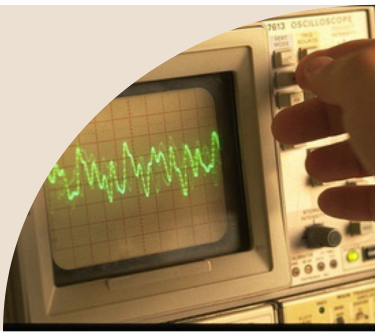
TNS Project Report : Part 1

Filière : GSTR1

HAMMOUCHI LOUAY



Part 1: Code Documentation: Convolution

Objectif :

The code is designed to simulate the output sequence y[n], which results from the convolution of two user-provided sequences, x[n] and h[n].

1. Import libraries:

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

- numpy: For numerical computations and array manipulations.
- matplotlib.pyplot: For plotting the resulting signals.
- re: For parsing user input strings (used to identify delta
- impulses and their parameters).

2. Define the delta function:

```
# Define the delta function
def delta(n):
    return np.where(n == 0, 1, 0)
```

Purpose: Defines a discrete delta function (δ[n]\delta[n]) for a given input array n.

- For n=0: Returns 1 (impulse).
- For n≠0: Returns 0.

Use Case: Conceptually represents impulses in the input

3. Parse Delta input

```
# Function to parse user input for delta impulses (handles positive and negative deltas)
def parse_delta_input(user_input, n_range):
    """
    Parses a string input (e.g., 'delta[n-3] + 2*delta[n+2] - delta[n]')
    and returns the signal array with proper handling of signs.
    """
    signal = np.zeros(len(n_range))
    matches = re.findall(r'([+-]?\d*)\*?delta\[n([+-]\d+)?\]', user_input)
```

Initialize the signal: A zero array (signal) of the same size as the time range (n_range).
Extract matches: The regular expression parses the input for terms like:

- Amplitudes: 2, -1, etc.
- Shifts: n-3, n+2, etc.
- Matches are tuples like ('2', '-3'), where 2 is the amplitude and -3 is the shift.

Handle each match

```
    for amp, shift in matches:
        # Handle amplitude
        if amp in ["", "+", "-"]:
            amplitude = 1 if amp == "" or amp == "+" else -1
        else:
            amplitude = int(amp)

        # Handle shift
        shift_value = int(shift) if shift else 0
        index = np.where(n_range == -shift_value)[0]  # Correct shift direction
        if len(index) > 0:
            signal[index] += amplitude  # Sum amplitudes if overlaps occur
    return signal
```

- **Amplitude Parsing:** Handles positive, negative, or omitted amplitudes (delta[n], -delta[n]).
- **Shift Parsing:** Translates shifts (n-3) into indices in the n_range.
- **Update Signal:** Adds the impulse amplitude at the computed index, summing amplitudes if overlapping impulses exist.

Return the Parsed Signal: The final signal array, with delta impulses represented by non-zero values at appropriate positions.

4. Find Delta Position

```
# Function to find positions of delta impulses
def find_delta_positions(signal, n_range):
    return n_range[np.where(signal != 0)]
```

- **Purpose**: Identifies positions in n_range where the signal has non-zero values (i.e., delta impulses).
- **Use Case**: Helps plot and label the impulses on the graph.

5. Main Program Flow

Define time range

```
# Range of time indices
n = np.arange(-10, 11)
```

Defines a range of discrete time indices (n) from -10 to 10.

User Input

```
# User input for x[n] and h[n]
print("Enter the input signal x[n] using Dirac impulses (e.g., delta[n-3] + 2*delta[n+2] - delta[n]):")
x_input = input("x[n]: ")
print("Enter the impulse response h[n] using Dirac impulses (e.g., delta[n] - delta[n-1]):")
h_input = input("h[n]: ")
```

Asks the user to insert the input signals x[n] and h[n], with examples provided for clarification.
Users describe signals using Dirac delta notations (e.g., 2*delta[n-3] - delta[n+2]).

