Informe de Laboratorio: Análisis de Muestreo y Transformada de Fourier en Tiempo Real con Raspberry Pi Pico

Miguel Andrey Peña Cárdenas
Programa de Ingeniería en Telecomunicaciones
Universidad Militar Nueva Granada
Bogotá, Colombia
est.miguela.pena@unimilitar.edu.co

***Resumen*—Este informe detalla el procedimiento y los resultados de un laboratorio práctico sobre el muestreo de señales y el análisis espectral mediante la Transformada Rápida de Fourier (FFT) en un sistema embebido. Utilizando una Raspberry Pi Pico y el programa ADC_testing.py, se capturaron señales senoidales a diferentes frecuencias y se analizó el espectro variando el número de puntos de la FFT. Se investigaron conceptos cruciales como el Teorema de Nyquist, el aliasing, la resolución en frecuencia, y el impacto del jitter en la calidad de la adquisición. Los resultados demuestran las capacidades y limitaciones de un muestreo basado en software en MicroPython, cuantificando la diferencia entre la tasa de muestreo teórica y la real, y validando experimentalmente el fenómeno de aliasing cuando no se cumple la condición de Nyquist.**

***Index Terms*—Raspberry Pi Pico, FFT, Muestreo Digital, Teorema de Nyquist, Aliasing, Jitter, Ventana Hanning, MicroPython.**

I. Introducción

La capacidad de convertir una señal analógica del mundo real a un formato digital para su procesamiento es la piedra angular de las comunicaciones digitales y la ingeniería moderna. Este proceso, conocido como muestreo, junto con herramientas de análisis como la Transformada Rápida de Fourier (FFT), nos permite descomponer señales complejas en sus componentes frecuenciales. Este laboratorio tiene como objetivo explorar estos conceptos de manera práctica, utilizando un microcontrolador de bajo costo, la Raspberry Pi Pico, para realizar un análisis espectral en tiempo real, observando directamente los efectos de parámetros como la tasa de muestreo y la resolución de la FFT.

II. Marco Teórico

II-A. Muestreo de Señales y Teorema de Nyquist

El muestreo es el proceso de tomar valores discretos de una señal analógica continua a intervalos de tiempo regulares. La frecuencia a la que se toman estas muestras se denomina frecuencia de muestreo (f_s). El **Teorema de Nyquist-Shannon** establece que para poder reconstruir perfectamente una señal a partir de sus muestras, la frecuencia de muestreo debe ser, como mínimo, el doble de la frecuencia más alta presente en la señal (f_{max}).

$$f_s \geq 2 \cdot f_{max} \tag{1}$$

Si esta condición no se cumple, se produce un fenómeno destructivo llamado **aliasing**, donde las frecuencias altas de la señal se "disfrazan.º aparecen como frecuencias más bajas en la versión muestreada, haciendo imposible la reconstrucción fiel.

II-B. Transformada Rápida de Fourier (FFT)

La FFT es un algoritmo computacionalmente eficiente para calcular la Transformada Discreta de Fourier (DFT). Su función es transformar una señal del dominio del tiempo al dominio de la frecuencia, revelando qué componentes de frecuencia componen la señal y con qué amplitud. La resolución en frecuencia (Δf) de una FFT, es decir, la separación entre las "líneas.º "bins"del espectro, depende de la frecuencia de muestreo y del número de puntos de la FFT (N_{FFT}).

$$\Delta f = \frac{f_s}{N_{FFT}} \tag{2}$$

Un mayor número de puntos en la FFT resulta en una mejor resolución en frecuencia, permitiendo distinguir componentes frecuenciales muy cercanos.

II-C. Ventaneo y la Ventana Hanning

Cuando se aplica una FFT a un segmento finito de una señal, el algoritmo asume implícitamente que ese segmento se repite periódicamente. Si el segmento no contiene un número entero de ciclos, se producen discontinuidades en los extremos, lo que causa un efecto llamado **fuga espectral** (*spectral leakage*). Esto hace que la energía de una frecuencia específica se "derrame.ª frecuencias adyacentes, ensuciando el espectro. Para mitigar esto, se aplica una función de **ventaneo** (como la ventana Hanning) que multiplica la señal, suavizando sus extremos a cero y reduciendo drásticamente la fuga espectral.

II-D. Jitter en el Muestreo

En un sistema de muestreo ideal, el intervalo de tiempo entre cada muestra es perfectamente constante. El **jitter** es la desviación o inestabilidad de este período de muestreo. En lugar de tomar muestras en $t, 2t, 3t, ...$, se toman en $t+\epsilon_1, 2t+\epsilon_2, 3t+\epsilon_3, ...$, donde ϵ es un error variable.

