Muestreo con Raspberry Pi Pico 2W

Lina Fernanda Guio Cabrera
est.lina.guio@unimilitar.edu.co

***Resumen*—Este laboratorio analizó el proceso de muestreo digital de una señal senoidal de 200 Hz utilizando una Raspberry Pi Pico 2W. Se estudió la tasa de muestreo real, el impacto del jitter y el efecto del número de puntos de la FFT en el análisis espectral. Se comparó el programa original en MicroPython con un código modificado para mejorar la estabilidad temporal, evidenciando que un control más preciso del muestreo redujo el jitter y mejoró la fidelidad de la señal en el dominio del tiempo y la frecuencia.**

***Abstract*—This laboratory analyzed the digital sampling process of a 200 Hz sine wave using a Raspberry Pi Pico 2W. The study focused on the real sampling rate, jitter effects, and the influence of the FFT length on spectral analysis. The original MicroPython program was compared with a modified version aimed at improving timing stability. Results showed that precise sampling control reduced jitter and enhanced signal fidelity in both time and frequency domains.**

I. INTRODUCCIÓN

El muestreo digital es un proceso fundamental en la conversión de señales analógicas a digitales y en la codificación de la fuente, ya que determina la calidad de la representación temporal y espectral de una señal. Su correcta implementación requiere no solo definir una frecuencia de muestreo adecuada, sino también garantizar la estabilidad temporal entre muestras, minimizando los efectos del **jitter**, entendido como la variabilidad en los instantes de adquisición. Estos factores influyen directamente en el análisis en el dominio del tiempo y en el cálculo de la **transformada rápida de Fourier (FFT)**, herramienta clave para caracterizar señales en el dominio de la frecuencia.

En este laboratorio se utilizó la **Raspberry Pi Pico 2W** como plataforma de adquisición. En una primera parte, se configuró un generador de señales para entregar una señal senoidal de 200 Hz con 1.2 Vpp y un offset de 1.6 V. Esta señal fue verificada en el osciloscopio y luego conectada a la entrada ADC del microcontrolador. Posteriormente, se ejecutó el programa **ADC_testing.py** para capturar las muestras, obtener los archivos generados (muestras.txt y fft.txt), y graficar los resultados en MATLAB en los dominios del tiempo y la frecuencia. Se exploró además el efecto de variar el número de puntos de la FFT (64, 128, 256, 512, 1024 y 2048) y la frecuencia de la señal de entrada (100 Hz a 1800 Hz), observando cómo estos cambios afectaban la resolución y fidelidad espectral.

En una segunda parte, se diseñó un programa alternativo al código original, con el propósito de mejorar la regularidad del muestreo y analizar el impacto del **jitter** en la adquisición. Este código permitió introducir un jitter controlado, registrar intervalos de muestreo y generar métricas estadísticas, lo que facilitó la comparación entre el muestreo ideal y uno afectado por variabilidad temporal. De esta manera, se evaluaron los efectos del jitter en la reconstrucción de la señal y en la calidad de los espectros obtenidos.

II. DESARROLLO

A. Marco teórico

RASPBERRY PI PICO 2 W

Es la actualización de la placa inalámbrica de la Fundación Raspberry Pi, ahora basada en el microcontrolador RP2350, que mejora significativamente al RP2040 al ofrecer mayor rendimiento, más memoria y nuevas funciones de seguridad, manteniendo la compatibilidad con los modelos anteriores de la serie Pico.

Este nuevo chip incorpora doble núcleo y doble arquitectura, permitiendo trabajar con Arm Cortex-M33, que incluye FPU y duplica la velocidad del Cortex-M0+, o con RISC-V Hazard3, ideal para proyectos en arquitecturas abiertas. Además, la memoria se duplicó: 512 KB de SRAM y 4 MB de flash, junto con más periféricos como tres bloques PIO y el periférico HSTX para transmisión de alta velocidad.

En el ámbito de la seguridad, el RP2350 integra Arm TrustZone, arranque seguro, aceleración SHA-256, almacenamiento seguro de claves y un generador de números aleatorios, lo que lo hace confiable para aplicaciones profesionales.