Parse Signals

```
# Parse user inputs to generate signals
x = parse_delta_input(x_input, n)
h = parse_delta_input(h_input, n)
```

Converts user inputs into discrete arrays representing the signals x[n] and h[n].

Convolution

```
# Perform convolution
y = np.convolve(x, h, mode='full')
n_y = np.arange(2 * n[0], 2 * n[-1] + 1)  # Time indices for y
```

- **np.convolve: Computes the convolution y[n]=x[n] ⋆ h[n].**
- **mode='full': Returns the entire convolution result.**
- **n_y: Defines the range of indices for the output signal y[n].**

Find impulse locations and amplitudes

```
# Find positions and amplitudes of impulses in y[n]
delta_positions = find_delta_positions(y, n_y)
amplitudes = y[np.where(y != 0)]
```

- **delta_positions:** Time indices of non-zero values in y[n].
- **amplitudes:** Corresponding non-zero values (amplitudes of impulses).

6. Plotting

Styling and labeling

```
# Check if there are any impulses to plot
if len(delta_positions) > 0 and len(amplitudes) > 0:
    # Plot the convolution result
    markerline, stemlines, baseline = plt.stem(delta_positions, amplitudes, basefmt=" ")
    plt.setp(stemlines, 'color', 'blue', 'linewidth', 1.5)  # Styling
    plt.setp(markerline, 'color', 'red', 'markersize', 5)   # Styling
    plt.setp(baseline, 'color', 'black', 'linewidth', 1.0)  # Styling
    plt.title("Representation of Convolution Result y[n]")
    plt.xlabel("n (Time Index)")
    plt.ylabel("Amplitude")
    plt.grid()
```

Ensure that only integer numbers are represented on the x-axis and y-axis

```
# Force integer ticks on both axes
plt.xticks(np.arange(min(delta_positions)-1, max(delta_positions)+2, 1))
plt.yticks(np.arange(int(np.floor(min(amplitudes)) - 1), int(np.ceil(max(amplitudes)) + 2), 1))
```

7. Handle no impulse case

```
    plt.show()
else:
    print("No impulses to plot in the convolution result.")
```

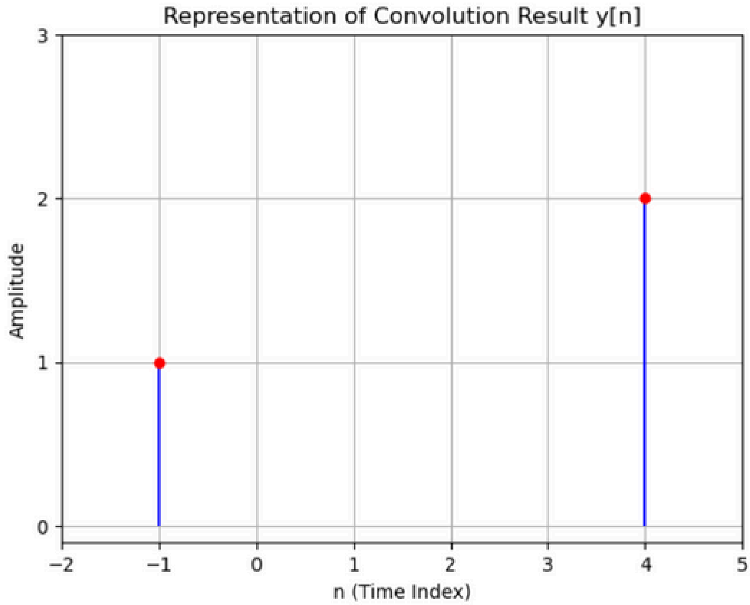
- **plt.show(): Displays the output signal y[n].**
- **else:**
- **print("No impulses to plot in the convolution result."): If there are no impulses in y[n], skips plotting and prints a message instead.**

