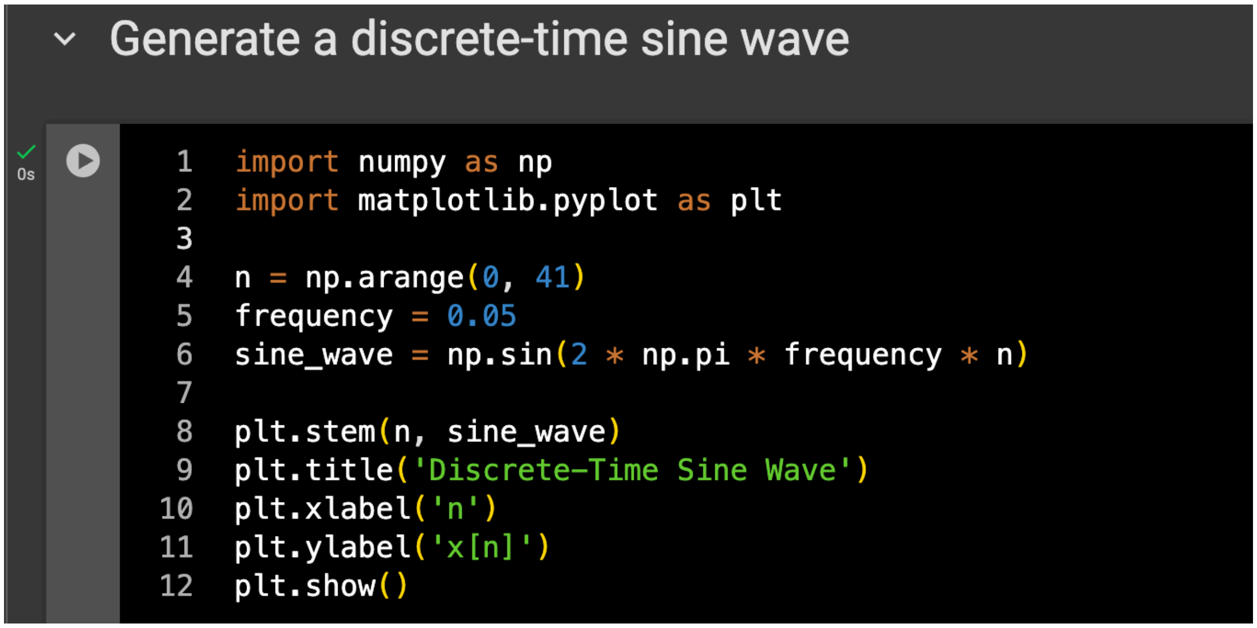
Theoretical Background:

1. Generate a discrete time sine wave:

A sine wave in the discrete-time domain is a fundamental periodic signal that is sampled at specific intervals from its continuous counterpart. It is mathematically expressed as,

$x[n] = A \sin(2\pi f n)$, where A is the amplitude, f is the frequency, n is the discrete time index. The amplitude determines the peak value of the wave, the frequency defines how many cycles occur per sample, and the phase shift specifies the starting point of the wave. In the discrete domain, the sine wave is represented as a sequence of values rather than a continuous curve, making it suitable for digital signal processing.

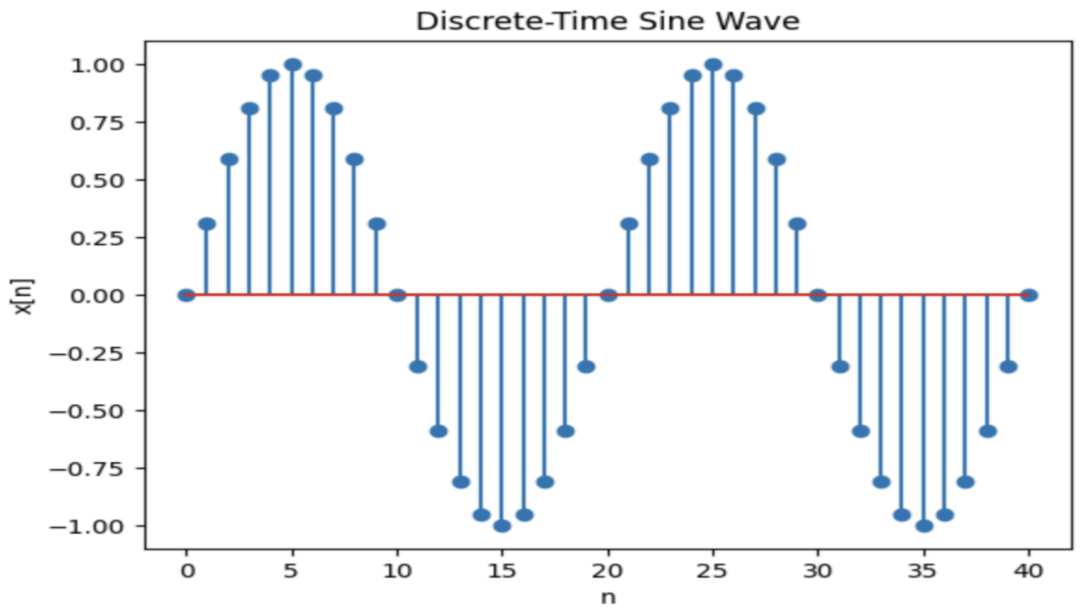
Code:



Generate a discrete-time sine wave

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

n = np.arange(0, 41)
frequency = 0.05
sine_wave = np.sin(2 * np.pi * frequency * n)