Finalmente, la Pico 2W sigue siendo programable en C/C++, MicroPython y CircuitPython, manteniendo su carácter accesible y versátil para educación y desarrollo profesional. No obstante, el chip RP2350 A2 presenta la errata E9, que afecta algunos GPIO, por lo que en ciertos casos se requieren resistencias externas para garantizar un funcionamiento correcto. [1]

MUESTREO Y FFT

La "Transformación rápida de Fourier", FFT para abreviar, es un importante método de medición en la tecnología de medición de audio y acústica. Descompone una señal en sus componentes espectrales individuales y así proporciona información sobre su composición. Los FFT se utilizan para el análisis de errores, el control de calidad y la monitorización de las condiciones de las máquinas o sistemas. Este artículo explica el cálculo del FFT, los parámetros relevantes y sus efectos en el resultado de la medición.[2]

JITTER

Jitter es el término utilizado para describir la variación en el momento del momento de una señal. En términos de música, el nerviosismo sería similar a un metrónomo donde las garrapatas son a veces tempranas y a veces tarde, causando estragos en una banda o orquesta tratando de tocar en sincronía. Como un metrónomo defectuoso, una señal de reloj agitada puede causar grandes problemas en un circuito digital. Los circuitos digitales requieren precisión en su momento; la presencia de nerviosismo puede afectar el rendimiento y la fiabilidad, causando varios problemas.[3]

TIEMPO DE MUESTREO

El *tiempo de muestreo* de un bloque es un parámetro que indica cuándo, durante la simulación, el bloque produce salidas y, si corresponde, actualiza su estado interno. El estado interno incluye, entre otros, los estados continuos y discretos que se registran.[4]

B. Procedimiento

Primero se configuró el generador de señales para que entregara una señal senoidal de 200 Hz, con una amplitud de 1.2 Vpp y una componente de DC de 1.6 V. Con esta configuración se garantizó que la señal cumpliera con los requisitos establecidos y se mantuviera dentro de los rangos permitidos por el conversor analógico–digital (ADC) de la Raspberry Pi Pico. Además, se validó que el valor máximo de la señal no superara los 3.3 V, ya que este es el límite de tensión de entrada del ADC, y sobrepasarlo podría ocasionar daños permanentes en la placa.

Posteriormente, se conectó la señal a una de las entradas ADC de la Raspberry Pi Pico 2W, específicamente al GPIO 17, el cual corresponde a uno de los canales de conversión analógica–digital del dispositivo. Asimismo, se cuidó que la referencia de tierra entre el generador de señales, el osciloscopio y la placa estuviera correctamente establecida, lo que aseguró la fidelidad en la captura de los datos y evitó errores de medición ocasionados por diferencias de potencial.

A continuación, **se ejecutó** el programa **ADC_testing.py**, disponible en el repositorio del docente.

En el programa ADC_testing.py la tasa de muestreo se establecía a partir de la variable `f_muestreo`, definida inicialmente con un valor de 2000 Hz. A partir de esta frecuencia, se calculaba el intervalo de muestreo en microsegundos mediante `dt_us = int(1_000_000 / f_muestreo)`. Este valor controlaba la función `utime.sleep_us(dt_us)`, la cual introducía pausas precisas entre cada adquisición de muestra, asegurando que la captura de datos se realizara con un periodo constante y, por lo tanto, con la frecuencia de muestreo deseada. Durante la adquisición, el programa también calculaba la frecuencia real alcanzada (`fs_real`) para verificar si coincidía con la programada y evaluaba la presencia de jitter, es decir, pequeñas variaciones en los intervalos de muestreo.

Cada función del código cumplía un papel específico en el proceso de análisis. La función `acquire_data()` realizaba la captura de los datos provenientes del ADC, guardaba los tiempos de adquisición, calculaba la frecuencia real de muestreo y registraba tanto las muestras como los intervalos en archivos de texto. Posteriormente, `convert_to_voltage()` transformaba los valores digitalizados (enteros entre 0 y 65535) a voltajes reales en un rango de 0 a 3.3 V, mientras que `remove_offset()` eliminaba el componente de corriente continua promediando las muestras y restando este valor, para centrar la señal en torno a cero. Por su parte, `apply_hanning_window()` multiplicaba los datos por una ventana Hanning, la cual suavizaba los extremos de la señal para minimizar las discontinuidades en el cálculo de la transformada de Fourier y reducir el efecto de fuga espectral.

