Luu Duc Lam 20203844

Biomedical signal processing

Instructor: Trinh Quang Duc

Assignment 1

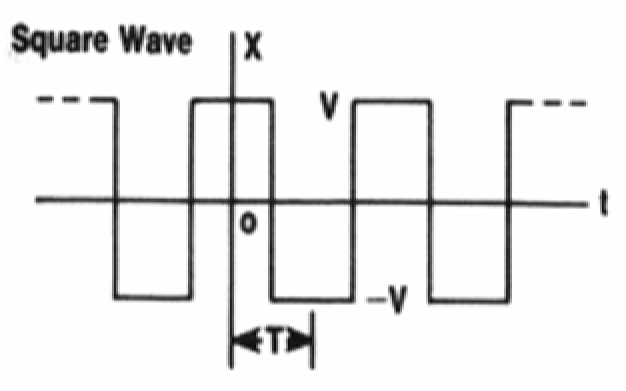
1.1 Mathematical model

Establishing a general mathematical model of waves:

- Square wave
- Triangle wave
- Sawtooth wave
- Pulse train

---

1. **Square Wave:**

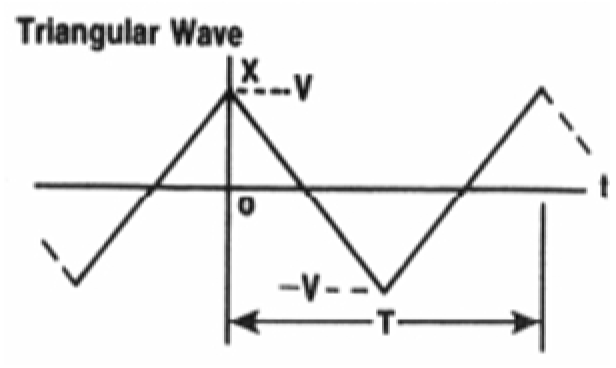


- General formula:

$$x(t) = \frac{4V}{\pi} \sum_{n=1,3,5...}^{\infty} \frac{1}{n} \sin(n\omega t + \phi_n)$$

where ($\omega = 2\pi f, V = amplitude, \phi_n = n\phi_0$)

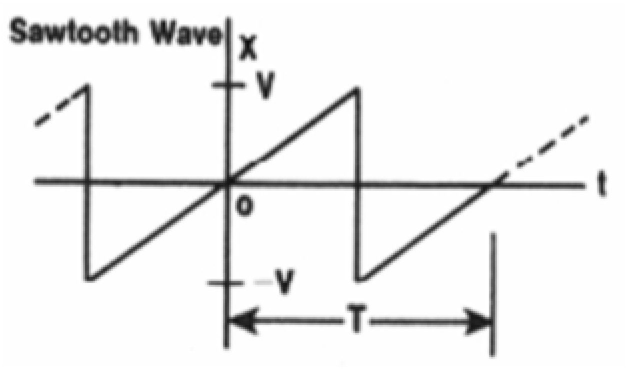
2. **Triangle Wave:**



$$x(t) = \frac{8}{\pi^2} \sum_{n=1,3,5...}^{\infty} \frac{(-1)^{\frac{n-1}{2}}}{n^2} \sin(n\omega t + \phi_n)(V = 1)$$

where ($\omega = 2\pi f, \phi_n = n\phi_0$)

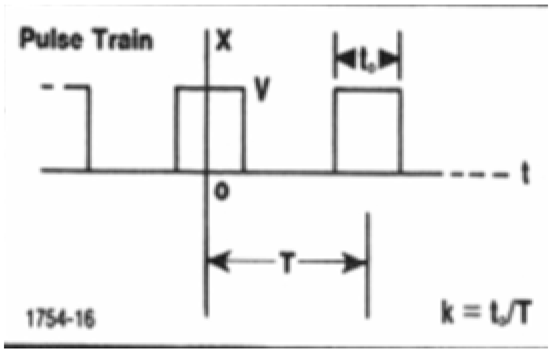
3. **Sawtooth Wave:**



$$x(t) = \frac{2}{\pi}\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n}\sin(n\omega t + \phi_0)(V = 1)$$

where ($\omega = 2\pi f, \phi_n = n\phi_0$)

4. **Pulse Train:**
   - A square wave with non-negative value is a form of pulse train



$$x(t) = max(\frac{4V}{\pi}\sum_{n=1,3,5...}^{\infty}\frac{1}{n}\sin(n\omega t + \phi_n), 0)$$

1.2. Code implementation

Write code to implement drawing the waves modeled mathematically:

- Survey with 1000 waves?
- Survey with phase angle phi = 2*pi/N?
- Find the amplitude of the wave?

- Plot the statistical amplitude of the synthesized function (waves) according to the total number of component waves running from 100 to 1000?

  Using the Fourier series model with frequency f = $1000 * sin(\frac{2\pi}{49} * N)$, where N is the student ID number (N = 27).

I define package as well as all constants value in my project which is described in Appendix

1.2.1. General approach

1. First I generates a wave with 1000 components

   ```
   wave1k = in_func(A=A, time=time, f=f, phi=phi, num_components=1000)
   ```

2. Second, I will find the maximum amplitude of this wave using the `find_amplitude` function.

   ```
   def find_amplitude(wave):
       return np.max(wave)

   print(f'\nAmplitude with 1000 components: {find_amplitude(wave1k)}')
   ```

3. Handling edge points (outlier): Outlier points in the wave with values greater than the specified amplitude are clipped to the amplitude value to filter out extreme values.

