Diseño e Implementación de Filtros FIR

Carrera: Ingeniería en Electrónica y Telecomunicaciones
Materia: Procesamiento Digital de Señales
Profesor: Dr. Alan David Blanco Miranda
Laboratorio: Electrónica
Fecha: 30/04/2025
Duración: 3 hrs.

OBJETIVO DE LA ACTIVIDAD:

- Comprender el concepto de filtros FIR (Finite Impulse Response) y sus propiedades
- Diseñar filtros FIR utilizando el método de la ventana
- Evaluar el efecto de diferentes ventanas en el comportamiento del filtro
- Implementar filtros FIR en Arduino para procesamiento de señales en tiempo real
- Verificar experimentalmente el funcionamiento de los filtros diseñados

HERRAMIENTAS, MATERIAL Y/O REACTIVOS A UTILIZAR:

1. Hardware:

- Generador de funciones (con capacidad hasta 5kHz)
- Osciloscopio digital (mínimo 2 canales)
- Arduino UNO o similar
- Convertidor Digital-Analógico MCP4725 (módulo I2C DAC)
- Protoboard
- Cables jumper
- Resistencias: 2x 10kΩ, 2x 1kΩ, 1x 4.7kΩ
- Capacitores: 1x 0.1µF, 1x 10µF
- Amplificador operacional LM358 o similar
- Cable USB
- Computadora con Python y Arduino IDE instalados

2. Software:

- Arduino IDE (última versión)
- Python 3.x con las siguientes bibliotecas:
    - numpy
    - matplotlib
    - scipy
    - pyserial

  - pandas
- Editor de código (VS Code recomendado)

PROCEDIMIENTO:

PARTE 1: SIMULACIÓN EN PYTHON

1. Crear un archivo Python llamado `fir_filter_design.py` con el siguiente código:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
import pandas as pd


# Funciones para diseño de filtros FIR
def ideal_lp_filter(cutoff, M):
    """
    Diseña un filtro paso-bajas ideal de orden M
    cutoff: frecuencia de corte normalizada (0 a 1, donde 1 es pi
rad/muestra)
    M: orden del filtro (M+1 coeficientes)
    """
    n = np.arange(M+1)
    # Calcular la respuesta al impulso ideal del filtro paso-bajas
    h = np.zeros(M+1)
    # Caso especial para evitar división por cero
    h[M//2] = cutoff
    # Calcular para el resto de los valores
    mask = np.ones(M+1, dtype=bool)
    mask[M//2] = False
    n_masked = n[mask] - M//2
    h[mask] = np.sin(np.pi * cutoff * n_masked) / (np.pi * n_masked)
    return h
```

```
def apply_window(h, window_type='rectangular'):
    """
    Aplica una ventana específica a los coeficientes del filtro
    """
    M = len(h) - 1
    if window_type.lower() == 'rectangular':
        window = np.ones(M+1)
    elif window_type.lower() == 'hanning':
        window = signal.windows.hann(M+1)
    elif window_type.lower() == 'hamming':
        window = signal.windows.hamming(M+1)
    elif window_type.lower() == 'blackman':
        window = signal.windows.blackman(M+1)
    else:
        raise ValueError(f"Tipo de ventana '{window_type}' no
reconocido")

    return h * window

def freqz_normalized(h, fs=1.0, nfft=4096):
    """
    Calcula la respuesta en frecuencia normalizada del filtro
    """
    w, H = signal.freqz(h, worN=nfft)
    f = w * fs / (2 * np.pi)  # Convertir a Hz si se proporciona fs
    return f, H

def plot_filter_response(h, title="Respuesta del Filtro", fs=1.0):
    """
    Grafica la respuesta en el tiempo y en frecuencia del filtro
    """
    M = len(h) - 1
```

```
    n = np.arange(M+1)

    # Obtener respuesta en frecuencia
    f, H = freqz_normalized(h, fs)
    H_mag = np.abs(H)
    H_phase = np.angle(H)
    H_db = 20 * np.log10(H_mag + 1e-10)  # Agregar pequeño valor para evitar log(0)

    fig, axs = plt.subplots(3, 1, figsize=(10, 12))

    # Respuesta al impulso
    axs[0].stem(n, h, use_line_collection=True)
    axs[0].set_title(f"Respuesta al Impulso ({title})")
    axs[0].set_xlabel('Muestra (n)')
    axs[0].set_ylabel('Amplitud')
    axs[0].grid(True)

    # Respuesta en magnitud
    axs[1].plot(f, H_mag)
    axs[1].set_title(f"Respuesta en Magnitud ({title})")
    axs[1].set_xlabel('Frecuencia Normalizada')
    axs[1].set_ylabel('Magnitud')
    axs[1].grid(True)

    # Respuesta en dB
    axs[2].plot(f, H_db)
    axs[2].set_title(f"Respuesta en dB ({title})")
    axs[2].set_xlabel('Frecuencia Normalizada')
    axs[2].set_ylabel('Magnitud (dB)')
    axs[2].set_ylim([-80, 5])
    axs[2].grid(True)
```

