EQUIPO DE MEDICIÓN DE ENERGÍA
ETAPA 2

Fernando Fortunati
e-mail: nfortunati@gmail.com

Sebastian Martínez
e-mail: sebajm1363@gmail.com

RESUMEN: *Se aplicarán los conceptos aprendidos en la cátedra Procesamiento Embebido de Señales para el diseño y realización de un medidor de energía.*

PALABRAS CLAVE: Calidad de energía, True RMS, THD, FFT, Ventana Hamming, Raspberry Pi Pico 2, Python, Tkinter.

1 INTRODUCCIÓN

El objetivo de la segunda etapa de este proyecto consiste en implementar el muestreo de una señal —de cualquier tipo— utilizando un microcontrolador **Raspberry Pi Pico 2**, para posteriormente realizar el **procesamiento digital** de dicha señal.

El procesamiento incluye la aplicación de la **Transformada Rápida de Fourier (FFT)** con el fin de identificar las componentes armónicas presentes y calcular, a partir de ellas, el **porcentaje de distorsión armónica total (THD)**. Asimismo, se determina el **valor eficaz (TRMS)** de la señal.

Una vez finalizado el procesamiento, el sistema transmite a través de la interfaz **UART** los datos obtenidos: las muestras temporales, los pares de valores **amplitud–frecuencia** correspondientes al espectro de la FFT, y los valores calculados de **TRMS** y **THD**.

Estos datos son luego recibidos por una aplicación en la computadora, donde se representan gráficamente tanto la señal como su espectro en frecuencia, junto con los valores característicos calculados.

2 ALCANCE

Se propone realizar el procesamiento de una señal de forma embebida en un microcontrolador, para determinar parámetros característicos, tales como el verdadero valor eficaz (True RMS) y la distorsión armónica total (THD).

Para el cálculo de la THD, se aplicará una Transformada Rápida de Fourier (FFT) a la señal previamente muestreada, obteniendo así los componentes espectrales (10 armonicos) necesarios para la estimación del valor mencionado.

Posteriormente, los valores correspondientes a la señal muestreada, los parámetros utilizados en la Transformada Rápida de Fourier (FFT), así como los resultados del cálculo de True RMS y del índice de Distorsión Armónica Total (THD) se enviaran por protocolo a un computadora, para asi poder verlos representados.

Para ello se diseñara una interfaz en Python, utilizando la librería Tkinter; de forma de poder ver los graficos de la señal y del espectro de la misma, asi como también los valores antes mencionados.

Esta interfaz, aparte de funcionar para realizar la visualización; tambien permitira cambiar la frecuencia de muestreo en el microcontrolador y descargar los valores a un archivo CSV.

3 MARCO TEÓRICO

3.1 Muestreo:

El muestreo consiste en convertir una señal continua en una secuencia discreta de valores tomados en instantes periódicos. Según el Teorema de Nyquist, para evitar aliasing, la frecuencia de muestreo f_s debe ser al menos el doble de la máxima frecuencia presente en la señal ($f_s \geq 2f_{max}$).

Para la selección de la frecuencia de muestreo se seguiran los siguientes lineamientos:

- Definir hasta que armónico se deseara mostrar en la FFT.
- Con la frecuencia máxima que se desea ver en el espectro, se seleccionara la frecuencia de muestreo de acuerdo al criterio de Nyquist.
- Se debera adecuar este valor de frecuencia para que sea potencia de 2, de esta forma evitamos el leakage.

EJEMPLO:

Fundamental: 50 Hz

Cantidad de armónicos: 10

Frecuencia Máxima (Fmax): 10 * 50 Hz = 500 Hz

Frecuencia de Muestreo (Fs): 2 * Fmax = 1000 Hz

Fs corregida: 1024 Hz (exactamente 20 ciclos)

En la práctica, el muestreo no se realiza con impulsos ideales, sino con un tren de pulsos rectangulares, lo que introduce un efecto adicional: la señal muestreada se convoluciona con una función sinc en frecuencia, debido al ancho finito de los pulsos.

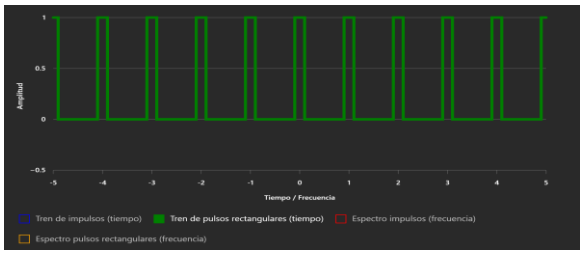


Figura 1: Tren de pulsos rectangulares

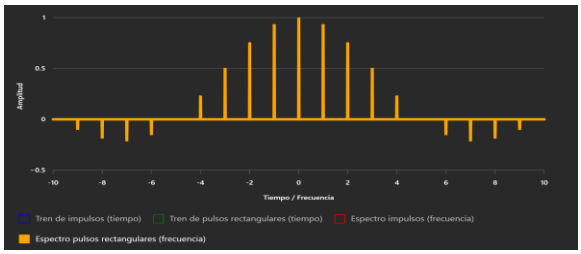


Figura 2: Respuesta en Frecuencia

3.2 Ventana de Hanning:

La **Ventana de Hanning** (o Hann) es una función utilizada en procesamiento digital de señales para reducir el **efecto de fuga espectral** cuando se aplica la Transformada de Fourier a señales finitas. En lugar de cortar la señal abruptamente, se multiplica por una ventana que suaviza los extremos, disminuyendo las discontinuidades.

Su fórmula es:

$$\omega[n] = 0{,}5 * \left(1 - cos\left(\frac{2\pi n}{N-1}\right)\right), 0 \leq n \leq N-1$$

Características:

- Reduce la fuga espectral mejor que una ventana rectangular.
- Atenúa los extremos de la señal a cero.
- Se usa en análisis espectral y filtrado.

