Captura y procesamiento digital de señales junto con Jitter

Luis Carlos Leal Gamboa

est.luis.cleal@unimilitar.edu.co

Docente: José de Jesús Rúgeles

***Resumen*— Este laboratorio analiza el proceso de muestreo de señales analógicas utilizando la Raspberry Pi Pico 2W, comparando dos enfoques de programación: uno basado en temporización activa con sleep_us (código del profesor) y otro con muestreo libre sin sincronización (código del estudiante). Se estudia el efecto del *jitter* en la calidad de la digitalización, la importancia de la ventana de Hanning para reducir *leakage* espectral, y cómo el número de puntos en la FFT afecta la resolución y visualización del espectro. Los resultados muestran que el código del profesor logra menor jitter y mayor precisión espectral, mientras que el código del estudiante, aunque funcional, sufre inestabilidad temporal por las limitaciones de MicroPython. Se concluye que para aplicaciones de precisión, es indispensable migrar a C++ y aprovechar el hardware de la Pico (timers, DMA, buffers).**

***Abstract*— This lab analyzes analog signal sampling using the Raspberry Pi Pico 2W, comparing two programming approaches: one using active timing with sleep_us (professor's code) and another using free-running sampling without synchronization (student's code). The impact of *jitter* on digitization quality, the role of the Hanning window in reducing spectral leakage, and the effect of FFT point count on spectral resolution are studied. Results show the professor's code achieves lower jitter and higher spectral accuracy, while the student's code, though functional, suffers from timing instability due to MicroPython's limitations. It is concluded that for precision applications, migrating to C++ and leveraging the Pico's hardware (timers, DMA, buffers) is essential.**

I. Introducción

El objetivo de este laboratorio es evaluar y comparar dos métodos de adquisición de datos mediante el ADC de la Raspberry Pi Pico 2W: uno sincronizado mediante pausas fijas (sleep_us) y otro basado en muestreo libre sin control temporal. Se busca entender cómo el *jitter*, la ventana de Hanning y el número de puntos de la FFT influyen en la fidelidad de la representación digital de una señal analógica. Las señales de prueba (senoidales de 100 Hz, 200 Hz, 900 Hz y 1800 Hz) se generan externamente y se analizan mediante FFT y Goertzel, registrándose los resultados en archivos para su posterior visualización en MATLAB. Este estudio permite identificar las limitaciones de MicroPython y la necesidad de optimización a bajo nivel para aplicaciones de procesamiento de señales en tiempo real.

II. Parte 1

Se establece una señal en el generador de señales de una señal senoidal de frecuencia de 200Hz, 1,2Vpp y una componente de 1,6 V DC, esta señal se comprueba en el osciloscopio, con el fin de observar y corroborar que la señal obtenida corresponde con lo establecido, como se observa a continuación.

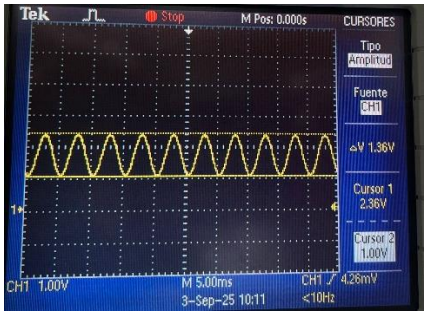

Ilustración 1 Señal senoidal establecida.

Como se observa en la ilustración número uno, la señal obtenida corresponde directamente con lo planteado en el generador de señales, así mismo, manteniendo el nivel DC, a partir de esto, se conecta esta señal al conversor A/D de la raspberry pi pico 2W, y posteriormente al ejecutar el código suministrado por el docente (ADC_testing.py) este código cuenta con las siguientes funciones.