```
    plt.tight_layout()
    return fig


def filter_signal(x, h):
    """
    Filtra la señal x con los coeficientes del filtro h
    """
    return np.convolve(x, h, mode='same')


def generate_test_signal(fs=1000, duration=1.0, frequencies=[10, 50, 100,
200]):
    """
    Genera una señal de prueba con múltiples frecuencias
    """
    t = np.arange(0, duration, 1/fs)
    x = np.zeros_like(t)
    for f in frequencies:
        x += np.sin(2 * np.pi * f * t)
    return t, x


def design_filter(filter_type, cutoff, order, window='hamming', fs=1.0):
    """
    Diseña un filtro FIR con los parámetros especificados
    filter_type: 'lowpass', 'highpass', 'bandpass', 'bandstop'
    cutoff: frecuencia de corte normalizada o tupla para
bandpass/bandstop
    order: orden del filtro (debe ser par para HP, BP, BS)
    window: tipo de ventana
    fs: frecuencia de muestreo (para visualización)
    """
    if order % 2 == 0:
```

```
        order += 1  # Asegurarse de que el orden sea impar para fase lineal

    if filter_type.lower() == 'lowpass':
        h_ideal = ideal_lp_filter(cutoff, order)

    elif filter_type.lower() == 'highpass':
        # Diseñar paso-bajas y convertir a paso-altas
        h_lp = ideal_lp_filter(cutoff, order)
        h_ideal = -h_lp  # Invertir
        h_ideal[order//2] += 1  # Agregar impulso en el centro

    elif filter_type.lower() == 'bandpass':
        # Asegurarse de que cutoff es una tupla
        if not isinstance(cutoff, (list, tuple)) or len(cutoff) != 2:
            raise ValueError("Para filtro bandpass, cutoff debe ser una tupla (f_low, f_high)")

        f_low, f_high = cutoff
        # Diseñar dos filtros paso-bajas y restar
        h_lp1 = ideal_lp_filter(f_high, order)
        h_lp2 = ideal_lp_filter(f_low, order)
        h_ideal = h_lp1 - h_lp2

    elif filter_type.lower() == 'bandstop':
        # Asegurarse de que cutoff es una tupla
        if not isinstance(cutoff, (list, tuple)) or len(cutoff) != 2:
            raise ValueError("Para filtro bandstop, cutoff debe ser una tupla (f_low, f_high)")

        f_low, f_high = cutoff
        # Diseñar dos filtros paso-bajas y combinar
        h_lp1 = ideal_lp_filter(f_low, order)
```

```
        h_lp2 = ideal_lp_filter(f_high, order)
        h_ideal = h_lp1 + (np.ones(order+1) - h_lp2)

    else:
        raise ValueError(f"Tipo de filtro '{filter_type}' no reconocido")

    # Aplicar ventana
    h = apply_window(h_ideal, window)

    return h

def export_filter_coeffs(h, filename="filter_coeffs.csv"):
    """
    Exporta los coeficientes del filtro a un archivo CSV y formato
Arduino
    """
    # Guardar en CSV
    pd.DataFrame({'coeff': h}).to_csv(filename, index=False)

    # Formato para Arduino
    arduino_code = "const float filter_coeffs[] = {\n"
    for i, coeff in enumerate(h):
        arduino_code += f"  {coeff:.10f}"
        if i < len(h) - 1:
            arduino_code += ","
        if (i + 1) % 4 == 0:
            arduino_code += "\n"
    if (len(h) % 4) != 0:
        arduino_code += "\n"
    arduino_code += "};\n\n"
    arduino_code += f"const int FILTER_ORDER = {len(h) - 1};\n"
    arduino_code += f"const int FILTER_LENGTH = {len(h)};\n"
```

```
    # Guardar el código Arduino
    with open(filename.replace('.csv', '.h'), 'w') as f:
        f.write(arduino_code)

    print(f"Coeficientes guardados en {filename} y formato Arduino en
{filename.replace('.csv', '.h')}")
    return arduino_code

# Ejemplo 1: Diseño de Filtro Paso-Bajas
def ejemplo_paso_bajas():
    # Parámetros
    fs = 1000  # Hz
    cutoff = 100 / (fs/2)  # Normalizada (100 Hz)
    order = 30  # Orden del filtro

    # Comparar diferentes ventanas
    windows = ['rectangular', 'hanning', 'hamming', 'blackman']
    plt.figure(figsize=(12, 8))

    for i, window in enumerate(windows, 1):
        h = design_filter('lowpass', cutoff, order, window, fs)
        f, H = freqz_normalized(h, fs)
        H_db = 20 * np.log10(np.abs(H) + 1e-10)

        plt.subplot(2, 2, i)
        plt.plot(f, H_db)
        plt.title(f"Paso-Bajas, Ventana {window.capitalize()}")
        plt.xlabel('Frecuencia (Hz)')
        plt.ylabel('Magnitud (dB)')
        plt.grid(True)
        plt.xlim([0, fs/2])
```

```
    plt.ylim([-100, 5])

plt.tight_layout()
plt.savefig("ejemplo_paso_bajas.png")
plt.show()

# Para la ventana Hamming, exportar coeficientes
h_hamming = design_filter('lowpass', cutoff, order, 'hamming', fs)
fig = plot_filter_response(h_hamming, "Paso-Bajas Hamming", fs)
fig.savefig("paso_bajas_hamming.png")
export_filter_coeffs(h_hamming, "lowpass_coeffs.csv")