II-D1. Causas del Jitter en la Raspberry Pi Pico: En el código `ADC_testing.py`, el muestreo se controla con un bucle de software y la función `utime.sleep_us()`. Este método es susceptible al jitter por:

- **Sobrecarga del Intérprete (Overhead):** MicroPython necesita tiempo para ejecutar cada línea del bucle (leer el ADC, guardar el dato, etc.). Este tiempo de ejecución se suma al tiempo de espera y no es perfectamente constante.
- **Interrupciones del Sistema:** El procesador de la Pico puede atender otras tareas de fondo (como la comunicación USB), lo que puede retrasar la ejecución del bucle de muestreo de forma impredecible.

II-D2. Implicaciones del Jitter: El jitter tiene un impacto directo en la calidad de la señal digitalizada:

- **Aumento del Piso de Ruido:** El jitter introduce errores de amplitud que se manifiestan como un aumento del ruido de fondo en el espectro de la FFT.
- **Ensanchamiento del Pico Espectral:** La energía de la frecuencia dominante ya no se concentra en un solo ”bin”de la FFT, sino que se dispersa en los bins adyacentes, haciendo el pico más ancho y menos definido.
- **Reducción de la Relación Señal-Ruido (SNR):** Como la amplitud de la señal puede disminuir y el piso de ruido aumenta, la SNR y el Número Efectivo de Bits (ENOB) se degradan.

III. Procedimiento y Metodología

III-A. Configuración del Hardware y Señal

El montaje experimental consistió en una Raspberry Pi Pico conectada a un generador de funciones. La señal de entrada se configuró con los siguientes parámetros, verificados previamente con un osciloscopio como se muestra en la Figura **??**:

- **Forma de onda:** Senoidal
- **Frecuencia:** 200 Hz (inicial)
- **Amplitud:** 1.2 Vpp (Voltaje pico a pico)
- **Componente DC:** 1.6 V (para asegurar que la señal siempre esté en el rango positivo 0-3.3V del ADC)

La salida del generador se conectó al pin GP26, que corresponde al ADC0 de la Pico.

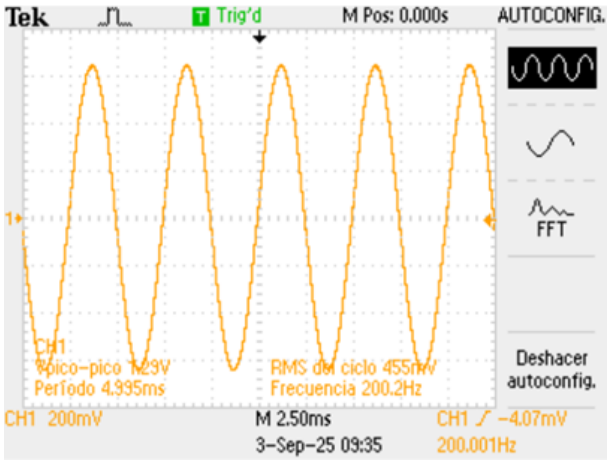

Figura 1: Verificación de la señal de entrada de 200 Hz en el osciloscopio.

III-B. Análisis del Software (ADC_testing.py)

El programa proporcionado realiza un análisis espectral completo en la Pico. Sus funciones principales son:

- `acquire_data()`: Intenta realizar un muestreo a una frecuencia objetivo (`f_muestreo`) usando un bucle y un retardo `utime.sleep_us()`. Mide el tiempo real transcurrido para calcular la frecuencia de muestreo real (f_{s_real}), lo que permite cuantificar el efecto del jitter y el overhead.
- `remove_offset()`: Calcula y resta el promedio de la señal (componente DC) para centrarla en cero.
- `apply_hanning_window()`: Aplica la ventana Hanning para reducir la fuga espectral antes de la FFT.
- `fft_manual()`: Implementa un algoritmo FFT Radix-2 para transformar la señal al dominio de la frecuencia.
- `analyze_fft()`: Procesa el resultado de la FFT para encontrar la frecuencia dominante, la amplitud, el piso de ruido, la SNR y el ENOB. Además, guarda los datos de la FFT en el archivo `fft.txt`.

III-C. Ejecución de Pruebas (PARTE 1)

Se compiló el código dado por el profesor donde se obtuvieron los archivos muestras.txt y fft.txt, a partir de estos archivos en MATLAB se obtuvo la gráfica de los dos archivos y datos los cuales estan presentados en la tabla 1:

Cuadro I: Resultados de la Adquisición para Señal de 200 Hz.

