TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS INFORMATICS

ANDMICROELECTRONICS

DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATION

Laboratory work 3

Subject: Frequency domain analysis of signals

Done by: **Konjevic Alexandra,**
st. gr. FAF-213

Verified by: **Railean Serghei,**
university lecturer

CHIŞINĂU, 2024

TASK OF THE LABORATORY WORK

The purpose of this lab work is to explore various aspects of Fourier Transform and signal processing techniques. It involves tasks such as generating discrete signals, performing Fourier Transform, analyzing frequency spectra, determining magnitudes and phases, modeling rectangular pulses, filtering noise signals, and examining continuous Fourier Transform. The tasks cover practical applications of Fourier analysis in signal processing and aim to deepen understanding of frequency domain representations of signals.

THEORETICAL CONSIDERATIONS

When we discretize a continuous signal $f(t)$ at intervals of T seconds, we get a sequence of values $f_k = f(kT)$. In MATLAB, vector indexing starts from 1, but signal notations often begin from 0 or any other value, including negatives.

In the frequency domain, signals can be represented by complex values, representing the sinusoids comprising the signal.

The Discrete Fourier Transform (DFT) algorithm transforms a digital signal from the time domain into a set of points in the frequency domain. It maps N time domain values $[f_k]$ to N complex values $[F_k]$, representing frequency domain information. The Fast Fourier Transform (FFT) is used when N is a power of 2.

To avoid aliasing, the sampling rate should be at least twice the frequency of the highest sinusoid in the signal. The Nyquist frequency, half of the sampling frequency, represents the upper limit of frequency components that can be represented in the digital signal.

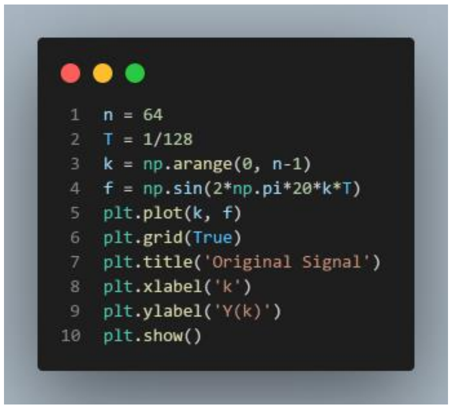
In MATLAB, the `fft` function is used for the DFT. It can take one or two arguments. With one argument (a vector containing the signal in the time domain), it returns another vector containing complex values representing the frequency domain content of the signal. With two arguments, the first argument is the signal vector, and the second is an integer L specifying the number of points in the output vector.

The frequency values obtained using fft correspond to frequencies separated by $1/NT$, where NT is the number of time-domain samples. Frequencies higher than the Nyquist frequency do not provide new information.

Understanding these concepts lays the foundation for applying Fourier analysis and signal processing techniques in MATLAB.

IMPLEMENTATIONS AND DIAGRAMS

1.1 Generate a discrete signal with 64 samples.



```python
n = 64
T = 1/128
k = np.arange(0, n-1)
f = np.sin(2*np.pi*20*k*T)
plt.plot(k, f)
plt.grid(True)
plt.title('Original Signal')
plt.xlabel('k')
plt.ylabel('Y(k)')
plt.show()
```

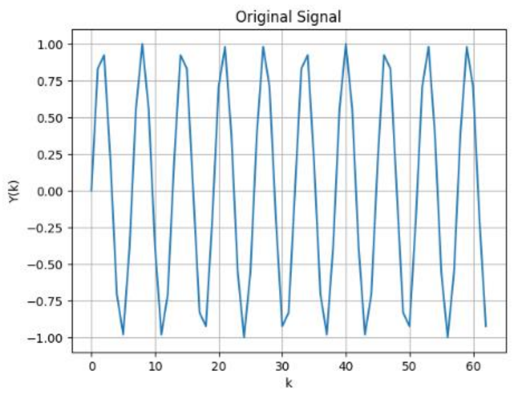
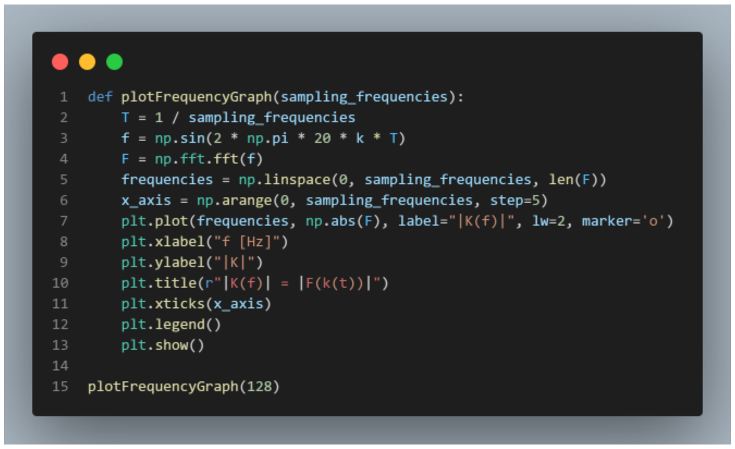


Figure 1a. Program for generating the signal

Figure 1b. Signal's graph

1.2 Determine F(k).



```python
def plotFrequencyGraph(sampling_frequencies):
    T = 1 / sampling_frequencies
    f = np.sin(2 * np.pi * 20 * k * T)
    F = np.fft.fft(f)
    frequencies = np.linspace(0, sampling_frequencies, len(F))
    x_axis = np.arange(0, sampling_frequencies, step=5)
    plt.plot(frequencies, np.abs(F), label="|K(f)|", lw=2, marker='o')
    plt.xlabel("f [Hz]")
    plt.ylabel("|K|")
    plt.title(r"|K(f)| = |F(k(t))|")
    plt.xticks(x_axis)
    plt.legend()
    plt.show()

plotFrequencyGraph(128)
```

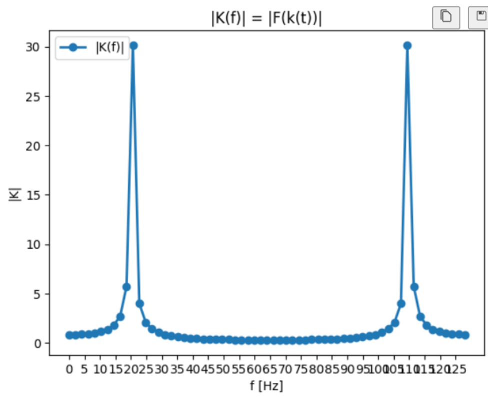
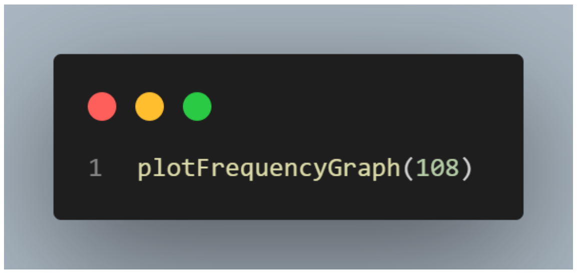


