Library Imports

In [ ]:
```
import numpy as np
import plotly.graph_objects as go
from scipy.integrate import simps
```

Imagine we have an LED within an optical detection apparatus. To detect various materials, we require multiple LED colors. If we wish to perform simultaneous detection with these LEDs and distinguish their signals, one way to achieve this is by rapidly switching them on and off. The signals detected by photodiodes can be combined through a shared mixer, while the microcontroller (MCU) can perform a Fast Fourier Transform (FFT) to separate the different detection channels.

Now, let's simulate one of these detection channels.

- $y(t) = A \cdot \sin(2\pi f t + \phi)$

  Where:

  - $y(t)$ is the value of the wave at time t.
  - A is the amplitude of the wave, determining its maximum and minimum values.
  - f is the frequency of the wave, which specifies how many cycles occur in one second (measured in Hertz, Hz).
  - ϕ is the phase angle, which determines the horizontal shift of the wave along the time axis.

- $y(t) = \sin(40.0 \cdot 2\pi t)$

  In this equation, the frequency (f) is set to 40.0 Hz.

The Simpson's rule for integration is given by:

$$\text{Simpson's Rule: } \int_a^b f(x)\,dx \approx \frac{h}{3}[f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h) + 2f(a -$$

here it is manually defined

In [ ]:
```
def simpsons_integration_manual(xf, yf, idx_start, idx_stop):
    # Ensure that the number of intervals is even
    n = idx_stop - idx_start
    if (idx_stop - idx_start) % 2 != 0:
        idx_stop -= 1

    h = (xf[idx_stop] - xf[idx_start]) / n
    result = 0

    # Loop in steps of 2 since Simpson's rule integrates over two intervals at once
    for i in range(0, n - 1, 2):
        result += (h/3) * (yf[idx_start + i] + 4*yf[idx_start + i + 1] + yf[idx_sta

    return result
```

and again with a library feature. we will compare the results

In [ ]:
```
def simpsons_integration_library(xf, yf, idx_start, idx_stop):
    # Extract the relevant x and y values
    x = xf[idx_start:idx_stop+1]
    y = yf[idx_start:idx_stop+1]

    # Use the scipy Simpson's integration
    result = simps(y, x)

    return result
```

In [ ]:
```
def generate_signal(timestep, numsamples):
    t = np.linspace(0, numsamples*timestep, numsamples)
    # windowing functions decrese side lobes. I am not going to implement a hamming
    windowed_signal = np.sin(40.0 * 2.0 * np.pi * t) * np.hamming(numsamples)
    return windowed_signal

def fft_calculate(data, timestep):
    yf = np.abs(np.fft.fft(data))
    numsamples = len(data)
    # same here, the fast fourier transform is being pulled from an industry standa
    freq = np.fft.fftfreq(numsamples, d=timestep)
    xf = freq[:numsamples//2]
    yf = yf[:numsamples//2] * 2.0 / numsamples
    return xf, yf

def find_nearest(array, value):
    idx = np.argmin(np.abs(array - value))
    nearestValue = array[idx]
    return idx, nearestValue
```

In [ ]:
```
def visualize_signal(signal, timestep, mod_freq_hz, channel_separation_hz):
    xf, yf = fft_calculate(signal, timestep)

    freq_start = mod_freq_hz - channel_separation_hz / 2
    freq_stop = mod_freq_hz + channel_separation_hz / 2

    idx_start, _ = find_nearest(xf, freq_start)
```

```
    idx_stop, _ = find_nearest(xf, freq_stop)

    # Ensure even number of intervals
    if (idx_stop - idx_start) % 2 == 0:
        idx_stop += 1

    integrated_area_manual = simpsons_integration_manual(xf, yf, idx_start, idx_sto
    integrated_area_library = simpsons_integration_library(xf, yf, idx_start, idx_s

    # Time-domain Signal plot
    fig1 = go.Figure()
    fig1.add_trace(go.Scatter(y=signal, mode='lines', name='Signal'))
    fig1.update_layout(title='Time-domain Signal')
    fig1.show()

    # FFT Magnitude plot
    fig2 = go.Figure()
    fig2.add_trace(go.Scatter(x=xf, y=yf, mode='lines', name='FFT'))
    fig2.add_trace(go.Scatter(x=xf[idx_start:idx_stop+1], y=yf[idx_start:idx_stop+1
    fig2.update_layout(title=f'FFT Magnitude - Integrated Simpsons Area Manual: {in
    fig2.show()
```

In [ ]:

```
def main():
    # Constants
    TIMESTEP = 0.01
    NUMSAMPLES = 1000
    MOD_FREQ_HZ = 40
    CHANNEL_SEPARATION_HZ = 40

    # Main script execution
    signal = generate_signal(TIMESTEP, NUMSAMPLES)
    visualize_signal(signal, TIMESTEP, MOD_FREQ_HZ, CHANNEL_SEPARATION_HZ)

main()
```

Now I have a distinct region to look for singnal for a given frequency. I could add another channel with a different modulation frequency now!