- acquire_data(): Toma N muestras con pausa fija, mide tiempo real, guarda en .txt.
- convert_to_voltage(): Pasa valores ADC (0-65535) a voltaje (0-3.3V).
- remove_offset(): Resta el promedio → elimina DC para no contaminar FFT.
- apply_hanning_window(): Suaviza bordes de la señal con ventana de Hanning.
- fft_manual(): Calcula FFT Radix-2 (rellena con ceros si N_FFT > N).
- analyze_fft(): Saca frecuencia dominante, SNR, ENOB y guarda espectro.
- main(): Ejecuta todo en orden.

¿Para qué sirve la ventana Hanning y cómo se usa?
Sirve para evitar falsas frecuencias en la FFT al suavizar los bordes de la señal. Se aplica multiplicando muestra por muestra la señal (ya sin DC) por la fórmula:

$$w[i] = 0.5*(1 - \cos(2\pi i/(N-1)))$$

A partir de esto se obtiene el valor de la frecuencia de muestreo real, la frecuencia de la señal suministrada, el valor del offset removido, el piso del ruido para la FFT de la señal, y así mismo el valor del SNR.

```
MPY: soft reboot
Iniciando adquisición y análisis...
Frecuencia deseada: 2000 Hz, frecuencia real: 1891.18 Hz
Offset DC removido: 0.727 V
Frecuencia dominante: 199.46 Hz
Amplitud señal: 0.015 V
Piso de ruido: 0.00020 V (-84.24 dB FS)
SNR: 37.62 dB, ENOB: 5.96 bits
```

Ilustración 2 Datos obtenidos por el programa suministrado por el profesor.

Como se observa, el valor de la frecuencia corresponde aproximadamente con respecto a lo establecido en el generador de señales, a partir de este programa, se generan dos archivos correspondientes a las muestras de la señal en el dominio del tiempo y su dominio en la frecuencia, estos dos archivos se grafican en el software de Matlab, siendo que se utiliza el siguiente código para graficar la FFT de la señal medida.

```
t1 = fft.Frecuencia_Hz_;
v2 = fft.Magnitud_V_;
plot(t1, v2);
xlabel('Frecuencia (Hz)');
ylabel('Magnitud (dBm)');
title('Magnitud vs Frecuencia');
hold;
stem(t1,v2);
```

En este caso, se hace uso de la función “steam”, la cual permite ver la cantidad de muestras tomadas de la señal.

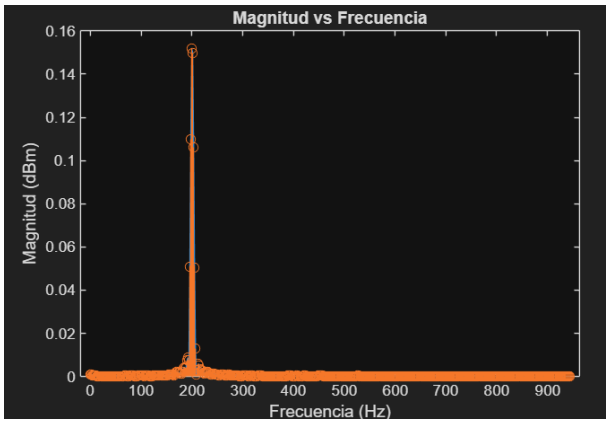

Ilustración 3 FFT de la señal muestreada.

Como se observa en la ilustracion numero tres, el pico de la señal corresponde a una frecuencia de 200Hz, lo cual nos indica un correcto desarrollo de la toma de muestras. Así mismo se realiza el grafico de la señal en el dominio del tiempo, a partir del siguiente código.

```
t = muestras.tiempo_s;
v = muestras.voltaje_v;
T = 0.005;
indices = t <= 5*T;
t_filtrado = t(indices);
v_filtrado = v(indices);

plot(t_filtrado, v_filtrado);
xlabel('Tiempo (s)');
ylabel('Voltaje (V)');
```

En este caso, el código permite observar únicamente cinco periodos de la señal, esto debido a que la frecuencia de muestreo es bastante alta, por lo cual la cantidad de periodos del total de las muestras realizadas es considerablemente alta, por lo cual la señal obtenida es.