Figure 2a. Program for determing the F(k)

Figure 2b. F(k) corresponding to 20 Hz

To confirm the result, retry the experiment with a sampling frequency of 108Hz. Then try some values below and above the Nyquist frequency (64Hz) and observe the results.



```python
plotFrequencyGraph(108)
```

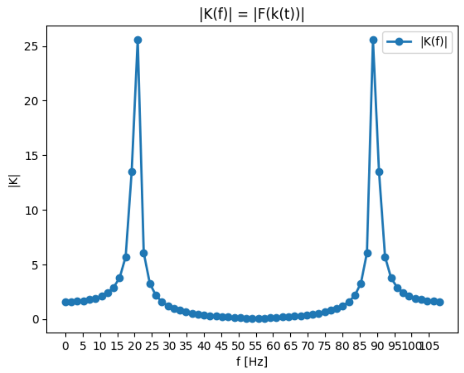
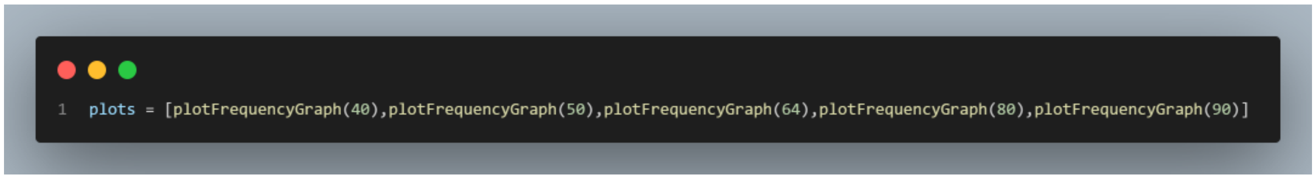


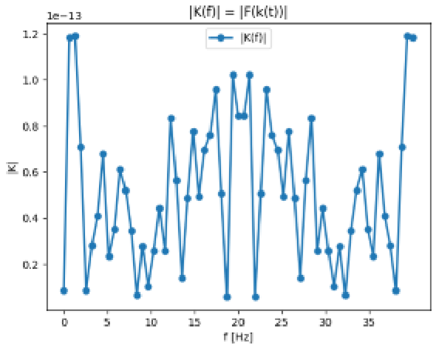
Figure 3a. Function for frequency f =108 Hz

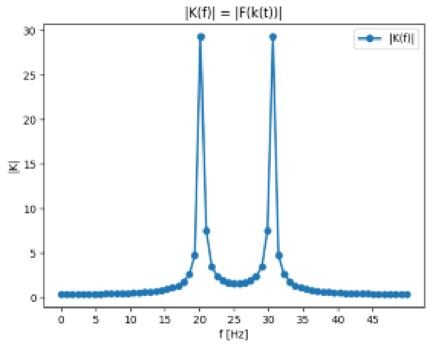
Figure 3b. Graph of F(k) with frequency 108 Hz



```python
plots = [plotFrequencyGraph(40),plotFrequencyGraph(50),plotFrequencyGraph(64),plotFrequencyGraph(80),plotFrequencyGraph(90)]
```

Figure 4a. Generation of plots for frequencies 40, 50, 64, 80, 90 Hz





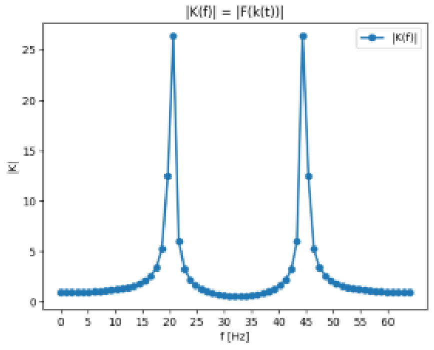
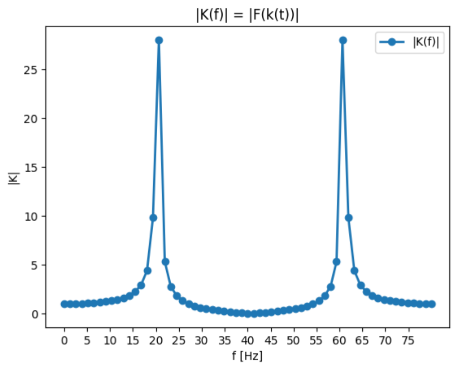


Figure 4b. f = 40 Hz ***Figure 4c. f = 50 Hz*** ***Figure 4d. f = 64 Hz***



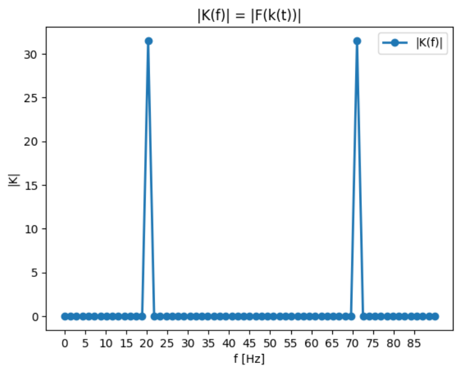
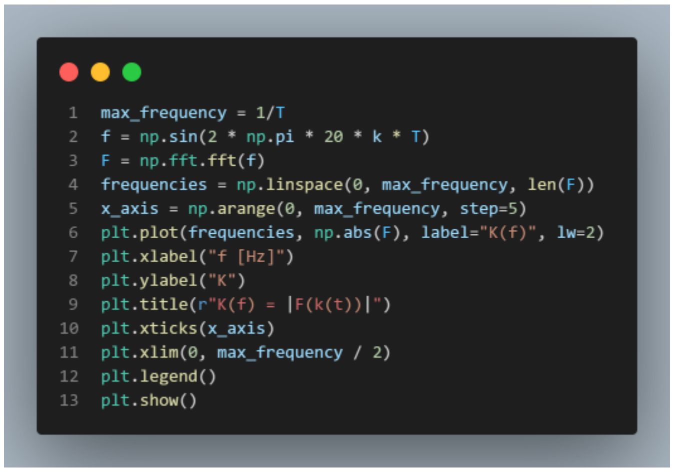