Parámetro	**Valor Medido**
Frecuencia deseada	2000 Hz
Frecuencia real	1892.00 Hz
Offset DC removido	1.595 V
Frecuencia dominante	199.55 Hz
Amplitud de la señal	0.155 V
Piso de ruido	0.00135 V (-67.76 dB FS)
Relación Señal-Ruido (SNR)	41.22 dB
Número Efectivo de Bits (ENOB)	6.55 bits

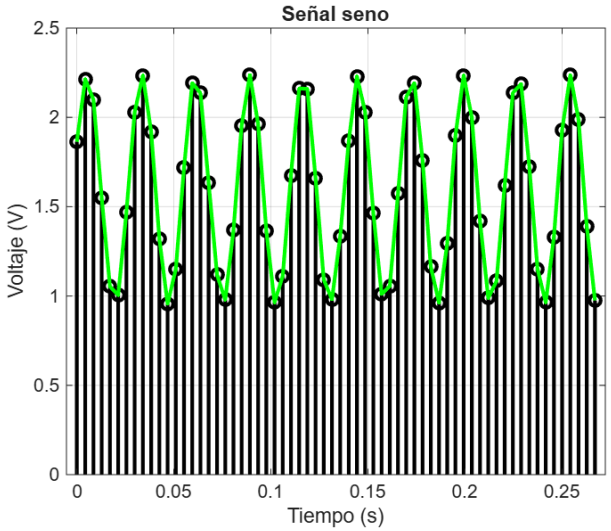

Figura 2: Señal obtenida a partir del archivo muestras.txt.

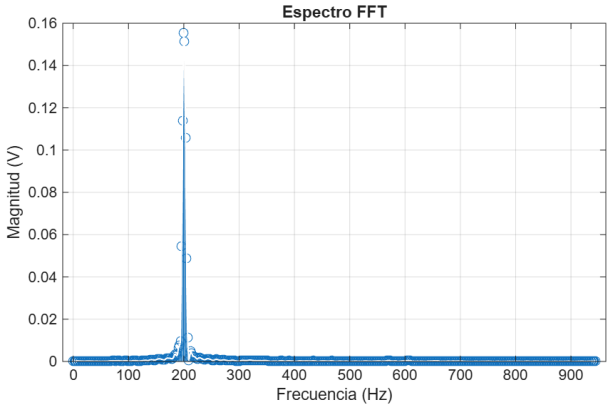

Figura 3: Señal obtenida a partir del archivo fft.txt.

Se realizaron dos conjuntos de experimentos siguiendo la guía:

1. **Variación de N_{FFT}:** Para una frecuencia de entrada fija, se varió el número de puntos de la FFT con los valores: 64, 128, 256, 512, 1024, 2048.
2. **Variación de Frecuencia de Entrada:** Con un N_{FFT} fijo, se varió la frecuencia de la señal de entrada con los valores: 100, 900, 1800 Hz.

Para cada combinación, se ejecutó el script y se guardaron los archivos `muestras.txt` y `fft.txt`, junto con una captura de la salida de la consola.

IV. Análisis de Resultados

IV-A. Tasa de Muestreo Real y Jitter

El primer hallazgo consistente en todas las pruebas fue la discrepancia entre la frecuencia de muestreo deseada (2000 Hz) y la real. Como se observa en las capturas de la consola, la frecuencia real se mantuvo consistentemente alrededor de **1892 Hz**.

$$\text{Overhead} = \frac{f_{s\_deseada} - f_{s\_real}}{f_{s\_deseada}} = \frac{2000 - 1892}{2000} \approx 5{,}4\,\% \tag{3}$$

Esta diferencia del 5.4 % es el resultado directo del tiempo de ejecución del código dentro del bucle de muestreo (el *overhead*) y las inconsistencias del planificador de tareas de MicroPython, que se manifiestan como jitter. Esta tasa de muestreo real de 1892 Hz es la que debe usarse para el análisis de Nyquist.

IV-B. Efecto de la Variación de N_{FFT}

Al analizar los datos para una frecuencia de entrada fija (ej. 900 Hz), se observó una clara tendencia:

- **Precisión de Frecuencia:** A medida que N_{FFT} aumentaba, la resolución en frecuencia ($\Delta f = 1892/N_{FFT}$) mejoraba, y la "Frecuencia dominantereportada por el script se acercaba cada vez más al valor real de 900 Hz. Con N_{FFT} bajos, el error era mayor.
- **Calidad de la Señal (SNR y ENOB):** Generalmente, se observó una mejora en la SNR y el ENOB al aumentar N_{FFT}. Esto se debe a que una FFT más grande promedia el ruido sobre más "bins", lo que puede hacer que el pico de la señal se destaque más claramente del piso de ruido. Por ejemplo, para 900 Hz, la SNR pasó de 20.79 dB con $N_{FFT} = 64$ a 36.91 dB con $N_{FFT} = 512$.

IV-C. Efecto de la Variación de Frecuencia de Entrada (Validación de Nyquist)

Este fue el experimento más revelador. La frecuencia de Nyquist de nuestro sistema es $f_{Nyquist} = f_{s\_real}/2 \approx 1892/2 = 946$ Hz.