# Probar con señal de prueba
t, x = generate_test_signal(fs, 1.0, [10, 50, 100, 200, 300])
y = filter_signal(x, h_hamming)

plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(t, x)
plt.title("Señal Original")
plt.xlabel('Tiempo (s)')
plt.ylabel('Amplitud')
plt.grid(True)

plt.subplot(2, 1, 2)
plt.plot(t, y)
plt.title("Señal Filtrada (Paso-Bajas)")
plt.xlabel('Tiempo (s)')
plt.ylabel('Amplitud')
plt.grid(True)

plt.tight_layout()
```

```
    plt.savefig("ejemplo_paso_bajas_senal.png")
    plt.show()

# Ejemplo 2: Diseño de Filtro Paso-Altas
def ejemplo_paso_altas():
    # Parámetros
    fs = 1000  # Hz
    cutoff = 150 / (fs/2)  # Normalizada (150 Hz)
    order = 30  # Orden del filtro

    # Comparar diferentes ventanas
    windows = ['rectangular', 'hanning', 'hamming', 'blackman']
    plt.figure(figsize=(12, 8))

    for i, window in enumerate(windows, 1):
        h = design_filter('highpass', cutoff, order, window, fs)
        f, H = freqz_normalized(h, fs)
        H_db = 20 * np.log10(np.abs(H) + 1e-10)

        plt.subplot(2, 2, i)
        plt.plot(f, H_db)
        plt.title(f"Paso-Altas, Ventana {window.capitalize()}")
        plt.xlabel('Frecuencia (Hz)')
        plt.ylabel('Magnitud (dB)')
        plt.grid(True)
        plt.xlim([0, fs/2])
        plt.ylim([-100, 5])

    plt.tight_layout()
    plt.savefig("ejemplo_paso_altas.png")
    plt.show()
```

```
    # Para la ventana Hamming, exportar coeficientes
    h_hamming = design_filter('highpass', cutoff, order, 'hamming', fs)
    fig = plot_filter_response(h_hamming, "Paso-Altas Hamming", fs)
    fig.savefig("paso_altas_hamming.png")
    export_filter_coeffs(h_hamming, "highpass_coeffs.csv")

    # Probar con señal de prueba
    t, x = generate_test_signal(fs, 1.0, [10, 50, 100, 200, 300])
    y = filter_signal(x, h_hamming)

    plt.figure(figsize=(12, 6))
    plt.subplot(2, 1, 1)
    plt.plot(t, x)
    plt.title("Señal Original")
    plt.xlabel('Tiempo (s)')
    plt.ylabel('Amplitud')
    plt.grid(True)

    plt.subplot(2, 1, 2)
    plt.plot(t, y)
    plt.title("Señal Filtrada (Paso-Altas)")
    plt.xlabel('Tiempo (s)')
    plt.ylabel('Amplitud')
    plt.grid(True)

    plt.tight_layout()
    plt.savefig("ejemplo_paso_altas_senal.png")
    plt.show()

# Ejemplo 3: Diseño de Filtro Paso-Banda
def ejemplo_paso_banda():
    # Parámetros
```

```
fs = 1000  # Hz
cutoff = (100/(fs/2), 250/(fs/2))  # Normalizada (100-250 Hz)
order = 50  # Orden del filtro

# Comparar diferentes ventanas
windows = ['rectangular', 'hanning', 'hamming', 'blackman']
plt.figure(figsize=(12, 8))

for i, window in enumerate(windows, 1):
    h = design_filter('bandpass', cutoff, order, window, fs)
    f, H = freqz_normalized(h, fs)
    H_db = 20 * np.log10(np.abs(H) + 1e-10)

    plt.subplot(2, 2, i)
    plt.plot(f, H_db)
    plt.title(f"Paso-Banda, Ventana {window.capitalize()}")
    plt.xlabel('Frecuencia (Hz)')
    plt.ylabel('Magnitud (dB)')
    plt.grid(True)
    plt.xlim([0, fs/2])
    plt.ylim([-100, 5])

plt.tight_layout()
plt.savefig("ejemplo_paso_banda.png")
plt.show()

# Para la ventana Hamming, exportar coeficientes
h_hamming = design_filter('bandpass', cutoff, order, 'hamming', fs)
fig = plot_filter_response(h_hamming, "Paso-Banda Hamming", fs)
fig.savefig("paso_banda_hamming.png")
export_filter_coeffs(h_hamming, "bandpass_coeffs.csv")
```

```
    # Probar con señal de prueba
    t, x = generate_test_signal(fs, 1.0, [10, 50, 150, 300, 400])
    y = filter_signal(x, h_hamming)

    plt.figure(figsize=(12, 6))
    plt.subplot(2, 1, 1)
    plt.plot(t, x)
    plt.title("Señal Original")
    plt.xlabel('Tiempo (s)')
    plt.ylabel('Amplitud')
    plt.grid(True)

    plt.subplot(2, 1, 2)
    plt.plot(t, y)
    plt.title("Señal Filtrada (Paso-Banda)")
    plt.xlabel('Tiempo (s)')
    plt.ylabel('Amplitud')
    plt.grid(True)

    plt.tight_layout()
    plt.savefig("ejemplo_paso_banda_senal.png")
    plt.show()

# Función principal que ejecuta los ejemplos
def main():
    print("=== DISEÑO DE FILTROS FIR MEDIANTE EL MÉTODO DE LA VENTANA
===")
    print("1. Filtro Paso-Bajas")
    print("2. Filtro Paso-Altas")
    print("3. Filtro Paso-Banda")
    print("4. Diseñar filtro personalizado")
    print("5. Ejecutar todos los ejemplos")
```