plt.stem(n, sine_wave)
plt.title('Discrete-Time Sine Wave')
plt.xlabel('n')
plt.ylabel('x[n]')
plt.show()
```

Generated Output:



2. Basic Signal Operations:

Time Shifting:

Time shifting involves shifting the signal in time by k samples. If k is positive, the signal is shifted to the right, delaying it. If k is negative, the signal is shifted to the left, advancing it. Mathematically, this is represented as: [$x[n - k]$] where k denotes the number of samples by which the signal is shifted.

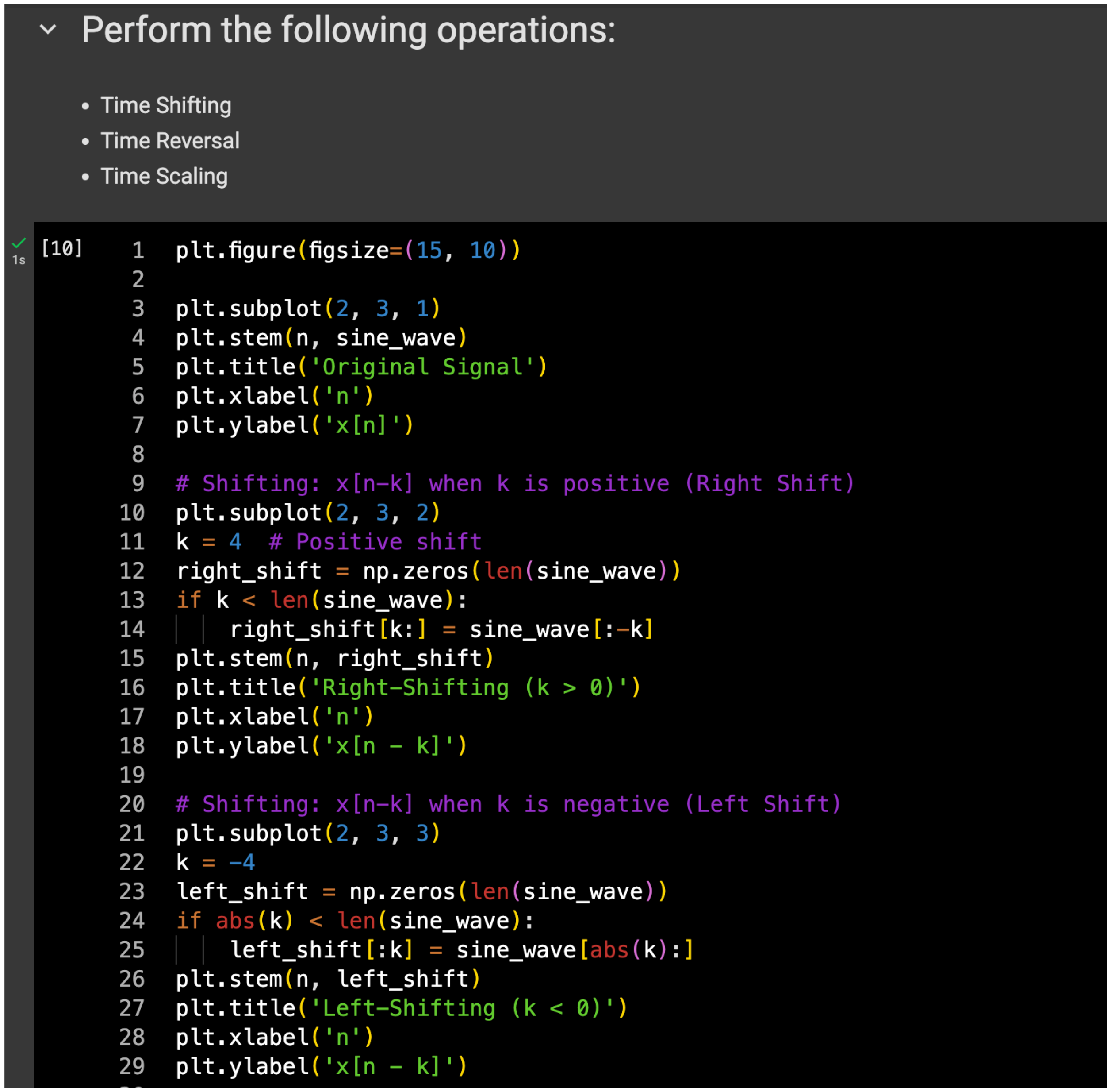
Time Reversal:

Time reversal involves reversing the signal in time. This operation flips the signal around the vertical axis, effectively reversing the order of the samples. Mathematically, this is represented as: [$x[-n]$] where the signal is mirrored around $n=0$.

Time Scaling:

Time scaling involves compressing or expanding the signal in time by a factor of 'a'. If 'a' is greater than 1, the signal is down scaled. Down scaling involves decrement of the number of samples in the resultant signal. On the other hand, if 'a' is between 0 and 1, the signal is upscaled. Up scaling involves increment of the number of samples in the resultant signal. It can be achieved by average, interpolation or by taking r.m.s value. Mathematically, this is represented as: $x[a.n]$ where a is the scaling factor applied to the time index n.

Code:

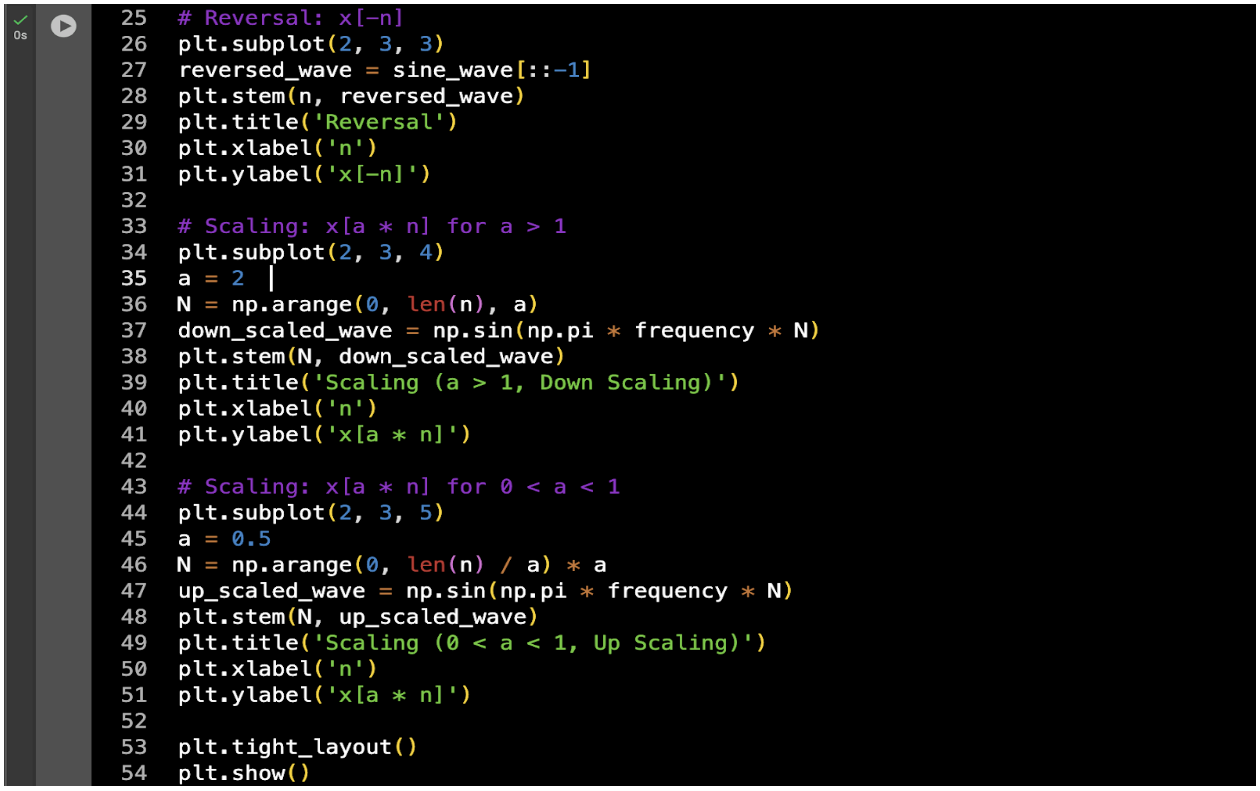


Perform the following operations:

- Time Shifting
- Time Reversal
- Time Scaling

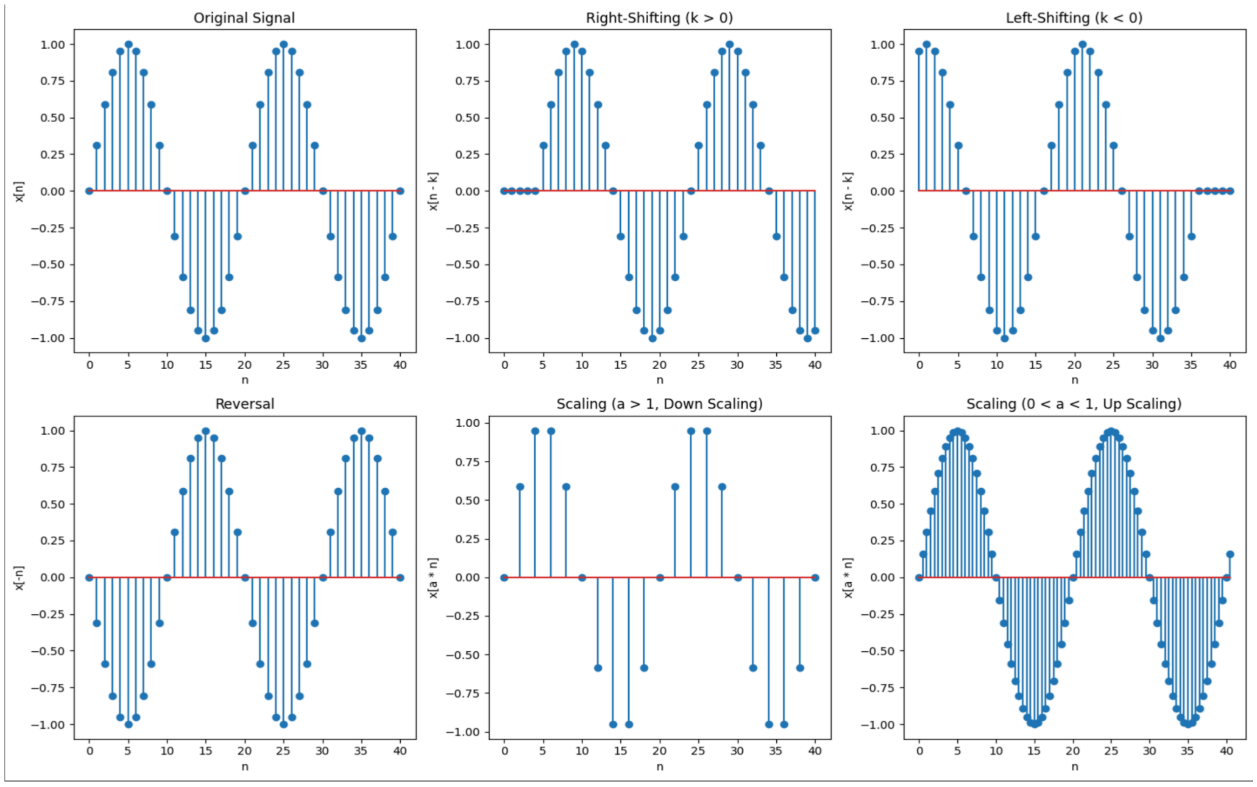
[10]

