**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**

A logo with text and symbols

AI-generated content may be incorrect.

**Faculty of Engineering & Technology**

**Department of Information and Communication Engineering**

LAB REPORT

**Course name: Data Structure and Algorithm Sessional**

**Course Code: ICE-2202**

Submitted By: Submitted To:

**MD. Junaid** Dr. Md. Imran Hossain

Roll: 220618 Associate Professor

Session: 2021-22 Department of ICE

2nd Year 2nd Semester Pabna University of Science and

Department of ICE, PUST Technology, Pabna.

Date of Submission: 03-03-2025

**Index**

|  |  |
| --- | --- |
| No | Experiment Name |
| 01 | System Sequence: Analysis of Impulse Signal, Step Signal, and Ramp Signal |
| 02 | Signal Operations: Addition, Multiplication, Scaling, Shifting, and Folding |
| 03 | Correlation: Analysis of Autocorrelation and Cross-Correlation of Signals |
| 04 | Convolution: Analysis of Autoconvolution and Convolution of Signals |
| 05 | Comprehensive PPG Signal Analysis for Heart Rate and Anomaly Detection |
| 06 | Implementation of Discrete Fourier Transform (DFT) in Python |
| 07 | Removing Noise from an Audio Signal using FFT and Inverse FFT |

**Experiment No:01**

**Title:**

System Sequence: Analysis of Impulse Signal, Step Signal, and Ramp Signal

**Objective:**  
The objective of this experiment is to analyze and understand the characteristics of fundamental discrete-time signals, namely the impulse signal, step signal, and ramp signal. These signals are widely used in signal processing and system analysis.

**Theory:**  
Fundamental signals play a crucial role in signal processing and control systems. They are often used as test signals to analyze system responses. The three basic signals examined in this experiment are:

1. Impulse Signal (): A discrete signal with a value of 1 at and 0 elsewhere. It is used to analyze system responses in convolution and system identification.
2. Step Signal (): A discrete signal with a value of 0 for and 1 for . It is used to study system stability and transient response.
3. Ramp Signal (): A discrete signal that increases linearly with time, defined as for , and 0 for . It is used to analyze system growth and acceleration behavior.

Mathematical Definitions:

* Impulse Signal:
* Step Signal:
* Ramp Signal:

**Source Code**

|  |
| --- |
| **import numpy as np**  **import matplotlib.pyplot as plt**  **# Define the range**  **n = np.arange(-10, 11)**  **# Define functions for signals**  **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, 9))**  **# Impulse Signal**  **plt.subplot(3, 1, 1)**  **plt.stem(n, impulse)**  **plt.title("Impulse Signal")**  **plt.xlabel("n")**  **plt.ylabel("Amplitude")**  **plt.grid()**  **# Step Signal**  **plt.subplot(3, 1, 2)**  **plt.stem(n, step)**  **plt.title("Step Signal")**  **plt.xlabel("n")**  **plt.ylabel("Amplitude")**  **plt.grid()**  **# Ramp Signal**  **plt.subplot(3, 1, 3)**  **plt.stem(n, ramp)**  **plt.title("Ramp Signal")**  **plt.xlabel("n")**  **plt.ylabel("Amplitude")**  **plt.grid()**  **plt.tight\_layout()**  **plt.show()** |

**Output**

|  |
| --- |
| **A graph of a function  AI-generated content may be incorrect.** |

**Experiment No: 02**

**Title:**

Signal Operations: Addition, Multiplication, Scaling, Shifting, and Folding

**Objective:**

To understand and implement the fundamental operations of signal processing, including addition, multiplication, scaling, shifting, and folding, and to analyze their effects on discrete-time signals.

**Theory:**  
Signal processing involves manipulating signals to extract information, enhance features, or analyze behavior. Basic operations like addition, multiplication, scaling, and shifting are fundamental in understanding signal behavior in both time and frequency domains.  
  