Figure 4e. f = 80 Hz ***Figure 4e. f = 90 Hz***

1.3 It's recommended to graph only the first half and display the frequency in Hz.



```python
max_frequency = 1/T
f = np.sin(2 * np.pi * 20 * k * T)
F = np.fft.fft(f)
frequencies = np.linspace(0, max_frequency, len(F))
x_axis = np.arange(0, max_frequency, step=5)
plt.plot(frequencies, np.abs(F), label="K(f)", lw=2)
plt.xlabel("f [Hz]")
plt.ylabel("K")
plt.title(r"K(f) = |F(k(t))|")
plt.xticks(x_axis)
plt.xlim(0, max_frequency / 2)
plt.legend()
plt.show()
```

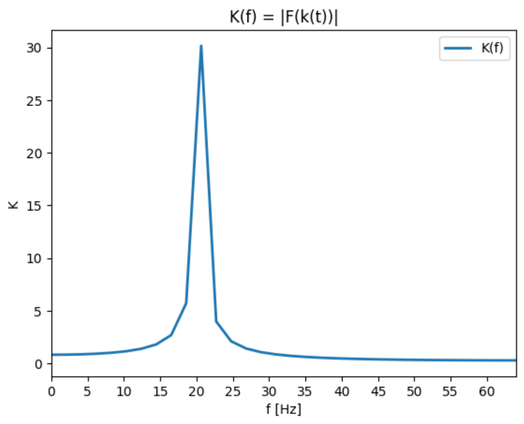
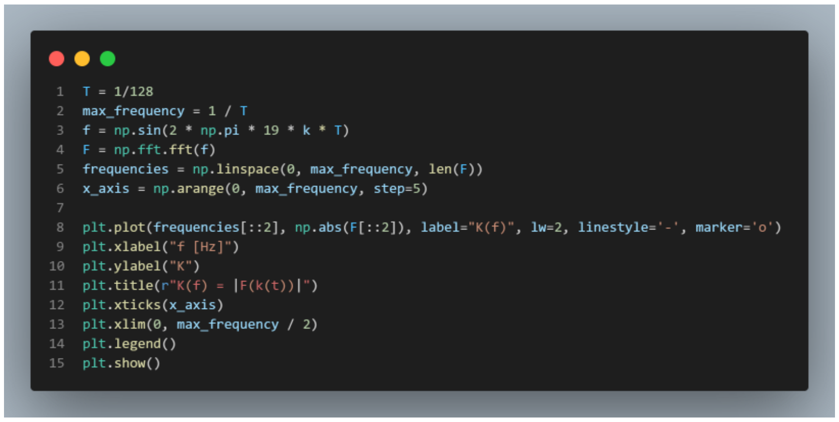


Figure 5a. Program for displaying only the first half ***Figure 5b. Graph of the first half***

1.4 Suppose the frequency of the sinusoid is 19Hz. The step in the graph is 2Hz. This means that the analyzed frequency will be distributed among its neighbors - 18Hz and 20Hz. Verify this.



```python
T = 1/128
max_frequency = 1 / T
f = np.sin(2 * np.pi * 19 * k * T)
F = np.fft.fft(f)
frequencies = np.linspace(0, max_frequency, len(F))
x_axis = np.arange(0, max_frequency, step=5)

plt.plot(frequencies[::2], np.abs(F[::2]), label="K(f)", lw=2, linestyle='-', marker='o')
plt.xlabel("f [Hz]")
plt.ylabel("K")
plt.title(r"K(f) = |F(k(t))|")
plt.xticks(x_axis)
plt.xlim(0, max_frequency / 2)
plt.legend()
plt.show()
```

Figure 6a. Frequency of sinusoid set to 19 Hz

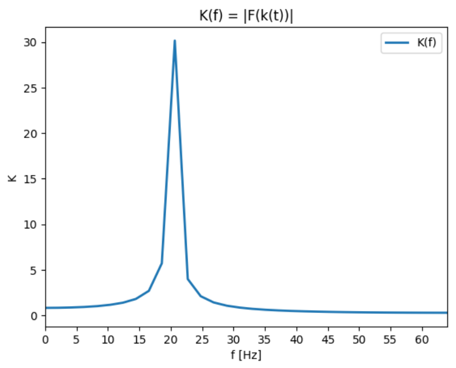
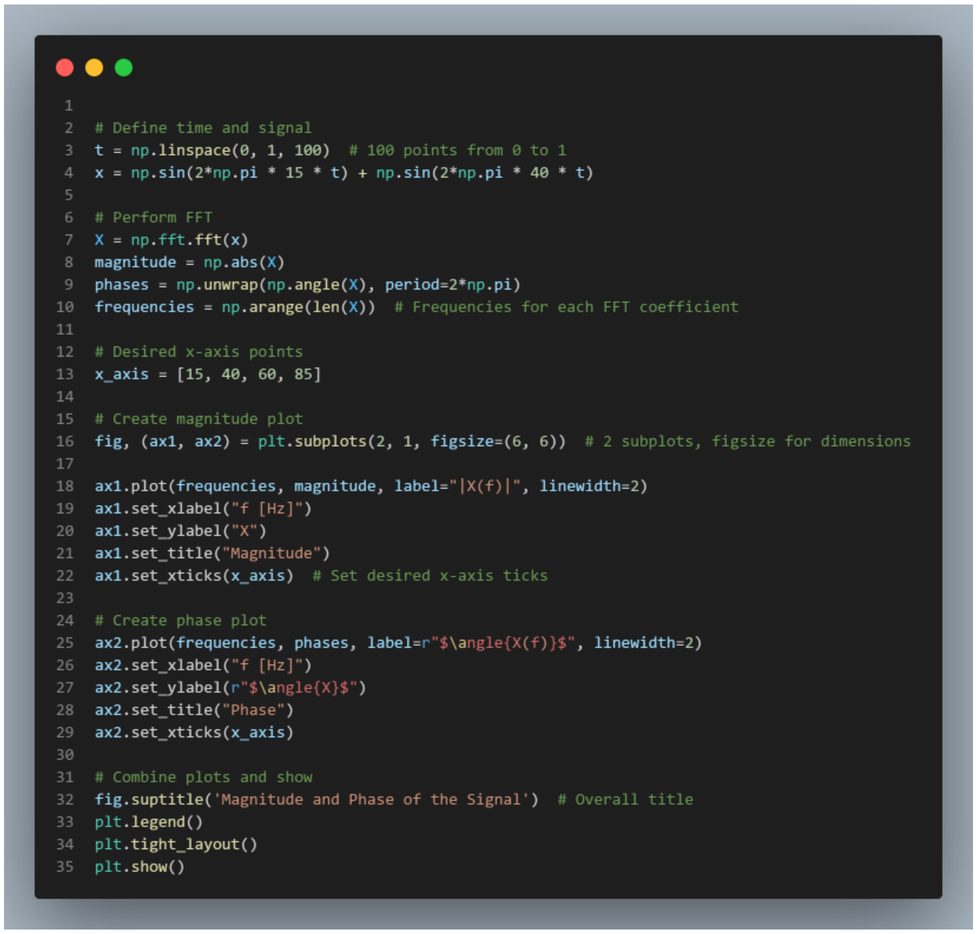