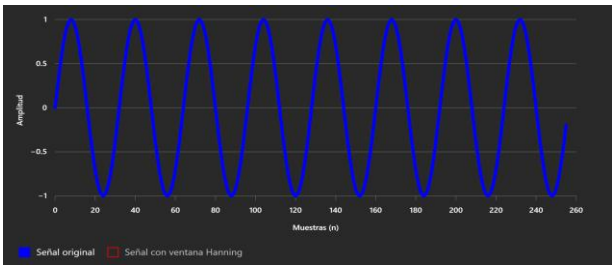


Figura 3: Señal Senoidal (Sin aplicarle la ventana)

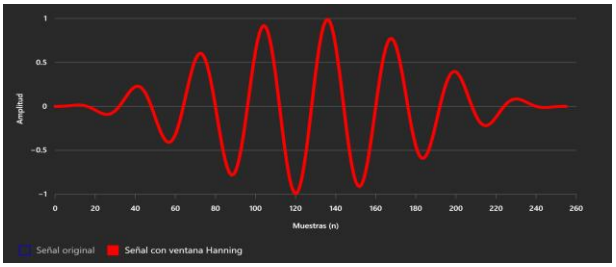


Figura 4: Señal Senoidal (Ventana Aplicada)

Para mas claridad se analizan ahora, los espectros antes y después de aplicarle la ventana:

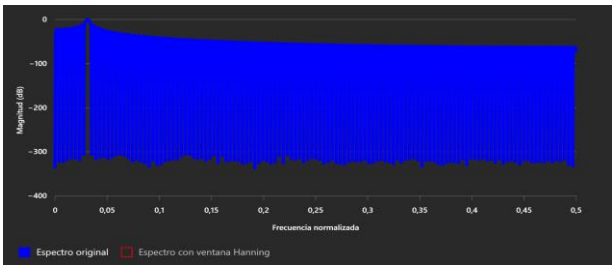


Figura 5: Espectro de la Señal Senoidal (sin ventana)

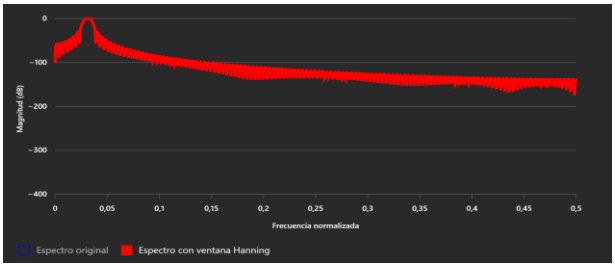


Figura 6: Espectro de la Señal Senoidal (Con ventana)

Como se puede apreciar, cuando no se aplica la ventana hay un alto nivel de componentes de alta frecuenia que no pertencen a la señal original. Aunque el pico principal está en la frecuencia correcta, la energía se dispersa en frecuencias adyacentes, dificultando la interpretación.

En cambio, luego de aplicar la ventana, se aprecia una atenuacion de las componentes de alta frecuencia. Esto se debe a que se suavizaron los extremos de la señal (Figura 4), lo que disminuye la Fuga Espectral. El resultado es un espectro más limpio, con menos energía fuera de la frecuencia principal.

3.3 Transformada Rápida Fourier (FFT):

La Transformada Rápida de Fourier (FFT) es un algoritmo eficiente para calcular la Transformada Discreta de Fourier (DFT), que convierte una señal del dominio del tiempo al dominio de la frecuencia. Esto permite analizar qué componentes frecuenciales forman la señal.

La DFT se define como:

$$X[k] = \sum_{n=0}^{N-1} x[n] * e^{-j2\pi kn/N} \ , k = 0, 1, 2, \dots, N-1$$

Diferencias:

- La DFT directa requiere $O(N^2)$operaciones para Nmuestras.
- La FFT reduce esto a $O(N\log\ N)$, lo que permite análisis rápido incluso para señales largas.
- El punto anterior, en el caso de sistemas embebidos. Mejora la velocidad de procesamiento y la utilizacion de recursos.

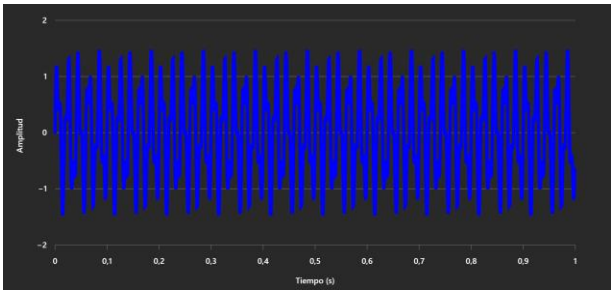


Figura 7: Señales senoidales de 50 y 120 Hz

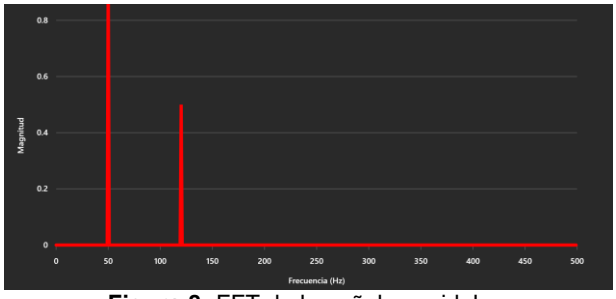


Figura 8: FFT de la señal senoidal

3.4 Distorsión Armónica Total (THD):

La THD (Total Harmonic Distortion), es una medida que indica cuánto una señal eléctrica (normalmente de corriente o tensión) se desvía de una onda senoidal pura debido a la presencia de armónicos.

En otras palabras, muestra el nivel de distorsión introducido por componentes de frecuencia múltiple de la fundamental (2°, 3°, 4°, etc.).

Matemáticamente se define como:

$$THD = \frac{\sqrt{V_2^2 + V_3^2 + V_4^2 + \cdots}}{V_1} \ x\ 100$$

donde:

- V_1= valor RMS del componente fundamental
- V_2, V_3, V_4, ...= valores RMS de los armónicos

Interpretación práctica:

- Una THD baja (por ejemplo < 5%) indica una señal casi senoidal, con buena calidad.
- Una THD alta implica que la señal contiene muchos armónicos, lo que puede causar calentamiento, interferencias y pérdida de eficiencia en equipos eléctricos y electrónicos.

4 DESARROLLO

Como se mencionó en este documento, se realizaran dos códigos para poder realizar la tarea que nos proponemos.

El primero se realizara en MicroPython para poder correrlo en un microcontrolador del tipo Raspberry Pi Pico 2.