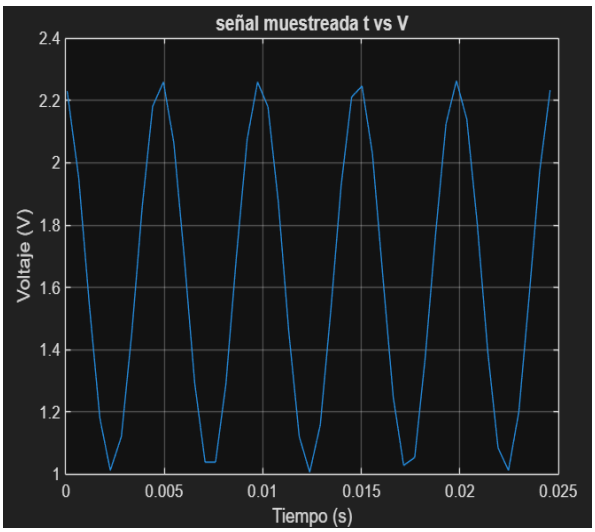

Ilustración 4 Señal muestreada en el dominio del tiempo.

Por lo cual podemos determinar que los datos obtenidos corresponden directamente con lo planteado y visto en el osciloscopio, a partir de esto, se realiza la variación de la cantidad de puntos para la FFT en este caso se asignan los siguientes valores, (64,128,256,512,1024,2048).

A partir de esto, se obtuvieron los datos los cuales se grafican utilizando Matlab, como se observa a continuación.

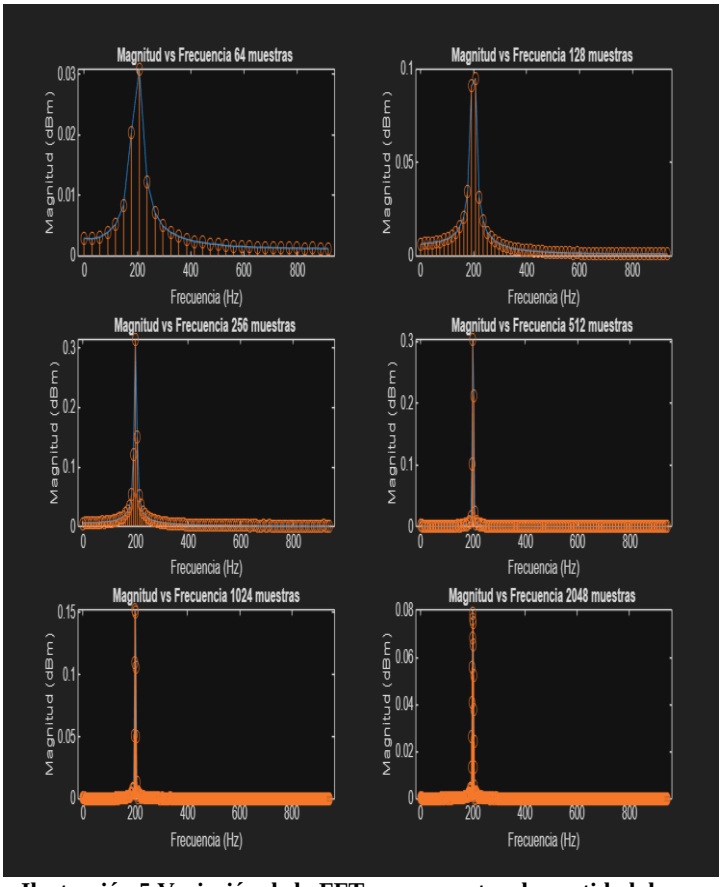

Ilustración 5 Variación de la FFT con respecto a la cantidad de muestras.

Como se observa, las señales obtenidas correspondientes a partir de los datos varían con respecto a la cantidad de puntos, podemos observar el como la cantidad de puntos y la forma que se llega a tener, en este caso se comprueba que a menor cantidad de puntos de la FFT la señal en su espectro va a variar y evidentemente no se observara correctamente con lo planteado teóricamente, la señal con 64 puntos únicamente tiene forma de montaña no de un pico a diferencia de la señal con 2048 puntos, por lo cual se determina que a mayor cantidad de puntos la señal se "cierra" y se ve de mejor manera el pico de la señal senoidal en 200 Hz, correspondiente a la frecuencia ingresada en el generador.

