Laboratorio Jitter

Belky Valentina Giron Lopez
est.belky.giron@unimilitar.edu.co
Docente: José De Jesús Rúgeles

Resumen — Durante la práctica se configuró un generador de señales para producir una señal senoidal de 200 Hz, 1.2 Vpp y un nivel DC de 1.6 V, la cual fue adquirida por el conversor A/D del Raspberry Pi Pico. Posteriormente, se procesaron los datos mediante el programa ADC_testing.py, obteniendo archivos con muestras y espectros en frecuencia, que fueron graficados en Matlab para su análisis. Se estudiaron los efectos de variar la cantidad de puntos de la FFT (64, 128, 256, 512, 1024 y 2048) y la frecuencia de la señal de entrada, observando los cambios en la resolución espectral y en la representación de la señal.

En una segunda etapa, se implementó un programa alternativo para el muestreo, evaluando la estabilidad temporal y comparando los resultados con los valores teóricos. Se analizó la influencia del jitter en la calidad de la digitalización.

Abstract **--** During the practical exercise, a signal generator was configured to produce a 200 Hz, 1.2 Vpp sinusoidal signal with a DC level of 1.6 V, which was acquired by the A/D converter of the Raspberry Pi Pico. The data was subsequently processed using the ADC_testing.py program, obtaining files with samples and frequency spectra, which were graphed in Matlab for analysis. The effects of varying the number of FFT points (64, 128, 256, 512, 1024, and 2048) and the input signal frequency were studied, observing changes in spectral resolution and signal representation.

In the second stage, an alternative sampling program was implemented, evaluating temporal stability and comparing the results with theoretical values. The influence of jitter on digitization quality was analyzed.

I. Introducción

En este laboratorio trabajamos con el microcontrolador Raspberry Pi Pico 2w para aprender cómo funciona el muestreo y el análisis de señales. La idea principal fue tomar una señal senoidal, digitalizarla y observar qué pasa cuando cambiamos parámetros como la frecuencia, la cantidad de puntos en la FFT y la forma en la que se define la tasa de muestreo en el programa. Además, no solo usamos el código base entregado por el profesor, sino que también desarrollamos uno propio para probar cómo se comporta el sistema en condiciones reales, teniendo en cuenta limitaciones como el jitter.

II. Desarrollo de experimentos

Parte 1

Para comenzar la práctica, lo primero que hicimos fue configurar el generador de señales. Ajustamos el equipo para que nos entregara una onda senoidal de 200 Hz, con una amplitud de 1.2 Vpp y un nivel DC de 1.6 V. Una vez configurada la señal, la verificamos en el osciloscopio para asegurarnos de que correspondiera a lo que necesitábamos antes de conectarla al microcontrolador.

Después de esa verificación inicial, llevamos la señal directamente a una de las entradas del conversor A/D del Raspberry Pi Pico, lo que nos permitió empezar el proceso de digitalización. En ese punto ejecutamos el programa ADC_testing.py, disponible en el repositorio del profesor. Este código se encargó de capturar las muestras de la señal y generar dos archivos de salida: muestras.txt y fft.txt.

Con estos archivos, el siguiente paso fue trabajar en Matlab. Allí importamos la información y generamos las gráficas correspondientes. Primero observamos la señal en el dominio del tiempo, donde usamos funciones como plot y stem para facilitar la visualización de los primeros periodos de la onda. Luego pasamos al dominio de la frecuencia, analizando el contenido espectral de la señal.

Código empleado:

practica5.m

/MATLAB Drive/practica5.m

```
% Script para visualizar datos desde MicroPython
% Archivos requeridos: 'muestras.txt' y 'fft.txt'

% --- Cargar datos de muestras ---
t = muestras1.Tiempo_s_;
V = muestras1.Voltaje_V_;

% --- Graficar señal en el dominio del tiempo ---
subplot(2,1,1)
plot(t, V, 'b'); hold on;
stem(t,V,'r');
grid on;
xlabel('Tiempo (s)');
ylabel('Voltaje (V)');
title('Señal Adquirida (Dominio del Tiempo)');
xlim([0 0.020]);

% --- Cargar datos de la FFT ---
f = fft11.Frecuencia_Hz_;
A = fft11.Magnitud_V_;

% --- Graficar espectro de frecuencia ---
subplot(2,1,2)
plot(f, A, 'r');
grid on;
xlabel('Frecuencia (Hz)');
ylabel('Magnitud (V)');
title('Espectro de Frecuencia (FFT)');
xlim([0 max(f)]);
```

Fig. 1. Código implementado en Matlab parte 1.