Código Raspberry Pi Pico 2 (MicroPython):

El bucle principal correra las distintas funciones del microcontrolador, como se ve a continuación:

```
# ====================================================
#                  BUCLE PRINCIPAL
# ====================================================
while True:
    recibir_fs()

    signal = muestrear()
    if signal is None:
        continue

    armonicos, Vrms, THD, f1 = procesar(signal)
    imprimir(armonicos, Vrms, THD, f1)
    OLED(f1, Vrms, THD)
    enviar_trama((signal-1.65), armonicos, Vrms, THD)

    time.sleep(2)
```

El proceso de muestreo se realizara mediante la función muestrear, donde se muestrea la señal por medio del timer interno del micro. A su vez, se convierte los valores muestreados a valores de tensión en Volts.

```
# ======================================================
#                 MUESTREO SINCRONIZADO
# ======================================================
def muestrear():
    signal = []
    sampling_done = False
    start_time = time.ticks_ms()

    # ---------- Esperar cruce por cero ----------
    esperar_cruce_cero()

    def sample_adc(timer):
        nonlocal signal, sampling_done
        if len(signal) < N:
            val = ((adc.read_u16() * 3.3) / 65535)
            signal.append(val)
        else:
            timer.deinit()
            sampling_done = True

    timer = Timer()
    timer.init(freq=fs, mode=Timer.PERIODIC, callback=sample_adc)

    while not sampling_done:
        time.sleep_ms(1)
        if time.ticks_diff(time.ticks_ms(), start_time) > 5000:
            print("Error: muestreo no completado en tiempo esperado")
            timer.deinit()
            return None

    return np.array(signal)
```

Si el muestreo finaliza de forma satisfactoria, a continuación se procesan las muestras para asi obtener los armonicos de la señal, el True RMS y el valor de la THD.

Algunas de las operaciónes que se realizan en esta funcion son:

- Ventana de Hanning:

```
#Ventana de Hanning
n = np.arange(N)
window = 0.5 - 0.5 * np.cos(2 * np.pi * n / (N - 1))
signal_windowed = signal * window
correction_factor = 1 / (np.sum(window) / N)
```

- Calculo de la FFT:

```
#Calculo de la FFT
spectrum = np.fft.fft(signal_windowed)
frequencies = np.linspace(0, fs, N)
magnitudes = np.sqrt(spectrum.real**2 + spectrum.imag**2)
amplitudes = correction_factor * (2 / N) * magnitudes

freqs_pos = frequencies[:N//2]
amplitudes_pos = amplitudes[:N//2]
```

- Calculo de THD:

```
#Filtro los primeros 10 armonicos reales
arm_temp = []

for fr, a in armonicos:
    if f1 <= 0:
        continue

    n = round(fr / f1)
    if 1 <= n <= 10:
        if abs(fr - n*f1) <= f1 * 0.10:  # tolerancia 10%
            arm_temp.append((n, fr, a))

arm_temp.sort(key=lambda x: x[0])
armonicos_ordenados = [(fr, a) for (n, fr, a) in arm_temp[:10]]

# THD con los 10 armónicos
suma = 0
for fr, a in armonicos_ordenados[1:]:
    suma += a*a
THD = (np.sqrt(suma) / (a1 + 1e-12)) * 100
```

Para terminar la funcionalidad de este codigo, se envian los datos recopilados de las función “procesar()” para enviarlos por UART y asi poder visualizarlos en una interfaz gráfica en la computadora.

```
# --- Enviar trama ---
def enviar_trama(signal, armonicos_ordenados, Vrms, THD):
    try:
        frame = bytearray()
        frame.extend(HEADER)

        # Frecuencia de muestreo
        frame.extend(struct.pack('<I', fs))

        # Número de muestras
        frame.extend(struct.pack('<H', N))

        # Muestras en int16
        samples_int16 = [max(min(int((v / 3.3) * 32767), 32767), -32767) for v in signal]
        for s in samples_int16:
            frame.extend(struct.pack('<h', s))

        # Armónicos
        M = len(armonicos_ordenados)
        frame.extend(struct.pack('<H', M))
        for f, a in armonicos_ordenados:
            frame.extend(struct.pack('<ff', f, a))

        # RMS y THD
        frame.extend(struct.pack('<f', Vrms))
        frame.extend(struct.pack('<f', THD))

        # Calcular CRC desde el byte 4 en adelante
        crc = binascii.crc32(frame[4:]) & 0xFFFFFFFF
        frame.extend(struct.pack('<I', crc))

        MAX_CHUNK = 512  # Tamaño seguro por bloque
        if len(frame) > MAX_CHUNK:
            print("Enviando trama en bloques...")
            for i in range(0, len(frame), MAX_CHUNK):
                uart.write(frame[i:i+MAX_CHUNK])
                time.sleep_ms(10)  # Pequeña pausa para evitar saturación
        else:
            uart.write(frame)

    except Exception as e:
        print("Error en enviar_trama:", e)
```

Código Interfaz Gráfica (Python):

En este programa se implementa una interfaz gráfica en Python, desarrollada con **Tkinter** y **Matplotlib**, que permite la visualización en tiempo real de señales adquiridas desde el microcontrolador a través de un puerto serie. La aplicación recibe los datos en formato binario, los decodifica y valida mediante un **CRC (Cyclic Redundancy Check)** para garantizar la integridad de la transmisión.

Una vez procesados, el sistema representa simultáneamente:

- La señal temporal adquirida, correspondiente a la forma de onda que ingresa al circuito.
- El espectro en frecuencia obtenido a partir del procesamiento FFT (Transformada Rápida de Fourier), que permite observar las componentes armónicas presentes.

Además, el programa muestra de forma dinámica los valores del voltaje eficaz (TRMS) y del porcentaje de distorsión armónica total (THD) calculados en el microcontrolador, actualizando estos parámetros a medida que llegan nuevas tramas de datos.

La interfaz incluye controles para:

- Seleccionar y conectar el puerto serie de comunicación.
- Pausar y reanudar la actualización de gráficos.
- Enviar una frecuencia deseada al dispositivo remoto.
- Guardar los datos en formato CSV para su posterior análisis.

En conjunto, esta herramienta cumple la función de monitorear y analizar señales eléctricas en tiempo real, integrando la adquisición desde hardware embebido con un entorno gráfico de escritorio de fácil interpretación.

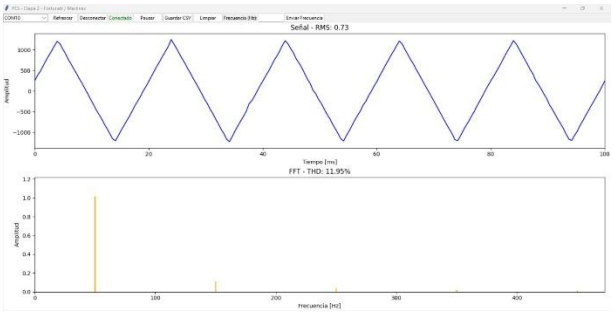


Figura 9: Interfaz Gráfica

5 CONCLUSIONES

Se logró cumplir con el objetivo propuesto de implementar el programa desarrollado en la Etapa 1 dentro de un microcontrolador. Durante el proceso se presentaron algunos inconvenientes técnicos que requirieron ajustes específicos.