8. Displays the convolution result array and the positions of the delta impulses in y[n]

```
# Output results with delta positions
print("y[n] (Result of x[n] * h[n]):", y)
print("Delta positions in y[n]:", delta_positions)
```

Example of an output signal:

```
Enter the input signal x[n] using Dirac impulses (e.g., delta[n-3] + 2*delta[n+2] - delta[n]):
x[n]:  2*delta[n-3] + delta[n+2]
Enter the impulse response h[n] using Dirac impulses (e.g., delta[n] - delta[n-1]):
h[n]:  delta[n-1]
```

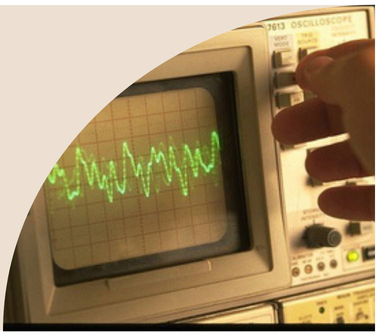


FIN

TNS Project Report : Part 2

Filière : GSTR1

EL Moraghi Saad



Part 2: Code Documentation: Code Documentation: Bilateral and Unilateral Spectrum

Objectif :

The code extracts frequencies and amplitudes from a mathematically defined periodic signal and plots the bilateral and unilateral spectra to analyze the signal's spectral components, considering sampling effects.

1. Import libraries:

```python
import matplotlib.pyplot as plt
import re
```

- matplotlib.pyplot: For plotting the resulting signals.
- re: For parsing user input strings (used to identify delta
- impulses and their parameters).

2. Extracting Frequencies and Amplitudes:

```python
def extract_frequencies_and_amplitudes(expression):
    """
    Extracts amplitudes and frequencies from a mathematical expression of the type x(t).
    Example: x(t) = 2*cos(2*pi*2000*t) + cos(2*pi*4000*t) + 0.5*cos(pi*12000*t)
    Returns:
    - A list of tuples (amplitude, frequency)
    """
    matches = re.findall(r'([0-9]*\.?[0-9]+)?\s*\*\s*cos\((?:2\*pi|pi)\s*\*\s*(\d+)', expression)
    result = []
    for amp, freq in matches:
        amplitude = float(amp) if amp else 1.0  # Default amplitude = 1.0
        frequency = int(freq)
        result.append((amplitude, frequency))
    return result
```

The function extract_frequencies_and_amplitudes processes a mathematical signal expression of the form:
It extracts the frequencies and amplitudes as a list of tuples .

- Detecting cosine components: The pattern matches expressions like A * cos(B * frequency), where amplitude is optional (default).
- Regular Expression: re.findall(r'([0-9]*\.?[0-9]+)?\s**\s*cos\((?:2*pi|pi)\s**\s*(\d+)', expression).
- Default Values: If amplitude is missing, it is automatically set to 1.0.

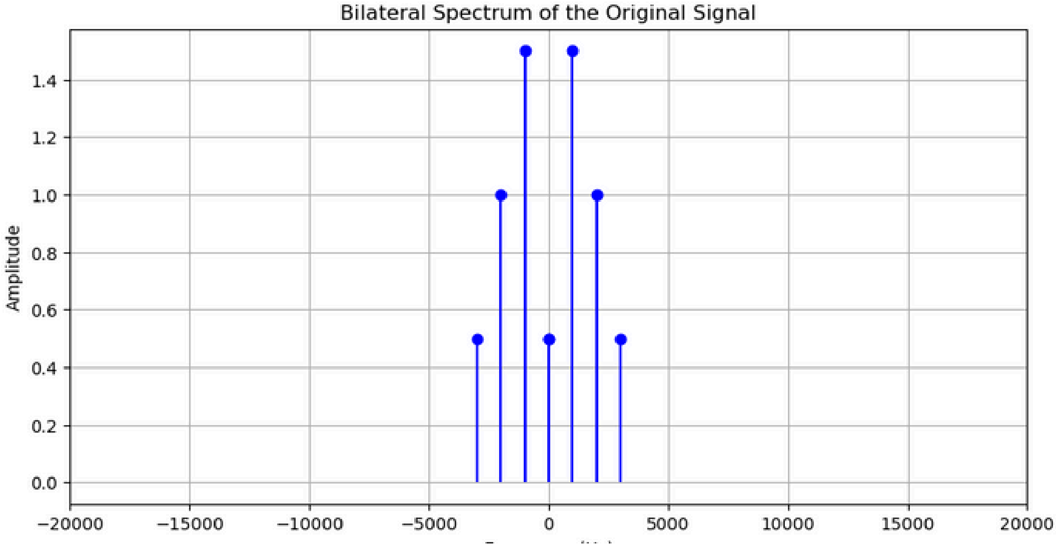
3. Plots the bilateral and unilateral spectrum of a signal (original and sampled).