```python
plt.figure(figsize=(15, 10))

plt.subplot(2, 3, 1)
plt.stem(n, sine_wave)
plt.title('Original Signal')
plt.xlabel('n')
plt.ylabel('x[n]')

# Shifting: x[n-k] when k is positive (Right Shift)
plt.subplot(2, 3, 2)
k = 4  # Positive shift
right_shift = np.zeros(len(sine_wave))
if k < len(sine_wave):
    right_shift[k:] = sine_wave[:-k]
plt.stem(n, right_shift)
plt.title('Right-Shifting (k > 0)')
plt.xlabel('n')
plt.ylabel('x[n - k]')

# Shifting: x[n-k] when k is negative (Left Shift)
plt.subplot(2, 3, 3)
k = -4
left_shift = np.zeros(len(sine_wave))
if abs(k) < len(sine_wave):
    left_shift[:k] = sine_wave[abs(k):]
plt.stem(n, left_shift)
plt.title('Left-Shifting (k < 0)')
plt.xlabel('n')
plt.ylabel('x[n - k]')
```



```python
# Reversal: x[-n]
plt.subplot(2, 3, 3)
reversed_wave = sine_wave[::-1]
plt.stem(n, reversed_wave)
plt.title('Reversal')
plt.xlabel('n')
plt.ylabel('x[-n]')

# Scaling: x[a * n] for a > 1
plt.subplot(2, 3, 4)
a = 2
N = np.arange(0, len(n), a)
down_scaled_wave = np.sin(np.pi * frequency * N)
plt.stem(N, down_scaled_wave)
plt.title('Scaling (a > 1, Down Scaling)')
plt.xlabel('n')
plt.ylabel('x[a * n]')

# Scaling: x[a * n] for 0 < a < 1
plt.subplot(2, 3, 5)
a = 0.5
N = np.arange(0, len(n) / a) * a
up_scaled_wave = np.sin(np.pi * frequency * N)
plt.stem(N, up_scaled_wave)
plt.title('Scaling (0 < a < 1, Up Scaling)')
plt.xlabel('n')
plt.ylabel('x[a * n]')