La versión estándar de **MicroPython**, obtenida desde su sitio oficial, incluye la librería **ulab** para operaciones numéricas, aunque esta versión no incorpora las funciones de **FFT** necesarias para el análisis espectral. Debido a esta limitación, fue necesario **compilar un firmware personalizado**, adaptado a los requerimientos del proyecto e incorporando dichas funcionalidades.

Por otro lado, una desventaja importante del uso de MicroPython en este tipo de aplicaciones es la **imposibilidad de aprovechar las capacidades del DMA (Direct Memory Access)** para la adquisición de muestras, dado que esta característica no está disponible en el entorno de ejecución de MicroPython.

Si bien el sistema implementado presenta un funcionamiento correcto y cumple con las especificaciones funcionales, se identificó que **una versión equivalente desarrollada en lenguaje C podría ofrecer un desempeño significativamente superior**, ya que permitiría el uso del DMA para optimizar la velocidad y eficiencia del proceso de muestreo.

6 BIBLIOGRAFÍA

[1] Documentación MicroPython [En Línea]
https://micropython.org/download/RPI_PICO2/

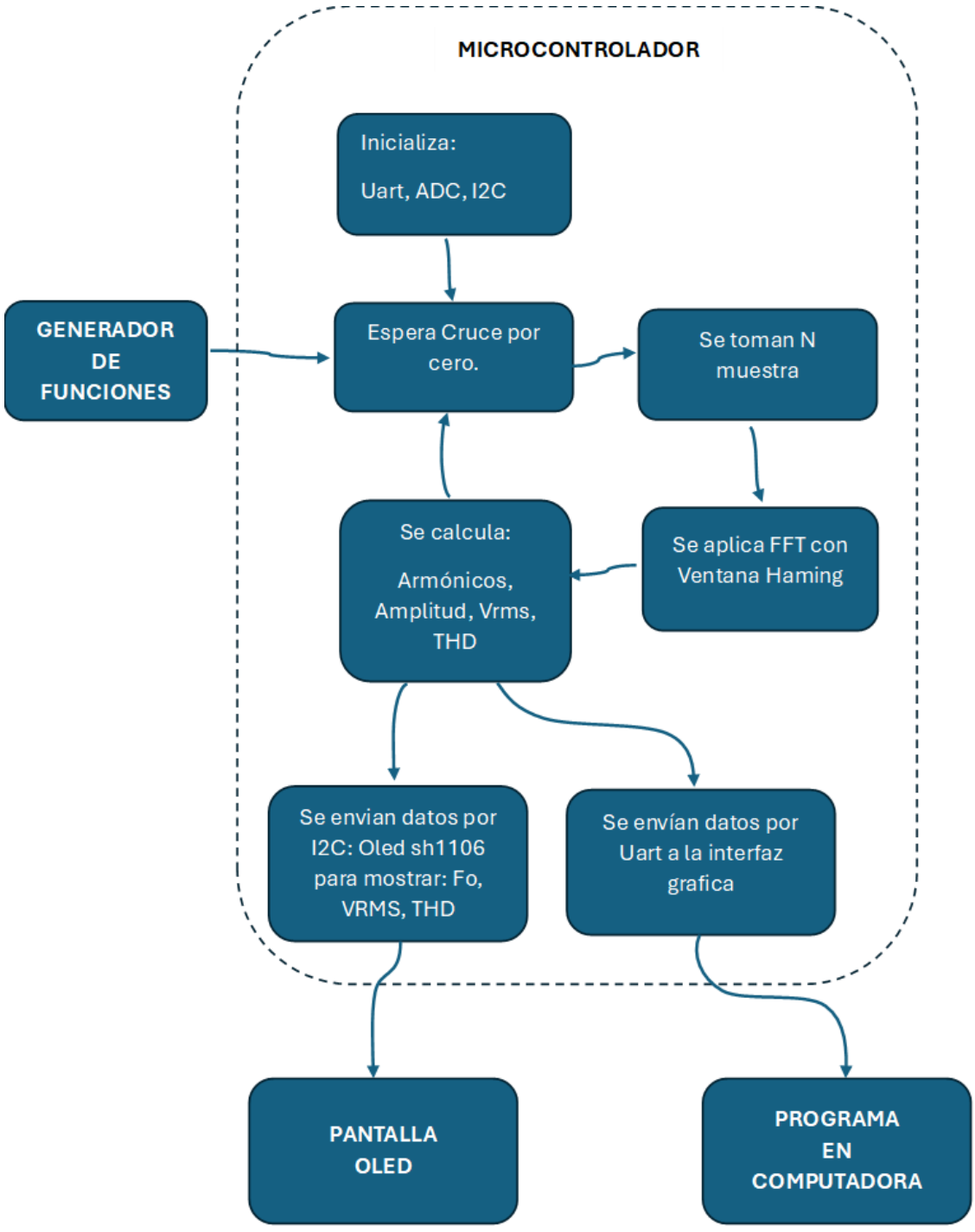
[2] Documentación Librería Ulab [En Linea]
https://github.com/v923z/micropython-ulab

[3] Documentación Librería Tkinter [En Linea]
https://docs.python.org/es/3/library/tkinter.html

[4] "Análisis Espectral" – Apunte de clase para Procesamiento Embebido de Señales – Facultad Regional Avellaneda, Universidad Tecnológica Nacional