La función `fft_manual()` implementaba la transformada rápida de Fourier (FFT) usando el algoritmo radix-2, con un procedimiento de inversión de bits y etapas de cálculo en mariposa. Esto permitía transformar la señal del dominio del tiempo al dominio de la frecuencia, identificando las componentes espectrales presentes. Finalmente, `analyze_fft()` procesaba los resultados de la FFT: calculaba magnitudes, determinaba la frecuencia dominante y su amplitud, estimaba el piso de ruido y a partir de estos datos calculaba la relación señal a ruido (SNR) y el número efectivo de bits (ENOB). Los resultados eran mostrados en pantalla y guardados en un archivo de texto para su posterior análisis.

La ventana Hanning servía específicamente para mejorar la precisión del análisis espectral. Cuando se trabajaba con señales finitas, al cortar un fragmento para la FFT se generaban bordes abruptos que introducían componentes no deseadas en el espectro, un fenómeno conocido como fuga espectral. Al aplicar la ventana Hanning, el inicio y el final del segmento de señal se reducían gradualmente hacia cero, suavizando las transiciones y disminuyendo este efecto. En el programa, la ventana se aplicaba antes de realizar la FFT mediante la función `apply_hanning_window`, de manera que el análisis de frecuencia resultaba más limpio y representativo de la señal real.

Se obtuvieron en el Shell los siguientes resultados

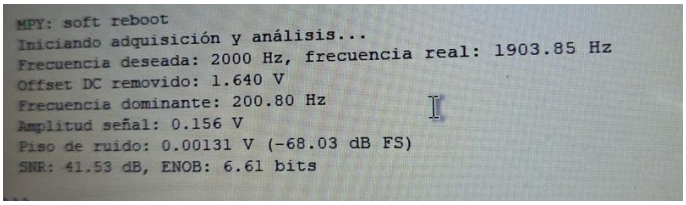


Figura 1. Valores obtenidos en el Shell

Posteriormente, **se utilizaron los archivos generados en MATLAB** para su análisis gráfico. Los archivos *muestras.txt* y *fft.txt*, previamente convertidos a formato CSV, **se graficaron** con el fin de observar el comportamiento de la señal tanto en el dominio del tiempo como en el dominio de la frecuencia. Para mejorar la visualización, **se ajustó el programa** de manera que mostrara únicamente los primeros periodos de la señal en el dominio temporal.

```
opts = delimitedTextImportOptions("NumVariables", 2);

% Rango y delimitador

opts.DataLines = [10, Inf];

opts.Delimiter = "\t";

% Nombres y tipos de columnas

opts.VariableNames = ["t", "x"];

opts.VariableTypes = ["double", "double"];

% Reglas adicionales

opts.ExtraColumnsRule = "ignore";

opts.EmptyLineRule = "read";

% Importar datos

muestras = readtable("C:\Users\Lenovo\MATLAB Drive\muestras100hz.csv", opts);

% Asignación de columnas

t = muestras.t;     % tiempo en segundos

x = muestras.x;     % amplitud de la señal

% Parámetros de la señal

f1 = 200;          % Frecuencia en Hz

T = 1/f1;          % Periodo

idx = t <= 2*T;    % Selección de dos periodos

% Gráfica

figure;

plot(t(idx), x(idx), 'b', 'LineWidth', 1.2); % Señal en azul

hold on;

stem(t(idx), x(idx), 'r');                % Muestras en rojo

hold off;

xlabel('Tiempo (s)');

ylabel('Amplitud de la señal (V)');

title('Señal muestreada en el dominio del tiempo');

legend('Señal continua aproximada','Muestras discretas');

grid on;
```

El código comenzó configurando las opciones de importación mediante la instrucción `delimitedTextImportOptions`, con lo cual se estableció que el archivo a leer contendría dos variables o columnas. Esta preparación fue necesaria para que MATLAB reconociera la estructura del archivo CSV o TXT que almacenaba los datos obtenidos en el muestreo.

Posteriormente, se indicó el rango de lectura y el delimitador. En este caso, los datos se empezaron a importar a partir de la línea 10 del archivo hasta el final, y se especificó que el separador de columnas fuera la tabulación. Esto aseguró que el software leyera únicamente la información relevante, ignorando encabezados o líneas adicionales que no hicieran parte de los datos de la señal.