```
    print("0. Salir")

    opcion = input("Seleccione una opción: ")

    if opcion == '1':
        ejemplo_paso_bajas()
    elif opcion == '2':
        ejemplo_paso_altas()
    elif opcion == '3':
        ejemplo_paso_banda()
    elif opcion == '4':
        # Diseño personalizado
        tipo = input("Tipo de filtro (lowpass, highpass, bandpass,
bandstop): ")
        fs = float(input("Frecuencia de muestreo (Hz): "))

        if tipo.lower() in ['bandpass', 'bandstop']:
            f_low = float(input("Frecuencia de corte inferior (Hz): "))
            f_high = float(input("Frecuencia de corte superior (Hz): "))
            cutoff = (f_low/(fs/2), f_high/(fs/2))
        else:
            f_c = float(input("Frecuencia de corte (Hz): "))
            cutoff = f_c/(fs/2)

        order = int(input("Orden del filtro: "))
        window = input("Tipo de ventana (rectangular, hanning, hamming,
blackman): ")

        h = design_filter(tipo, cutoff, order, window, fs)
        fig = plot_filter_response(h, f"{tipo.capitalize()}
{window.capitalize()}", fs)
        plt.show()
```

```
        exportar = input("¿Desea exportar los coeficientes? (s/n): ")
        if exportar.lower() == 's':
            nombre = input("Nombre del archivo (sin extensión): ") or
f"{tipo}_filter"
            export_filter_coeffs(h, f"{nombre}.csv")

    elif opcion == '5':
        ejemplo_paso_bajas()
        ejemplo_paso_altas()
        ejemplo_paso_banda()
    elif opcion == '0':
        print("¡Gracias por utilizar el diseñador de filtros FIR!")
    else:
        print("Opción no válida")

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

2. Ejecutar el código y analizar los resultados para comprender:
    - La respuesta al impulso de los diferentes tipos de filtros FIR
    - El efecto de las diferentes ventanas en la respuesta en frecuencia
    - El compromiso entre ancho de la banda de transición y atenuación en la banda de rechazo
    - La relación entre el orden del filtro y su rendimiento

PARTE 2: MONTAJE DE HARDWARE

1. Configuración del generador de funciones
    - Encender el generador
    - Configurar:
        - Forma de onda: Senoidal
        - Frecuencia: Variable (comenzar con 100 Hz)
        - Amplitud: 2Vpp
        - Offset: +1V (para mantener la señal positiva)
2. Montaje del circuito de adquisición

```
[Generador] ---> [R1 10kΩ] ---> |punto A| ---> [R2 10kΩ] ---> GND
                                |punto A| ---> [C1 0.1µF] ---> GND
```

```
|punto A| ---> Arduino A0
```

3. Montaje del circuito de salida con DAC

```
Arduino ---> I2C (SDA: A4, SCL: A5) ---> MCP4725 ---> |salida DAC|
---> [R3 4.7kΩ] ---> |punto B|


|punto B| ---> [C2 10µF] ---> GND


|punto B| ---> Osciloscopio CH2
```

4. Verificación con Osciloscopio
   - Conectar Canal 1 al punto A (entrada)
   - Conectar Canal 2 al punto B (salida)
   - Configurar:
     - Base de tiempo: 2ms/div
     - Voltaje CH1 y CH2: 500mV/div
   - Verificar que ambas señales sean visibles y estables