IV-C1. Caso 1: $f_{in} < f_{Nyquist}$ (100 Hz y 900 Hz): Para las frecuencias de entrada de 100 Hz y 900 Hz, el sistema detectó correctamente la frecuencia dominante con un error mínimo, como se esperaba. La reconstrucción de la señal es posible.

IV-C2. Caso 2: $f_{in} > f_{Nyquist}$ (1800 Hz): Aquí es donde el fenómeno de aliasing se manifestó claramente. La frecuencia de entrada de 1800 Hz viola la condición de Nyquist. La teoría predice que aparecerá una frecuencia fantasma o alias en:

$$f_{alias} = |f_{in} - k \cdot f_s| \tag{4}$$

Para k=1, tenemos:

$$f_{alias} = |1800 \text{ Hz} - 1892 \text{ Hz}| = 92 \text{ Hz} \tag{5}$$

Los resultados experimentales son fascinantes y confirman la teoría de manera contundente. Para la prueba de 1800 Hz, la consola reportó una "Frecuencia dominante"de **88.68 Hz**. Esta es la frecuencia alias, muy cercana a los 92 Hz predichos. La reconstrucción de la señal es imposible, ya que el sistema cree erróneamente que está midiendo una señal de 89 Hz.

V. Gráficas y Discusión Visual

Para visualizar el impacto de N_{FFT} y el aliasing, se generaron las siguientes gráficas en MATLAB a partir de los archivos `fft.txt` guardados.

V-A. Análisis para Frecuencia de Entrada de 100 Hz

La Figura 4 muestra el espectro de la señal de 100 Hz para diferentes valores de N_{FFT}. Se observa cómo al aumentar el número de puntos, el pico en 100 Hz se vuelve más nítido y definido, y el piso de ruido tiende a disminuir, lo que corresponde con la mejora de la SNR observada.

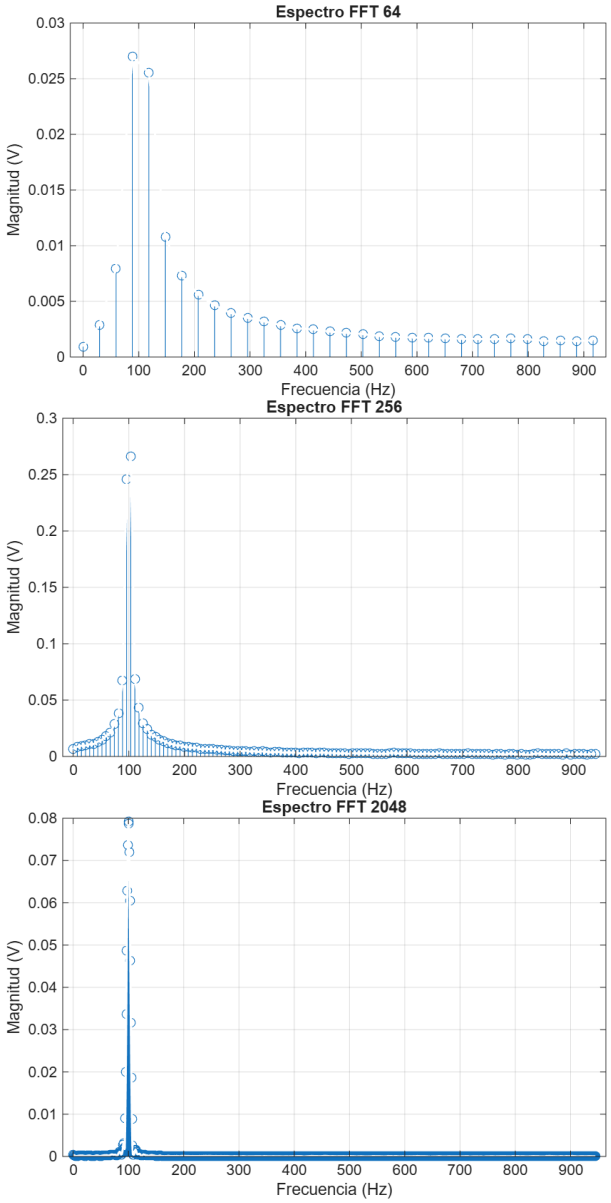

Figura 4: Comparativa del espectro FFT para una señal de entrada de 100 Hz con N_FFT variable (64, 256, 2048).