Este proceso se repite para 3 diferentes frecuencias.

- 100Hz.

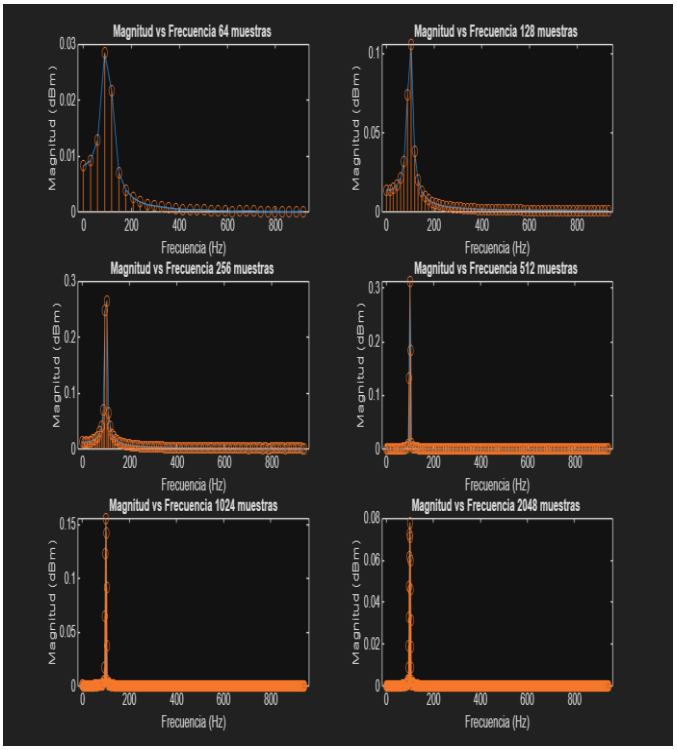

Ilustración 6 Variación de muestras de la FFT para una señal de 100Hz

Como podemos observar este proceso se repite y se evidencia adecuadamente cada uno de los picos, ya se de manera mas cerrada o abierta de acuerdo con la cantidad de puntos suministrados para el proceso de muestreo, en este caso todos los puntos tienen el pico en 100Hz como se estableció en el generador de señales.

- 900Hz

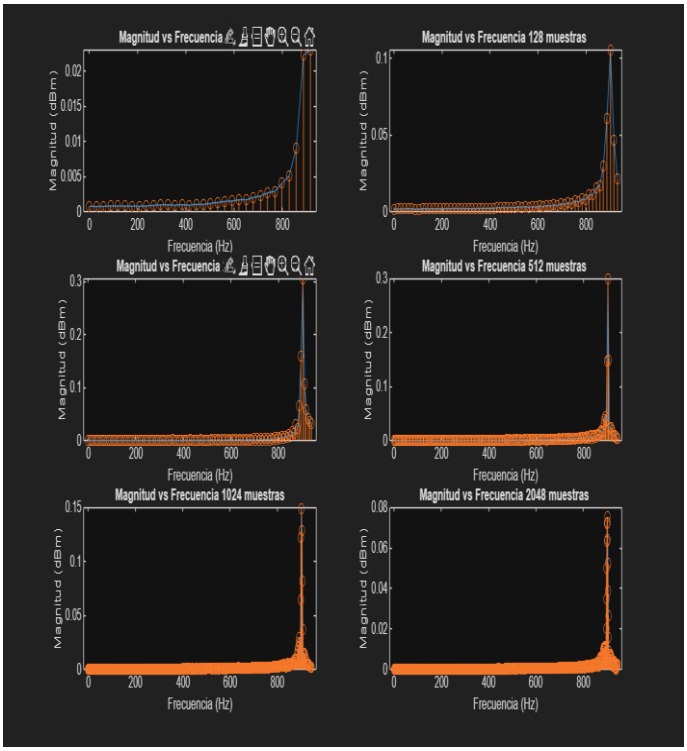

Ilustración 7 Variación de muestras de la FFT para una señal de 900Hz

Como se observa, la señal correspondiente se acerca al límite de la frecuencia de muestreo establecida, siendo que esto afecta considerablemente la resolución de los picos de la señal, por lo cual indica abiertamente que, en menor medida de puntos utilizados, el pico de la señal llega a distorsionarse y perderse.