PARTE 3: IMPLEMENTACIÓN EN ARDUINO

1. Código Arduino para implementación del filtro FIR Crear un nuevo sketch en Arduino IDE con el siguiente código:

```
/*
 * Implementación de Filtro FIR en Arduino
 *
 * Este código implementa un filtro FIR utilizando coeficientes
 * generados por el script de Python.
 */


#include <Wire.h>
#include <Adafruit_MCP4725.h>


// Incluir archivo de coeficientes generado por Python
// Descomentar el filtro que se desea utilizar
#include "lowpass_coeffs.h"
//#include "highpass_coeffs.h"
```

```
//#include "bandpass_coeffs.h"

// Crear objeto DAC
Adafruit_MCP4725 dac;

// Configuración de pines
const int analogInPin = A0;  // Pin analógico de entrada

// Buffer circular para almacenar muestras de entrada
float samples[FILTER_LENGTH];
int sampleIndex = 0;

// Variables para temporización
const unsigned long SAMPLE_PERIOD_US = 1000;  // Período de muestreo en microsegundos (1kHz)
unsigned long lastSampleTime = 0;

// Variables para estadísticas
unsigned long sampleCount = 0;
unsigned long processingTime = 0;

void setup() {
  // Iniciar comunicación serial
  Serial.begin(115200);

  // Iniciar DAC
  dac.begin(0x62);  // Dirección I2C del MCP4725

  // Inicializar buffer de muestras
  for (int i = 0; i < FILTER_LENGTH; i++) {
    samples[i] = 0.0;
  }
```

```
  // Configuración del ADC
  analogReference(DEFAULT);  // Referencia de 5V

  Serial.println("Filtro FIR iniciado");
  Serial.print("Orden del filtro: ");
  Serial.println(FILTER_ORDER);
  Serial.print("Período de muestreo: ");
  Serial.print(SAMPLE_PERIOD_US);
  Serial.println(" us");

  // Esperar estabilización
  delay(1000);
}

void loop() {
  unsigned long currentTime = micros();

  // Verificar si es tiempo de tomar una nueva muestra
  if (currentTime - lastSampleTime >= SAMPLE_PERIOD_US) {
    // Guardar tiempo para cálculo de procesamiento
    unsigned long startProcessingTime = micros();

    // Leer valor analógico (0-1023)
    int sensorValue = analogRead(analogInPin);

    // Convertir a voltaje (0.0-5.0V)
    float voltage = sensorValue * (5.0 / 1023.0);

    // Almacenar en el buffer circular
    samples[sampleIndex] = voltage;
```

```
// Aplicar el filtro FIR
float outputValue = 0.0;
int index = sampleIndex;

for (int i = 0; i < FILTER_LENGTH; i++) {
  outputValue += filter_coeffs[i] * samples[index];
  index = (index > 0) ? index - 1 : FILTER_LENGTH - 1;
}

// Limitar salida entre 0 y 5V
outputValue = constrain(outputValue, 0.0, 5.0);

// Convertir a valor para el DAC (0-4095)
uint16_t dacValue = (uint16_t)(outputValue * 4095.0 / 5.0);

// Enviar al DAC
dac.setVoltage(dacValue, false);

// Actualizar índice del buffer
sampleIndex = (sampleIndex + 1) % FILTER_LENGTH;

// Guardar tiempo de muestreo
lastSampleTime = currentTime;

// Calcular tiempo de procesamiento
processingTime = micros() - startProcessingTime;

// Incrementar contador de muestras
sampleCount++;

// Cada 1000 muestras, mostrar estadísticas
if (sampleCount % 1000 == 0) {
```

```
      Serial.print("Tiempo de procesamiento: ");
      Serial.print(processingTime);
      Serial.println(" us");
    }
  }
}
```

2. Cargar el código en el Arduino
   - Conectar Arduino al PC
   - Seleccionar la placa y puerto correctos
   - Cargar el código
   - Verificar que no haya errores de compilación
3. Configuración del sistema para verificación
   - Abrir el Monitor Serial a 115200 baudios
   - Verificar que el sistema inicia correctamente y muestra la información del filtro
   - Observar los tiempos de procesamiento para asegurar que no superen el período de muestreo

PARTE 4: EXPERIMENTOS CON SEÑALES REALES

1. Configuración inicial
   - Generar una señal senoidal de 100 Hz
   - Observar en el osciloscopio la señal original (CH1) y la señal filtrada (CH2)
   - Verificar que el sistema funciona correctamente
2. Código Python para análisis en tiempo real Crear un archivo `fir_analysis.py` con el siguiente código:

```
import serial
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import time
from scipy import signal


# Configuración
PUERTO_SERIE = 'COM3'  # Cambiar según corresponda
BAUDRATE = 115200
MAX_POINTS = 1000  # Número máximo de puntos a mostrar
```

```
SAMPLE_RATE = 1000  # Hz (corresponde a 1000 us de período de muestreo)

# Inicialización
ser = None
try:
    ser = serial.Serial(PUERTO_SERIE, BAUDRATE)
    print(f"Conectado a {PUERTO_SERIE} a {BAUDRATE} baudios")
    time.sleep(2)  # Esperar inicialización
except Exception as e:
    print(f"Error al conectar: {e}")
    exit()

# Crear figura para visualización
plt.style.use('dark_background')
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))
fig.suptitle('Análisis de Filtro FIR en Tiempo Real')

# Inicializar datos
times = np.linspace(0, MAX_POINTS/SAMPLE_RATE, MAX_POINTS)
input_signal = np.zeros(MAX_POINTS)
output_signal = np.zeros(MAX_POINTS)

# Líneas para gráficos
line_input, = ax1.plot(times, input_signal, 'r-', label='Entrada')
line_output, = ax1.plot(times, output_signal, 'g-', label='Salida')
ax1.set_xlim(0, MAX_POINTS/SAMPLE_RATE)
ax1.set_ylim(0, 5)
ax1.set_xlabel('Tiempo (s)')
ax1.set_ylabel('Voltaje (V)')
ax1.grid(True)
ax1.legend()
```

```
# Gráfico de respuesta en frecuencia
freqs = np.fft.fftfreq(MAX_POINTS, 1/SAMPLE_RATE)
input_fft = np.zeros(MAX_POINTS)
output_fft = np.zeros(MAX_POINTS)
line_input_fft, = ax2.semilogy(freqs[:MAX_POINTS//2],
np.ones(MAX_POINTS//2), 'r-', label='Entrada')
line_output_fft, = ax2.semilogy(freqs[:MAX_POINTS//2],
np.ones(MAX_POINTS//2), 'g-', label='Salida')
ax2.set_xlim(0, SAMPLE_RATE/2)
ax2.set_ylim(1e-3, 1e3)
ax2.set_xlabel('Frecuencia (Hz)')
ax2.set_ylabel('Magnitud')
ax2.grid(True)
ax2.legend()

# Función para actualizar gráficos
def update(frame):
    global input_signal, output_signal, input_fft, output_fft

    # Capturar datos del Arduino
    if ser.in_waiting:
        try:
            line = ser.readline().decode().strip()
            if line.startswith("Input:") and "Output:" in line:
                parts = line.split()
                input_val = float(parts[1])
                output_val = float(parts[3])

                # Actualizar señales
                input_signal = np.roll(input_signal, -1)
                input_signal[-1] = input_val
```

```
                output_signal = np.roll(output_signal, -1)
                output_signal[-1] = output_val

                # Calcular FFT
                input_fft = np.abs(np.fft.fft(input_signal))
                output_fft = np.abs(np.fft.fft(output_signal))

                # Actualizar gráficos
                line_input.set_ydata(input_signal)
                line_output.set_ydata(output_signal)
                line_input_fft.set_ydata(input_fft[:MAX_POINTS//2])
                line_output_fft.set_ydata(output_fft[:MAX_POINTS//2])
            except Exception as e:
                print(f"Error procesando datos: {e}")

    return line_input, line_output, line_input_fft, line_output_fft

# Modificar el código Arduino para enviar datos en formato
# "Input: <valor> Output: <valor>"
print("Iniciando análisis en tiempo real... Presione Ctrl+C para
finalizar")

# Animación
ani = FuncAnimation(fig, update, interval=50, blit=True)
plt.tight_layout()
plt.subplots_adjust(top=0.9)

try:
    plt.show()
except KeyboardInterrupt:
    print("Análisis finalizado")
finally:
```

```
    if ser is not None and ser.is_open:
        ser.close()
        print("Puerto serial cerrado")
```