1. **Signal Addition**: Combining two or more signals, often used in overlaying or superimposing information.  
 y(t) = x1(t) + x2(t)  
  
2. **Signal Multiplication:** Multiplying two signals results in a combined signal with modulated characteristics, often used in amplitude modulation.  
 y(t) = x1(t) \* x2(t)  
  
3**. Scaling:** Modifying the amplitude or duration of a signal.  
 - Amplitude Scaling: Changes the signal's magnitude by a constant.  
 - Time Scaling: Compresses or expands the signal along the time axis.  
 y(t) = k \* x(at)  
  
4. **Shifting:** In this operation, each sample of x(n) is shifted by an amount k to obtain a shifted sequence y(n).

y(n)={x(n−k)}

If we let m = n−k, then n = m+k and the above operation is given by

y(m+k) = {x(m)}  
These operations help in signal transformation, modulation, and system analysis.

5. **Folding:** In this operation each sample of x(n)is flipped around n =0 to obtain a folded sequence y(n).

y(n)={x(−n)}

**Source Code**

|  |
| --- |
| **import numpy as np**  **import matplotlib.pyplot as plt**  **# Parameters**  **t = np.arange(-10, 10, 0.01)  # Time vector**  **# Define two signals**  **x1 = np.sin(2 \* np.pi \* 1 \* t)  # Signal 1: Sine wave**  **x2 = np.cos(2 \* np.pi \* 0.5 \* t)  # Signal 2: Cosine wave**  **# Signal Addition**  **y\_add = x1 + x2**  **# Signal Multiplication**  **y\_mult = x1 \* x2**  **# Amplitude Scaling**  **k = 2  # Scaling factor**  **y\_scaled = k \* x1**  **# Time Shifting**  **shift = 2  # Shift value (in seconds)**  **y\_shifted = np.sin(2 \* np.pi \* 1 \* (t - shift))  # Delayed signal**  **# Signal Folding**  **y\_folded = np.sin(2 \* np.pi \* 1 \* (-t))  # Folded sine wave**  **# Plot Results**  **plt.figure(figsize=(12, 15))**  **# Original Signals**  **plt.subplot(6, 1, 1)**  **plt.plot(t, x1, label='x1: Sine Wave', color='b')**  **plt.plot(t, x2, label='x2: Cosine Wave', color='r')**  **plt.title('Original Signals')**  **plt.legend()**  **plt.grid()**  **# Signal Addition**  **plt.subplot(6, 1, 2)**  **plt.plot(t, y\_add, color='m')**  **plt.title('Signal Addition')**  **plt.grid()**  **# Signal Multiplication**  **plt.subplot(6, 1, 3)**  **plt.plot(t, y\_mult, color='k')**  **plt.title('Signal Multiplication')**  **plt.grid()**  **# Amplitude Scaling**  **plt.subplot(6, 1, 4)**  **plt.plot(t, y\_scaled, color='g')**  **plt.title('Amplitude Scaling')**  **plt.grid()**  **# Time Shifting**  **plt.subplot(6, 1, 5)**  **plt.plot(t, y\_shifted, color='c')**  **plt.title('Time Shifting')**  **plt.grid()**  **# Signal Folding**  **plt.subplot(6, 1, 6)**  **plt.plot(t, y\_folded, label='Folded Signal', color='purple')**  **plt.title('Signal Folding')**  **plt.legend()**  **plt.grid()**  **# Display the plots**  **plt.tight\_layout()**  **plt.show()** |

**Output**

|  |
| --- |
| A graph of a graph of a wave |

**Experiment No: 03**

**Title**

**Correlation: Analysis of Autocorrelation and Cross-Correlation of Signals**

**Objective:**  
The objective of this experiment is to analyze the correlation properties of signals by computing and interpreting their autocorrelation and cross-correlation functions. This helps in understanding how a signal is related to itself over time and how two different signals are related.

**Theory:**  
Correlation is a fundamental signal processing technique used to determine the similarity between signals. It is widely applied in communications, radar, image processing, and pattern recognition. The two main types of correlation are:

1. Autocorrelation: Measures the similarity of a signal with a delayed version of itself. It is useful in identifying periodicity and detecting repeating patterns in a signal.
2. Cross-Correlation: Measures the similarity between two different signals as a function of time delay. It helps in detecting the presence of a known signal in another signal.

Mathematically, the autocorrelation of a discrete signal is given by:

where k is the time shift.

The cross-correlation between two discrete signals and is given by:

where k represents the time lag.

**Source 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() |

**Output**

|  |
| --- |
| **A screenshot of a graph  AI-generated content may be incorrect.** |

**Experiment No: 04**

**Title:**

Convolution: Analysis of Autoconvolution and Convolution of Signals

**Objective:**The objective of this experiment is to analyze the properties of convolution, including autoconvolution and convolution of signals, and understand their significance in signal processing applications.