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

__Generated Output__:



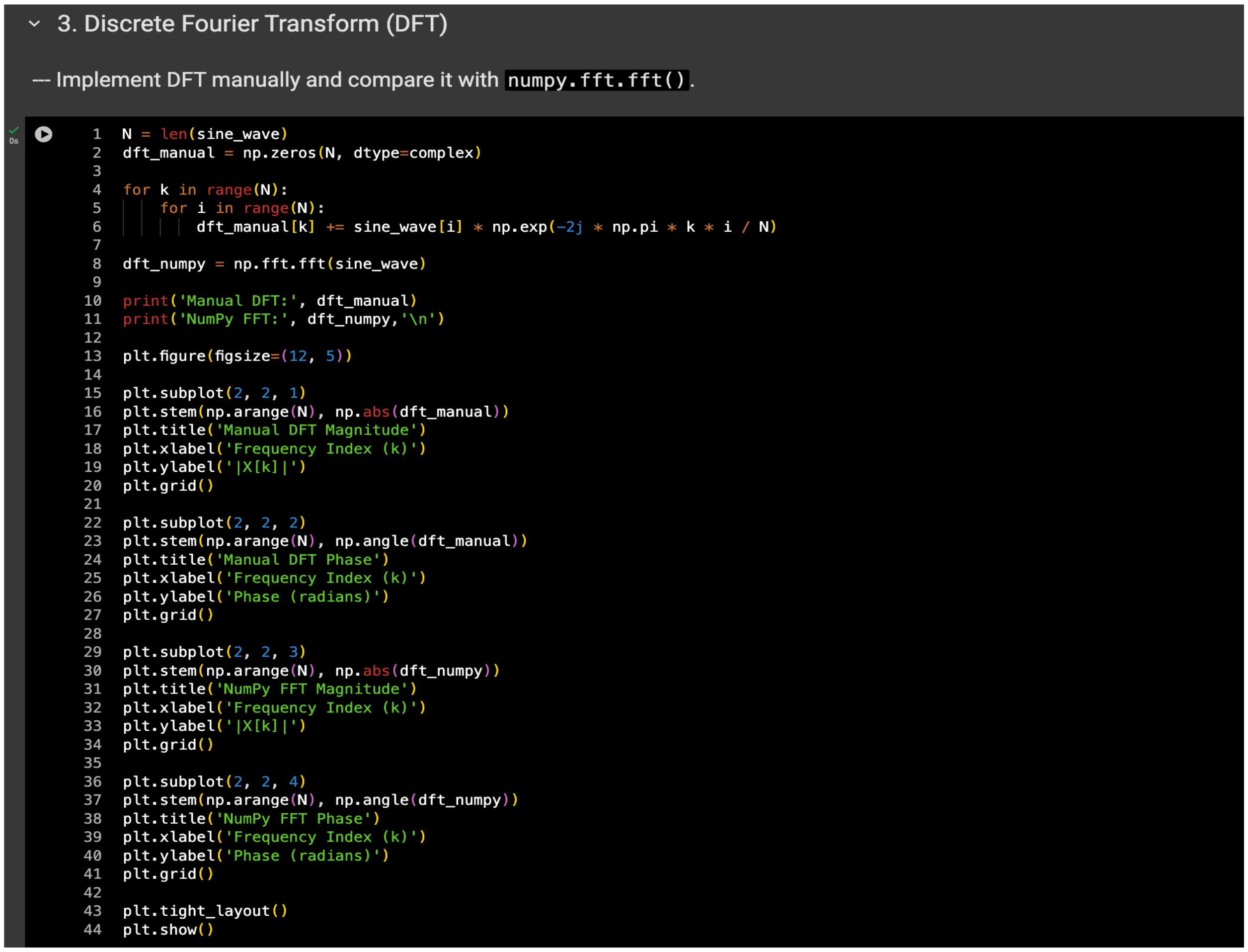
3. Discrete Fourier Transform (DFT)

The DFT is given by:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{\left\{-j\left(\frac{2\pi}{N}\right)kn\right\}}$$

The DFT transforms a discrete-time signal x[n] into its frequency-domain representation. It is computed by summing the product of the signal x[n] and a complex exponential term, which represents sinusoidal basis functions. The result provides the amplitude and phase of the signal's frequency components.

Code:

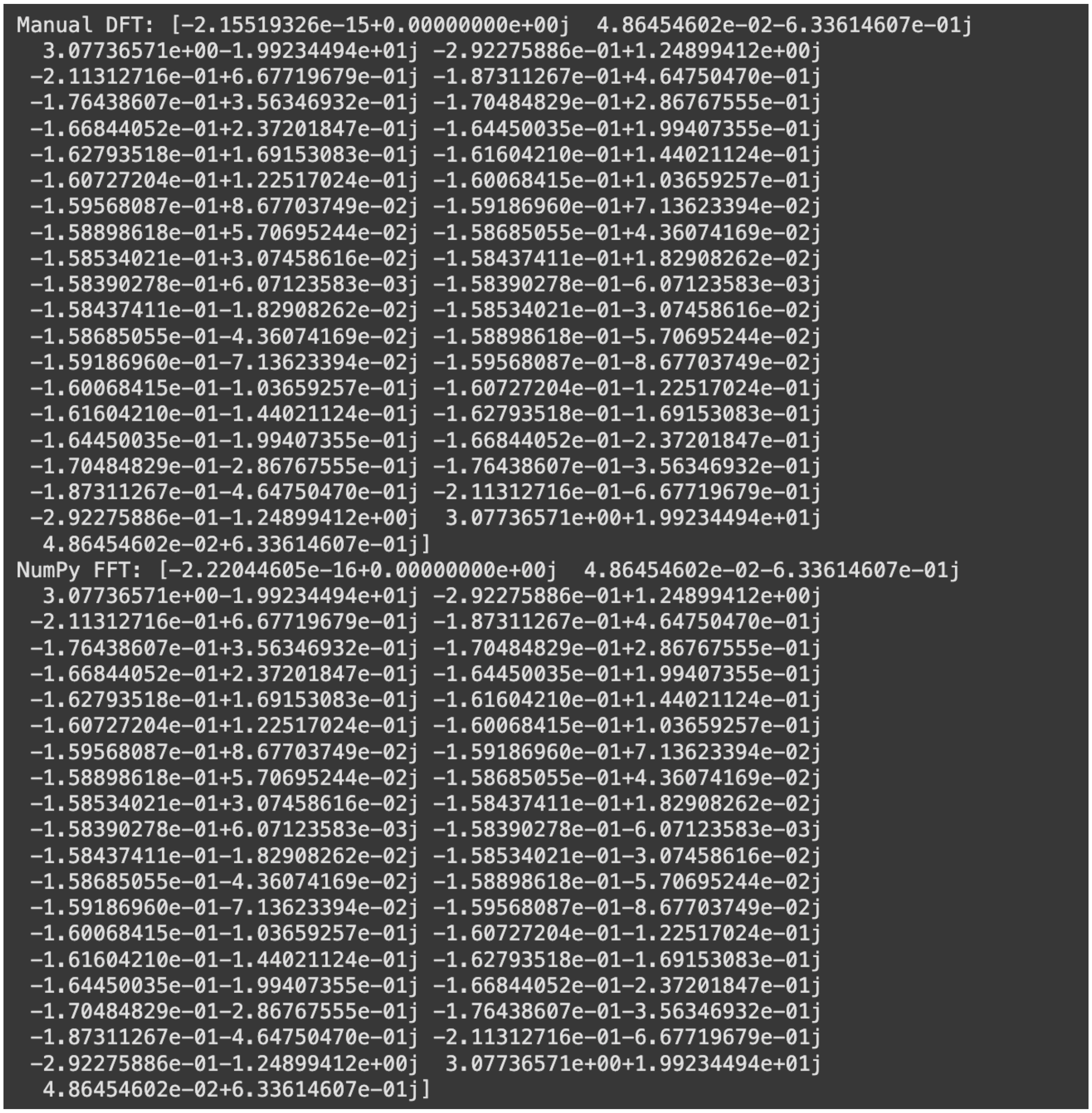


3. Discrete Fourier Transform (DFT)

--- Implement DFT manually and compare it with `numpy.fft.fft()`.