3. Modificar el código Arduino para enviar datos de entrada y salida Añadir el siguiente código dentro del loop principal, justo después de enviar al DAC:

```
// Enviar datos de entrada y salida para análisis
if (sampleCount % 10 == 0) {  // Enviar cada 10 muestras para no saturar
  Serial.print("Input: ");
  Serial.print(voltage);
  Serial.print(" Output: ");
  Serial.println(outputValue);
}
```

4. Experimento 1: Respuesta a Señales de Diferentes Frecuencias
   - Cargar el filtro paso-bajas en Arduino
   - Ejecutar el script de análisis en Python
   - Generar señales senoidales de diferentes frecuencias:
     - 50 Hz (debería pasar)
     - 100 Hz (cerca de la frecuencia de corte)
     - 200 Hz (debería atenuarse)
     - 400 Hz (debería atenuarse significativamente)
   - Observar la respuesta del filtro en el dominio del tiempo y la frecuencia
   - Anotar las observaciones
5. Experimento 2: Respuesta a Señal con Múltiples Componentes de Frecuencia
   - Configurar el generador para producir una señal cuadrada de 100 Hz
   - Observar en el analizador espectral las múltiples componentes de frecuencia
   - Verificar cómo el filtro paso-bajas elimina las componentes de alta frecuencia
   - Repetir con filtro paso-altas y paso-banda
6. Experimento 3: Efecto del Orden del Filtro
   - Diseñar filtros paso-bajas de diferentes órdenes (15, 31, 63)
   - Cargar cada uno en Arduino y comparar:
     - Ancho de la banda de transición
     - Atenuación en la banda de rechazo
     - Tiempo de procesamiento
   - Encontrar el mejor compromiso para la aplicación
7. Experimento 4: Eliminación de Ruido
   - Añadir ruido a la señal (acercar un teléfono móvil o usar el generador de ruido)
   - Aplicar el filtro paso-banda para extraer la señal deseada

- Comparar la señal original ruidosa con la señal filtrada
- Evaluar la capacidad del filtro para rechazar el ruido

ANÁLISIS DE RESULTADOS

Para cada experimento, analice:

1. Respuesta en frecuencia del filtro:
   - Compare la respuesta teórica (simulación) con la respuesta real
   - Identifique discrepancias y explique sus posibles causas
   - Evalúe el ancho de la banda de transición y la atenuación en la banda de rechazo
2. Efecto del tipo de ventana:
   - ¿Qué ventana ofrece la mejor atenuación en la banda de rechazo?
   - ¿Qué ventana proporciona la banda de transición más estrecha?
   - ¿Cuál es el compromiso entre ancho de banda y atenuación?
3. Comparación entre tipos de filtros:
   - ¿Cómo se comporta cada tipo de filtro (paso-bajas, paso-altas, paso-banda)?
   - ¿Qué tipo de filtro es más eficiente en términos de procesamiento?
   - ¿Cuál es más efectivo para eliminar ruido?
4. Limitaciones del sistema:
   - ¿Cuál es el orden máximo del filtro que puede implementar Arduino sin problemas?
   - ¿Cómo afecta la frecuencia de muestreo al rendimiento del filtro?
   - ¿Qué mejoras podrían realizarse al sistema?

PREGUNTAS DE COMPRENSIÓN

1. ¿Qué es un filtro FIR y cuáles son sus principales ventajas respecto a los filtros IIR?
2. Explique el método de la ventana para el diseño de filtros FIR y sus cuatro pasos principales.
3. ¿Por qué es importante considerar la fase lineal en los filtros FIR? ¿Qué implicaciones tiene en aplicaciones de tiempo real?
4. Compare las ventanas Rectangular, Hanning, Hamming y Blackman en términos de:
   - Ancho del lóbulo principal
   - Atenuación de lóbulos secundarios
   - Aplicaciones recomendadas para cada una
5. ¿Cómo se relaciona el orden del filtro con:
   - La selectividad en frecuencia
   - El costo computacional
   - El retardo de grupo
6. Explique el fenómeno de las oscilaciones de Gibbs en los filtros FIR y cómo las ventanas ayudan a mitigarlo.