Luego obtenemos la siguiente gráfica:

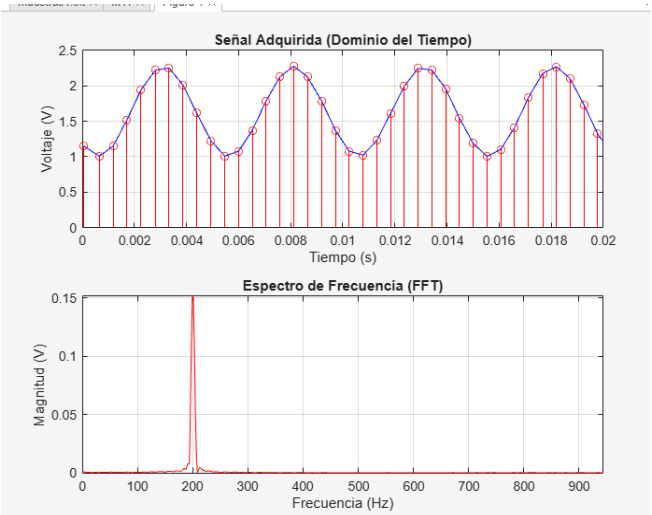

Fig. 2. Señal senoidal 200 Hz, A= 1.2 Vpp y DC= 1.6.

Al observar la gráfica en el dominio del tiempo, se nota claramente que la señal adquirida corresponde a una onda senoidal. Los puntos rojos que representan las muestras siguen muy bien la forma de la curva azul, lo que significa que el proceso de muestreo fue exitoso y que la señal no perdió su forma original.

En el espectro de frecuencia también se ve un resultado muy limpio: aparece un pico bien definido en los 200 Hz, que es exactamente la frecuencia de la señal que generamos. No hay picos extraños ni armónicos fuertes, lo que demuestra que el sistema capturó la señal de manera precisa y sin distorsiones importantes.

Una vez obtuvimos esos resultados base, empezamos a hacer variaciones en el programa. Lo primero fue modificar el número de puntos de la FFT (N_FFT). Probamos con 64, 128, 256, 512, 1024 y 2048 puntos, esto para las frecuencias de 200, 600 y 900 Hz guardando los archivos generados en cada caso y comparando cómo cambiaba la resolución en frecuencia. Allí notamos que con pocos puntos la señal se veía borrosa y poco precisa, mientras que con un mayor número de puntos se lograba distinguir mejor la frecuencia fundamental y sus armónicos.

Para 200 Hz

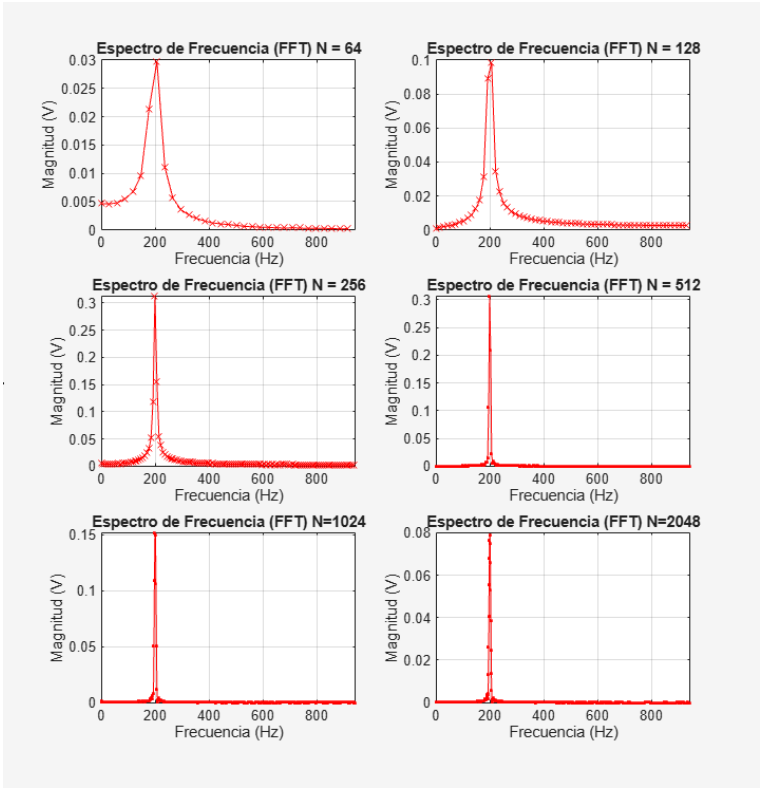

Fig. 3. Señal senoidal 200 Hz diferentes puntos en la FFT.