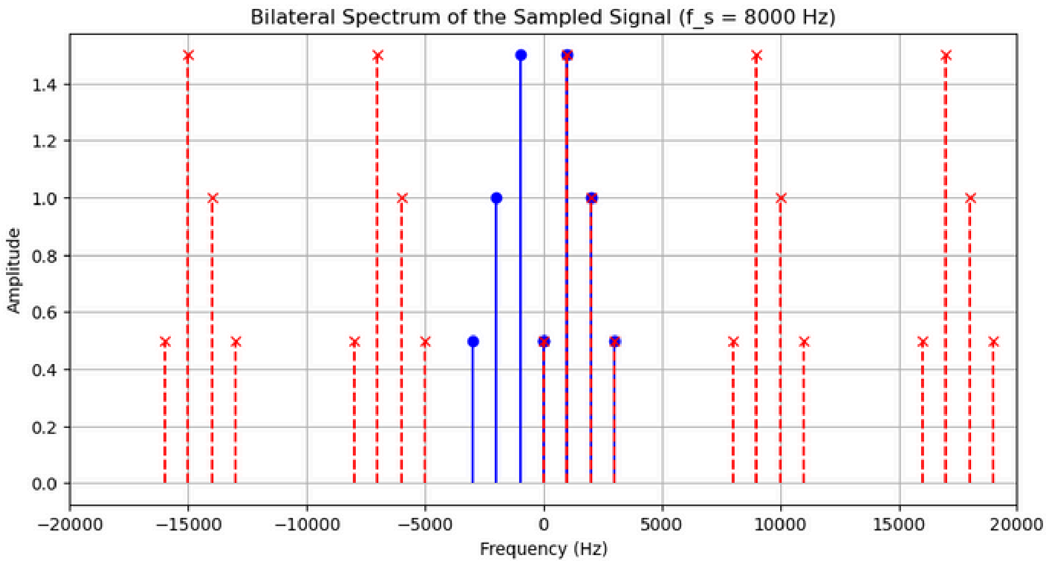
```
def plot_signal_spectrum(frequencies_and_amplitudes, sampling_rate, frequency_limit):
    """
    Plots the bilateral and unilateral spectrum of a signal (original and sampled).
    """
    def plot_bilateral_spectrum(frequencies_and_amplitudes, title, sampling_freq=None, limit=20000):
        plt.figure(figsize=(10, 5))
        # Bilateral spectrum: divide amplitudes by 2
        for amp, f in frequencies_and_amplitudes:
            plt.stem([f, -f], [amp / 2, amp / 2], linefmt='b-', markerfmt='bo', basefmt=" ")

        # Aliases
        if sampling_freq:
            for amp, f in frequencies_and_amplitudes:
                for k in range(-3, 4):  # Copies around multiples of f_s
                    alias = f + k * sampling_freq
                    if abs(alias) <= limit:
                        plt.stem([alias], [amp / 2], linefmt='r--', markerfmt='rx', basefmt=" ")

        plt.title(title)
        plt.xlabel("Frequency (Hz)")
        plt.ylabel("Amplitude")
        plt.xlim(-limit, limit)
        plt.grid()
        plt.show()

    def plot_unilateral_spectrum(frequencies_and_amplitudes, title, sampling_freq=None, limit=20000):
        plt.figure(figsize=(10, 5))
        # Unilateral spectrum: original amplitudes
        for amp, f in frequencies_and_amplitudes:
            plt.stem([f], [amp], linefmt='b-', markerfmt='bo', basefmt=" ")

        # Aliases
        if sampling_freq:
            for amp, f in frequencies_and_amplitudes:
                for k in range(1, 4):  # Only positive aliases
                    alias = f + k * sampling_freq
                    if alias <= limit:
                        plt.stem([alias], [amp], linefmt='r--', markerfmt='rx', basefmt=" ")

        plt.title(title)
        plt.xlabel("Frequency (Hz)")
        plt.ylabel("Amplitude")
        plt.xlim(0, limit)
        plt.grid()
        plt.show()
```