   ```
   # filter points at edge (outliers which value > amplitude)
   wave1k = np.clip(wave1k, -amplitude, amplitude)
   ```

For easy, I encapsulate my general approach in a function with input is each function in part 1.1. So easily, I just need to implementation from mathematical model and then put it into my general approach function.

Here are my general approach function which including all steps above

```
def plot_and_stats(in_func, amplitude=A) -> None:
    """
    Plot input function, find amplitude and survey amplitude 
wave
    as changing num_components
    params:
        in_func: input function,
        amplitude: amplitude
    return: None
    """

    wave1k, amps = in_func(A=amplitude)

    def find_amplitude(wave):
        return np.max(wave)

    print(f'\nAmplitude with 1000 components: {find_amplitude
(wave1k)}')

    # filter points at edge (outliers which value > amplitud
e)
    wave1k = np.clip(wave1k, -amplitude, amplitude)

    fig, axes = plt.subplots(1, 2, figsize=(12, 6))

    # Plot
    axes[0].plot(time, wave1k)
    axes[0].set_title('Wave (filtered) from Fourier Series')
    axes[0].set_xlabel('Time [s]')
    axes[0].set_ylabel('Amplitude')
    axes[0].grid(True)

    ##### amplitude STATS

    # Amplitude statistic of sum function as
    # the number of components change for 100 to 1000?
    components = list(range(100, len(amps), 1))
    axes[1].plot(components, amps[100:len(amps)])
    axes[1].set_title('Amplitude Statistics')
    axes[1].set_xlabel('Number of Components')
    axes[1].set_ylabel('Amplitude')
    axes[1].grid(True)

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

1.2.2. Square wave experiment

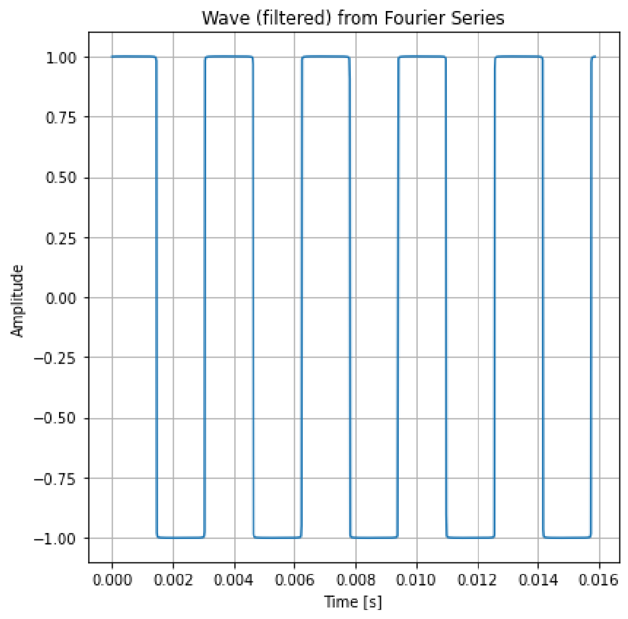
I define Python code for square wave function from mathematical above

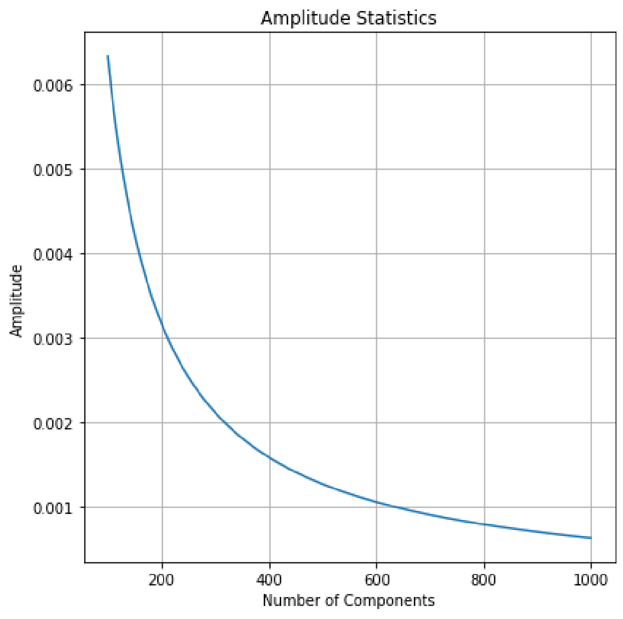
```
def square_wave(A=A, time=time, f=f, phi=phi, num_components=
1000):
    wave = 4/np.pi * A * sum(
                            1/n * np.sin((2*np.pi*f) * n *  t
ime + phi * n)
                            for n in range(1, 2*num_component
s+1, 2)
                        )
    amp = [4 * A / (np.pi * n) for n in range(1, 2*num_compon
ents+1, 2)]
    return wave, amp
```

Then i put it into function defining in general approach.