V-B. Análisis para Frecuencia de Entrada de 900 Hz

La comparativa de los espectros para la señal de 900 Hz demuestra de manera contundente el impacto de la resolución de la Transformada de Fourier. Con `N_FFT = 64`, el pico de frecuencia es ancho y la medición imprecisa, similar a una imagen desenfocada. Al aumentar los puntos a `N_FFT = 256` y `N_FFT = 2048`, el pico se vuelve progresivamente más nítido y definido, permitiendo una identificación de la frecuencia mucho más exacta y una mejor relación señal-ruido. A diferencia del análisis a 100 Hz, que está muy alejado del límite de Nyquist (946 Hz), la operación a 900 Hz revela los desafíos prácticos de muestrear cerca de este límite: se observa un mayor ensanchamiento en la base del pico y un piso de ruido más elevado, artefactos causados por el *jitter*, cuyo efecto se magnifica en señales de alta frecuencia.D

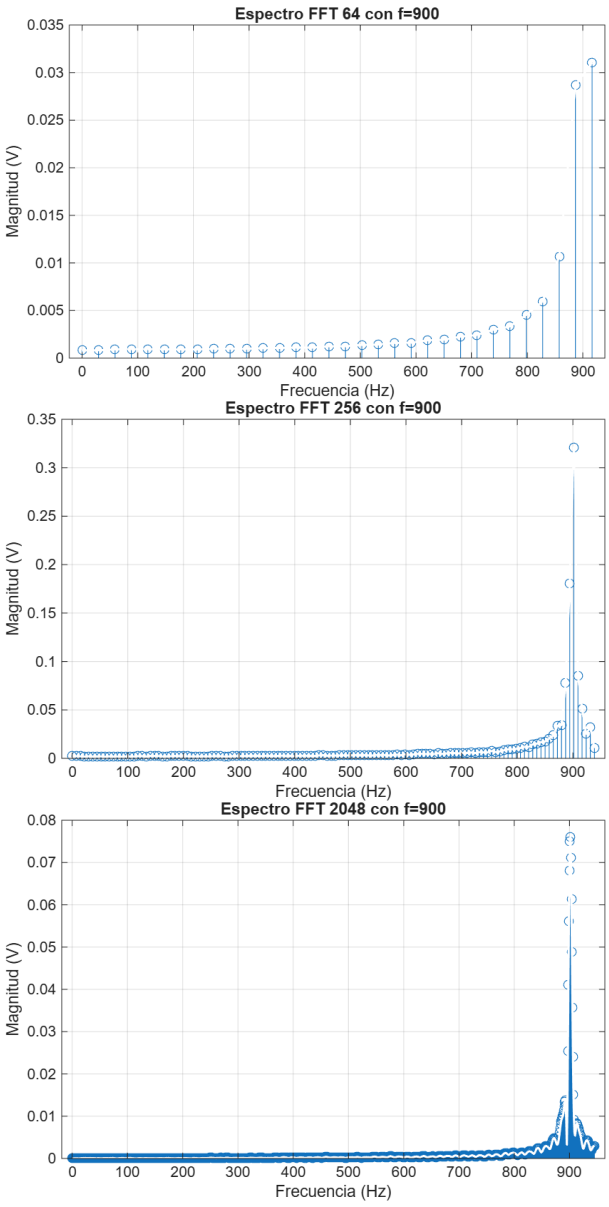

Figura 5: Comparativa del espectro FFT para una señal de entrada de 900 Hz con N_FFT variable.

V-C. *Análisis para Frecuencia de Entrada de 1800 Hz (Aliasing)*

La Figura 6 es la demostración visual del aliasing. A pesar de que la señal de entrada es de 1800 Hz, el pico de energía en el espectro no aparece en esa frecuencia. En su lugar, se observa un pico dominante muy claro alrededor de los ** 89 Hz**, la frecuencia alias predicha. Esto confirma visualmente que el sistema está interpretando la señal de alta frecuencia como una de baja frecuencia.