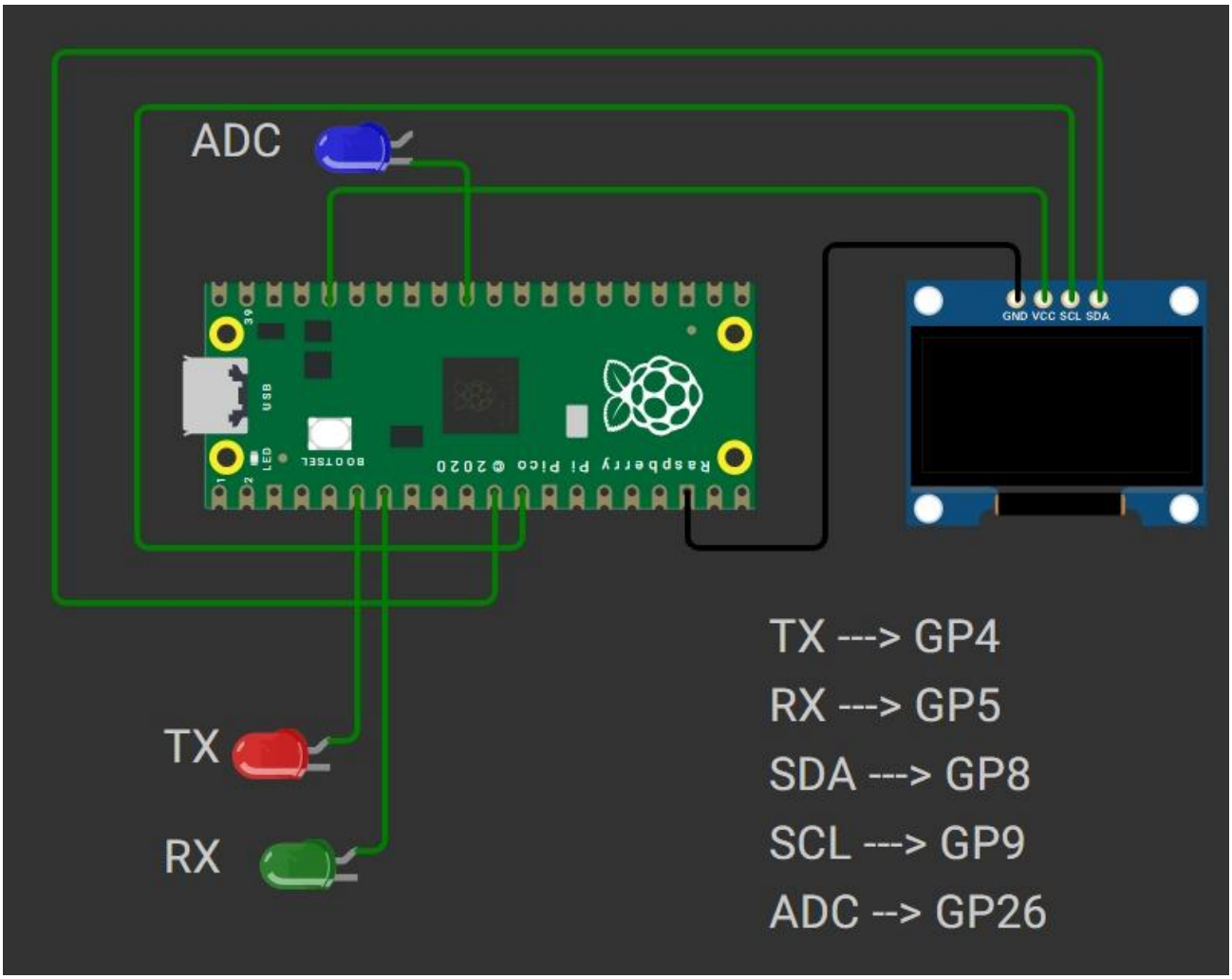
ANEXO I

Diagrama en Bloques



ANEXO II

Esquema de conexión:



ANEXO III

Código Raspberry Pi Pico 2 (MicroPython)

```
from ulab import numpy as np
from machine import UART, Pin, ADC, Timer, I2C
import struct, binascii, time
from sh1106 import SH1106_I2C

# --- Configuración OLED ---
WIDTH, HEIGHT = 128, 64
i2c = I2C(0, scl=Pin(9), sda=Pin(8), freq=100000)
oled = SH1106_I2C(WIDTH, HEIGHT, i2c)
oled_enabled = True

# --- UART ---
uart = UART(1, baudrate=115200, tx=Pin(4), rx=Pin(5))

# --- ADC ---
adc = ADC(Pin(26))

# --- Parámetros ---
fs = 1024
N = 1024
HEADER = b'PICO'

# ======================================================
#   DETECCIÓN DE CRUCE POR CERO ASCENDENTE (VCC/2)
# ======================================================
def esperar_cruce_cero():
    UMBRAL = 1.65   # Punto medio VCC/2
    prev = (adc.read_u16() * 3.3) / 65535

    while True:
        actual = (adc.read_u16() * 3.3) / 65535

        # Cruce ascendente: antes abajo del medio y ahora arriba
        if prev < UMBRAL and actual >= UMBRAL:
            return

        prev = actual
        time.sleep_us(50)   # Lectura rápida


# --- Función recepción frecuencia de muestreo ---
def recibir_fs():
    global fs
    if uart.any() >= 4:
        data = uart.read(4)
        if data and len(data) == 4:
            new_fs = struct.unpack('<I', data)[0]
            if 10 <= new_fs <= 20000:
                fs = new_fs
                print("Nueva frecuencia de muestreo:", fs)
                if fs > 10000:
                    print("Advertencia: fs demasiado alta, ajustando a 10000 Hz")
                    fs = 10000
            else:
                print("Frecuencia fuera de rango:", new_fs)
```

```
# ======================================================
#               MUESTREO SINCRONIZADO
# ======================================================
def muestrear():
    signal = []
    sampling_done = False
    start_time = time.ticks_ms()

    # ---------- Esperar cruce por cero ----------
    esperar_cruce_cero()

    def sample_adc(timer):
        nonlocal signal, sampling_done
        if len(signal) < N:
            val = ((adc.read_u16() * 3.3) / 65535)
            signal.append(val)
        else:
            timer.deinit()
            sampling_done = True

    timer = Timer()
    timer.init(freq=fs, mode=Timer.PERIODIC, callback=sample_adc)

    while not sampling_done:
        time.sleep_ms(1)
        if time.ticks_diff(time.ticks_ms(), start_time) > 5000:
            print("Error: muestreo no completado en tiempo esperado")
            timer.deinit()
            return None

    return np.array(signal)


# ======================================================
#               PROCESAMIENTO FFT + ARMÓNICOS
# ======================================================
def procesar(signal):
    signal = signal - np.mean(signal)

    #Ventana de Hanning
    n = np.arange(N)
    window = 0.5 - 0.5 * np.cos(2 * np.pi * n / (N - 1))
    signal_windowed = signal * window
    correction_factor = 1 / (np.sum(window) / N)

    #Calculo de la FFT
    spectrum = np.fft.fft(signal_windowed)
    frequencies = np.linspace(0, fs, N)
    magnitudes = np.sqrt(spectrum.real**2 + spectrum.imag**2)
    amplitudes = correction_factor * (2 / N) * magnitudes

    freqs_pos = frequencies[:N//2]
    amplitudes_pos = amplitudes[:N//2]

    # Pico detection
    umbral = 0.01 * np.max(amplitudes_pos)
    armonicos = []

    for i in range(1, len(amplitudes_pos) - 1):
        if (amplitudes_pos[i] > amplitudes_pos[i - 1] and
            amplitudes_pos[i] > amplitudes_pos[i + 1] and
            amplitudes_pos[i] > umbral):
            armonicos.append((freqs_pos[i], amplitudes_pos[i]))
```

```
    if len(armonicos) > 0:
        armonicos.sort(key=lambda x: x[1], reverse=True)
        f1, a1 = armonicos[0]
    else:
        f1, a1 = 0.0, 0.0

    #Calculo de TRMS
    Vrms = np.sqrt(np.mean(signal ** 2))

    #Filtro los primeros 10 armonicos reales
    arm_temp = []

    for fr, a in armonicos:
        if f1 <= 0:
            continue

        n = round(fr / f1)
        if 1 <= n <= 10:
            if abs(fr - n*f1) <= f1 * 0.10:  # tolerancia 10%
                arm_temp.append((n, fr, a))

    arm_temp.sort(key=lambda x: x[0])
    armonicos_ordenados = [(fr, a) for (n, fr, a) in arm_temp[:10]]

    # THD con los 10 armónicos
    suma = 0
    for fr, a in armonicos_ordenados[1:]:
        suma += a*a
    THD = (np.sqrt(suma) / (a1 + 1e-12)) * 100

    return armonicos_ordenados, Vrms, THD, f1


# --- Imprimir resultados ---
def imprimir(armonicos_ordenados, Vrms, THD, f1):
    print("\n==============================")
    print(f"Frecuencia fundamental: {f1:.0f} Hz")
    print(f"Vrms: {Vrms:.3f} V")
    print(f"THD: {THD:.2f} %")
    print("------------------------------")
    print("ARMÓNICOS (max 10):")
    for i, (fr, a) in enumerate(armonicos_ordenados):
        print(f"{i+1:>2d}: {fr:>9.0f} Hz   {a:>9.4f} V")


# --- Enviar trama ---
def enviar_trama(signal, armonicos_ordenados, Vrms, THD):
    try:
        frame = bytearray()
        frame.extend(HEADER)

        # Frecuencia de muestreo
        frame.extend(struct.pack('<I', fs))

        # Número de muestras
        frame.extend(struct.pack('<H', N))

        # Muestras en int16
        samples_int16 = [max(min(int((v / 3.3) * 32767), 32767), -32767) for v in signal]
        for s in samples_int16:
            frame.extend(struct.pack('<h', s))
```

```
        # Armónicos
        M = len(armonicos_ordenados)
        frame.extend(struct.pack('<H', M))
        for f, a in armonicos_ordenados:
            frame.extend(struct.pack('<ff', f, a))

        # RMS y THD
        frame.extend(struct.pack('<f', Vrms))
        frame.extend(struct.pack('<f', THD))

        # Calcular CRC desde el byte 4 en adelante
        crc = binascii.crc32(frame[4:]) & 0xFFFFFFFF
        frame.extend(struct.pack('<I', crc))

        MAX_CHUNK = 512  # Tamaño seguro por bloque
        if len(frame) > MAX_CHUNK:
            print("Enviando trama en bloques...")
            for i in range(0, len(frame), MAX_CHUNK):
                uart.write(frame[i:i+MAX_CHUNK])
                time.sleep_ms(10)  # Pequeña pausa para evitar saturación
        else:
            uart.write(frame)

    except Exception as e:
        print("Error en enviar_trama:", e)


# --- OLED ---
def OLED(f1, Vrms, THD):
    oled.fill(0)
    oled.text("f1={:.0f}Hz".format(f1), 0, 0)
    oled.text("Vrms={:.3f}V".format(Vrms), 0, 20)
    oled.text("THD={:.2f}%".format(THD), 0, 40)
    oled.show()


# ======================================================
#                 BUCLE PRINCIPAL
# ======================================================
while True:
    recibir_fs()

    signal = muestrear()
    if signal is None:
        continue

    armonicos, Vrms, THD, f1 = procesar(signal)
    imprimir(armonicos, Vrms, THD, f1)
    OLED(f1, Vrms, THD)
    enviar_trama((signal-1.65), armonicos, Vrms, THD)

    time.sleep(2)
```