```
plot_and_stats(square_wave, 1)
```

Here are results





1.2.2. Triangle wave experiment

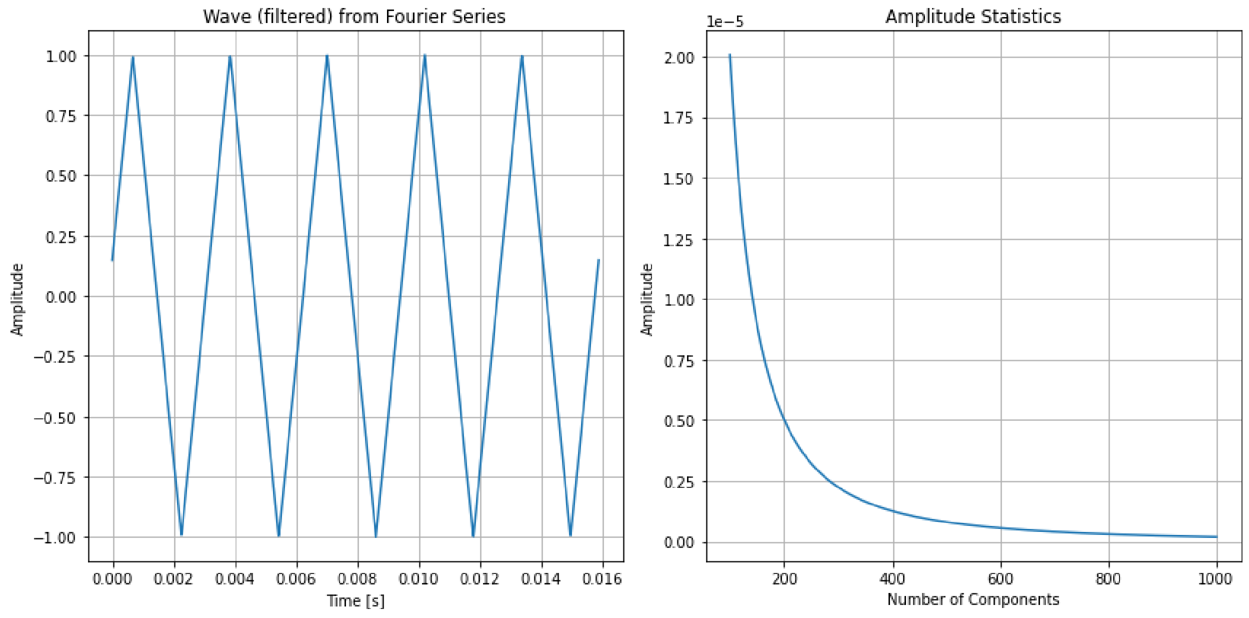
I define Python code for Triangle wave function from mathematical above

```
def triangle_wave(A=A, time=time, f=f, phi=phi, num_component
s=1000):
    wave = (8*A / np.pi**2) * sum(
                                   (-1)**((k-1)//2)/ k**2  *
                                   np.sin(2 * np.pi * k * f * ti
me + k * phi)
                                   for k in range(1, 2*num_compo
nents+3, 2)
                                   )
    amps = [8*A / (np.pi**2 * n**2) for n in range(1, 2*num_c
omponents+3, 2)]
    return wave, amps
```

Then i put it into function defining in general approach.

```
plot_and_stats(triangle_wave, A)
```

Here are results



1.2.3. Sawtooth wave experiment

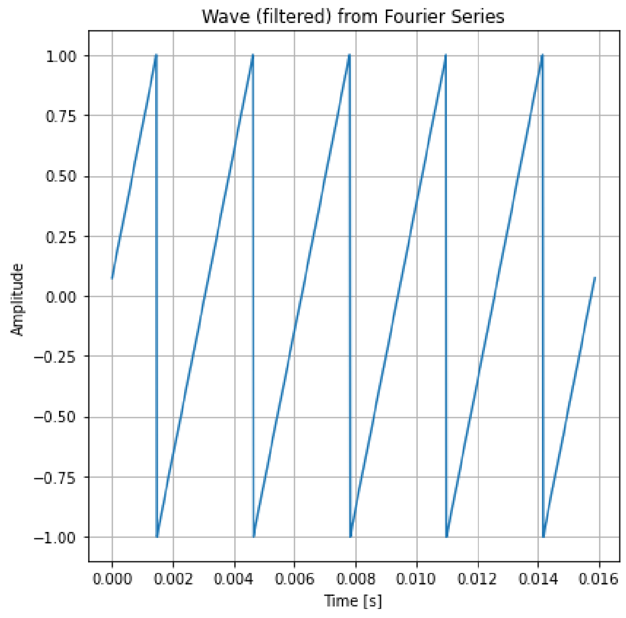
I define Python code for Sawtooth wave function from mathematical above

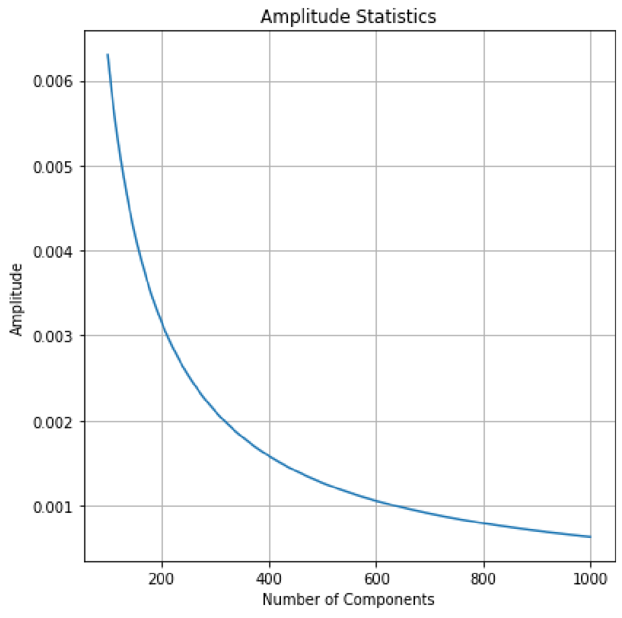
```
def sawtooth_wave(A=A, time=time, f=f, phi=phi, num_component
s=1000):
    wave = (2*A / np.pi) * sum(
                                            ((-1)**(n+1) / n) *
                                            np.sin(2 * np.pi * f * n * ti
me + n * phi)
                                            for n in range(1, num_compone
nts+1)
                                        )
    amps = [2*A / (np.pi*n) for n in range(1, num_components+
1) ]
    return wave, amps
```

Then i put it into function defining in general approach.