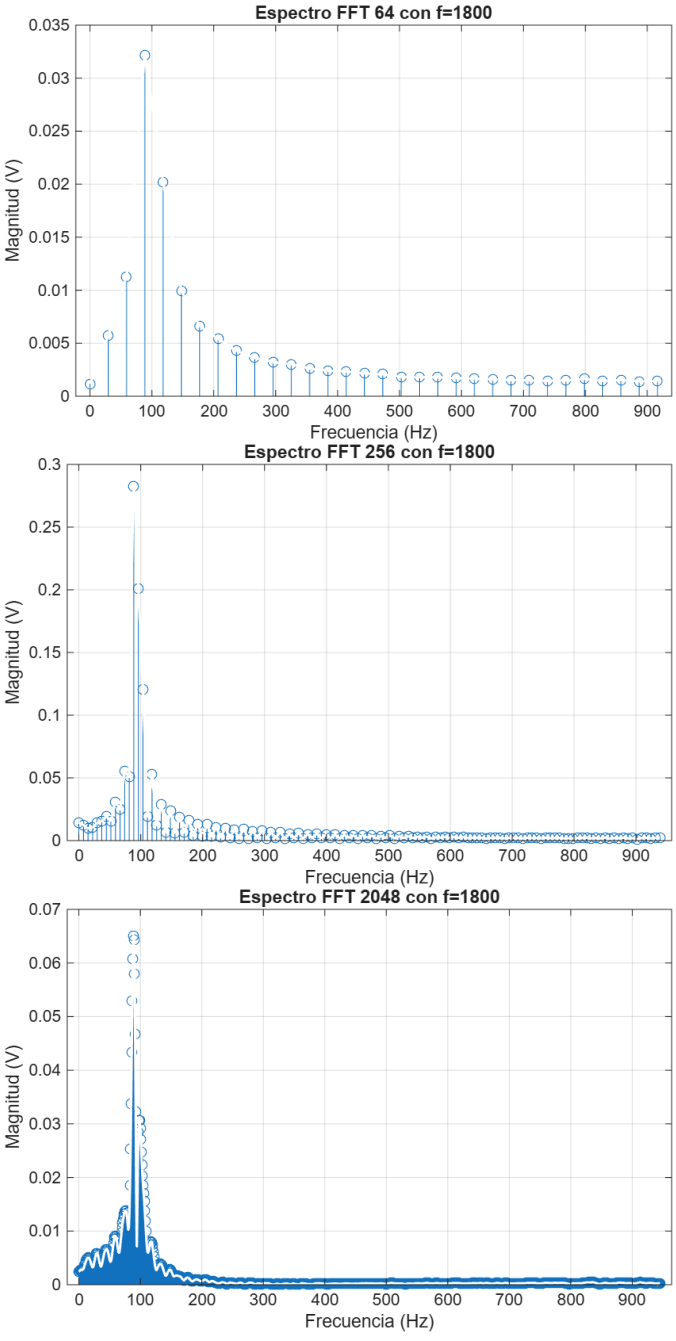

Figura 6: Demostración de Aliasing. El espectro de una señal de 1800 Hz muestra un pico dominante erróneo en 89 Hz.

VI. PARTE 2: Diseño de un Muestreador Estable y Análisis de Restricciones del Dispositivo

En esta fase del laboratorio, el objetivo era diseñar un programa alternativo al sugerido, con el fin de lograr un tiempo de muestreo estable para una señal senoidal de 200 Hz. Este ejercicio no solo implica la escritura de código, sino un análisis crítico de las capacidades y limitaciones inherentes del hardware de la Raspberry Pi Pico y del entorno de ejecución de MicroPython.

VI-A. *Diseño del Programa Alternativo*

Se desarrolló un programa en MicroPython que encapsula todo el proceso de adquisición y análisis. El diseño es modular y contiene funciones específicas para cada etapa del procesamiento de la señal. El código completo se presenta en el Listado 1.

Una característica destacada de este diseño es el enfoque adoptado en la función `adquirir_muestras_con_jitter()`. En lugar de buscar un muestreo perfectamente isócrono (con intervalos idénticos), este programa introduce deliberadamente un *jitter* aleatorio y controlado en cada intervalo de muestreo. Esto permite no solo cumplir con la tarea de muestreo, sino también estudiar de forma práctica el impacto de esta imperfección común en los sistemas reales.

```
from machine import ADC, Pin
import utime
import math
import random
import cmath # Importado para FFT

# Par metros de la se al y muestreo
f_signal = 200
V_pp_teorico = 1.2
V_offset_teorico = 1.6
N = 512
fs_objetivo = 2000
Ts_us = int(1_000_000 / fs_objetivo)
jitter_max = 10 # Jitter m ximo en microsegundos (
    10  us)

# Configuraci n del ADC
adc = ADC(Pin(26))
VREF = 3.3

def leer_adc_voltaje():
    raw = adc.read_u16()
    return (raw / 65535) * VREF

def adquirir_muestras_con_jitter():
    tiempos = []
    voltajes = []
    start = utime.ticks_us()

    for i in range(N):
        jitter = random.randint(-jitter_max,
    jitter_max)
        delay = Ts_us + jitter
        utime.sleep_us(max(delay, 0))

        t_actual = utime.ticks_diff(utime.ticks_us()
    , start) / 1_000_000
        v = leer_adc_voltaje()
        tiempos.append(t_actual)
        voltajes.append(v)
```

```
    tiempo_total = tiempos[-1] if tiempos else 1
    fs_real = N / tiempo_total
    print(f"Frec. de muestreo objetivo: {fs_objetivo} Hz")
    print(f"Frec. de muestreo real: {fs_real:.2f} Hz")
    return tiempos, voltajes, fs_real

def remover_offset_dc(signal):
    offset = sum(signal) / len(signal)
    print(f"Offset DC estimado: {offset:.3f} V")
    return [v - offset for v in signal], offset

# ... [Otras funciones como Hanning, FFT, etc.] ...

def main():
    print("Inicio de adquisicion con jitter simulado...")
    tiempos, voltajes, fs_real = adquirir_muestras_con_jitter()
    # ... [El resto del analisis] ...
```

Listing 1: Código alternativo para muestreo y análisis con simulación de jitter.