Para 600 Hz

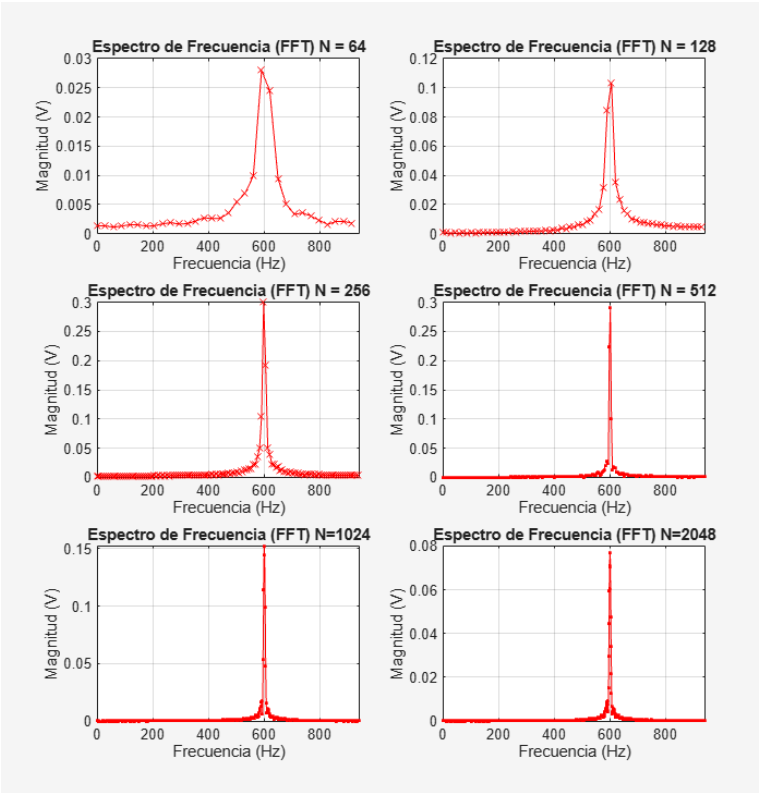

Fig. 4. Señal senoidal 600 Hz diferentes puntos en la FFT.

Para 900 Hz

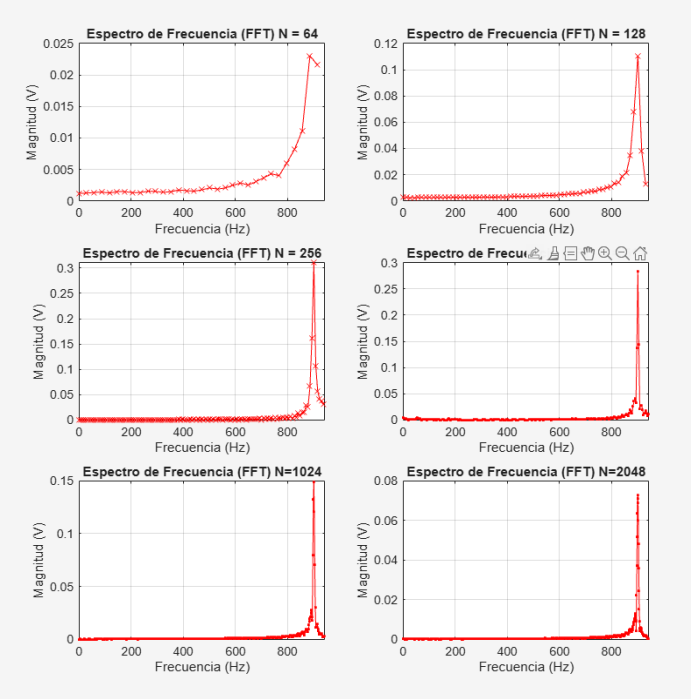

Fig. 5. Señal senoidal 900 Hz diferentes puntos en la FFT.