Figure 6b. Graph when f =19 Hz

2.1 Create a signal composed of 2 sinusoids and plot the spectrum of modules and phases.



```python

# Define time and signal
t = np.linspace(0, 1, 100)  # 100 points from 0 to 1
x = np.sin(2*np.pi * 15 * t) + np.sin(2*np.pi * 40 * t)

# Perform FFT
X = np.fft.fft(x)
magnitude = np.abs(X)
phases = np.unwrap(np.angle(X), period=2*np.pi)
frequencies = np.arange(len(X))  # Frequencies for each FFT coefficient

# Desired x-axis points
x_axis = [15, 40, 60, 85]

# Create magnitude plot
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 6))  # 2 subplots, figsize for dimensions

ax1.plot(frequencies, magnitude, label="|X(f)|", linewidth=2)
ax1.set_xlabel("f [Hz]")
ax1.set_ylabel("X")
ax1.set_title("Magnitude")
ax1.set_xticks(x_axis)  # Set desired x-axis ticks

# Create phase plot
ax2.plot(frequencies, phases, label=r"$\angle{X(f)}$", linewidth=2)
ax2.set_xlabel("f [Hz]")
ax2.set_ylabel(r"$\angle{X}$")
ax2.set_title("Phase")
ax2.set_xticks(x_axis)

# Combine plots and show
fig.suptitle('Magnitude and Phase of the Signal')  # Overall title
plt.legend()
plt.tight_layout()
plt.show()
```

Figure 7a. Signal composed of two sinusoids

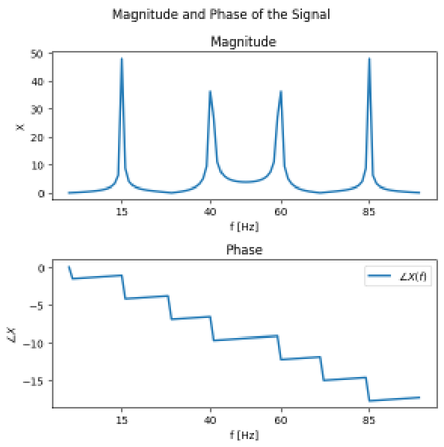
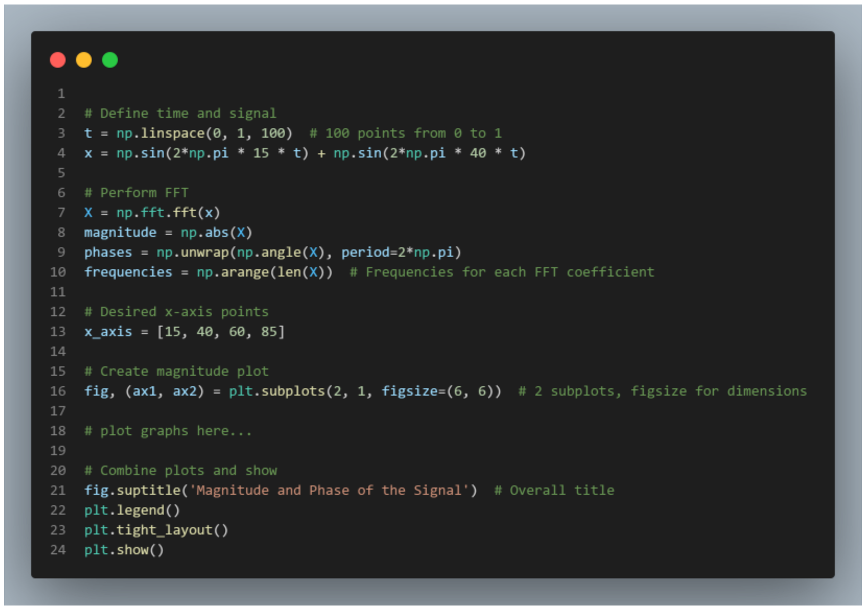


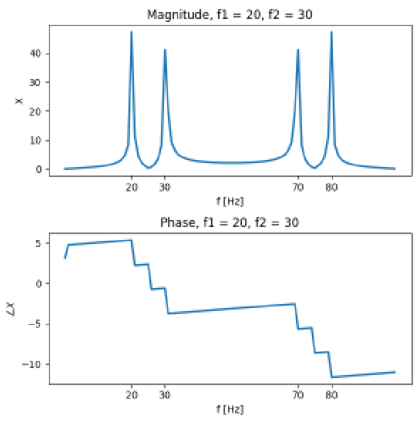
Figure 7b. Magnitude and phase of the sinusoids

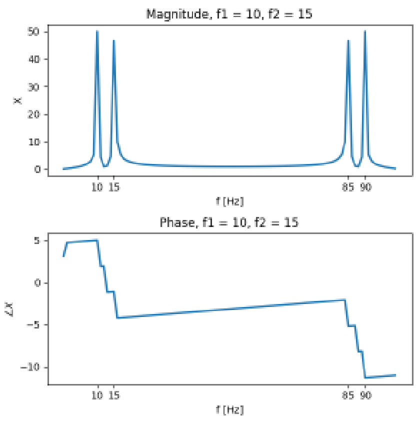
Repeat the procedure with 2-3 other frequencies of the sinusoids and observe the results.



```python

# Define time and signal
t = np.linspace(0, 1, 100)  # 100 points from 0 to 1
x = np.sin(2*np.pi * 15 * t) + np.sin(2*np.pi * 40 * t)

# Perform FFT
X = np.fft.fft(x)
magnitude = np.abs(X)
phases = np.unwrap(np.angle(X), period=2*np.pi)
frequencies = np.arange(len(X))  # Frequencies for each FFT coefficient

# Desired x-axis points
x_axis = [15, 40, 60, 85]

# Create magnitude plot
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 6))  # 2 subplots, figsize for dimensions

# plot graphs here...

# Combine plots and show
fig.suptitle('Magnitude and Phase of the Signal')  # Overall title
plt.legend()
plt.tight_layout()
plt.show()
```