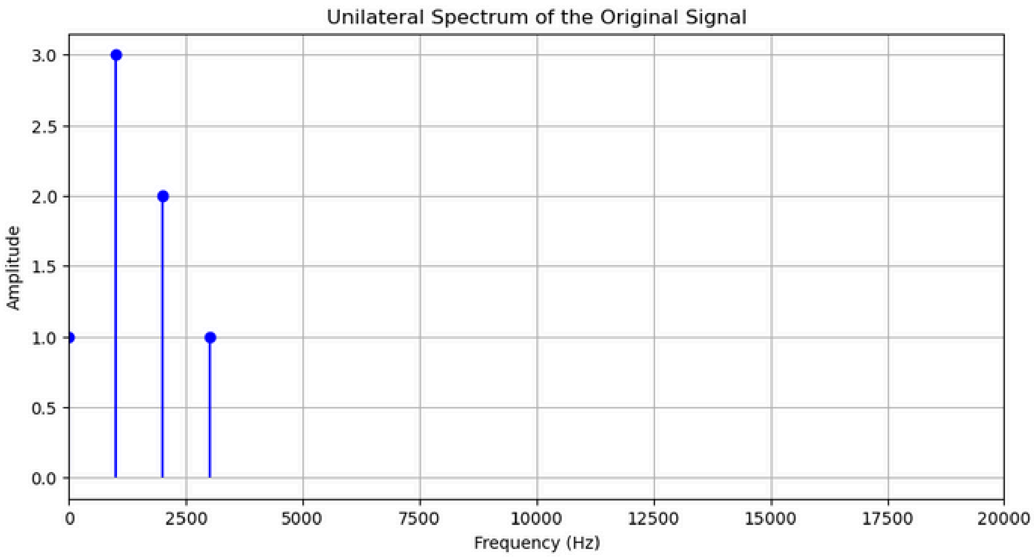
The scond for loop calculates aliases of the frequencies by shifting them by multiples of the sampling frequency, both positively and negatively.