**Theory:**Convolution is a fundamental mathematical operation in signal processing used to determine the response of a system to an input signal. It is widely applied in filtering, image processing, and system analysis. The two main types of convolution are:

1. Autoconvolution: The convolution of a signal with itself, which helps in enhancing periodic patterns and extracting key signal features.
2. Convolution: The convolution of two different signals, which is commonly used to determine how an input signal is modified by a system or filter.

Mathematically, the convolution of two discrete signals x[n] and h[n] is given by:

k is the summation index representing time shifts.

For autoconvolution, x[n]is convolved with itself:

**Source 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() |

**Output**

|  |
| --- |
| A graph of a function  AI-generated content may be incorrect. |

**Experiment No: 5**

**Title**

Comprehensive PPG Signal Analysis for Heart Rate and Anomaly Detection

**Theory**

Photoplethysmography (PPG) is a non-invasive optical technique used to detect blood volume changes in the microvascular tissue. It is widely used for monitoring heart rate and detecting cardiovascular abnormalities. In this experiment, we implement and analyze a PPG signal processing pipeline, which includes data loading, filtering, R-peak detection, heart rate computation, and abnormality detection.

**Objectives**

* Load and preprocess a PPG signal.
* Apply a bandpass filter to remove noise and artifacts.
* Detect R-peaks in the signal.
* Calculate RR intervals and heart rate.
* Identify abnormalities such as bradycardia, tachycardia, and irregular rhythms.
* Visualize all processing steps for better interpretation.

**Data Acquisition**

The PPG signal is loaded from a CSV file, which contains amplitude values sampled at a predefined sampling rate.

**Signal Preprocessing**

A bandpass filter (0.5–8 Hz) is applied using a Butterworth filter to remove high-frequency noise and low-frequency baseline wander.

**R-Peak Detection**

R-peaks, representing significant pulse events, are detected using the find\_peaks function from SciPy.

**Heart Rate and RR Interval Analysis**

The RR intervals are calculated as the time difference between consecutive R-peaks. The heart rate is computed as: where RR intervals are measured in seconds.

**Abnormality Detection**

Abnormalities are classified into three categories:

* **Bradycardia:** Heart rate < 60 BPM
* **Tachycardia:** Heart rate > 100 BPM
* **Irregular Rhythm:** Significant deviation from the mean RR interval