En la gráfica de la señal a 900 Hz con 64 puntos en la FFT, notamos que el pico en frecuencia no se veía tan limpio como en los demás casos. Esto pasa porque estamos muy cerca del límite que nos permite la frecuencia de muestreo (1000 Hz), así que el sistema apenas alcanza a capturar unas pocas muestras por cada ciclo. Además, al usar tan pocos puntos, la resolución en frecuencia es muy baja y la FFT no logra ubicar con exactitud la señal en 900 Hz. El resultado es que la energía se reparte entre varios bins, lo que hace que el pulso se vea más ancho y distorsionado.

En el código proporcionado por el profesor se establece la tasa de muestreo de manera fija, primero se fija cuántas muestras por segundo queremos (f_muestreo), luego se convierte a un intervalo de tiempo (dt_us), y finalmente ese tiempo se aplica entre cada lectura usando sleep_us. Al final el programa comprueba si lo que logramos realmente coincide con lo que pedimos, lo que nos permite ver de manera práctica que siempre hay una diferencia entre la frecuencia "ideal" y la frecuencia real que se alcanza en el microcontrolador.

Parte 2

En esta segunda parte se creó un código como alternativa para muestrear la señal senoidal de **200 Hz**, con los mismos parámetros de amplitud y nivel DC. La idea era lograr un muestreo más estable y comparar los resultados con lo que debería verse en teoría.

El siguiente código fue el que se implementó para el muestreo:

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

# Configuración inicial
adc = ADC(Pin(27))
N = 512
N_FFT = 1024
f_muestreo = 2000  # Frecuencia objetivo de muestreo (Hz)
dt_us = int(1_000_000 / f_muestreo)
data = array.array('H', [0] * N)

# Adquisición de datos con tiempos uniformes

def acquire_data():
    tiempos = []
    muestras = []
    start = utime.ticks_us()
    next_t = start

    for i in range(N):
        # Espera activa hasta el instante objetivo
        while utime.ticks_diff(next_t, utime.ticks_us()) > 0:
            pass

        # Captura de muestra y tiempo relativo
        t_actual = utime.ticks_diff(utime.ticks_us(), start) / 1_000_000
        tiempos.append(t_actual)
        muestras.append(adc.read_u16())

        # Programa el siguiente instante de muestreo
        next_t = utime.ticks_add(next_t, dt_us)

    # Tiempo total y frecuencia real
    elapsed_time = tiempos[-1] - tiempos[0]
    fs_real = (N - 1) / elapsed_time
    print(f"Frecuencia deseada: {f_muestreo} Hz, frecuencia real: {fs_real:.2f} Hz")

    # Guardar en archivo
    with open("muestras.txt", "w") as f:
        f.write("Tiempo(s)\tVoltaje(V)\n")
        voltajes = [(x / 65535) * 3.3 for x in muestras]
        for t, v in zip(tiempos, voltajes):
            f.write(f"{t:.6f}\t{v:.5f}\n")

    return muestras, fs_real


# Convertir datos a voltaje y eliminar offset DC
def convert_to_voltage(data, VREF=3.3):
    return [(x / 65535) * VREF for x in data]

def remove_offset(data):
    avg_dc = sum(data) / len(data)
    print(f"Offset DC removido: {avg_dc:.3f} V")
```

```
        return [d - avg_dc for d in data]

    # Aplicar ventana Hanning
    def apply_hanning_window(data):
        N = len(data)
        window = [0.5 * (1 - math.cos(2 * math.pi * i / (N - 1))) for i in range(N)]
        return [d * w for d, w in zip(data, window)]

    # FFT manual (Radix-2)
    def fft_manual(x, N_FFT):
        def bit_reversal(n, logN):
            rev = 0
            for i in range(logN):
                if (n >> i) & 1:
                    rev |= 1 << (logN - 1 - i)
            return rev

        X = [complex(v, 0) for v in x[:N_FFT]] + [0]*(N_FFT - len(x))
        logN = int(math.log2(N_FFT))

        for i in range(N_FFT):
            j = bit_reversal(i, logN)
            if j > i:
                X[i], X[j] = X[j], X[i]

        for s in range(logN):
            m = 2 ** (s + 1)
            half_m = m // 2
            w_m = cmath.exp(-2j * math.pi / m)
            for k in range(0, N_FFT, m):
                w = 1
                for j in range(half_m):
                    t = w * X[k + j + half_m]
                    u = X[k + j]
                    X[k + j] = u + t
                    X[k + j + half_m] = u - t
                    w *= w_m
        return X