Figure 8a. Magnitude and phase of some other sinusoids





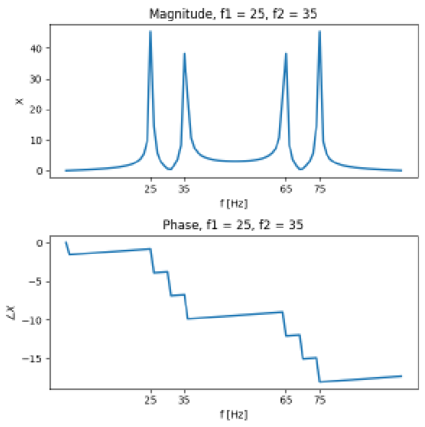
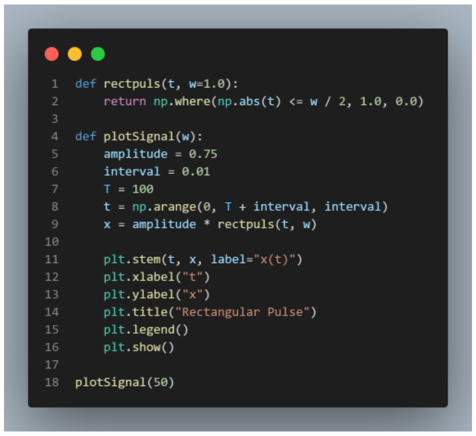


Figure 8b. f1 = 20, f2 = 30   *Figure 8c. f1 = 10, f2 = 15*   *Figure 8d. f1 = 25, f2 = 35*

3.1 Model and plot a rectangular pulse signal.



```python
def rectpuls(t, w=1.0):
    return np.where(np.abs(t) <= w / 2, 1.0, 0.0)

def plotSignal(w):
    amplitude = 0.75
    interval = 0.01
    T = 100
    t = np.arange(0, T + interval, interval)
    x = amplitude * rectpuls(t, w)

    plt.stem(t, x, label="x(t)")
    plt.xlabel("t")
    plt.ylabel("x")
    plt.title("Rectangular Pulse")
    plt.legend()
    plt.show()

plotSignal(50)
```

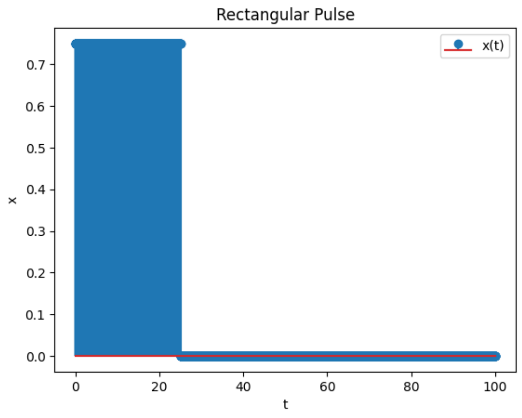
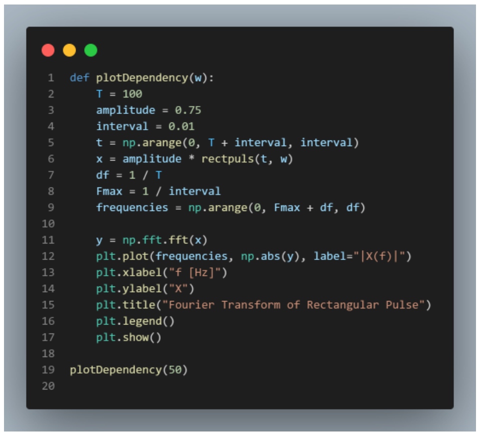


Figure 9a. Generation of rectangular pulse signal ***Figure 9b. Rectangular pulse signal***

3.2 Specify df = 1/T, Fmax = 1/interval and f = 0:df:Fmax. Apply the the Fast Fourier Transform `y = fft(x)` and plot the spectrum of the signal on a stem graph.



```python
def plotDependency(w):
    T = 100
    amplitude = 0.75
    interval = 0.01
    t = np.arange(0, T + interval, interval)
    x = amplitude * rectpuls(t, w)
    df = 1 / T
    Fmax = 1 / interval
    frequencies = np.arange(0, Fmax + df, df)

    y = np.fft.fft(x)
    plt.plot(frequencies, np.abs(y), label="|X(f)|")
    plt.xlabel("f [Hz]")
    plt.ylabel("X")
    plt.title("Fourier Transform of Rectangular Pulse")
    plt.legend()
    plt.show()

plotDependency(50)

```

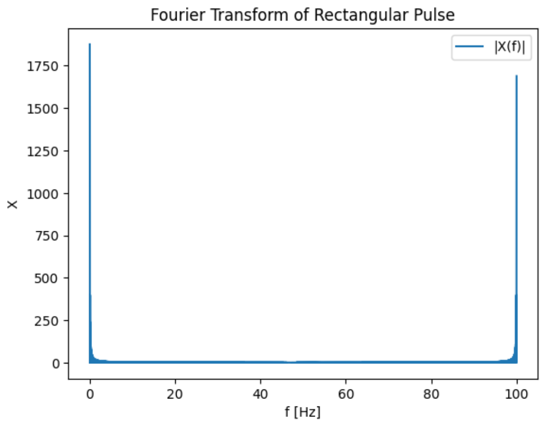
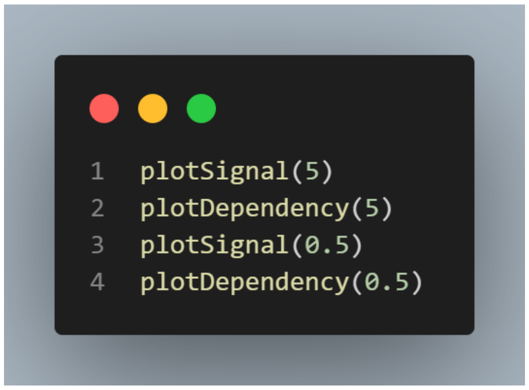


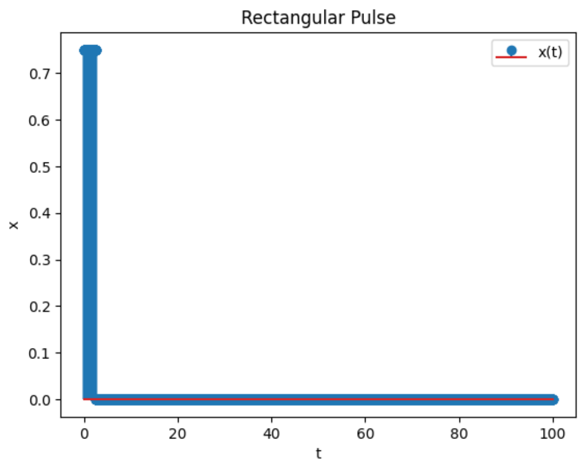
Figure 10a. Generation of rectangular pulse signal ***Figure 10b. Spectrum of the signal***