A continuación, se definieron los nombres y tipos de variables correspondientes a las columnas del archivo. La primera columna se nombró como `f`, la cual representó los valores numéricos (ya sea tiempo o frecuencia, según el archivo). La segunda columna se llamó `v`, que contenía los valores asociados a la señal muestreada. De esta manera, MATLAB reconoció a `f` como datos numéricos (`double`) y a `v` como texto (`string`), permitiendo una correcta asignación de la información.

Después de esto, se ajustaron ciertas reglas de lectura para que MATLAB ignorara posibles columnas extra y tratara adecuadamente las líneas vacías o los espacios en blanco. Esto fue útil para prevenir errores en la importación y garantizar que los datos se cargaran de forma limpia.

Finalmente, se realizó la lectura del archivo con la instrucción `readtable`, lo que permitió guardar los datos en una tabla llamada `muestras`. Posteriormente, se asignaron variables para representar el tiempo (`t`) y los valores de la señal (`x`). A partir de allí, se filtraron únicamente los primeros dos periodos de la señal y se generó la gráfica correspondiente. En el gráfico, se incluyeron los ejes de tiempo (s) y valores de la señal, además de un título descriptivo que indicó el tipo de información representada.

Las graficas obtenidas tanto de muestreo como de FFT fueron las siguientes.

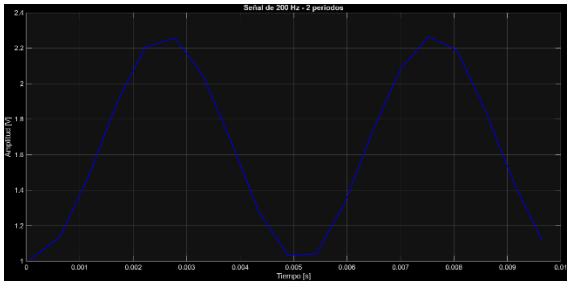


Figura 2. Muestreo de FFT 1024

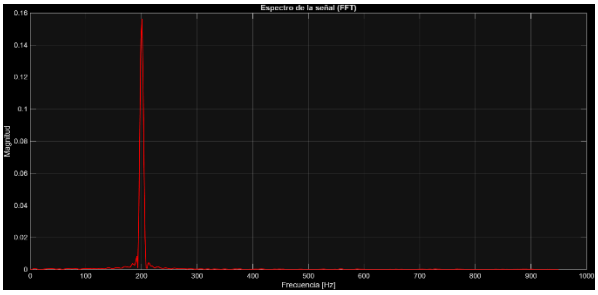


Figura 3. FFT de 1024

Después de realizar la primera parte del procedimiento, se repitió el proceso modificando el número de puntos utilizados en la Transformada Rápida de Fourier (FFT) dentro del programa *ADC_testing.py*. Inicialmente, el valor por defecto estaba establecido en N_FFT = 1024, sin embargo, se ajustó este parámetro a diferentes valores: 64, 128, 256, 512, 1024 y 2048.

Cada vez que se cambió el valor de N_FFT, se ejecutó nuevamente el programa para adquirir las muestras y calcular la transformada, lo que permitió observar cómo variaba la resolución espectral en función del número de puntos considerados en la FFT.

Posteriormente, se guardaron los archivos generados para cada caso, tanto las muestras en el dominio temporal como los resultados en el dominio de la frecuencia, con el fin de mantener un registro organizado de las pruebas y facilitar su análisis comparativo posterior en MATLAB.

La tabla registra los valores obtenidos de muestras, FFT y en la consola.