ANEXO IV

Código Interfaz Gráfica (Python):

```
import math
import tkinter as tk
from tkinter import ttk, filedialog, messagebox
import threading, queue, struct, binascii, serial, serial.tools.list_ports, csv
import numpy as np
import matplotlib
matplotlib.use('TkAgg')
from matplotlib.figure import Figure
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2Tk

fs = 1024          # Frecuencia de muestreo por defecto (Hz)
HEADER = b'PICO'   # Cabecera del paquete binario recibido por UART


# Clase encargada de la lectura asíncrona del puerto serie

class SerialReader(threading.Thread):
    def __init__(self, ser, data_queue):
        super().__init__(daemon=True)
        self.ser = ser
        self.data_queue = data_queue
        self.running = True
        self.buffer = b''  # Buffer temporal para armar paquetes completos

    def run(self):
        #Hilo principal de lectura del puerto serie
        while self.running:
            try:
                data = self.ser.read(self.ser.in_waiting or 1)
                if data:
                    self.buffer += data
                    self._process_buffer()
            except Exception:
                break

    def _process_buffer(self):
        #Analiza el buffer, valida el CRC y extrae muestras + FFT + RMS + THD
        while True:
            header_index = self.buffer.find(HEADER)
            if header_index == -1 or len(self.buffer) < header_index + 4:
                return  # No hay encabezado completo
            self.buffer = self.buffer[header_index:]

            try:
                # Verifica que haya suficientes bytes para los campos fijos
                if len(self.buffer) < 4 + 4 + 2:
                    return

                # Decodificación del encabezado del paquete
                fs = struct.unpack_from('<I', self.buffer, 4)[0]
                n_samples = struct.unpack_from('<H', self.buffer, 8)[0]

                # Calcula posiciones de cada sección del paquete
                samples_start = 10
                samples_end = samples_start + n_samples * 2
                if len(self.buffer) < samples_end + 2:
                    return

                m_samples = struct.unpack_from('<H', self.buffer, samples_end)[0]
                fft_start = samples_end + 2
                fft_end = fft_start + m_samples * 8
                rms_start = fft_end
                thd_start = rms_start + 4
                crc_start = thd_start + 4
                expected_total_len = crc_start + 4
```

```
                if len(self.buffer) < expected_total_len:
                    return  # Paquete incompleto

                # Extrae el paquete completo
                payload = self.buffer[:expected_total_len]
                self.buffer = self.buffer[expected_total_len:]

                # --- Validación del CRC ---
                crc_recv = struct.unpack_from('<I', payload, crc_start)[0]
                crc_calc = binascii.crc32(payload[4:crc_start]) & 0xffffffff
                if crc_recv != crc_calc:
                    continue  # Se descarta si el CRC no coincide

                # --- Decodificación de los datos binarios ---
                samples = (np.frombuffer(payload[samples_start:samples_end], dtype=np.int16))/10
                fft_data = payload[fft_start:fft_end]
                fft_pairs = struct.iter_unpack('<ff', fft_data)
                freqs, amps = zip(*fft_pairs) if m_samples > 0 else ([], [])
                rms = struct.unpack_from('<f', payload, rms_start)[0]
                thd = struct.unpack_from('<f', payload, thd_start)[0]

                # Envía los datos procesados al hilo principal mediante la cola
                self.data_queue.put((samples, np.array(freqs), np.array(amps), rms, thd, fs))
            except Exception:
                return

    def stop(self):
        """Detiene el hilo de lectura"""
        self.running = False


# Clase principal de la aplicación (interfaz gráfica + lógica)

class PicoFFTApp:
    def __init__(self, root):
        self.root = root
        self.root.title("PES - Etapa 2 - Fortunati / Martinez")
        self.data_queue = queue.Queue()    # Cola para recibir datos del hilo serie
        self.serial_thread = None
        self.ser = None
        self.connected = False
        self.paused = False

        # Variables de datos
        self.samples = np.array([])
        self.fft_freqs = np.array([])
        self.fft_amps = np.array([])
        self.rms = 0
        self.thd = 0
        self.annotation = None

        # Construcción de la UI
        self._build_ui()

        # Eventos de ventana y actualización periódica
        self.root.bind('<Configure>', self._on_resize)
        self.root.after(50, self.update_plot_loop)

    # Construcción de la interfaz gráfica

    def _build_ui(self):
        self.root.rowconfigure(1, weight=1)
        self.root.columnconfigure(0, weight=1)

        # --- Barra superior con controles ---
        top_frame = ttk.Frame(self.root)
        top_frame.grid(row=0, column=0, sticky="ew")

        # Selección de puerto serie y botones de control
        self.port_cb = ttk.Combobox(top_frame, width=15)
```

```
        self.port_cb.grid(row=0, column=0)
        ttk.Button(top_frame, text="Refrescar", command=self._populate_ports).grid(row=0, column=1)
        self.connect_btn = ttk.Button(top_frame, text="Conectar", command=self.toggle_connection)
        self.connect_btn.grid(row=0, column=2)
        self.status_lbl = ttk.Label(top_frame, text="Desconectado", foreground="red")
        self.status_lbl.grid(row=0, column=3)
        self.pause_btn = ttk.Button(top_frame, text="Pausar", command=self.toggle_pause)
        self.pause_btn.grid(row=0, column=4)
        ttk.Button(top_frame, text="Guardar CSV", command=self.save_csv).grid(row=0, column=5)
        ttk.Button(top_frame, text="Limpiar", command=self.clear_graphs).grid(row=0, column=6)

        # Campo para enviar frecuencia al microcontrolador
        ttk.Label(top_frame, text="Frecuencia (Hz):").grid(row=0, column=7)
        self.freq_entry = ttk.Entry(top_frame, width=10)
        self.freq_entry.grid(row=0, column=8)
        ttk.Button(top_frame, text="Enviar Frecuencia", command=self.enviar_frecuencia).grid(row=0, column=9)

        # --- Frame para gráficos (señal y FFT) ---
        plot_frame = ttk.Frame(self.root)
        plot_frame.grid(row=1, column=0, sticky="nsew")
        plot_frame.rowconfigure(0, weight=1)
        plot_frame.columnconfigure(0, weight=1)

        # Figura con dos subgráficos
        self.fig = Figure(figsize=(6, 4), constrained_layout=True)
        self.ax_time = self.fig.add_subplot(211)
        self.ax_fft = self.fig.add_subplot(212)
        self.ax_time.set_title("Señal en el tiempo")
        self.ax_fft.set_title("FFT")
        self.ax_time.set_xlabel("Tiempo [ms]")
        self.ax_time.set_ylabel("Amplitud")
        self.ax_fft.set_xlabel("Frecuencia [Hz]")
        self.ax_fft.set_ylabel("Amplitud")

        # Inserta la figura en el GUI
        self.canvas = FigureCanvasTkAgg(self.fig, master=plot_frame)
        self.canvas.get_tk_widget().grid(row=0, column=0, sticky="nsew")

        # Barra de herramientas de Matplotlib
        toolbar_frame = ttk.Frame(self.root)
        toolbar_frame.grid(row=2, column=0, sticky="ew")
        self.toolbar = NavigationToolbar2Tk(self.canvas, toolbar_frame)
        self.toolbar.update()

        # Evento del mouse para mostrar info en el espectro
        self.canvas.mpl_connect("motion_notify_event", self._on_mouse_move)

        # Carga inicial de los puertos disponibles
        self._populate_ports()

    # Funciones de comunicación serie

    def _populate_ports(self):
        #Actualiza la lista de puertos serie disponibles
        ports = [p.device for p in serial.tools.list_ports.comports()]
        self.port_cb['values'] = ports
        if ports:
            self.port_cb.current(0)

    def toggle_connection(self):
        #Conecta o desconecta el puerto serie
        if not self.connected:
            try:
                # Apertura del puerto serie
                self.ser = serial.Serial(self.port_cb.get(), 115200, timeout=1)
                self.serial_thread = SerialReader(self.ser, self.data_queue)
                self.serial_thread.start()
                self.connected = True
                self.status_lbl.config(text="Conectado", foreground="green")
                self.connect_btn.config(text="Desconectar")
```

```
            except Exception as e:
                messagebox.showerror("Error", str(e))
        else:
            # Cierre ordenado de la conexión
            if self.serial_thread:
                self.serial_thread.stop()
            if self.ser:
                self.ser.close()
            self.connected = False
            self.status_lbl.config(text="Desconectado", foreground="red")
            self.connect_btn.config(text="Conectar")

    def enviar_frecuencia(self):
        #Envía una frecuencia al microcontrolador vía UART (ajustada a potencia de 2)
        if not self.connected or not self.ser:
            messagebox.showwarning("Aviso", "Debe conectar primero el dispositivo.")
            return
        try:
            frecuencia = int(self.freq_entry.get())
            if frecuencia <= 0:
                messagebox.showerror("Error", "La frecuencia debe ser mayor que 0.")
                return

            # Ajuste a la potencia de 2 más cercana
            potencia = round(math.log2(frecuencia))
            frecuencia_corregida = 2 ** potencia

            # Envío binario (4 bytes, formato little-endian)
            data = struct.pack('<I', frecuencia_corregida)
            self.ser.write(data)

            messagebox.showinfo("Éxito",
                                f"Frecuencia corregida enviada: {frecuencia_corregida} Hz "
                                f"(original: {frecuencia} Hz)")
        except ValueError:
            messagebox.showerror("Error", "Ingrese un valor numérico válido.")


    # Funciones de control y gráficos

    def toggle_pause(self):
        #Pausa o reanuda la actualización de los gráficos
        self.paused = not self.paused
        self.pause_btn.config(text="Continuar" if self.paused else "Pausar")

    def save_csv(self):
        #Guarda los datos de señal y FFT en un archivo CSV
        if self.samples.size == 0:
            messagebox.showwarning("Aviso", "No hay datos para guardar")
            return
        file_path = filedialog.asksaveasfilename(defaultextension=".csv")
        if file_path:
            with open(file_path, 'w', newline='') as f:
                writer = csv.writer(f)
                writer.writerow(["Muestras"])
                writer.writerow(self.samples.tolist())
                writer.writerow(["Frecuencia", "Amplitud"])
                for freq, amp in zip(self.fft_freqs, self.fft_amps):
                    writer.writerow([freq, amp])
            messagebox.showinfo("Éxito", "Datos guardados en CSV")

    def clear_graphs(self):
        #Limpia los gráficos y reinicia las variables
        self.samples = np.array([])
        self.fft_freqs = np.array([])
        self.fft_amps = np.array([])
        self.rms = 0
        self.thd = 0
        self.ax_time.cla()
        self.ax_fft.cla()
```

```
        self.ax_time.set_title("Señal en el tiempo")
        self.ax_fft.set_title("FFT")
        self.canvas.draw()

    def update_plot_loop(self):
        #Bucle periódico que actualiza los gráficos con los datos recibidos
        if not self.paused and not self.data_queue.empty():
            self.samples, self.fft_freqs, self.fft_amps, self.rms, self.thd, self.fs = self.data_queue.get()
            self._redraw_plots()
        self.root.after(200, self.update_plot_loop)

    def _redraw_plots(self):
        #Actualiza los gráficos de tiempo y FFT con los nuevos datos
        if hasattr(self, 'fs') and self.fs > 0:
            t = (np.arange(len(self.samples)) / self.fs) * 1000
        else:
            t = np.arange(len(self.samples))

        self.ax_time.cla()
        self.ax_fft.cla()

        # Gráfico de la señal
        self.ax_time.plot(t, self.samples, color='blue')
        self.ax_time.set_xlim(0, 100)
        #self.ax_time.set_xlim(t[0], t[-1])
        self.ax_time.set_ylim(min(self.samples) * 1.1,
                              max(self.samples) * 1.1 if max(self.samples) > 0 else 1)
        self.ax_time.set_title(f"Señal - RMS: {self.rms:.2f}")
        self.ax_time.set_xlabel("Tiempo [ms]")
        self.ax_time.set_ylabel("Amplitud")

        # Gráfico del espectro FFT
        self.fft_bar = self.ax_fft.bar(self.fft_freqs, self.fft_amps, color='orange')
        self.ax_fft.set_xlim(0, max(self.fft_freqs) * 1.05 if self.fft_freqs.size > 0 else 1)
        self.ax_fft.set_ylim(0, max(self.fft_amps) * 1.2 if self.fft_amps.size > 0 else 1)
        self.ax_fft.set_title(f"FFT - THD: {self.thd :.2f}%")
        self.ax_fft.set_xlabel("Frecuencia [Hz]")
        self.ax_fft.set_ylabel("Amplitud")

        self.canvas.draw()

    # Eventos de interfaz y visualización

    def _on_resize(self, event):
        #Redibuja la figura al cambiar el tamaño de la ventana
        if hasattr(self, '_resize_pending') and self._resize_pending:
            return
        self._resize_pending = True
        self.root.after(300, self._perform_resize)

    def _perform_resize(self):
        self._resize_pending = False
        self.canvas.draw()

    def _on_mouse_move(self, event):
        #Muestra información de frecuencia y amplitud al pasar el mouse sobre la FFT
        if event.inaxes == self.ax_fft and self.fft_freqs.size > 0:
            x = event.xdata
            if x is None:
                return
            idx = (np.abs(self.fft_freqs - x)).argmin()
            freq = self.fft_freqs[idx]
            amp = self.fft_amps[idx]

            # Quita la anotación anterior
            if self.annotation:
                self.annotation.set_visible(False)

            # Crea una nueva anotación cerca del punto
            self.annotation = self.ax_fft.annotate(
```

```
                f"Freq: {freq:.2f} Hz\nAmp: {amp:.4f}",
                xy=(freq, amp),
                xytext=(freq, amp + max(self.fft_amps) * 0.1),
                arrowprops=dict(facecolor='black', shrink=0.05),
                bbox=dict(boxstyle="round,pad=0.3", fc="yellow", alpha=0.7)
            )
            self.canvas.draw()

# Punto de entrada del programa

if __name__ == "__main__":
    root = tk.Tk()
    app = PicoFFTApp(root)
    root.mainloop()
```