Repeat the procedure with a rectangular pulse with a duration of 5 and 0.5 seconds and observe the results.



```python
plotSignal(5)
plotDependency(5)
plotSignal(0.5)
plotDependency(0.5)
```

Figure 11a. Generation of two signals with w=5 and w=0.5



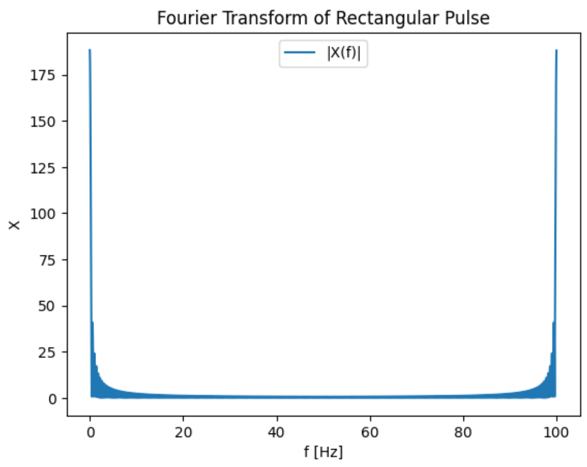
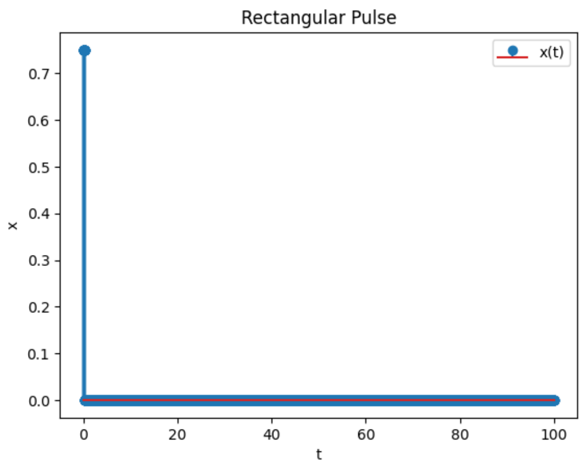


Figure 11b. Rectangulare pulse, w=5 ***Figure11c. Fourier tr. of rect. pulse, w=5***



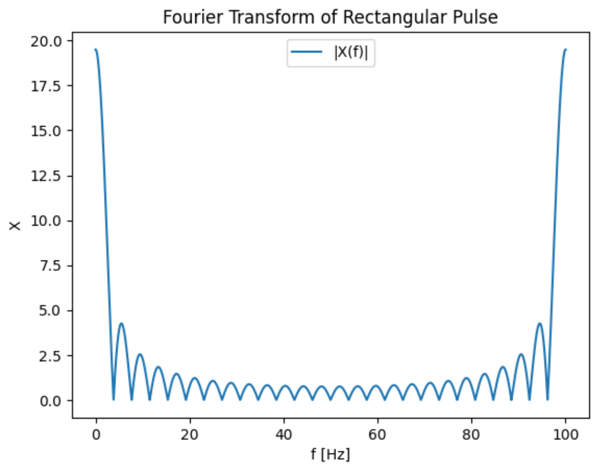
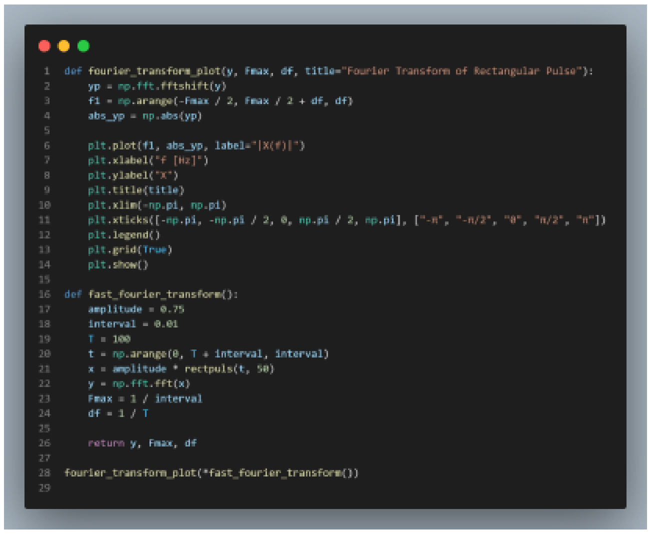


Figure 11d. Rectangulare pulse, w=0.5 ***Figure 11e. Fourier tr. of rect. pulse, w=0.5***

3.3 Apply fftshift to the fft result and show the spectrum of the signal between -π and π.



```
def fourier_transform_plot(y, Fmax, df, title="Fourier Transform of Rectangular Pulse"):
    yp = np.fft.fftshift(y)
    f1 = np.arange(-Fmax / 2, Fmax / 2 + df, df)
    abs_yp = np.abs(yp)

    plt.plot(f1, abs_yp, label="|X(f)|")
    plt.xlabel("f [Hz]")
    plt.ylabel("X")
    plt.title(title)
    plt.xlim(-np.pi, np.pi)
    plt.xticks([-np.pi, -np.pi / 2, 0, np.pi / 2, np.pi], ["-π", "-π/2", "0", "π/2", "π"])
    plt.legend()
    plt.grid(True)
    plt.show()

def fast_fourier_transform():
    amplitude = 0.75
    interval = 0.01
    T = 100
    t = np.arange(0, T + interval, interval)
    x = amplitude * rectpuls(t, 50)
    y = np.fft.fft(x)
    Fmax = 1 / interval
    df = 1 / T

    return y, Fmax, df

fourier_transform_plot(*fast_fourier_transform())
```