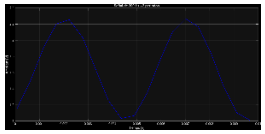
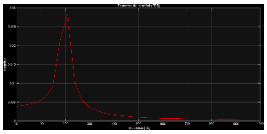
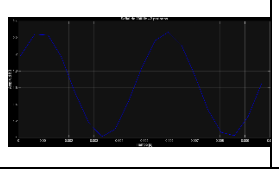
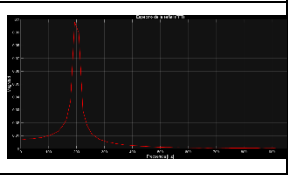
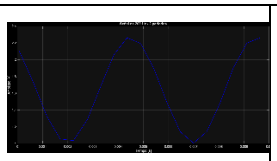
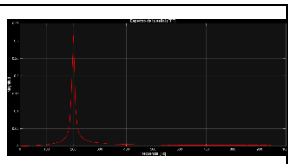
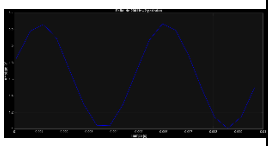
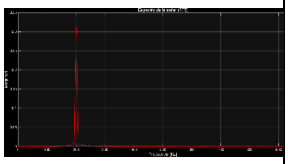
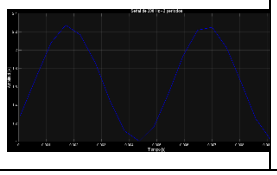
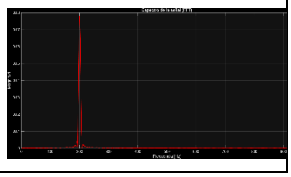
VALOR FFT CODI	MUESTRAS	FFT
64		
128		
256		
512		
2048		

Tabla 1. Resultados experimentales cambiando FFT

Al modificar el número de puntos de la FFT en el programa, se observó un cambio significativo en las gráficas obtenidas. Este comportamiento se explicó porque el parámetro `N_FFT` determinaba la resolución en frecuencia del análisis espectral.

Cuando se utilizó un valor bajo, como 64 o 128 puntos, las gráficas mostraron picos anchos y menos definidos. Esto se debió a que cada intervalo de frecuencia abarcaba un rango mayor, lo que redujo la capacidad de distinguir con precisión la frecuencia dominante de la señal. En consecuencia, el espectro resultó “grueso” y con menos detalle.

Por el contrario, al aumentar el número de puntos de la FFT a valores más altos, como 1024 o 2048, la resolución en frecuencia mejoró notablemente. En este caso, los picos en el espectro se representaron de manera más estrecha y bien definidos, lo que permitió identificar con mayor exactitud la frecuencia fundamental (200 Hz en este caso) y el nivel de ruido asociado. No obstante, también se notó que el cálculo con un mayor número de puntos implicaba un procesamiento más pesado y una mayor cantidad de muestras necesarias.

En conclusión, el análisis evidenció que la elección del número de puntos en la FFT representaba un compromiso entre la resolución en frecuencia y el costo computacional. Para el caso de la Raspberry Pi Pico, el uso de 1024 puntos permitió un equilibrio adecuado entre precisión espectral y tiempo de procesamiento.

Después de realizar las pruebas variando el número de puntos en la FFT, el parámetro se dejó nuevamente en su valor inicial de **1024 puntos**, con el fin de mantener una base constante de comparación. Una vez fijado este valor, se procedió a modificar la frecuencia de la señal de entrada en diferentes etapas: **100, 900 y 1800 Hz**.

Los valores obtenidos fueron:

F	Muestras	FFT

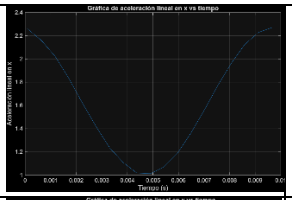
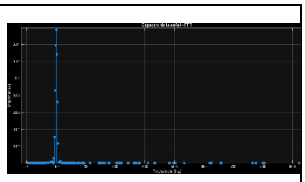
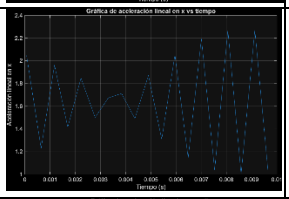
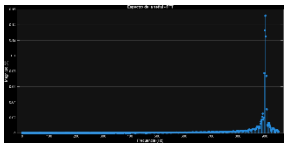
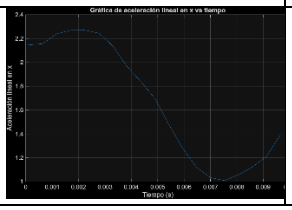
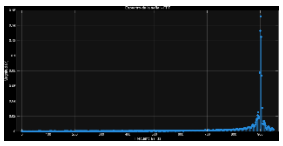
100		
900		
1800		