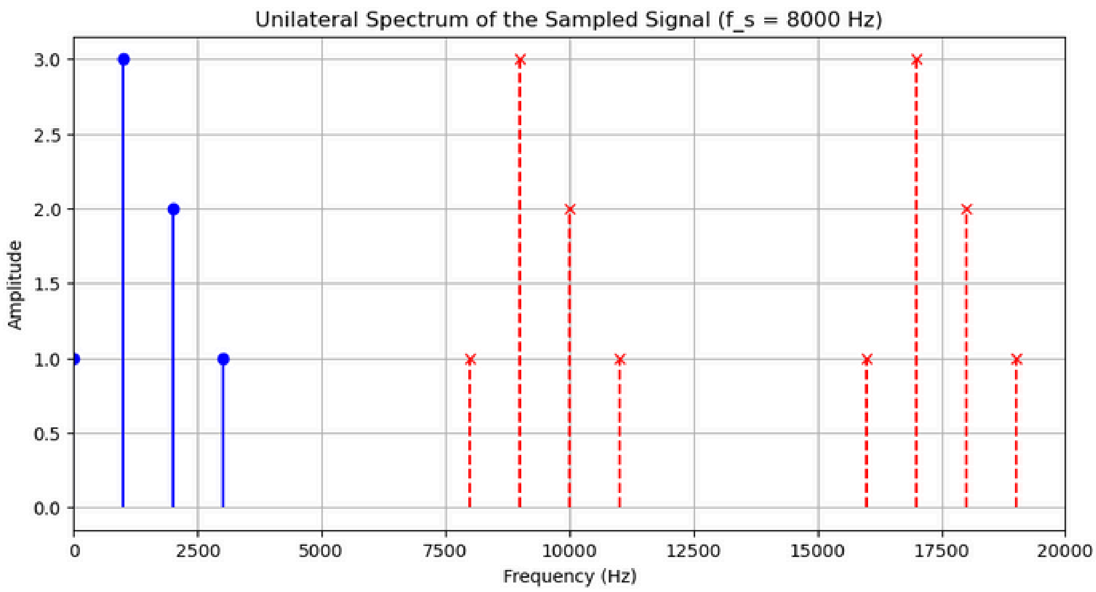
Example of an output signal:

```
=== Resolution of Bilateral and Unilateral Spectra ===
Enter the full expression of x(t) (e.g., x(t) = 2*cos(2*pi*2000*t)) :  1*cos(2*pi*0*t)+3*cos(2*pi*1000*t)+2*cos(2*pi*2000*t)+1*cos(2*pi*3000*t)
Extracted Amplitudes and Frequencies: [(1.0, 0), (3.0, 1000), (2.0, 2000), (1.0, 3000)]
Enter the sampling frequency (in Hz):  8000
Enter the frequency limit for display (in Hz):  20000
```





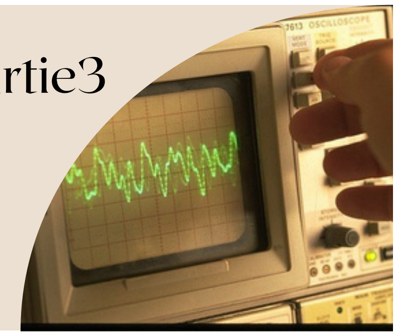

Unilateral Spectrum of the Original Signal
Amplitude
Frequency (Hz)
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
2500
5000
7500
10000
12500
15000
17500
20000


Unilateral Spectrum of the Sampled Signal (f_s = 8000 Hz)
Amplitude
Frequency (Hz)
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
2500
5000
7500
10000
12500
15000
17500
20000

Compte rendu sur le projet TNS : Partie3

Filière : GSTR1

CHEMMAM FATIMA EZZAHRA



Partie 3 : Documentation du Code : Traitement de Signal avec Filtrage Passe-Bas et Analyse FFT

Objectif :

Le code simule un signal périodique composé d'impulsions, applique un filtre passe-bas et analyse le signal avant et après filtrage à l'aide de la transformée de Fourier rapide (FFT).

1. Importation des Bibliothèques :

```
[50]: import numpy as np
      import matplotlib.pyplot as plt
      from scipy.signal import butter, lfilter
```

- numpy : Utilisé pour la manipulation de données numériques et les calculs mathématiques.
- matplotlib.pyplot : Utilisé pour la création de graphiques et la visualisation des signaux.
- scipy.signal : Contient des fonctions pour la conception de filtres et le traitement du signal.

2. Création du Signal Périodique :

```
sampling_frequency = 8000  # Hz (sampling rate)
duration = 0.5  # seconds (duration of the signal)
t = np.linspace(0, duration, int(sampling_frequency * duration), endpoint=False)
```

- sampling_frequency : La fréquence d'échantillonnage est de 8000 Hz, ce qui signifie que le signal est échantillonné 8000 fois par seconde.
- duration : La durée du signal est de 0.5 seconde.
- t : Vecteur temporel généré avec np.linspace pour représenter le temps, de 0 à 0.5 seconde, avec un échantillonnage à 8000 Hz.