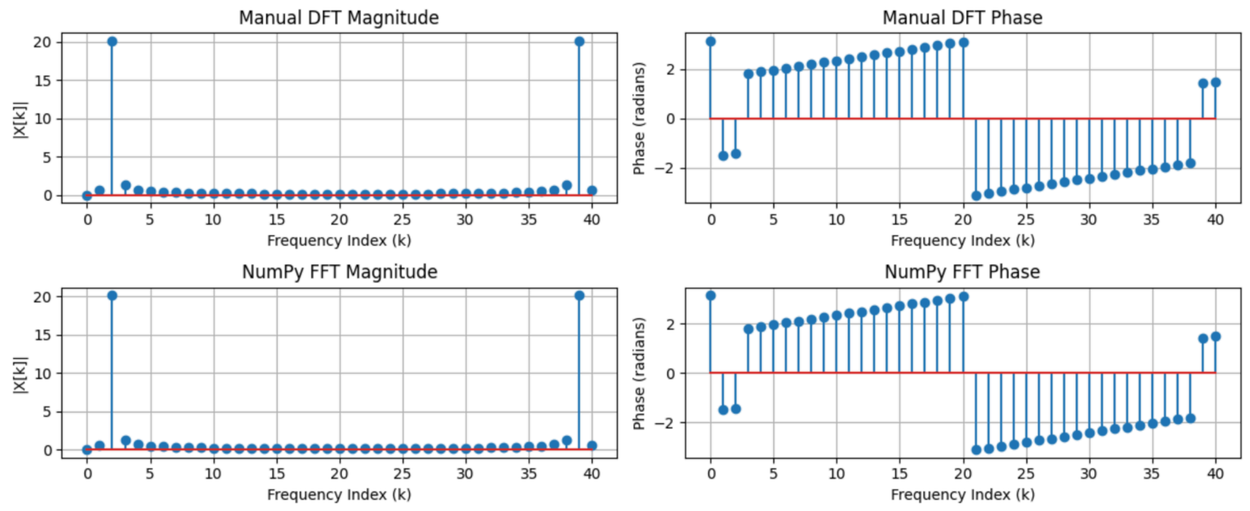
```python
N = len(sine_wave)
dft_manual = np.zeros(N, dtype=complex)

for k in range(N):
    for i in range(N):
        dft_manual[k] += sine_wave[i] * np.exp(-2j * np.pi * k * i / N)

dft_numpy = np.fft.fft(sine_wave)

print('Manual DFT:', dft_manual)
print('NumPy FFT:', dft_numpy,'\n')

plt.figure(figsize=(12, 5))

plt.subplot(2, 2, 1)
plt.stem(np.arange(N), np.abs(dft_manual))
plt.title('Manual DFT Magnitude')
plt.xlabel('Frequency Index (k)')
plt.ylabel('|X[k]|')
plt.grid()

plt.subplot(2, 2, 2)
plt.stem(np.arange(N), np.angle(dft_manual))
plt.title('Manual DFT Phase')
plt.xlabel('Frequency Index (k)')
plt.ylabel('Phase (radians)')
plt.grid()

plt.subplot(2, 2, 3)
plt.stem(np.arange(N), np.abs(dft_numpy))
plt.title('NumPy FFT Magnitude')
plt.xlabel('Frequency Index (k)')
plt.ylabel('|X[k]|')
plt.grid()

plt.subplot(2, 2, 4)
plt.stem(np.arange(N), np.angle(dft_numpy))
plt.title('NumPy FFT Phase')
plt.xlabel('Frequency Index (k)')
plt.ylabel('Phase (radians)')
plt.grid()

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

Generated Output:



```
Manual DFT: [-2.15519326e-15+0.00000000e+00j  4.86454602e-02-6.33614607e-01j
  3.07736571e+00-1.99234494e+01j -2.92275886e-01+1.24899412e+00j
 -2.11312716e-01+6.67719679e-01j -1.87311267e-01+4.64750470e-01j
 -1.76438607e-01+3.56346932e-01j -1.70484829e-01+2.86767555e-01j
 -1.66844052e-01+2.37201847e-01j -1.64450035e-01+1.99407355e-01j
 -1.62793518e-01+1.69153083e-01j -1.61604210e-01+1.44021124e-01j
 -1.60727204e-01+1.22517024e-01j -1.60068415e-01+1.03659257e-01j
 -1.59568087e-01+8.67703749e-02j -1.59186960e-01+7.13623394e-02j
 -1.58898618e-01+5.70695244e-02j -1.58685055e-01+4.36074169e-02j
 -1.58534021e-01+3.07458616e-02j -1.58437411e-01+1.82908262e-02j
 -1.58390278e-01+6.07123583e-03j -1.58390278e-01-6.07123583e-03j
 -1.58437411e-01-1.82908262e-02j -1.58534021e-01-3.07458616e-02j
 -1.58685055e-01-4.36074169e-02j -1.58898618e-01-5.70695244e-02j
 -1.59186960e-01-7.13623394e-02j -1.59568087e-01-8.67703749e-02j
 -1.60068415e-01-1.03659257e-01j -1.60727204e-01-1.22517024e-01j
 -1.61604210e-01-1.44021124e-01j -1.62793518e-01-1.69153083e-01j
 -1.64450035e-01-1.99407355e-01j -1.66844052e-01-2.37201847e-01j
 -1.70484829e-01-2.86767555e-01j -1.76438607e-01-3.56346932e-01j
 -1.87311267e-01-4.64750470e-01j -2.11312716e-01-6.67719679e-01j
 -2.92275886e-01-1.24899412e+00j  3.07736571e+00+1.99234494e+01j
  4.86454602e-02+6.33614607e-01j]
NumPy FFT: [-2.22044605e-16+0.00000000e+00j  4.86454602e-02-6.33614607e-01j
  3.07736571e+00-1.99234494e+01j -2.92275886e-01+1.24899412e+00j
 -2.11312716e-01+6.67719679e-01j -1.87311267e-01+4.64750470e-01j
 -1.76438607e-01+3.56346932e-01j -1.70484829e-01+2.86767555e-01j
 -1.66844052e-01+2.37201847e-01j -1.64450035e-01+1.99407355e-01j
 -1.62793518e-01+1.69153083e-01j -1.61604210e-01+1.44021124e-01j
 -1.60727204e-01+1.22517024e-01j -1.60068415e-01+1.03659257e-01j
 -1.59568087e-01+8.67703749e-02j -1.59186960e-01+7.13623394e-02j
 -1.58898618e-01+5.70695244e-02j -1.58685055e-01+4.36074169e-02j
 -1.58534021e-01+3.07458616e-02j -1.58437411e-01+1.82908262e-02j
 -1.58390278e-01+6.07123583e-03j -1.58390278e-01-6.07123583e-03j
 -1.58437411e-01-1.82908262e-02j -1.58534021e-01-3.07458616e-02j
 -1.58685055e-01-4.36074169e-02j -1.58898618e-01-5.70695244e-02j
 -1.59186960e-01-7.13623394e-02j -1.59568087e-01-8.67703749e-02j
 -1.60068415e-01-1.03659257e-01j -1.60727204e-01-1.22517024e-01j
 -1.61604210e-01-1.44021124e-01j -1.62793518e-01-1.69153083e-01j
 -1.64450035e-01-1.99407355e-01j -1.66844052e-01-2.37201847e-01j
 -1.70484829e-01-2.86767555e-01j -1.76438607e-01-3.56346932e-01j
 -1.87311267e-01-4.64750470e-01j -2.11312716e-01-6.67719679e-01j
 -2.92275886e-01-1.24899412e+00j  3.07736571e+00+1.99234494e+01j
  4.86454602e-02+6.33614607e-01j]
```

Visualization:



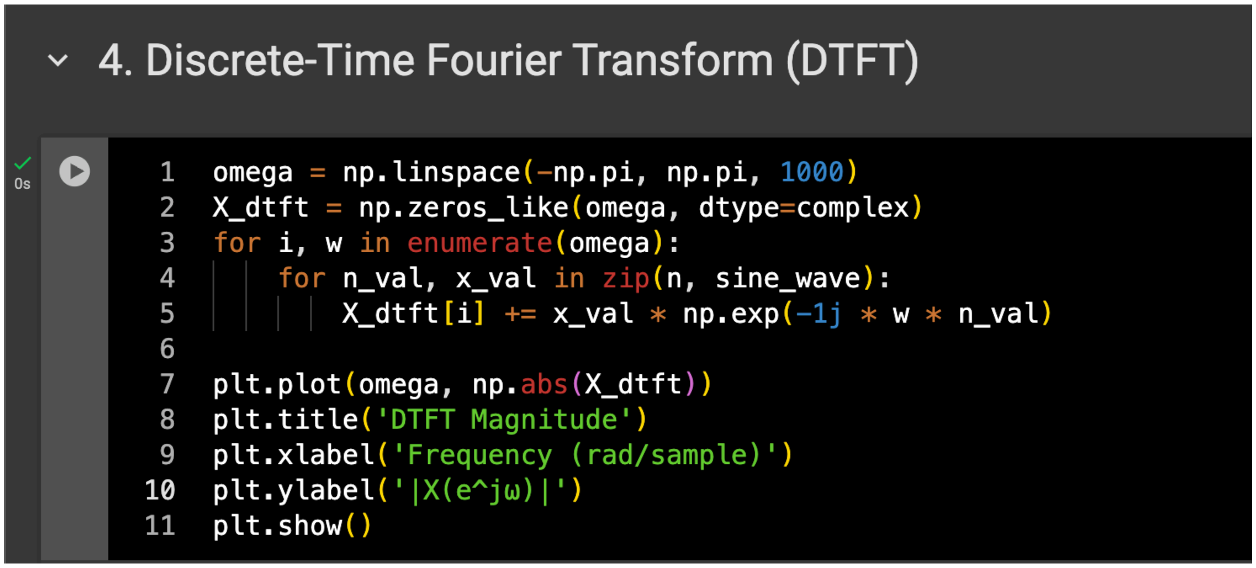
4. Discrete-Time Fourier Transform (DTFT)

The DTFT is computed as:

$$X\left(e^{j\omega}\right) = \sum_{\{n=-\infty\}}^{\{\infty\}} x[n]\, e\hat{}\{-\ j\omega n\}$$

The Discrete-Time Fourier Transform (DTFT) is a mathematical representation that transforms a discrete-time signal from the time domain into the frequency domain. where e^jω (ω=omega) represents the frequency-domain function of the signal x[n]. This function is continuous with respect to the angular frequency (omega) which is measured in radians per sample. The term x[n] is the discrete-time signal in the time domain, which consists of a sequence of values indexed by n. The summation from -infinity to +infinity denotes contributions from all time-domain samples to compute the frequency-domain representation. The complex exponential e^jωn, which can be expressed as cos(-ωn)-jsin(ωn) using Euler's formula can be acts as a basis function that extracts the frequency components of the signal. The DTFT provides a continuous spectrum for signals, making it particularly useful for analyzing the frequency content of infinite-length discrete-time signals.

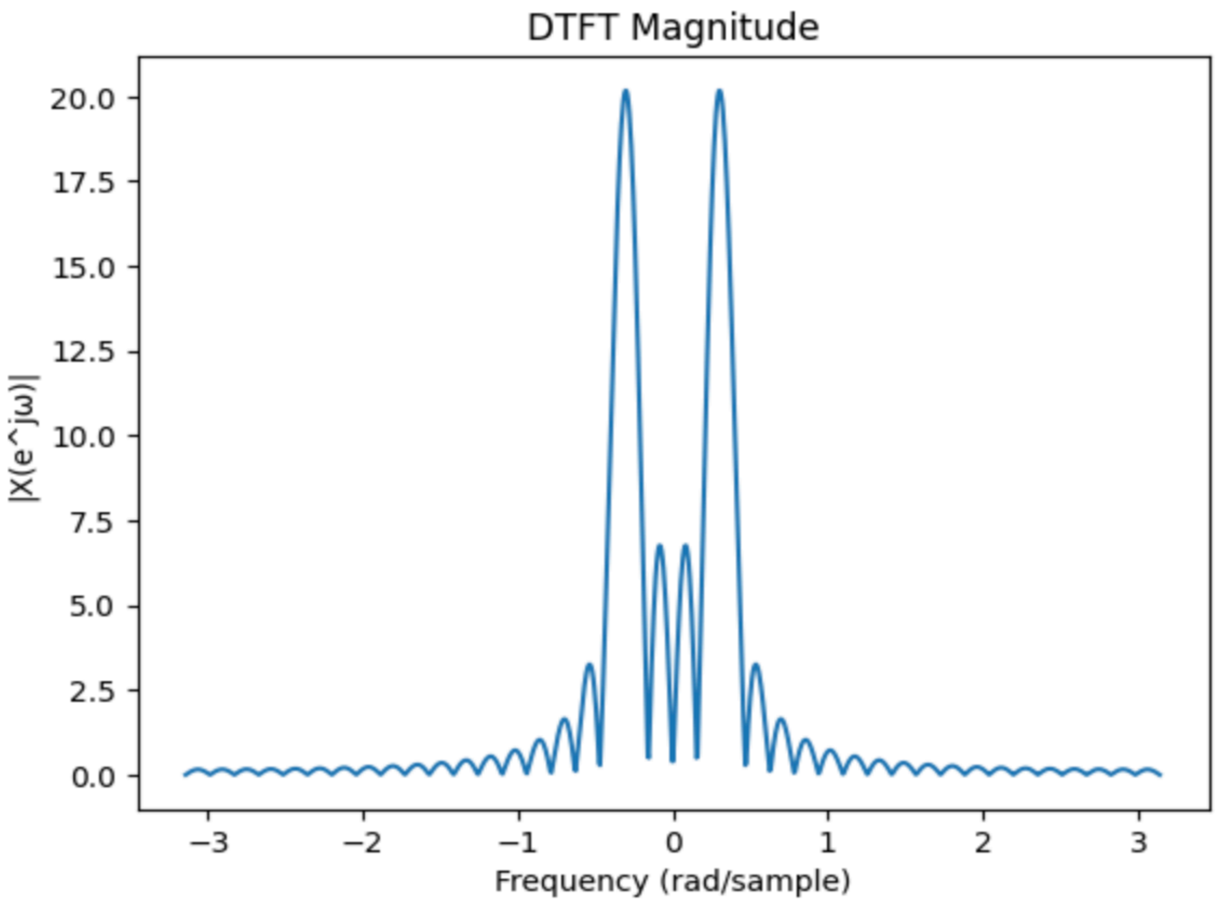
<u>Code</u>:



4. Discrete-Time Fourier Transform (DTFT)