ANEXO IV

Resultados de la medición y comparación con valores esperados:

Señal	Frecuencia fundamental	THD (10 armónicos)	THD	THD (medido)
Cuadrada 50 Hz, 1 Vpp, D=0.3	50 Hz	**71%**	76%	**72.7 %**
Cuadrada 50 Hz, 1 Vpp, D=0.4	50 Hz	**51.2 %**	53.4 %	**51.7 %**
Cuadrada 50 Hz, 1 Vpp, D=0.5	50 Hz	**45.0 %**	48.3 %	**42.7 %**
Cuadrada 50 Hz, 1 Vpp, D=0.6	50 Hz	**51.2 %**	53.4 %	**48.8 %**
Triangular 50 Hz, 1 Vpp, D=0.5	50 Hz	**12.0 %**	12.1 %	**11.96 %**
Suma Senoidal (50 Hz 1 Vpp + 100 Hz 0.25 Vpp)	50 Hz	**25.0 %**	25.0 %	**25%**
Suma Senoidal (50 Hz 1 Vpp + 100 Hz 0.5 Vpp)	50 Hz	**50.0 %**	50.0 %	**50%**
Suma Senoidal (50 Hz 1 Vpp + 100 Hz 0.75 Vpp)	50 Hz	**75.0 %**	75.0 %	**75%**
Suma Senoidal (50 Hz 1 Vpp + 100 Hz 1 Vpp)	50 Hz	**100.0 %**	100.0 %	**100%**