7. ¿Cómo afecta la causalidad al diseño de filtros FIR? ¿Qué estrategias pueden usarse para implementar filtros causales?
8. ¿Qué consideraciones deben tenerse al implementar filtros FIR en sistemas con recursos limitados como Arduino?

PARTE 4: EXPERIMENTOS CON SEÑALES REALES GENERADAS POR LOS ESTUDIANTES

1. Generación de Señales Básicas con Arduino A. Generación de Señal Senoidal
   - Crear un nuevo sketch de Arduino llamado `signal_generator.ino` con el siguiente código:

```
/*
 * Generador de Señales con Arduino
 * Genera señales de prueba para el análisis de filtros FIR
 */

#include <Wire.h>
#include <Adafruit_MCP4725.h>

// Crear objeto DAC
Adafruit_MCP4725 dac;

// Parámetros de la señal
const float SAMPLE_RATE = 1000.0;  // Hz
const float SIGNAL_FREQ = 100.0;   // Hz (frecuencia inicial)
byte signalType = 0;  // 0: senoidal, 1: cuadrada, 2: triangular, 3:
rampa, 4: multicomponente

// Variables para temporización
unsigned long lastTime = 0;
const unsigned long SAMPLE_PERIOD_US = 1000000UL / SAMPLE_RATE;

// Variables para control interactivo
```

```
unsigned long lastSerialCheck = 0;
const unsigned long SERIAL_CHECK_PERIOD = 500000;  // 500ms

// Variables para señal multicomponente
const float FREQ_1 = 50.0;   // Hz
const float FREQ_2 = 150.0;  // Hz
const float FREQ_3 = 350.0;  // Hz

void setup() {
  // Iniciar comunicación serial
  Serial.begin(115200);

  // Iniciar DAC
  dac.begin(0x62);  // Dirección I2C del MCP4725

  // Mensaje de inicio
  Serial.println("=== Generador de Señales para Prueba de Filtros FIR
===");
  Serial.println("Comandos disponibles:");
  Serial.println("  's': Señal senoidal");
  Serial.println("  'q': Señal cuadrada");
  Serial.println("  't': Señal triangular");
  Serial.println("  'r': Señal rampa");
  Serial.println("  'm': Señal multicomponente (50Hz + 150Hz + 350Hz)");
  Serial.println("  '+': Aumentar frecuencia");
  Serial.println("  '-': Disminuir frecuencia");
  Serial.println("Generando señal senoidal de 100Hz...");

  // Esperar estabilización
  delay(1000);
}
```

```
void loop() {
  unsigned long currentTime = micros();

  // Verificar comandos seriales
  if (currentTime - lastSerialCheck >= SERIAL_CHECK_PERIOD) {
    checkSerialCommands();
    lastSerialCheck = currentTime;
  }

  // Generar muestra a la frecuencia de muestreo especificada
  if (currentTime - lastTime >= SAMPLE_PERIOD_US) {
    // Calcular el tiempo normalizado (0.0 - 1.0 para un ciclo completo)
    float t = (float)(currentTime) / 1000000.0;  // tiempo en segundos

    // Generar valor según el tipo de señal
    float value = 0.0;

    switch (signalType) {
      case 0:  // Senoidal
        value = 2.5 + 2.0 * sin(2.0 * PI * SIGNAL_FREQ * t);
        break;

      case 1:  // Cuadrada
        value = (sin(2.0 * PI * SIGNAL_FREQ * t) >= 0) ? 4.5 : 0.5;
        break;

      case 2:  // Triangular
        value = 2.5 + 2.0 * (2.0 * abs(((SIGNAL_FREQ * t) -
floor(SIGNAL_FREQ * t + 0.5))) - 0.5);
        break;

      case 3:  // Rampa
```

```
        value = 0.5 + 4.0 * ((SIGNAL_FREQ * t) - floor(SIGNAL_FREQ * t));
        break;

      case 4:  // Multicomponente
        value = 2.5 +
                1.0 * sin(2.0 * PI * FREQ_1 * t) +
                0.5 * sin(2.0 * PI * FREQ_2 * t) +
                0.25 * sin(2.0 * PI * FREQ_3 * t);
        break;
    }

    // Asegurar que el valor esté en el rango válido (0-5V)
    value = constrain(value, 0.0, 5.0);

    // Convertir a valor para el DAC (0-4095)
    uint16_t dacValue = (uint16_t)(value * 4095.0 / 5.0);

    // Enviar al DAC
    dac.setVoltage(dacValue, false);

    // Actualizar tiempo de la última muestra
    lastTime = currentTime;
  }
}

// Función para verificar comandos seriales
void checkSerialCommands() {
  if (Serial.available() > 0) {
    char cmd = Serial.read();

    switch (cmd) {
      case 's':
```

```
    signalType = 0;
    Serial.print("Señal senoidal, ");
    break;

  case 'q':
    signalType = 1;
    Serial.print("Señal cuadrada, ");
    break;

  case 't':
    signalType = 2;
    Serial.print("Señal triangular, ");
    break;

  case 'r':
    signalType = 3;
    Serial.print("Señal rampa, ");
    break;

  case 'm':
    signalType = 4;
    Serial.println("Señal multicomponente (50Hz + 150Hz + 350Hz)");
    return;

  case '+':
    if (SIGNAL_FREQ < 450.0) {
      *((float*)&SIGNAL_FREQ) += 10.0;
    }
    break;

  case '-':
    if (SIGNAL_FREQ > 20.0) {
```

```
            *((float*)&SIGNAL_FREQ) -= 10.0;
          }
          break;

        default:
          return;
      }

      if (signalType != 4) {
        Serial.print("Frecuencia: ");
        Serial.print(SIGNAL_FREQ);
        Serial.println(" Hz");
      }
    }
}
```