```
plot_and_stats(sawtooth_wave, A)
```

Here are results





1.2.3. Pulse train wave experiment

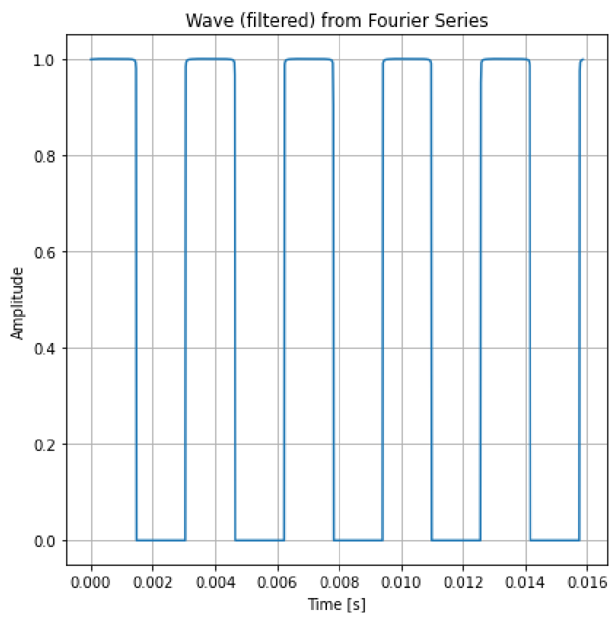
I define Python code for Pulse train wave function from mathematical above

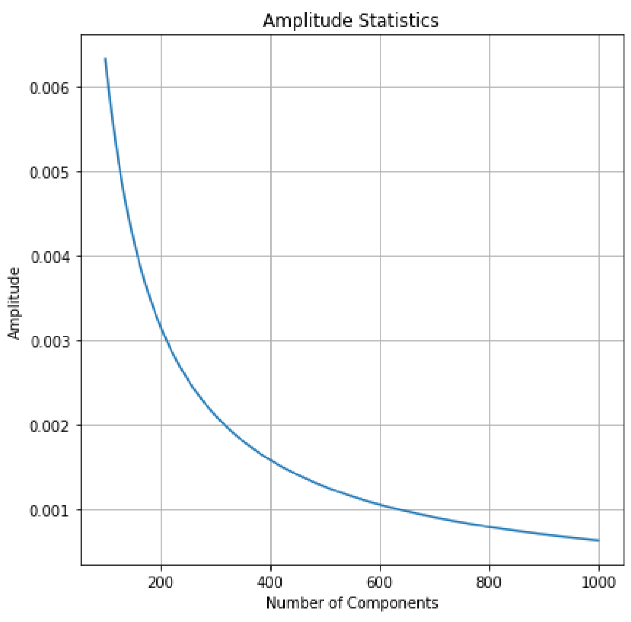
```
def pulse_train(A=A, time=time, f=f, phi=phi, num_components=
1000):
    square_w = 4/np.pi * A * sum(
                            1/n * np.sin((2*np.pi*f) * n *  t
ime + phi * n)
                            for n in range(1, 2*num_component
s+1, 2)
                        )
    max_a = np.max(square_w)
    pulse_train = np.clip(square_w, 0, max_a) # square wave w
ith non-negative is a form of pulse train
    amps = [4 * A / (np.pi * n) for n in range(1, 2*num_compo
nents+1, 2)]
    return pulse_train, amps
```

Then i put it into function defining in general approach.

```
plot_and_stats(pulse_train, A)
```

Here are results





Assignment 2

2.1. DFT algorithm

Establish the DFT algorithm:

- Continuous mathematical model of DFT
- Discrete mathematical model of DFT
- Representation of discrete DFT algorithm
- Experiment with the function $y = N\sin(2\pi 20t) + 2/3N\sin(2\pi 35t) + 2N\sin(2\pi 45t)$, representing the spectral graph in the range from 0 to 60 Hz (N=27)

N: Student ID number

---

1. **Continuous mathematical model of DFT**

- **The continuous Fourier transform** of a function ($f(t)$)

$$F(\omega) = \int_{-\infty}^{+\infty} f(t)e^{-j\omega t}dt$$

$F(\omega)$ - Fourier transform of $f(t)$

ω - the angular frequency

$e^{-j\omega t}$- the complex exponential function.

The integral runs over the entire real line, transforming the time domain function $f(t)$ into its frequency domain representation $F(\omega)$

2. **Discrete mathematical model of DFT**

   Discrete Fourier Transform (DFT) of a sequence $x[n]$:

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

$X[k]$ - the DFT of the sequence $x[n]$

N - the total number of samples

k - the index of the DFT output

n - the index of the input sequence.

The exponential term $e^{-j\frac{2\pi}{N}kn}$represents the complex sinusoids used as the basis functions in the transformation

3. **Representation of discrete DFT algorithm**

   Pseudocode of pure DFT

```
DFT(input_sequence):
    N = length(input_sequence)
    output_sequence = new complex array of size N
    for k from 0 to N-1:
        sum = 0
        for n from 0 to N-1:
            angle = 2 * PI * k * n / N
            sum += input_sequence[n] * e^(-j * angle)
        output_sequence[k] = sum
    return output_sequence
```

Implementation in Python: here i use matrix multiplication so my code is compact and succinct

```
def _dft(signal):
    # ## Version 1
    # N = len(signal)
    # output_sequence = np.zeros(N, dtype=complex)  # Initial

    # for k in range(N):
    #     sum_val = 0
    #     for n in range(N):
    #         angle = 2 * np.pi * k * n / N
    #         sum_val += signal[n] * np.exp(-1j * angle)  # (
    #     output_sequence[k] = sum_val