- 1800Hz

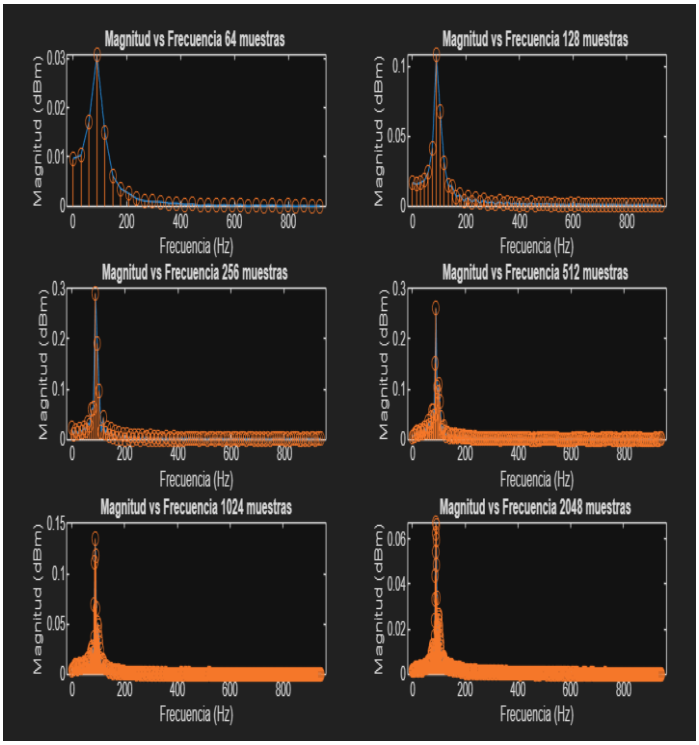

Ilustración 8 Variación de muestras de la FFT para una señal de 1800Hz

Podemos observar, que la frecuencia obtenida no corresponde directamente con la frecuencia asignada en el generador, a

diferencia se encuentra alrededor de 88Hz, esto lo podemos determinar por diferentes causas, lo principal puede deberse a la capacidad de la Raspberry pi pico 2W, siendo que el procesador no está adecuado para recibir frecuencias tan altas y su límite ronda alrededor de 1kHz, a diferencia de los 1,8kHz suministrados.

¿Cómo se establece la tasa de muestreo en el programa ?.
La tasa de muestreo del programa se establece, se establece mediante la variable "f_muestreo" y se implementa usando un retraso fijo entre muestras con "utime.sleep_us(dt_us)", esto se ve en las siguientes líneas del código del maestro.

```
f_muestreo = 2000
dt_us = int(1_000_000 / f_muestreo)
```

En este caso la variable "dt_us=int(100000/f_muestreo)" indica el intervalo de muestras en microsegundos(500µs), y así mismo la función.

```
for i in range(N):
    ...
    utime.sleep_us(dt_us)  # Espera fija entre lecturas
```

Indica una espera fija entre cada una de las muestras, esto con el fin de que estas sean uniformes.