**Source Code**

|  |
| --- |
| import numpy as np  from scipy.signal import butter, filtfilt, find\_peaks  import matplotlib.pyplot as plt  import pandas as pd  class PPGAnalyzer:      def \_\_init\_\_(self, sampling\_rate=100):          self.fs = sampling\_rate          self.ppg\_data = None          self.filtered\_data = None          self.r\_peaks = None          self.rr\_intervals = None          self.heart\_rates = None        def load\_data(self, file\_path):          try:              self.ppg\_data = pd.read\_csv(file\_path).iloc[:, 0].values              return True          except Exception as e:              print(f"Error loading data: {e}")              return False        def bandpass\_filter(self, lowcut=0.5, highcut=8.0):          nyquist = 0.5 \* self.fs          low = lowcut / nyquist          high = highcut / nyquist          order = 2            b, a = butter(order, [low, high], btype='band')          self.filtered\_data = filtfilt(b, a, self.ppg\_data)        def detect\_r\_peaks(self, height=None, distance=None):          if height is None:              height = 0.6 \* np.max(self.filtered\_data)          if distance is None:              distance = int(0.5 \* self.fs)            self.r\_peaks, \_ = find\_peaks(self.filtered\_data, height=height, distance=distance)        def analyze\_heart\_rate(self):          if self.r\_peaks is None:              print("Please detect R-peaks first")              return            self.rr\_intervals = np.diff(self.r\_peaks) / self.fs          self.heart\_rates = 60 / self.rr\_intervals        def detect\_abnormalities(self):          if self.rr\_intervals is None:              print("Please analyze heart rate first")              return {}            abnormalities = {'bradycardia': [], 'tachycardia': [], 'irregular': []}            bradycardia\_idx = np.where(self.heart\_rates < 60)[0]          abnormalities['bradycardia'] = self.r\_peaks[bradycardia\_idx]            tachycardia\_idx = np.where(self.heart\_rates > 100)[0]          abnormalities['tachycardia'] = self.r\_peaks[tachycardia\_idx]            rr\_std = np.std(self.rr\_intervals)          rr\_mean = np.mean(self.rr\_intervals)          irregular\_idx = np.where(np.abs(self.rr\_intervals - rr\_mean) > 2 \* rr\_std)[0]          abnormalities['irregular'] = self.r\_peaks[irregular\_idx]            return abnormalities      def visualize\_all\_steps(self, abnormalities=None, output\_file='ppg\_analysis.png'):          """          Create separate visualizations for each processing step and save the output as a PNG file          """          time = np.arange(len(self.ppg\_data)) / self.fs            # Create a figure with 7 subplots          fig = plt.figure(figsize=(15, 22))            # 1. Raw Signal          ax1 = fig.add\_subplot(711)          ax1.plot(time, self.ppg\_data)          ax1.set\_title('1. Raw PPG Signal')          ax1.set\_xlabel('Time (s)')          ax1.set\_ylabel('Amplitude')            # 2. Filtered Signal          ax2 = fig.add\_subplot(712)          ax2.plot(time, self.filtered\_data)          ax2.set\_title('2. Bandpass Filtered Signal (0.5-8 Hz)')          ax2.set\_xlabel('Time (s)')          ax2.set\_ylabel('Amplitude')            # 3. Raw vs. Filtered Signal Comparison          ax3 = fig.add\_subplot(713)          ax3.plot(time, self.ppg\_data, label='Raw PPG Signal', alpha=0.7)          ax3.plot(time, self.filtered\_data, label='Filtered PPG Signal', linewidth=1.2)          ax3.set\_title('3. Raw vs. Filtered PPG Signal')          ax3.set\_xlabel('Time (s)')          ax3.set\_ylabel('Amplitude')          ax3.legend()            # 4. R-Peak Detection          ax4 = fig.add\_subplot(714)          ax4.plot(time, self.filtered\_data)          ax4.plot(time[self.r\_peaks], self.filtered\_data[self.r\_peaks], 'rx', label='R-peaks')          ax4.set\_title('4. R-Peak Detection')          ax4.set\_xlabel('Time (s)')          ax4.set\_ylabel('Amplitude')          ax4.legend()            # 5. RR Intervals          ax5 = fig.add\_subplot(715)          if self.rr\_intervals is not None:              rr\_time = time[self.r\_peaks[:-1]]              ax5.plot(rr\_time, self.rr\_intervals)              ax5.set\_title('5. RR Intervals')              ax5.set\_xlabel('Time (s)')              ax5.set\_ylabel('Interval (s)')            # 6. Heart Rate Trend          ax6 = fig.add\_subplot(716)          if self.heart\_rates is not None:              hr\_time = time[self.r\_peaks[1:]]              ax6.plot(hr\_time, self.heart\_rates)              ax6.set\_title('6. Heart Rate Trend')              ax6.set\_xlabel('Time (s)')              ax6.set\_ylabel('Heart Rate (BPM)')              ax6.axhline(y=60, color='r', linestyle='--', alpha=0.5, label='Bradycardia threshold')              ax6.axhline(y=100, color='r', linestyle='--', alpha=0.5, label='Tachycardia threshold')              ax6.legend()            # 7. Abnormalities          ax7 = fig.add\_subplot(717)          ax7.plot(time, self.filtered\_data, label='Filtered Signal')          if abnormalities:              colors = {'bradycardia': 'blue', 'tachycardia': 'red', 'irregular': 'green'}              for abnorm\_type, peaks in abnormalities.items():                  if len(peaks) > 0:                      ax7.plot(time[peaks], self.filtered\_data[peaks], 'o', label=abnorm\_type, color=colors[abnorm\_type])          ax7.set\_title('7. Detected Abnormalities')          ax7.set\_xlabel('Time (s)')          ax7.set\_ylabel('Amplitude')          ax7.legend()            # Adjust spacing instead of using tight\_layout          plt.subplots\_adjust(hspace=0.5)  # Adjusts vertical spacing            # Save the figure as a PNG file          plt.savefig(output\_file, dpi=300)          plt.close()            # Print summary statistics          print("\nAnalysis Summary:")          print(f"Average Heart Rate: {np.mean(self.heart\_rates):.1f} BPM")          print(f"Heart Rate Variability: {np.std(self.heart\_rates):.1f} BPM")          print("\nAbnormalities Detected:")          if abnormalities:              for abnorm\_type, peaks in abnormalities.items():                  print(f"{abnorm\_type}: {len(peaks)} instances")      def analyze\_ppg\_file(file\_path, sampling\_rate=100, output\_file='ppg\_analysis.png'):          """          Analyze a PPG signal file and display results with detailed visualizations          """          analyzer = PPGAnalyzer(sampling\_rate)            if analyzer.load\_data(file\_path):              analyzer.bandpass\_filter()              analyzer.detect\_r\_peaks()              analyzer.analyze\_heart\_rate()                abnormalities = analyzer.detect\_abnormalities()                analyzer.visualize\_all\_steps(abnormalities, output\_file) |