    # return output_sequence

        # Version 2: matrix multiplication
    N = len(signal)

    # complex part
    n = np.arange(N) # 1xN
    k = n.reshape((N, 1)) # Nx1
    e = np.exp(-2j * np.pi * k * n / N) # NxN

    output = np.matmul(e, signal) # matrix multiplication of

    return output
```

4. **Experiment with the function** $y = N\sin(2\pi 20t) + 2/3N\sin(2\pi 35t) + 2N\sin(2\pi 45t)$**, representing the spectral graph in the range from 0 to 60 Hz (N=27)**

- Define function with 1000 points with time interval from 0 to 1 second

```
# t = time
t = [i / 1000 for i in range(1000)]  # 1000 points from
0 to 1 second
# our signal: y= N\sin(2\pi20t) + 2/3N\sin(2\pi35t)+2N
\sin(2\pi45t)
y = [N * np.sin(2*np.pi * 20*t_step) + 2/3*N * np.sin(2
*np.pi * 35*t_step) + 2*N * np.sin(2*np.pi * 45*t_step)
for t_step in t]
```

- Define DFT transform function

```
# DFT transform function
def _dft(signal):
    N = len(signal)

    # complex part
    n = np.arange(N) # 1xN
    k = n.reshape((N, 1)) # Nx1
    e = np.exp(-2j * np.pi * k * n / N)

    output = np.dot(e, signal)

    return output
```

- Computes the DFT of the signal `y` using the `_dft` function, then computes the frequency values corresponding to the DFT coefficients and finally computes the magnitude spectrum by taking the absolute value of the DFT coefficients and normalizing by the length of the time array `t`.

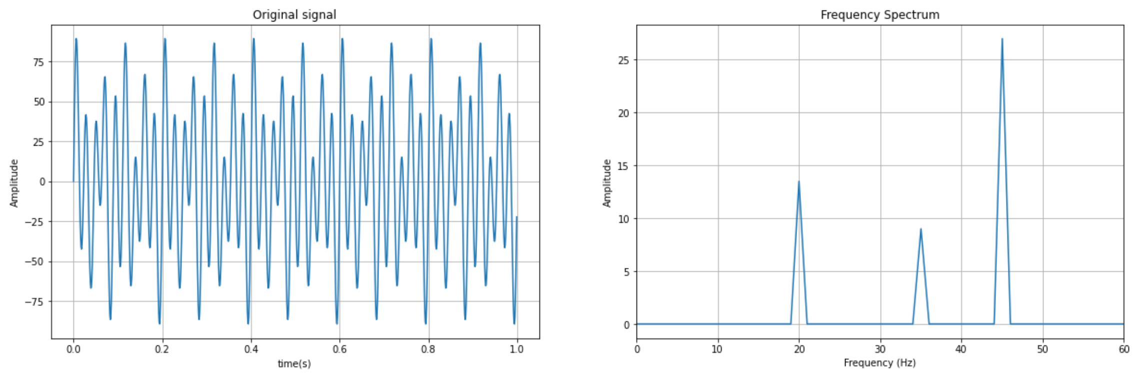
```
# DFT transform
y_dft = _dft(y)

# Compute frequency values
```

```
freqs = [k / (t[-1] - t[0]) for k in range(len(y_dft))]

# Compute magnitude spectrum
magnitude = [abs(x) / len(t) for x in y_dft]
```

- Full code and results

```
# t = time
t = [i / 1000 for i in range(1000)]  # 1000 points from
0 to 1 second
# our signal: y= N\sin(2\pi20t) + 2/3N\sin(2\pi35t)+2N
\sin(2\pi45t)
y = [N * np.sin(2*np.pi * 20*t_step) + 2/3*N * np.sin(2
*np.pi * 35*t_step) + 2*N * np.sin(2*np.pi * 45*t_step)
for t_step in t]

# DFT transform function
def _dft(signal):
    """
    Function to calculate the
    discrete Fourier Transform
    of a 1D real-valued signal signal
    """
    N = len(signal)

    # complex part
    n = np.arange(N) # 1xN
    k = n.reshape((N, 1)) # Nx1
    e = np.exp(-2j * np.pi * k * n / N)

    output = np.dot(e, signal)

    return output
```

```
# Compute the Discrete Fourier Transform (DFT)
y_dft = _dft(y)

# Compute frequency values
freqs = [k / (t[-1] - t[0]) for k in range(len(y_dft))]

# Compute magnitude spectrum
magnitude = [abs(x) / len(t) for x in y_dft]

# Plotting
fig, axes = plt.subplots(1, 2, figsize=(20, 6))

# original signal
axes[0].plot(t, y)
axes[0].set_title('Original signal')
axes[0].set_xlabel('time(s)')
axes[0].set_ylabel('Amplitude')
axes[0].grid(True)

axes[1].plot(freqs, magnitude)
axes[1].set_title('Frequency Spectrum')
axes[1].set_xlabel('Frequency (Hz)')
axes[1].set_ylabel('Amplitude')
axes[1].set_xlim(0, 60)  # Set the x-axis limit from 0
to 60 Hz
axes[1].grid(True)
plt.show()
```



Result of assignment 2.1 part d

2.2. Code practice

Write code to perform Fourier transform with the functions established in exercise 1:

- Plot the spectral graphs of the signals with maximum observation range?
- Determine the amplitudes of the center frequencies?
- Plot frequency response?
- Plot the power spectrum distribution of the signal?

2.2.1. General approach

Similar to assignment 1, I firstly define general approach steps and then encapsulate all of them into a function

1. Define DFT function which describe above
2. Performing the DFT
3. Calculating Frequency Range:
    - The frequency range is computed using the formula:

      $$frequency = \frac{k}{total\ time}$$

      where k is the index of the DFT coefficient and the total time is the duration covered by the `time_array`.