II. PARTE 2.

1. Se establece el siguiente código con un tipo de muestreo diferente, en este caso se hace uso del siguiente código que está diseñado para muestrear una señal analógica usando el ADC del pin 26 de una Raspberry Pi Pico 2W, con el objetivo de simular una frecuencia de muestreo de 2000 Hz (una muestra cada 500 microsegundos). En cada iteración, lee el valor del ADC, lo convierte a voltaje (escalando de 0-65535 a 0-3.3V), y lo guarda en un archivo CSV junto con un tiempo teórico basado en la posición de la muestra (i/fs). Además, aplica el algoritmo de Goertzel en línea —muestra por muestra— para detectar la magnitud de tres frecuencias específicas (50 Hz, 200 Hz y 400 Hz), actualizando variables internas en cada paso. Al finalizar, calcula y guarda las magnitudes finales de esas frecuencias en el archivo "time3,csv".

```
from machine import ADC, Pin
import utime, math

adc = ADC(26)
fs = 2000       # Frecuencia de muestreo [Hz]
N = 512         # Número de muestras
dt_us = int(1_000_000 / fs)
frequencies = [50, 200, 400]

# Guardar resultados de FFT por frecuencia
fft_results = {f: 0.0 for f in frequencies}

# Jitter
intervals = []

# Archivo para la señal en tiempo real
with open("time3.csv", "w") as f_time:
    f_time.write("tiempo_s,voltaje_v\n")
    t_prev = utime.ticks_us()
    for i in range(N):
        # Tomar muestra
        sample = adc.read_u16()
        t_curr = utime.ticks_us()
        elapsed_us = utime.ticks_diff(t_curr, t_prev)
        intervals.append(elapsed_us)
        t_prev = t_curr

        # Convertir a voltaje
        volt = sample * 3.3 / 65535
        f_time.write(f"{i/fs:.6f},{volt:.5f}\n")

        # Procesar Goertzel en línea
        for freq in frequencies:
            # Recálculo simple Goertzel por muestra
            # Mantener variables internas por frecuencia
            if i == 0:
                fft_results[freq] = {'s_prev':0.0,'s_prev2':0.0}
            k = int(0.5 + N * freq / fs)
            omega = 2.0 * math.pi * k / N
            coeff = 2.0 * math.cos(omega)
            s = volt + coeff * fft_results[freq]['s_prev'] - fft_results[freq]['s_prev2']
            fft_results[freq]['s_prev2'] = fft_results[freq]['s_prev']
            fft_results[freq]['s_prev'] = s
    # Calcular magnitudes finales de Goertzel
    with open("fft3.csv", "w") as f_fft:
        f_fft.write("frecuencia_hz,magnitud_v\n")
        for freq in frequencies:
            s_prev = fft_results[freq]['s_prev']
            s_prev2 = fft_results[freq]['s_prev2']
            k = int(0.5 + N * freq / fs)
            omega = 2.0 * math.pi * k / N
            coeff = 2.0 * math.cos(omega)
            magnitude = math.sqrt(s_prev2**2 + s_prev**2 - coeff * s_prev * s_prev2) * 2 / N
            f_fft.write(f"{freq},{magnitude:.5f}\n")
            print(f"Frecuencia {freq} Hz -> Magnitud {magnitude:.5f} V")
```

A partir del archivo obtenido por el programa se obtiene la siguiente señal graficada en Matlab.

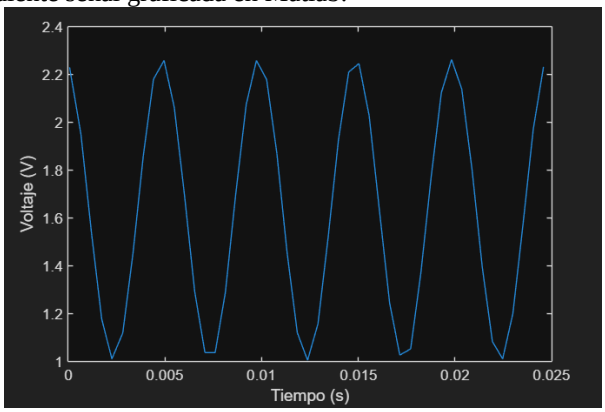

Ilustración 9 Señal muestreada a partir del código del estudiante.