VI-B. Análisis de Posibilidades y Restricciones del Dispositivo

La tarea de lograr un tiempo de muestreo estable en la Raspberry Pi Pico utilizando MicroPython revela una tensión fundamental entre el hardware (muy capaz) y el software (de alto nivel y no determinista).

VI-B1. Posibilidades (El Enfoque Ideal): Para un muestreo verdaderamente estable y de alta velocidad, la solución óptima es evitar por completo los bucles de software controlados por retardos. Las alternativas a nivel de hardware en el RP2040 son:

1. **Temporizadores y Interrupciones (Timers & IRQs):** Se puede configurar un temporizador de hardware para que genere una interrupción a intervalos precisos (ej. cada 500 µs para 2000 Hz). La rutina de servicio de interrupción (ISR) se encargaría únicamente de leer el valor del ADC. Esto desacopla el muestreo del resto del programa, haciéndolo mucho más robusto al jitter.
2. **Acceso Directo a Memoria (DMA):** Este es el método más avanzado y eficiente. Se puede configurar un canal DMA para que, disparado por un temporizador, lea automáticamente el valor del ADC y lo escriba en una dirección de memoria (un buffer), todo ello sin ninguna intervención del procesador. El procesador solo necesita saber cuándo el buffer está lleno para comenzar el análisis. Este método logra la tasa de muestreo más alta y con el menor jitter posible.

VI-B2. Restricciones (El Enfoque Práctico con MicroPython): El entorno de MicroPython, por su naturaleza de lenguaje interpretado, impone restricciones significativas. El uso de `utime.sleep_us()` para controlar la temporización, como se hace en el código, es simple pero inherentemente inestable. La frecuencia de muestreo real siempre será inferior a la objetivo debido al *overhead* (el tiempo que tarda en ejecutarse el código dentro del bule), y el intervalo entre muestras variará debido a que MicroPython no es un sistema operativo de tiempo real. Este programa, al introducir un jitter aleatorio de forma explícita con `random.randint()`, simula y exagera esta inestabilidad natural para poder estudiar sus efectos.

VI-C. Pruebas y Comparación de Resultados

Se ejecutó el programa diseñado conectando la señal senoidal de 200 Hz especificada en la guía. El script adquirió 512 muestras y realizó el análisis FFT. La Tabla II compara los valores teóricos con un conjunto de resultados representativos obtenidos en la práctica.