Output

|  |
| --- |
| A screenshot of a graph  AI-generated content may be incorrect. |

***Analysis Summary:***

***Average Heart Rate: 55.5 BPM***

***Heart Rate Variability: 25.6 BPM***

***Abnormalities Detected:***

***bradycardia: 10 instances***

***tachycardia: 1 instances***

***irregular: 1 instances***

**Experiment No: 06**

**Title:**

Implementation of Discrete Fourier Transform (DFT) in Python

**Objective:**  
To implement and understand the Discrete Fourier Transform (DFT) and analyze its frequency spectrum through a combination of two sine waves. The DFT allows us to convert a time-domain signal into its frequency-domain representation.

**Theory:**  
The Discrete Fourier Transform (DFT) is a mathematical technique used to convert a discrete time-domain signal into a frequency-domain representation. The DFT of a sequence x[n] is given by:

, k = 0,1,2,….,N-1

Where:

* x[n] is the input signal (time-domain),
* X[k] is the output signal (frequency-domain),
* N is the number of samples in the signal,
* k represents the frequency bins.

The output X[k] is a complex number, representing both the magnitude and phase of the frequency component at the k-th frequency bin. The magnitude of X[k] is used to plot the frequency spectrum.

**Procedure:**

1. **Signal Creation:**
   * A combination of two sine waves with frequencies 50 Hz and 120 Hz is generated.
   * The sampling rate is set to 1000 Hz, meaning that the signal is sampled 1000 times per second.
   * The signal is a sum of two sine waves, one with frequency 50 Hz and another with frequency 120 Hz.
2. **DFT Computation:**
   * The DFT function is implemented in Python. It calculates the Fourier transform of the signal by iterating over each frequency bin and summing up the contributions from each time sample, using the DFT formula.
3. **Frequency Bins:**
   * The frequency bins are computed using the numpy function np.fft.fftfreq, which generates a frequency axis corresponding to the DFT output.
4. **Plotting the Frequency Spectrum:**
   * The magnitude spectrum (real part) of the DFT output is plotted against the frequency bins.
   * A single-sided spectrum is shown by plotting only the first half of the DFT output, which contains the positive frequencies.

**Source Code:**

|  |
| --- |
| **import numpy as np**  **import matplotlib.pyplot as plt**  **def DFT(x):**  **"""**  **Compute the Discrete Fourier Transform (DFT) of a 1D signal.**  **"""**  **N = len(x)**  **X = np.zeros(N, dtype=complex) # Output array (complex numbers)**  **for k in range(N): # Loop over frequency bins**  **for n in range(N): # Loop over time samples**  **X[k] += x[n] \* np.exp(-2j \* np.pi \* k \* n / N)**    **return X**  **# Create a sample signal (two sine waves)**  **Fs = 1000 # Sampling rate**  **T = 1 / Fs # Sampling interval**  **t = np.linspace(0, 1, Fs, endpoint=False) # 1 second duration**  **# Signal: Combination of 50 Hz and 120 Hz sine waves**  **f1, f2 = 50, 120**  **signal = np.sin(2 \* np.pi \* f1 \* t) + 0.5 \* np.sin(2 \* np.pi \* f2 \* t)**  **# Compute DFT**  **dft\_output = DFT(signal)**  **# Compute frequency bins**  **freqs = np.fft.fftfreq(len(dft\_output), T)**  **# Plot magnitude spectrum (single-sided)**  **plt.figure(figsize=(10, 5))**  **plt.plot(freqs[:Fs//2], np.abs(dft\_output[:Fs//2])) # Single-sided spectrum**  **plt.title("DFT Frequency Spectrum")**  **plt.xlabel("Frequency (Hz)")**  **plt.ylabel("Magnitude")**  **plt.grid()**  **plt.show()** |