Como se observa, la señal obtenida cuenta con características muy similares con respecto a la señal muestreada obtenida a

partir del código del docente, por lo cual se identifica un correcto funcionamiento del código, no obstante, es posible identificar las siguientes falencias al utilizar y ejecutar el MicroPython. La Raspberry Pi Pico 2W cuenta con un ADC de 12 bits (accesible como 16 bits en MicroPython), doble núcleo ARM Cortex-M0+ y 264 KB de RAM, suficiente para muestreo y procesamiento de señales pero al programarse en MicroPython, sufre sobrecarga de interpretación, recolección de basura no determinista y operaciones bloqueantes (como escritura en archivo), lo que rompe la periodicidad del muestreo y genera jitter inaceptable en aplicaciones de precisión; por eso, es mejor programarla en C++ con el SDK oficial: permite usar timers de hardware, DMA para transferencia sin CPU, interrupciones de baja latencia y buffers en RAM, logrando muestreo sincronizado, eficiente y en tiempo real, con control total sobre el rendimiento y los recursos del sistema, algo imposible de garantizar en MicroPython.

2.
Jitter: es la inestabilidad temporal en la periodicidad de una señal o proceso que debería ocurrir a intervalos fijos. Se mide como la desviación (media o RMS) respecto al tiempo ideal entre eventos.
¿Qué implicaciones tiene en el proceso de codificación de la fuente?
En el contexto de la codificación de una fuente analógica (como la conversión analógico-digital), el jitter tiene consecuencias directas en la fidelidad y precisión de la representación digital:

1. Introduce ruido en la señal digitalizada: El jitter actúa como un error de muestreo en el dominio del tiempo, lo que se traduce en una distorsión equivalente a ruido en amplitud —especialmente crítico en señales de alta frecuencia o con pendientes pronunciadas (dV/dt altas).
2. Degrada la resolución efectiva del ADC: Aunque el ADC tenga alta resolución (por ejemplo, 12 o 16 bits), el jitter puede limitar la precisión real, ya que el valor muestreado corresponde a un instante erróneo, perdiendo información útil.
3. Compromete la sincronización en sistemas de comunicación o control: Si la fuente codificada se usa para reconstrucción, modulación o control en tiempo real, el jitter puede causar errores de temporización, pérdida de sincronía o inestabilidad en lazos de control.

Con el fin de medir el jitter a partir de las muestras realizadas, se adiciona tanto al código generado por el estudiante como por el profesor el cálculo de este, a partir de las siguientes líneas de código.

```
# Calcular jitter
jitter_mean = sum(intervals)/len(intervals) - 1e6/fs
jitter_rms = (sum([(x - 1e6/fs)**2 for x in
    intervals])/len(intervals))**0.5

with open("jitter.csv", "w") as f_j:
   f_j.write("jitter_mean_us,jitter_rms_us\n")
   f_j.write(f"{jitter_mean:.3f},{jitter_rms:.3f}\n")

    print(f"Jitter medio = {jitter_mean:.2f} us, RMS =
    {jitter_rms:.2f} us")
```

Este código calcula el jitter midiendo el tiempo real (en microsegundos) entre cada par de muestras consecutivas, almacenado en la lista “intervals;” luego, compara cada intervalo con el tiempo ideal de muestreo (1e6/fs = 500 µs para 2000 Hz): el jitter medio es la diferencia promedio entre los intervalos reales y el ideal, indicando si el sistema tiende a muestrear más rápido o más lento; el jitter RMS (raíz de la media de los cuadrados de las desviaciones) mide la variabilidad o inestabilidad del muestreo, siendo un indicador más robusto de la dispersión temporal. Finalmente, guarda ambos valores en un archivo CSV y los imprime, permitiendo evaluar cuán estable fue la temporización del proceso de adquisición.
Esto se evidencia en los resultados.

```
MPY: soft reboot
Frecuencia 50 Hz -> Magnitud 0.01269 V
Frecuencia 200 Hz -> Magnitud 0.01684 V
Frecuencia 400 Hz -> Magnitud 0.00511 V
Jitter medio = 419.62 us, RMS = 3505.38 us
```

Ilustración 10 Jitter medido código alumno.