Tabla 2. Resultados experimentales cambiando frecuencia

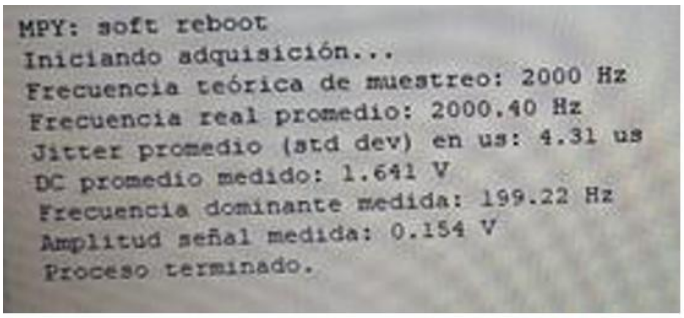
Para frecuencias bajas, como 100 Hz, la señal se representó claramente en el dominio de la frecuencia, y la frecuencia dominante coincidió con los valores esperados. La amplitud se mantuvo estable, y el piso de ruido se mantuvo bajo, lo que permitió una buena relación señal/ruido.

Al aumentar la frecuencia a 900 Hz, la precisión en la identificación de la frecuencia fundamental se conservó, pero se notó que el contenido espectral comenzaba a desplazarse hacia valores más altos en el eje de frecuencia. En estos casos, la resolución de la FFT dependió del número de puntos (`N_FFT`) utilizado: con valores bajos de puntos, los picos aparecieron menos definidos, mientras que con 1024 o 2048 puntos se pudo identificar la frecuencia con gran exactitud.

Finalmente, con frecuencias más altas, como 1800 Hz, se evidenció una limitación impuesta por la tasa de muestreo. Dado que la frecuencia de muestreo estaba alrededor de 2000 Hz, al acercarse al límite de Nyquist (1000 Hz), comenzaron a aparecer distorsiones y aliasing en las gráficas. En el caso de 1500 Hz y especialmente en 1800 Hz, la señal no pudo ser representada de manera fiel, ya que estas frecuencias superaban la mitad de la frecuencia de muestreo, lo que provocó que la señal se "reflejara" en el espectro en frecuencias más bajas, mostrando resultados erróneos.

Se había continuado el trabajo de laboratorio tomando como base el programa inicial `ADC_testing.py`, pero en este caso se diseñó un código diferente con el fin de muestrear una señal senoidal de 200 Hz, con una amplitud de 1.2 Vpp y un nivel DC de 1.6 V. La idea principal fue garantizar un muestreo estable en el tiempo, evitando las variaciones que podían presentarse cuando el control de adquisición no estaba estrictamente sincronizado.

El nuevo programa se estructuró de manera que el ADC realizara lecturas periódicas a intervalos definidos por un temporizador. Con ello se buscó asegurar que cada muestra fuera tomada de manera uniforme en el tiempo, condición necesaria para que la reconstrucción de la señal y su análisis en frecuencia resultaran confiables. Además, se implementó un almacenamiento ordenado de los datos para su posterior procesamiento



Los resultados obtenidos permitieron observar que la tasa de muestreo se mantuvo bastante estable. La frecuencia teórica programada fue de 2000 Hz, mientras que la frecuencia real promedio medida fue de 2000.40 Hz, lo que indicó que el sistema operó con un error prácticamente despreciable. Este comportamiento confirmó que la Raspberry Pi Pico 2W tuvo la capacidad de generar un muestreo confiable dentro de los límites técnicos del dispositivo.

En cuanto al **jitter**, el valor promedio calculado fue de **4.31 µs**, una magnitud pequeña si se compara con el periodo de muestreo nominal (500 µs). Esto significó que las variaciones en el instante de captura de cada muestra fueron mínimas y no afectaron de manera crítica la reconstrucción de la señal. Sin embargo, se evidenció que este parámetro no fue nulo, lo cual era esperado debido a la naturaleza de los temporizadores por software y a las posibles interrupciones internas del microcontrolador.