3. Rectification du Signal :

```
signal = np.zeros_like(t)
impulse_indices = np.arange(0, len(t), step=200)  # Impulse every 200 samples
signal[impulse_indices] = 1  # Set impulses to amplitude 1
```

- Rectification du signal pour ne conserver que les valeurs positives.

4. Filtrage Passe-Bas :

```
# Define the low-pass filter function
def low_pass_filter(signal, cutoff_frequency, sampling_rate):
    nyquist = sampling_rate / 2
    normal_cutoff = cutoff_frequency / nyquist
    b, a = butter(4, normal_cutoff, btype='low', analog=False)
    filtered_signal = lfilter(b, a, signal)
    return filtered_signal

# Apply low-pass filtering
cutoff_frequency = 400  # Hz
filtered_signal = low_pass_filter(rectified_signal, cutoff_frequency, sampling_frequency)
```

- La fonction low_pass_filter applique un filtre passe-bas à l'aide de la fonction butter de SciPy.
- Le cutoff_frequency définit la fréquence de coupure du filtre.
- normal_cutoff est la fréquence de coupure normalisée par rapport à la fréquence de Nyquist (moitié de la fréquence d'échantillonnage).
- lfilter applique ce filtre au signal rectifié.

5. Application de la FFT :

```
# Compute FFT for the original and filtered signals
fft_rectified = np.fft.fft(rectified_signal)
fft_filtered = np.fft.fft(filtered_signal)
frequencies_fft = np.fft.fftfreq(len(fft_rectified), 1 / sampling_frequency)

# Plot the results
plt.figure(figsize=(12, 8))
```

- fft_rectified et fft_filtered contiennent les coefficients de la FFT pour les signaux rectifiés et filtrés respectivement.
- frequencies_fft génère les fréquences correspondantes aux composantes du signal dans le domaine fréquentiel, en utilisant np.fft.fftfreq.

6. Visualisation des Résultats :

```
# Original rectified signal in the time domain
plt.subplot(3, 1, 1)
plt.plot(t, rectified_signal, label="Rectified Signal", color='magenta')
plt.title("Rectified Signal (Time Domain)")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.legend()

# Frequency domain (rectified signal)
plt.subplot(3, 1, 2)
plt.stem(frequencies_fft[:len(frequencies_fft)//2], np.abs(fft_rectified[:len(fft_rectified)//2]), basefmt=" ", use_line_collection=True, label="FFT of FFT of Rectified Signal")
plt.title("Frequency Spectrum of Rectified Signal")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Amplitude")
plt.legend()

# Frequency domain (filtered signal)
plt.subplot(3, 1, 3)
plt.stem(frequencies_fft[:len(frequencies_fft)//2], np.abs(fft_filtered[:len(fft_filtered)//2]), basefmt=" ", use_line_collection=True, label="FFT of FiT of Filtered Signal")
plt.title("Frequency Spectrum of Filtered Signal")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Amplitude")
plt.legend()

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

Résultats de l'Exécution du Code :

- À l'issue de l'exécution du code, les résultats obtenus sont les suivants :
- Signal rectifié dans le domaine temporel : Un graphique montre le signal périodique rectifié, où seules les valeurs positives sont conservées.
- Spectre de fréquence du signal rectifié : Un graphique de la transformée de Fourier rapide (FFT) du signal rectifié révèle les fréquences présentes dans le signal avant le filtrage.
- Spectre de fréquence du signal filtré : Après application du filtre passe-bas, un autre graphique de la FFT montre l'atténuation des fréquences supérieures à 400 Hz, illustrant l'effet du filtrage.
- Ces résultats permettent d'observer l'impact du filtrage sur la composition fréquentielle du signal.