Esto mismo se adiciona en el programa del docente por lo cual se observa.

```
>>> %Run -c $EDITOR_CONTENT

MPY: soft reboot
Iniciando adquisición y análisis...
Frecuencia deseada: 2000 Hz, frecuencia real: 1891.18 Hz
Offset DC removido: 0.727 V
Frecuencia dominante: 199.46 Hz
Amplitud señal: 0.015 V
Piso de ruido: 0.00020 V (-84.24 dB FS)
SNR: 37.62 dB, ENOB: 5.96 bits
Jitter medio: 0.75 us | Jitter RMS: 10.54 us
```

Ilustración 11 Jitter medido del código del docente.

Como se observa, el jitter medido en el código del profesor es menor con respecto al jitter obtenido por el código del alumno, esto nos indica que el código del maestro es mas preciso con respecto a la toma de los datos siendo que es mas exacto en cada uno de los tiempos.

Como se comprobó a partir de los dos diferentes códigos se deduce para lograr un muestreo exitoso con la Raspberry Pi Pico 2W —es decir, que las muestras se tomen en los momentos exactos y sin retrasos— lo más efectivo es programarla en C++ (no en MicroPython), porque así puedes usar herramientas del sistema que trabajan como un reloj de precisión: un temporizador que ordena al ADC cuándo tomar cada muestra, una memoria intermedia (buffer) para guardarlas sin interrumpir el flujo, y hasta el segundo procesador de la Pico para que mientras uno toma muestras, el otro las guarda o analiza. En MicroPython, todo se hace en secuencia y se generan retrasos impredecibles; en C++, el sistema se organiza como una línea de producción bien sincronizada. Si insistes en MicroPython, al menos evita escribir en archivos o hacer cálculos durante el muestreo —pero nunca será tan preciso. La clave: para muestreo confiable, hay que hablarle al hardware en su propio lenguaje: C++.

IX. Análisis.

El código del profesor, basado en pausas fijas con `sleep_us`, mostró menor jitter y espectros más limpios en comparación con el código del estudiante, donde la ejecución en MicroPython generó inestabilidad temporal y pérdida de precisión. La ventana de Hanning resultó clave para reducir el leakage espectral y mejorar la fidelidad de la FFT. Además, se evidenció que un mayor número de puntos en la FFT permite una mejor visualización del espectro, aunque no aumenta la resolución real. Finalmente, se comprobó que la Raspberry Pi Pico 2W presenta limitaciones al trabajar con frecuencias altas bajo MicroPython, por lo que para aplicaciones de precisión es necesario programar en C++ y aprovechar timers y DMA del hardware

X. Conclusiones.

1. El muestreo sincronizado con sleep_us (código del profesor) es superior en precisión y estabilidad frente al muestreo libre (código del estudiante), evidenciado por su menor jitter y espectros más limpios.
2. La ventana de Hanning es esencial para reducir el *leakage* espectral y obtener FFTs más precisas, especialmente cuando la señal no contiene un número entero de ciclos.
3. Aumentar el número de puntos de la FFT (zero-padding) mejora la visualización del espectro, pero no la resolución real, que depende únicamente de la duración de la captura.
4. La Raspberry Pi Pico 2W, aunque potente, no es apta para muestreo de alta precisión bajo MicroPython debido a su naturaleza interpretada y no determinista.
5. Para aplicaciones serias de procesamiento de señales, es obligatorio migrar a C++, aprovechando timers de hardware, DMA y buffers, logrando así un muestreo verdaderamente en tiempo real y de alta fidelidad.

Referencias

[1] Movistar. (2025). Jitter: qué es, cómo te afecta y cómo reducirlo en casa. Blog Movistar Colombia. https://blog.movistar.com.co/guias/jitter/

[2] Codificación de fuentes: técnicas & ejemplos | StudySmarter. (s. f.). StudySmarter ES. https://www.studysmarter.es/resumenes/ingenieria/ingenieria-de-telecomunicaciones-ingenieria/codificacion-de-fuentes/