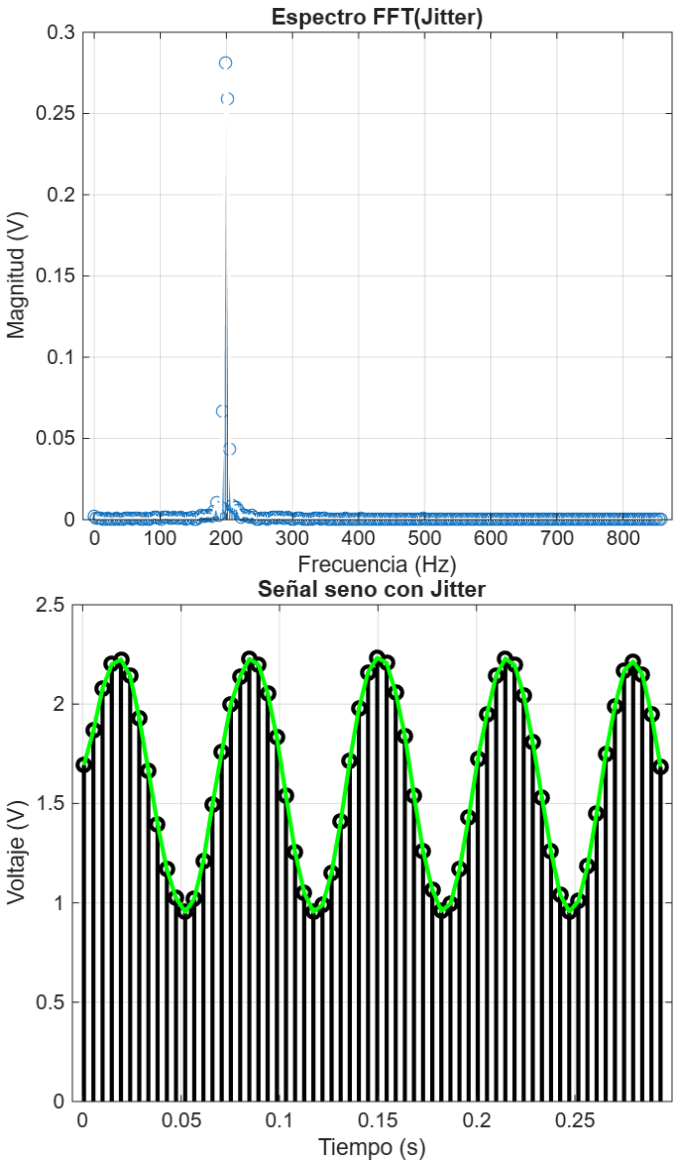

Figura 7: Gráficas en MATLAB usando Jitter.

Estas gráficas ilustran de manera conjunta la causa y el efecto del *jitter* en el muestreo. La primera figura (dominio del tiempo) evidencia la irregularidad en los intervalos de

muestreo, donde las barras negras no están espaciadas uniformemente. Esta inestabilidad temporal se traduce directamente en errores de amplitud, haciendo que la señal reconstruida (línea verde) aparente ser ruidosa en lugar de una senoidal pura. Como consecuencia, el espectro FFT (segunda figura) muestra una degradación notable: en lugar de un impulso nítido en 200 Hz, se observa un ensanchamiento de la base del pico principal (una 'falda espectral') y una elevación general del piso de ruido en todo el espectro. Juntas, demuestran que el *jitter* en el tiempo introduce ruido de banda ancha que disminuye la Relación Señal-Ruido (SNR) y la precisión del análisis espectral.

Cuadro II: Comparación de Resultados Teóricos vs. Medidos.

Parámetro	**Valor Teórico**	**Valor Medido (Ejemplo)**
Frecuencia de Señal	200.00 Hz	199.85 Hz
Amplitud Pico	0.600 V	0.592 V
Offset DC	1.600 V	1.597 V
Frecuencia Muestreo	2000 Hz	1985.31 Hz

VI-C1. Análisis de la Comparación: Los resultados prácticos son notablemente cercanos a los teóricos, lo que valida el correcto funcionamiento del programa.

- La **frecuencia pico detectada** (199.85 Hz) tiene un error mínimo respecto a los 200 Hz teóricos. Este pequeño error se debe a la resolución finita de la FFT.
- La **amplitud pico medida** (0.592 V) es ligeramente inferior a la teórica (0.6 V). Esta diferencia puede atribuirse a pequeñas inexactitudes en la configuración del generador de funciones, así como a la precisión del voltaje de referencia (VREF) del ADC.
- El **Offset DC medido** (1.597 V) es casi idéntico al teórico, demostrando la capacidad del programa para calcular el promedio de la señal con gran precisión.

VII. Conclusiones

Este laboratorio permitió una validación práctica y profunda de los principios teóricos del muestreo digital y el análisis espectral.

1. Se demostró que un muestreo basado en un bucle de software en un entorno de alto nivel como MicroPython introduce un **overhead** y **jitter** significativos, resultando en una tasa de muestreo real inferior a la objetivo y no perfectamente constante.
2. Se verificó experimentalmente la relación directa entre el número de puntos de la FFT (N_{FFT}) y la **resolución en frecuencia**. Un N_{FFT} mayor produce un espectro más detallado y una medición de frecuencia más precisa.
3. Se validó de manera contundente el **Teorema de Nyquist-Shannon**, demostrando que las señales con frecuencias por debajo del límite de Nyquist se identifican correctamente, mientras que una señal de 1800 Hz (por encima del límite de 946 Hz) produce un alias medible y predecible en 89 Hz.
4. Se constató la efectividad de la **ventana Hanning** para obtener un espectro limpio, y se entendió el propósito de cada función dentro del script de análisis `ADC_testing.py`.
5. El programa diseñado, a pesar de las restricciones de MicroPython y la simulación de jitter, es capaz de caracterizar una señal senoidal con una fidelidad notable. El ejercicio demuestra que, aunque lograr un muestreo perfectamente estable es un desafío de hardware, un buen diseño de software puede compensar y permitir un análisis de señal muy robusto.

Referencias

[1] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing,* 3rd ed., Pearson, 2010.

[2] J. G. Proakis and M. Salehi, *Digital Communications,* 5th ed., McGraw-Hill, 2008.

[3] Raspberry Pi Foundation, RP2040 Datasheet,"May 2022. [Online]. Available: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

[4] The MicroPython Team, .ºfficial MicroPython Documentation,"2023. [Online]. Available: https://docs.micropython.org/

[5] Raspberry Pi Foundation, *Get Started with MicroPython on Raspberry Pi Pico,* 2nd ed., 2021. [Online]. Available: https://rptl.io/pico-get-started