**Output**

|  |
| --- |
| **A graph with a line graph  AI-generated content may be incorrect.** |

**Experiment No: 07**

**Title:**

Removing Noise from an Audio Signal using FFT and Inverse FFT

**Objective:**  
To demonstrate the process of removing noise from a noisy audio signal by applying the Fast Fourier Transform (FFT), filtering out high-frequency noise, and then using the Inverse FFT to reconstruct the cleaned signal.

**Theory:**  
The Fast Fourier Transform (FFT) is an efficient algorithm to compute the Discrete Fourier Transform (DFT), which transforms a time-domain signal into its frequency-domain representation. Once in the frequency domain, it is easier to identify and filter out unwanted noise, particularly high-frequency noise.

The process follows these steps:

1. **FFT:** Transforms the noisy signal from the time domain to the frequency domain.
2. **Filtering:** High-frequency noise is identified and removed by setting the corresponding frequency components to zero.
3. **Inverse FFT:** The cleaned signal is reconstructed by applying the Inverse FFT, returning it to the time domain.

**Procedure:**

1. **Signal Generation:**
   * A pure sine wave of frequency 440 Hz is generated, which mimics an "A4" musical note.
   * Random noise is added to this pure sine wave to simulate a noisy audio signal.
2. **FFT Transformation:**
   * The noisy signal is transformed from the time domain to the frequency domain using the FFT.
3. **Noise Filtering:**
   * Frequencies above 500 Hz are assumed to be noise and are set to zero in the frequency domain. This step filters out the high-frequency noise from the signal.
4. **Inverse FFT:**
   * The filtered signal is transformed back to the time domain using the Inverse FFT, resulting in a cleaned audio signal.
5. **Plotting Results:**
   * The original sine wave, noisy signal, and cleaned signal are plotted to visualize the effects of noise removal.

**Source Code:**

|  |
| --- |
| **import numpy as np**  **import matplotlib.pyplot as plt**  **from scipy.fft import fft, ifft, fftfreq**  **# Generate a sample audio signal**  **Fs = 1000 # Sampling rate (1000 Hz)**  **T = 1 / Fs # Sampling interval**  **t = np.linspace(0, 1, Fs, endpoint=False) # 1 second time vector**  **# Generate a pure sine wave (440 Hz, like an "A4" musical note)**  **freq\_signal = 440**  **pure\_signal = np.sin(2 \* np.pi \* freq\_signal \* t)**  **# Add random noise**  **noise = np.random.normal(0, 0.5, pure\_signal.shape)**  **noisy\_signal = pure\_signal + noise**  **# Apply FFT**  **fft\_signal = fft(noisy\_signal)**  **freqs = fftfreq(len(fft\_signal), T) # Frequency bins**  **# Filter: Remove frequencies higher than 500 Hz**  **fft\_filtered = fft\_signal.copy()**  **fft\_filtered[np.abs(freqs) > 500] = 0 # Zero out high frequencies (noise)**  **# Apply Inverse FFT to get the cleaned signal**  **cleaned\_signal = ifft(fft\_filtered).real**  **# Plot the results**  **plt.figure(figsize=(12, 6))**  **plt.subplot(3, 1, 1)**  **plt.plot(t, pure\_signal, label="Original Signal (440 Hz)")**  **plt.legend()**  **plt.title("Original Pure Signal")**  **plt.subplot(3, 1, 2)**  **plt.plot(t, noisy\_signal, label="Noisy Signal", color="red")**  **plt.legend()**  **plt.title("Noisy Signal")**  **plt.subplot(3, 1, 3)**  **plt.plot(t, cleaned\_signal, label="Cleaned Signal (After FFT Filtering)", color="green")**  **plt.legend()**  **plt.title("Filtered Signal (Noise Removed)")**  **plt.tight\_layout()**  **plt.show()** |

**Output**

|  |
| --- |
| **A group of colorful waves  AI-generated content may be incorrect.** |