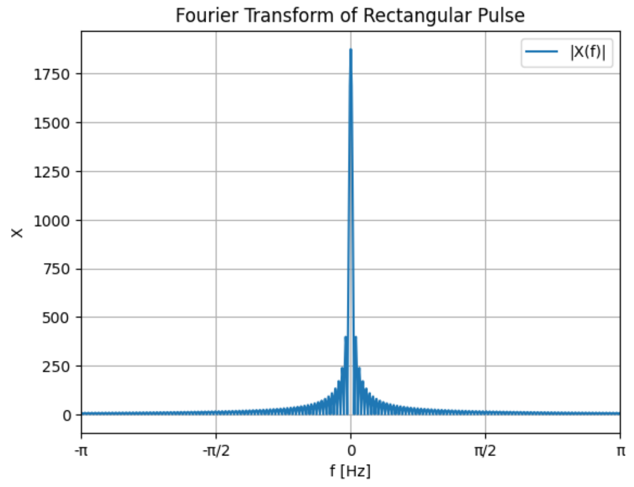
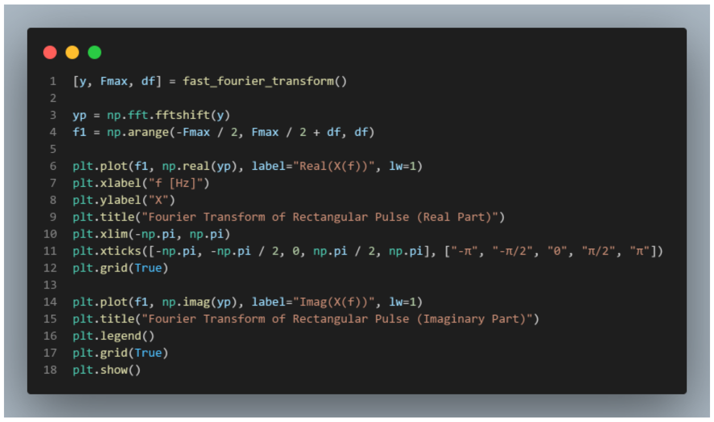


Figure 12a. FFT with fftshift ***Figure 12b. FFT of rectangular pulse***

3.4 Display the real and imaginary parts of the signal on a single graph.



```python
[y, Fmax, df] = fast_fourier_transform()

yp = np.fft.fftshift(y)
f1 = np.arange(-Fmax / 2, Fmax / 2 + df, df)

plt.plot(f1, np.real(yp), label="Real(X(f))", lw=1)
plt.xlabel("f [Hz]")
plt.ylabel("X")
plt.title("Fourier Transform of Rectangular Pulse (Real Part)")
plt.xlim(-np.pi, np.pi)
plt.xticks([-np.pi, -np.pi / 2, 0, np.pi / 2, np.pi], ["-π", "-π/2", "0", "π/2", "π"])
plt.grid(True)

plt.plot(f1, np.imag(yp), label="Imag(X(f))", lw=1)
plt.title("Fourier Transform of Rectangular Pulse (Imaginary Part)")
plt.legend()
plt.grid(True)
plt.show()
```

Figure 13a. Code for graphs real, imag. part

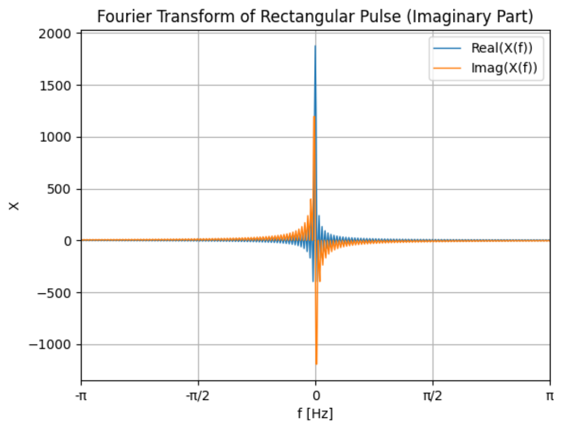
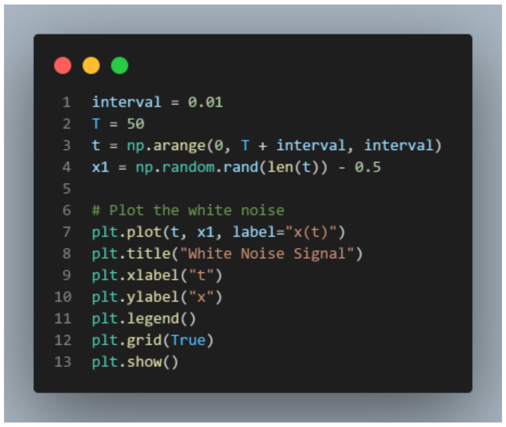


Figure 13b. Real, imag. parts Fourier tr.

4.1 Generate white noise and plot it.



```python
interval = 0.01
T = 50
t = np.arange(0, T + interval, interval)
x1 = np.random.rand(len(t)) - 0.5

# Plot the white noise
plt.plot(t, x1, label="x(t)")
plt.title("White Noise Signal")
plt.xlabel("t")
plt.ylabel("x")
plt.legend()
plt.grid(True)
plt.show()
```

Figure 14a. Generation of white noise

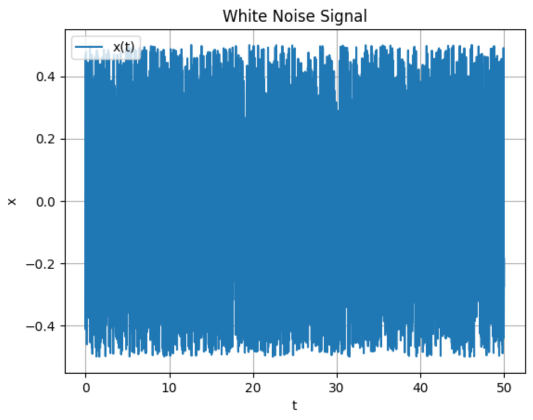
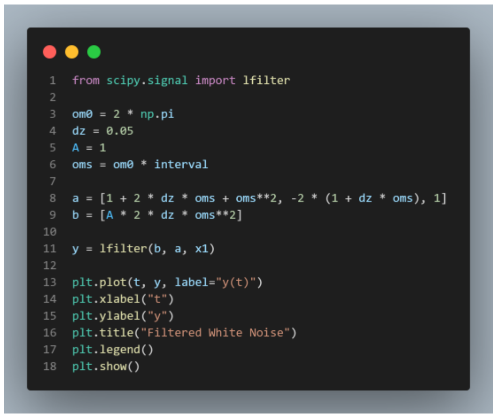


Figure 14b. White noise signal graph

4.2 Project a filter and plot the result.



```python
from scipy.signal import lfilter

om0 = 2 * np.pi
dz = 0.05
A = 1
oms = om0 * interval

a = [1 + 2 * dz * oms + oms**2, -2 * (1 + dz * oms), 1]
b = [A * 2 * dz * oms**2]

y = lfilter(b, a, x1)

plt.plot(t, y, label="y(t)")
plt.xlabel("t")
plt.ylabel("y")
plt.title("Filtered White Noise")
plt.legend()
plt.show()
```