```python
omega = np.linspace(-np.pi, np.pi, 1000)
X_dtft = np.zeros_like(omega, dtype=complex)
for i, w in enumerate(omega):
    for n_val, x_val in zip(n, sine_wave):
        X_dtft[i] += x_val * np.exp(-1j * w * n_val)

plt.plot(omega, np.abs(X_dtft))
plt.title('DTFT Magnitude')
plt.xlabel('Frequency (rad/sample)')
plt.ylabel('|X(e^jω)|')
plt.show()
```

Generated Output:

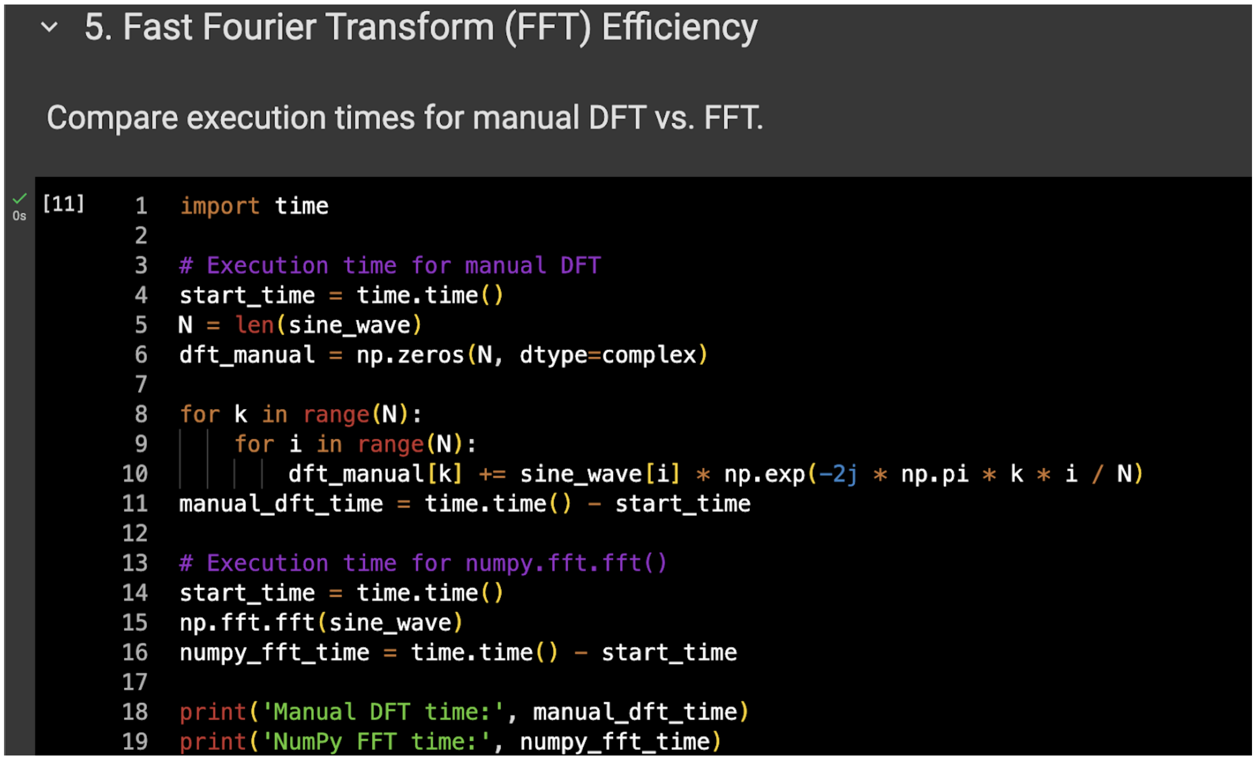


5. Fast Fourier Transform (FFT) Efficiency

The Fast Fourier Transform (FFT) is an optimized algorithm for computing the Discrete Fourier Transform (DFT). While the DFT provides a way to transform a signal from the time domain to the frequency domain, it is computationally expensive for large signals. The FFT significantly reduces the computational complexity, making it much faster and more efficient.

Manual DFT requires complex multiplications and additions for a signal of length N. So, it's computational complexity is O(N^2).But FFT reduces the number of computations by exploiting the symmetry and periodicity properties of the DFT. It divides the DFT computation into smaller sub-problems and tries to solve them recursively. Computational Complexity: O(Nlog N), which is much faster than O(N^2) for large N .

Code:

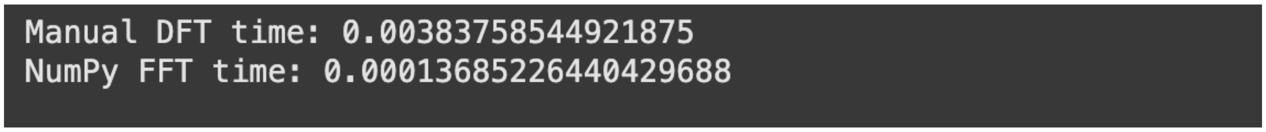


5. Fast Fourier Transform (FFT) Efficiency

Compare execution times for manual DFT vs. FFT.