```
# frequency range
freqs = [k / (time_array[-1] - time_array[0]) for k in ran
ge(len(y_dft))]
```

4. Computing Magnitude Spectrum:
   - The magnitude spectrum is computed by taking the absolute value of each DFT coefficient and normalizing by the length of the time array.

```
# Compute magnitude spectrum
magnitude = [abs(a) / len(t) for a in y_dft]
```

5. Compute power spectrum distribution

```
power_spectrum = [a ** 2 for a in magnitude]
```

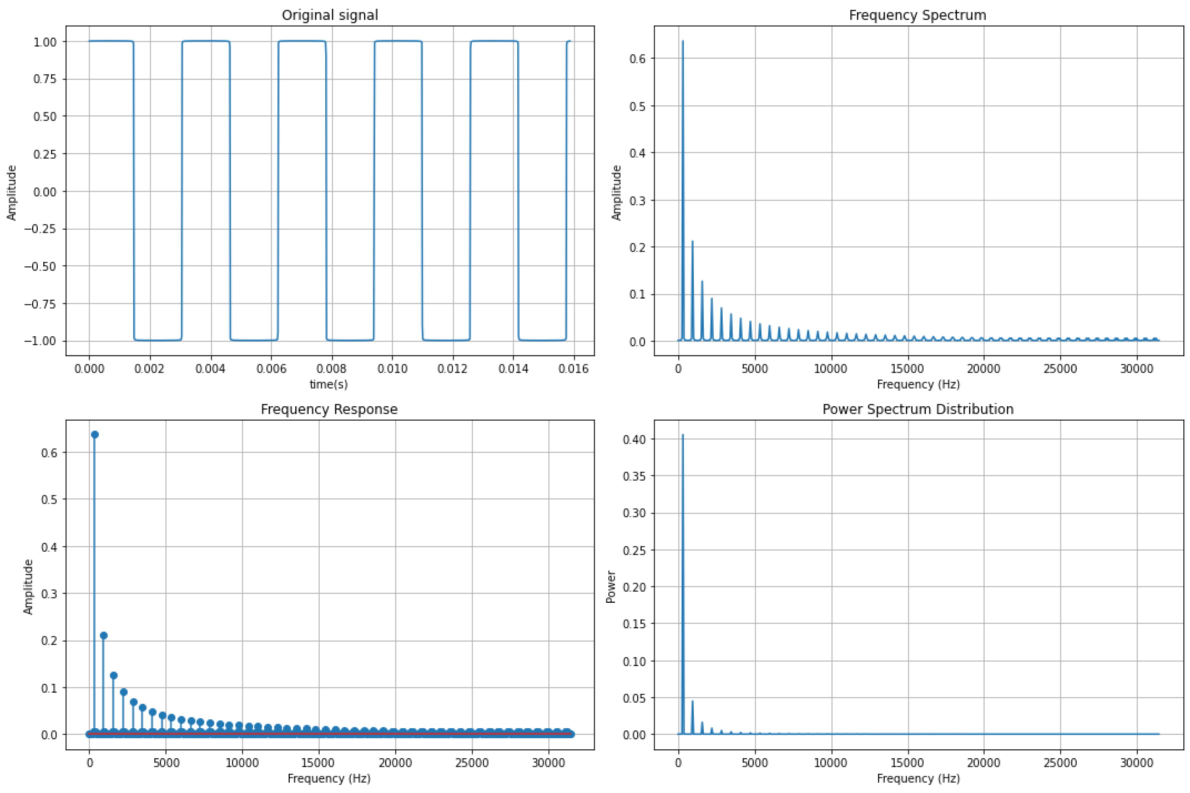
Put everything together, i have my function

```
def dft_and_plot(in_signal, time_array=time,  A=None):
    """
    Apply discrete fourier transform to input function and th
en plot
    params:
        - in_signal: input signal
        - time_array:  a time vector of time interval
    """

    if A is not None:
        # filter signal (clipping)
        in_signal = np.clip(in_signal, -A, A)
    # DFT transform function
    def _dft(signal):
        N = len(signal)

        # complex part
        n = np.arange(N) # 1xN
        k = n.reshape((N, 1)) # Nx1
        e = np.exp(-2j * np.pi * k * n / N)

        output = np.dot(e, signal)

        return output


    # DFT process
    y_dft = _dft(in_signal)

    # frequency range
    freqs = [k / (time_array[-1] - time_array[0]) for k in ra
nge(len(y_dft))]

    # Compute magnitude spectrum
    magnitude = [abs(a) / len(t) for a in y_dft]


    # Find maximum observation range
    max_obs_range_index = np.argmax(magnitude)
    center_frequency = abs(freqs[max_obs_range_index])
    max_amplitude = magnitude[max_obs_range_index]

    # divide spectrum
    di = int(len(magnitude)/2)

    # Print center frequency and corresponding amplitude
    print("Center Frequency:", center_frequency, "Hz")
    print("Corresponding Amplitude:", max_amplitude)

    ## PLOTTING
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))

    # original signal
    axes[0, 0].plot(time_array, in_signal)
    axes[0, 0].set_title('Original signal')
    axes[0, 0].set_xlabel('time(s)')
    axes[0, 0].set_ylabel('Amplitude')
    axes[0, 0].grid(True)

    # full range spectrum
    axes[0, 1].plot(freqs[:di], magnitude[:di])
    axes[0, 1].set_title('Frequency Spectrum')
    axes[0, 1].set_xlabel('Frequency (Hz)')
    axes[0, 1].set_ylabel('Amplitude')
    axes[0, 1].grid(True)