El análisis espectral corroboró que la **frecuencia dominante medida fue de 199.22 Hz**, en concordancia con la señal senoidal de entrada de 200 Hz. La amplitud medida fue de 0.154 V, ligeramente inferior a la teórica (0.156 V), diferencia atribuida tanto al offset removido como a la cuantización del ADC y al efecto del jitter sobre la exactitud de las muestras.

En términos prácticos, el jitter introdujo un **ensanchamiento leve en el espectro de frecuencia y un aumento mínimo del piso de ruido**, aunque el impacto fue bajo debido a que la magnitud de la variación temporal fue pequeña frente al periodo de muestreo. Por lo tanto, se concluyó que el sistema mantuvo una **relación señal/ruido adecuada** y un desempeño satisfactorio para la codificación de la fuente en señales de este rango de frecuencias.

Se reconoció que las posibilidades técnicas del dispositivo podían optimizarse aún más con la implementación de módulos como el PIO, el uso de DMA o la configuración de temporizadores de hardware, lo que permitiría reducir aún más el jitter y mejorar la fidelidad en la digitalización de señales de mayor frecuencia.

Por último, se diseñó un programa diferente al sugerido por el profesor, cuyo objetivo principal fue **medir y simular los efectos del jitter** en el proceso de muestreo.

```
from machine import ADC, Pin
```

```
import array, utime, urandom

adc = ADC(Pin(27))

N = 512

fs = 2000

dt = int(1_000_000 / fs)

jmax = 20

buf = array.array('H', [0]*N)

ts = array.array('I', [0]*N)

VREF = 3.3

print("Inicio de muestreo con jitter")

t0 = utime.ticks_us()

for k in range(N):

    buf[k] = adc.read_u16()

    ts[k] = utime.ticks_diff(utime.ticks_us(), t0)

    delta = urandom.randint(-jmax, jmax)

    utime.sleep_us(dt + delta)

print("Muestreo finalizado")

vals = [(x / 65535) * VREF for x in buf]

with open("datos_jitter_alt.txt", "w") as f:

    f.write("n\tus\tV\n")

    for n, (t, v) in enumerate(zip(ts, vals)):

        f.write(f"{n}\t{t}\t{v:.5f}\n")

print("Archivo listo: datos_jitter_alt.txt")
```

A diferencia del código original, que mantenía un tiempo de muestreo constante mediante pausas de duración fija, en este nuevo programa se introdujo una variación aleatoria en el retardo entre muestras. Dicha variación se controló con la instrucción `urandom.randint(-jmax, jmax)`, donde `jmax` definió el rango máximo de desviación en microsegundos.

Una diferencia importante respecto al programa del profesor fue que en este caso se almacenaron no solo los valores de voltaje, sino también las marcas de tiempo exactas de cada muestra (`ts[k]`). Esto permitió cuantificar directamente la variación temporal entre muestras y, en consecuencia, observar de forma explícita el jitter generado en el proceso.

Los resultados esperados mostraron que los archivos obtenidos contenían ligeras oscilaciones en los intervalos de tiempo entre muestras. Comparado con el código base del profesor, que alcanzó un jitter promedio del orden de 4.31 µs, el programa diseñado introdujo variaciones controladas de hasta ±20 µs, lo cual produjo un **jitter más alto y visible en el registro de datos**.

Si bien este tipo de simulación no reflejó exactamente el comportamiento interno del microcontrolador, sí permitió **evaluar el impacto del jitter en el espectro de la señal**, mostrando un ensanchamiento más pronunciado en las gráficas de la FFT y un aumento del piso de ruido.

Repositorio:

III. REFERENCIAS

[1] "Pico 2 W Download." Accessed: Aug. 17, 2025. [Online]. Available: https://circuitpython.org/board/raspberry_pi_pico2_w/

[2] "FFT." Accessed: Sep. 09, 2025. [Online]. Available: https://www.nti-audio.com/es/servicio/conocimientos/transformacion-rapida-de-fourier-fft

[3] "¿Qué es el jitter? Definición y explicación - IONOS." Accessed: Sep. 09, 2025. [Online]. Available: https://www.ionos.com/es-us/digitalguide/servidores/know-how/jitter/

[4] "¿Qué es el tiempo de muestreo? - MATLAB & Simulink." Accessed: Sep. 09, 2025. [Online]. Available: https://la.mathworks.com/help/simulink/ug/what-is-sample-time.html