```
[11]
import time

# Execution time for manual DFT
start_time = time.time()
N = len(sine_wave)
dft_manual = np.zeros(N, dtype=complex)

for k in range(N):
    for i in range(N):
        dft_manual[k] += sine_wave[i] * np.exp(-2j * np.pi * k * i / N)
manual_dft_time = time.time() - start_time

# Execution time for numpy.fft.fft()
start_time = time.time()
np.fft.fft(sine_wave)
numpy_fft_time = time.time() - start_time

print('Manual DFT time:', manual_dft_time)
print('NumPy FFT time:', numpy_fft_time)
```

Generated Output:

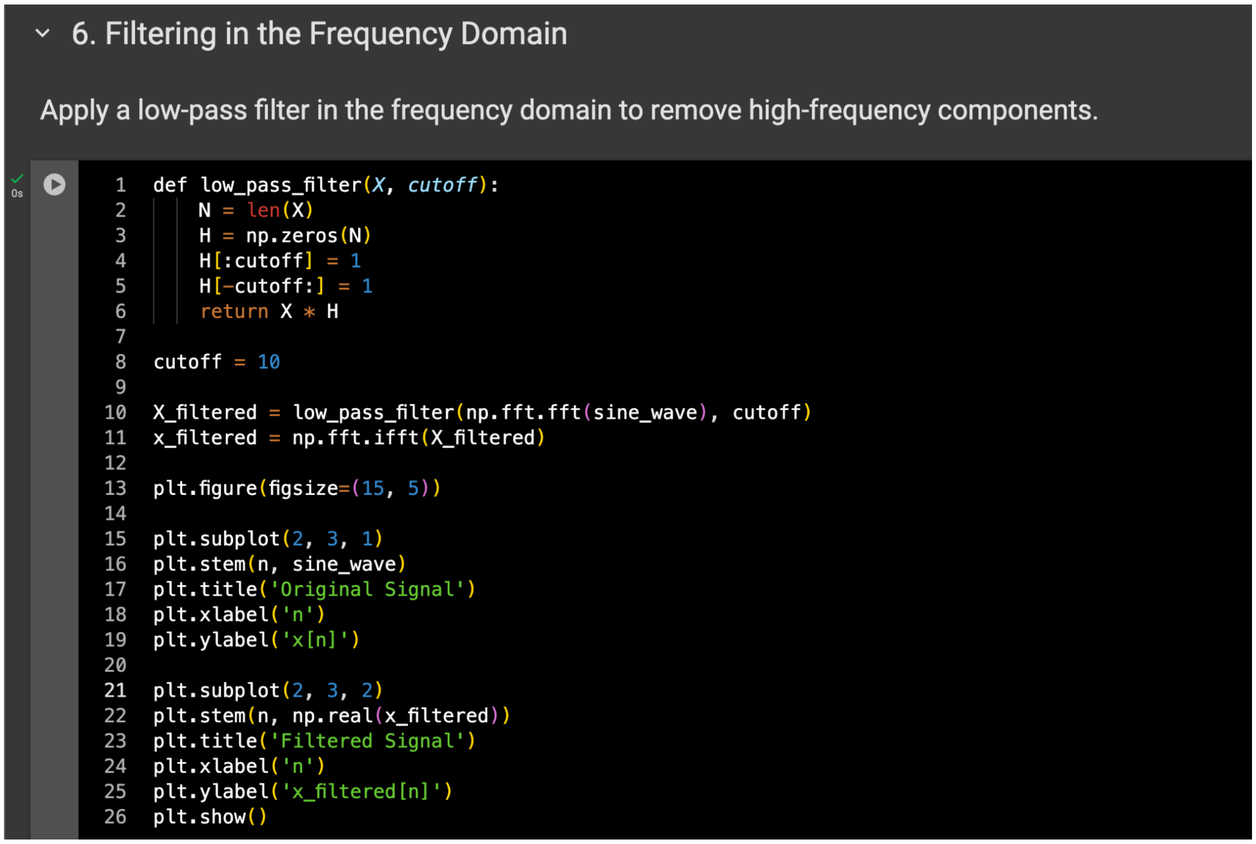


```
Manual DFT time: 0.003837585449211875
NumPy FFT time: 0.00013685226440429688
```

6. Filtering in the Frequency Domain

A low-pass filter is a signal processing technique that allows frequencies below a specified cutoff frequency to pass through while attenuating frequencies above the cutoff. In the frequency domain, signals are represented using the Fourier Transform (e.g., DFT or FFT), making filtering a straightforward operation by modifying the frequency components. The low-pass filter is defined by a cutoff frequency, where frequencies below the cutoff are preserved, and those above are removed. This is achieved by applying a frequency-domain mask or transfer function to the signal. The filtered signal is then transformed back to the time domain using the inverse Fourier Transform. Low-pass filters are commonly used for noise reduction, smoothing signals, and retaining essential low-frequency components in applications like audio processing, image processing, and communications.

Code:

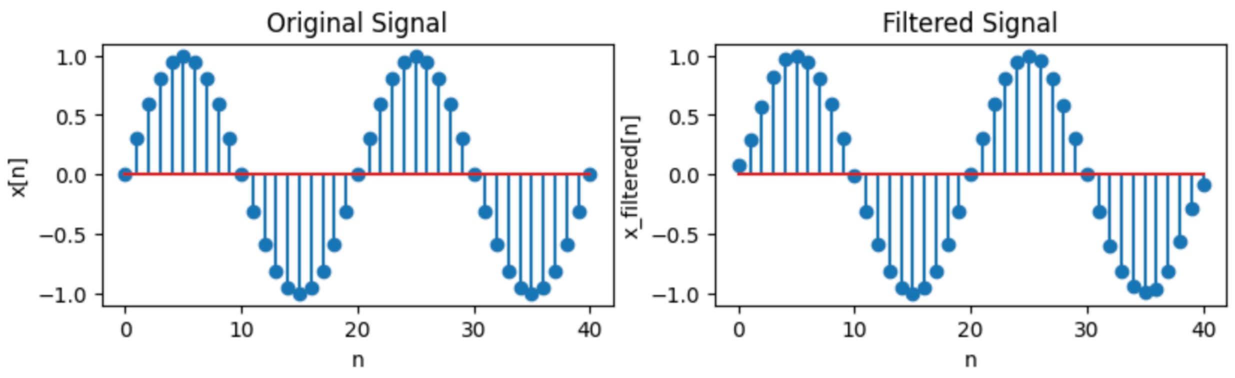


6. Filtering in the Frequency Domain

Apply a low-pass filter in the frequency domain to remove high-frequency components.

```python
def low_pass_filter(X, cutoff):
    N = len(X)
    H = np.zeros(N)
    H[:cutoff] = 1
    H[-cutoff:] = 1
    return X * H

cutoff = 10

X_filtered = low_pass_filter(np.fft.fft(sine_wave), cutoff)
x_filtered = np.fft.ifft(X_filtered)

plt.figure(figsize=(15, 5))

plt.subplot(2, 3, 1)
plt.stem(n, sine_wave)
plt.title('Original Signal')
plt.xlabel('n')
plt.ylabel('x[n]')

plt.subplot(2, 3, 2)
plt.stem(n, np.real(x_filtered))
plt.title('Filtered Signal')
plt.xlabel('n')
plt.ylabel('x_filtered[n]')
plt.show()
```

Generated Output:



Conclusion:

This assignment provided a deeper understanding of basic signal operations such as shifting, reversing, and scaling. We implemented the Discrete Fourier Transform (DFT) using Python and NumPy to transform signals into the frequency domain. By comparing the manual DFT implementation with the Fast Fourier Transform (FFT), we observed the significant efficiency gains of FFT, which is much faster and more computationally efficient. Additionally, we applied the frequency-domain filtering techniques to modify and enhance signals.