    # Plot the frequency response
    axes[1, 0].stem(freqs[:di], magnitude[:di], use_line_coll
ection=True)
    axes[1, 0].set_title('Frequency Response')
    axes[1, 0].set_xlabel('Frequency (Hz)')
    axes[1, 0].set_ylabel('Amplitude')
    axes[1, 0].grid(True)

    # Plot the power spectrum distribution
    power_spectrum = [a ** 2 for a in magnitude]
    axes[1, 1].plot(freqs[:di], power_spectrum[:di])
    axes[1, 1].set_title('Power Spectrum Distribution')
    axes[1, 1].set_xlabel('Frequency (Hz)')
    axes[1, 1].set_ylabel('Power')
    axes[1, 1].grid(True)

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

2.2.2. Put functions from exercise 1

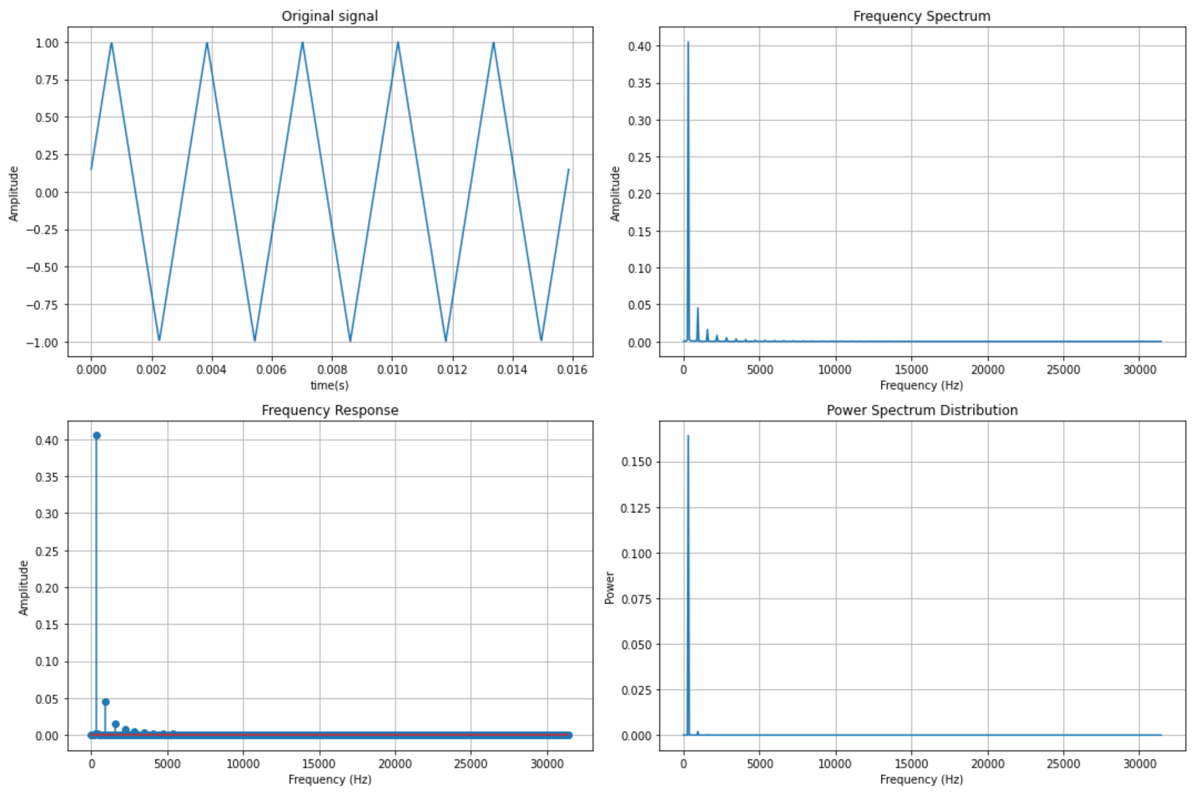
- Result with square wave

```
square_wave1000, _ = square_wave()
dft_and_plot(square_wave1000)
```