    # Análisis FFT
    def analyze_fft(fft_result, fs_real, N_FFT):
        magnitudes = [abs(c) / (N_FFT / 2) for c in fft_result[:N_FFT // 2]]
        frequencies = [i * fs_real / N_FFT for i in range(N_FFT // 2)]

        max_index = magnitudes.index(max(magnitudes[1:]))
        dominant_freq = frequencies[max_index]
        signal_amplitude = magnitudes[max_index]

        noise_magnitudes = magnitudes[:]
        noise_magnitudes[max_index] = 0
        noise_floor_v = sum(noise_magnitudes) / (N_FFT // 2 - 1)
        noise_floor_db = 20 * math.log10(noise_floor_v / 3.3)
        SNR = 20 * math.log10(signal_amplitude / noise_floor_v)
        ENOB = (SNR - 1.76) / 6.02

        print(f"Frecuencia dominante: {dominant_freq:.2f} Hz")
        print(f"Amplitud señal: {signal_amplitude:.3f} V")
        print(f"Piso de ruido: {noise_floor_v:.5f} V ({noise_floor_db:.2f} dB FS)")
        print(f"SNR: {SNR:.2f} dB, ENOB: {ENOB:.2f} bits")

        # Guardar FFT en archivo
        with open("fft.txt", "w") as f:
            f.write("Frecuencia(Hz)\tMagnitud(V)\n")
            for freq, mag in zip(frequencies, magnitudes):
                f.write(f"{freq:.2f}\t{mag:.5f}\n")

        return frequencies, magnitudes

    # Programa principal
    def main():
        print("Iniciando adquisición y análisis...")
        muestras, fs_real = acquire_data()
        voltajes = convert_to_voltage(muestras)
        voltajes_sin_offset = remove_offset(voltajes)
        senal_ventaneada = apply_hanning_window(voltajes_sin_offset)

        fft_result = fft_manual(senal_ventaneada, N_FFT)
        analyze_fft(fft_result, fs_real, N_FFT)

    if __name__ == "__main__":
    main()
```

luego implementamos el mismo código de la parte 1 para graficar y obtuvimos la siguiente gráfica:

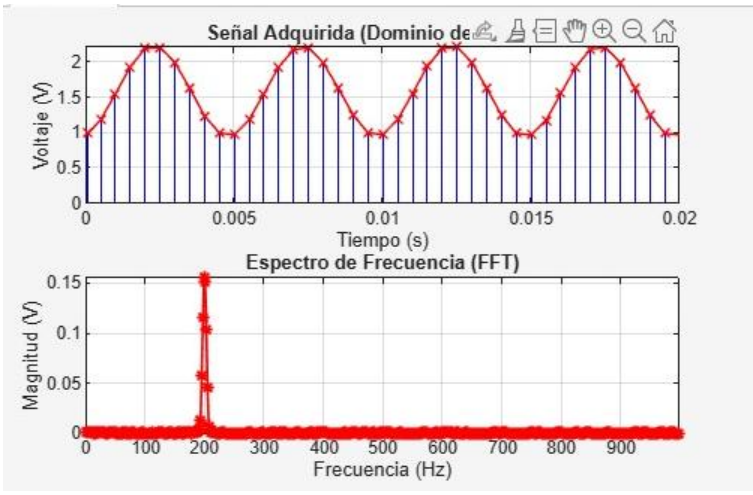

Fig. 6. Parte 2

- ¿Qué es el Jitter? ¿Qué implicaciones tiene en el proceso de codificación de la fuente?

El jitter son esas pequeñas variaciones en el tiempo exacto en que se toman las muestras. Aunque el muestreo debería ser totalmente uniforme, en la práctica siempre hay retrasos que

hacen que unas muestras se tomen un poquito antes o después.

- ¿Qué alternativas existen con el dispositivo Raspberry Pi pico 2W para realizar un muestreo exitoso considerando las características técnicas y posibilidades del dispositivo?

Con la Raspberry Pi Pico 2W existen varias formas de mejorar el muestreo y hacerlo más estable, una opción pueden ser agregar retardos, otra opción es usar temporizadores por hardware, que permiten generar interrupciones precisas en cada intervalo de muestreo, si se necesita aún más precisión, se recomienda trabajar en C/C++, ya que el código se ejecuta más rápido y con menos variaciones de tiempo.

Link repositorio GitHub:
https://github.com/belkyvalentina11/Comunicaci-n-digital-.git