Figure 15a. Adding a filter

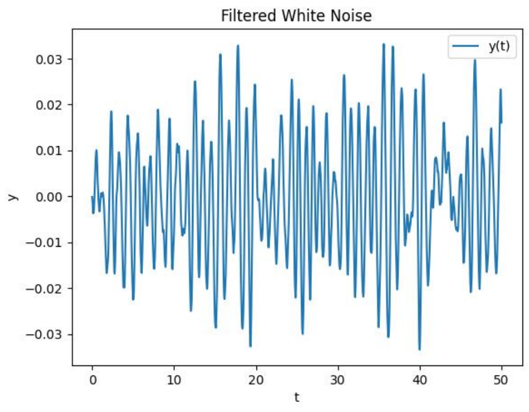
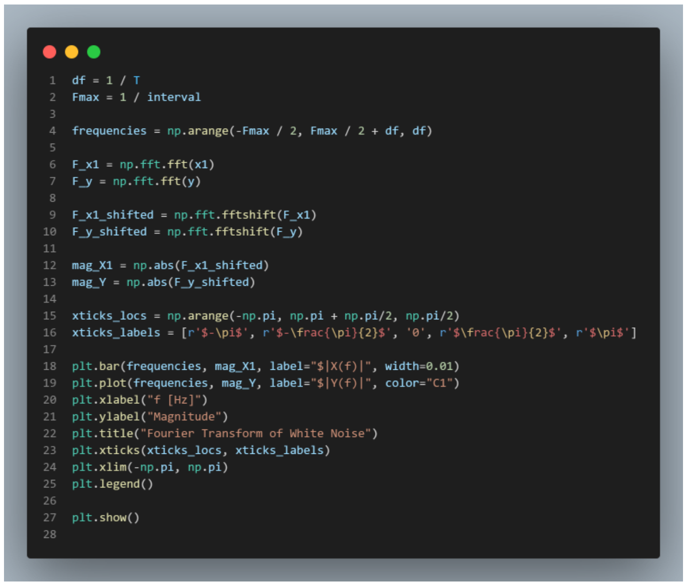


Figure 15b. White noise with filter

4.3 Display the FFT of the white noise and the filtered signal on the same graph.



```python
df = 1 / T
Fmax = 1 / interval

frequencies = np.arange(-Fmax / 2, Fmax / 2 + df, df)

F_x1 = np.fft.fft(x1)
F_y = np.fft.fft(y)

F_x1_shifted = np.fft.fftshift(F_x1)
F_y_shifted = np.fft.fftshift(F_y)

mag_X1 = np.abs(F_x1_shifted)
mag_Y = np.abs(F_y_shifted)

xticks_locs = np.arange(-np.pi, np.pi + np.pi/2, np.pi/2)
xticks_labels = [r'$-\pi$', r'$-\frac{\pi}{2}$', '0', r'$\frac{\pi}{2}$', r'$\pi$']

plt.bar(frequencies, mag_X1, label="$|X(f)|", width=0.01)
plt.plot(frequencies, mag_Y, label="$|Y(f)|", color="C1")
plt.xlabel("f [Hz]")
plt.ylabel("Magnitude")
plt.title("Fourier Transform of White Noise")
plt.xticks(xticks_locs, xticks_labels)
plt.xlim(-np.pi, np.pi)
plt.legend()

plt.show()

```

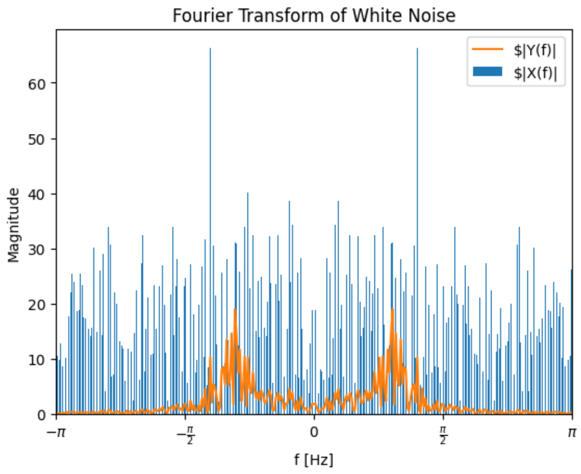


Figure 16a. Generation of two signals　　*Figure 16b. FFT white noise; filtered signal*

CONTROL QUESTIONS

1. The Fourier series definition for continuous periodic signals states that any periodic function $f(t)$ with period T can be represented as a sum of sinusoidal functions (sine and cosine) of integer multiples of the fundamental frequency $\omega_0=2\pi/T$
2. The Fourier transform definition for continuous aperiodic signals states that any aperiodic function $f(t)$ can be decomposed into a continuous spectrum of sinusoidal functions across all frequencies, represented by the integral of the function multiplied by complex exponential terms.
3. The Dirichlet conditions are a set of criteria that must be met by a periodic function in order for its Fourier series representation to converge and accurately represent the function. These conditions include the function being absolutely integrable over one period, having a finite number of discontinuities in each period, and having a finite number of maxima and minima in each period.
4. The energy/power spectral density represents the distribution of signal energy/power across different frequencies. It provides information about how much energy or power the signal contains at each frequency component.
5. The energy spectral density of a rectangular pulse represents the distribution of signal energy across different frequencies for a rectangular pulse signal. It shows the contribution of each

frequency component to the total energy of the signal.

6. The shape of the energy spectral density of a periodic discrete signal will be a series of impulses located at discrete frequencies. The height of each impulse represents the energy content at that particular frequency component, and the spacing between impulses corresponds to the fundamental frequency and its harmonics.

CONCLUSION

In conclusion, this lab work has provided me with a comprehensive exploration of Fourier analysis and signal processing techniques. Through practical tasks, I gained hands-on experience in generating discrete signals, performing Fourier transforms, analyzing frequency spectra, and understanding the principles of signal discretization and sampling.

I learned how to apply the Discrete Fourier Transform (DFT) algorithm using MATLAB's fft function, and I observed how frequency domain representations reveal the spectral characteristics of various signals. The concepts of aliasing, Nyquist frequency, and the importance of sampling rate were elucidated, highlighting the critical considerations in digital signal processing.

Moreover, by examining the Fourier series and Fourier transform definitions for both continuous periodic and aperiodic signals, I grasped the fundamental principles underlying Fourier analysis. The Dirichlet conditions were identified as essential criteria for ensuring the convergence and accuracy of Fourier series representations.

Through tasks involving rectangular pulses and white noise signals, I observed the effects of filtering and spectral analysis on different types of signals. This practical experience enhanced my understanding of how Fourier analysis can be applied to analyze and manipulate real-world signals affected by noise and other disturbances.

Overall, this lab work deepened my knowledge of Fourier analysis and its applications in signal processing, laying a solid foundation for further exploration and practical utilization in various engineering and scientific fields.