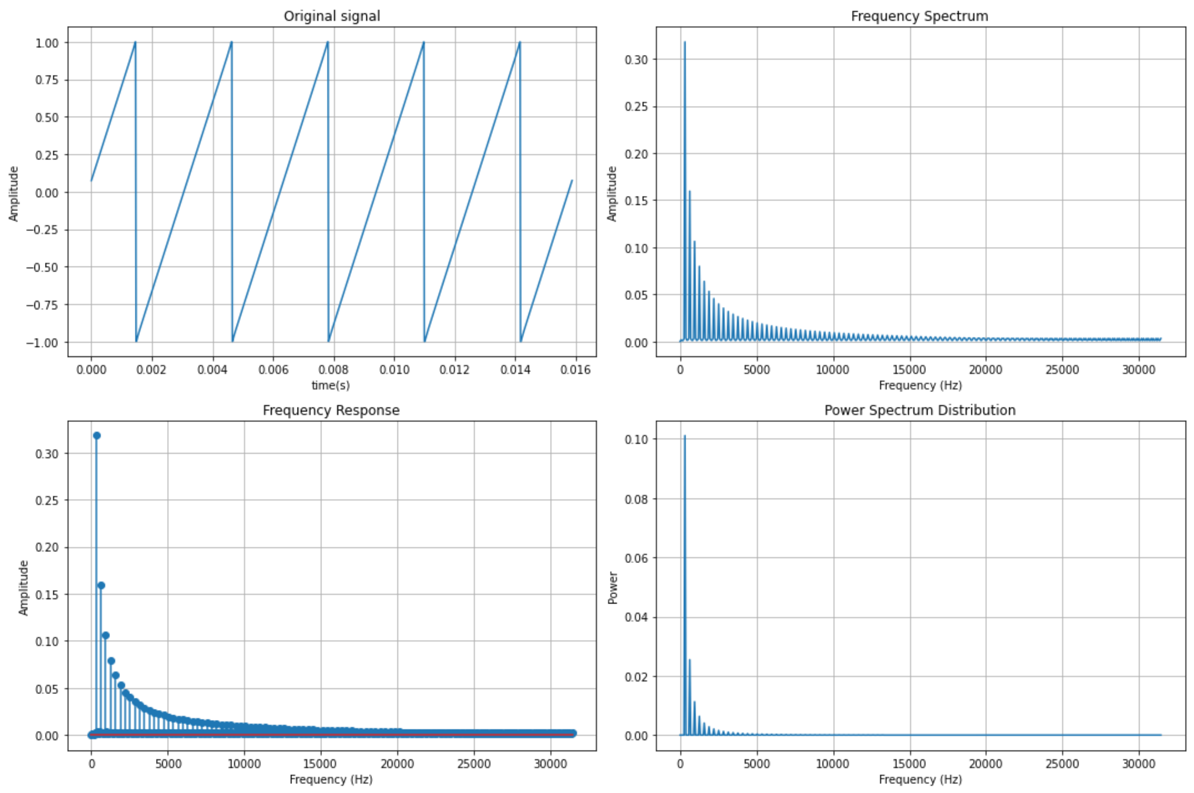
- Result with triangle wave

```
triangle_wave1000, _ = triangle_wave()
dft_and_plot(triangle_wave1000)
```



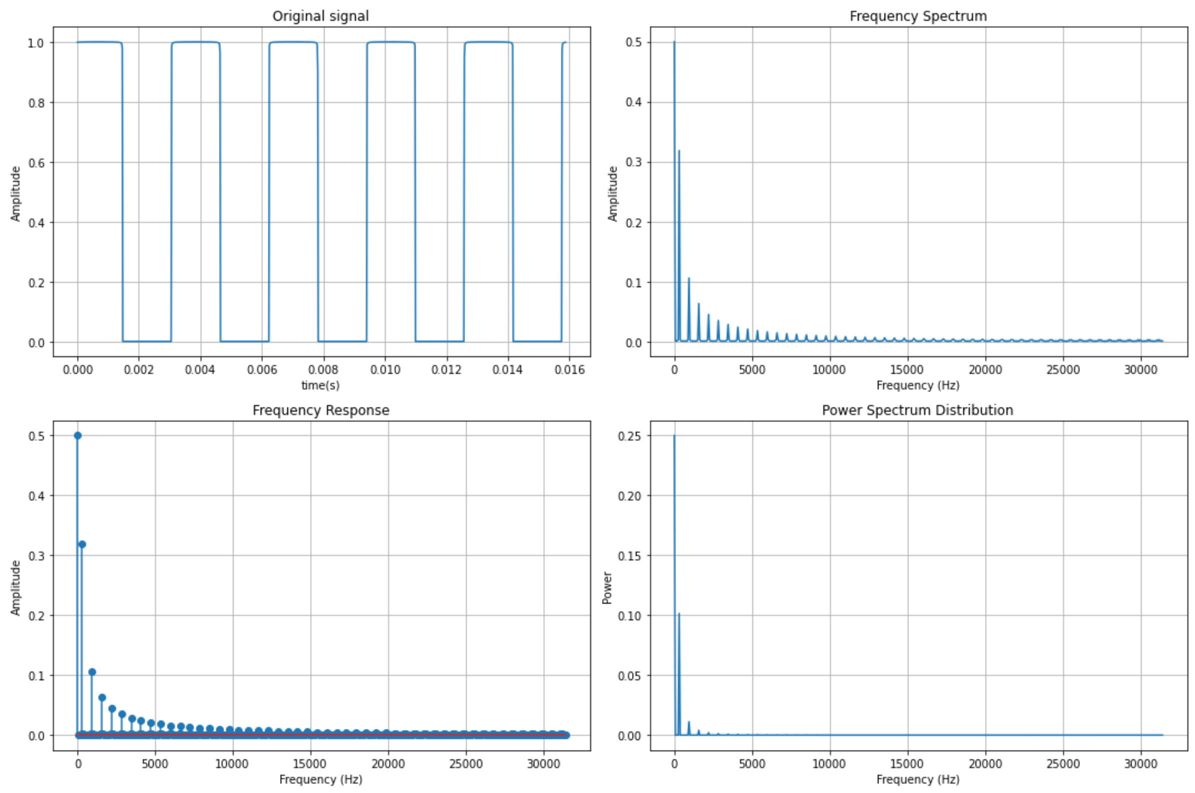
- Result with saw tooth

```
sawtooth_wave1000, _ = sawtooth_wave()
dft_and_plot(sawtooth_wave1000, A=1)
```



- Result with pulse train

```
pulse_train1000, _ = pulse_train()
dft_and_plot(pulse_train1000)
```



Appendix

- Constants and Packages in our project

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

## CONSTANTS
N = 27 # id number

f = abs(1000 * np.sin( (2 * np.pi / 49) * N)) # Frequency
phi = 2 * np.pi / N # Phase
A = 1 # amplitude


num_components = 1000  # 1000 component waves
sample_rate = 1000 # sample rate
time_interval = 5 * 1/f
```

```
time = np.linspace(0, time_interval, sample_rate)  # 3 per
iod

print(
    f'\n Amplitude: {A}',
    f'\n Frequency: {f}',
    f'\n Init Phase: {phi}',
)
```