B. Montaje del Circuito Generador de Señales

```
Arduino (con signal_generator.ino) ---> I2C (SDA: A4, SCL: A5) --->
MCP4725 ---> |salida DAC| ---> Osciloscopio CH1

|salida DAC| ---> [Divisor de Voltaje] ---> Arduino Filtro A0
```

C. Verificación de Señales

- Cargar el código en el Arduino
- Abrir el Monitor Serial a 115200 baudios
- Probar los diferentes tipos de señales (s, q, t, r, m)
- Observar las señales en el osciloscopio
- Experimentar con diferentes frecuencias (+ para aumentar, - para disminuir)

2. Generación de Señales con Ruido

   A. Modificar el Generador para Incluir Ruido Controlado

   - Actualizar el código `signal_generator.ino` añadiendo las siguientes líneas justo después de las constantes FREQ_X:

```
// Variables para control de ruido
```

```
float noiseLevel = 0.0;  // Nivel de ruido (0.0 - 1.0)
```

- Añadir un nuevo comando al método `checkSerialCommands()`:

```
case 'n':
  // Cambiar el nivel de ruido
  float newLevel = Serial.parseFloat();
  if (newLevel >= 0.0 && newLevel <= 1.0) {
    noiseLevel = newLevel;
    Serial.print("Nivel de ruido: ");
    Serial.println(noiseLevel);
  }
  break;
```

- Modificar la generación de valores añadiendo ruido al final del switch:

```
cpp
// Añadir ruido controlado
if (noiseLevel > 0.0) {
  // Generar ruido aleatorio entre -1.0 y 1.0 y escalarlo según el nivel
  float noise = (random(-1000, 1000) / 1000.0) * noiseLevel;
  value += noise;
  value = constrain(value, 0.0, 5.0);  // Asegurar que se mantiene en
rango
}
```

B. Experimentación con Ruido

- Recargar el código modificado
- Generar una señal senoidal pura (comando 's')
- Añadir gradualmente ruido utilizando el comando 'n' seguido del nivel (ej: 'n0.1', 'n0.2', etc.)
- Observar el efecto del ruido en la señal en el osciloscopio

3. Experimento 1: Filtrado de Señales Generadas por Estudiantes

   A. Configuración del Sistema de Filtrado

- Conectar la salida del generador de señales al Arduino con filtro FIR
- Cargar diferentes tipos de filtros FIR (paso-bajas, paso-altas, paso-banda)
- Utilizar el osciloscopio para observar la señal original y filtrada

B. Procedimiento paso a paso:

- Generar señal senoidal a 100Hz
- Aplicar filtro paso-bajas (fc = 150Hz) y observar que la señal pasa sin atenuación
- Aumentar la frecuencia gradualmente hasta 200Hz y observar la atenuación
- Generar señal cuadrada a 100Hz
- Observar en el osciloscopio las componentes armónicas
- Aplicar filtro paso-bajas y observar cómo se suaviza la señal (eliminación de armónicos)
- Aplicar filtro paso-altas (fc = 80Hz) y observar cómo se resaltan los bordes

4. Experimento 2: Eliminación de Ruido

A. Procedimiento paso a paso:

- Generar señal multicomponente (m)
- Añadir ruido al 20% (n0.2)
- Aplicar filtro paso-banda centrado en 150Hz
- Observar cómo se aísla la componente de 150Hz
- Aplicar filtro paso-bajas para eliminar ruido de alta frecuencia
- Comparar la SNR (relación señal-ruido) antes y después del filtrado

5. Experimento 3: Efecto de las Ventanas en Señales Reales

A. Procedimiento paso a paso:

1. Generar señal multicomponente sin ruido
2. Aplicar filtro paso-banda con diferentes ventanas:
   - Rectangular
   - Hanning
   - Hamming
   - Blackman
3. Comparar capacidad de separación de componentes cercanas
4. Documentar diferencias en la forma de la señal filtrada

6. Experimento 4: Diseño Personalizado para Aplicación Específica

A. Diseño de filtro a medida:

1. Generar una combinación de señales senoidales (ej: 50Hz + 200Hz + 350Hz)
2. Diseñar un filtro para extraer solo la componente de 200Hz
3. Determinar:
   - Orden mínimo necesario
   - Mejor ventana para la aplicación
   - Ancho de banda óptimo
4. Implementar el filtro y verificar su rendimiento

PARTE 5: ANÁLISIS AVANZADO CON PYTHON EN TIEMPO REAL

1. Código Python para Análisis Espectral en Tiempo Real
   - Crear un archivo `real_time_analyzer.py` con el siguiente código:

```
import serial
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import time
from scipy import signal
import tkinter as tk
from tkinter import ttk
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

# Configuración
PUERTO_SERIE = 'COM3'  # Cambiar según corresponda
BAUDRATE = 115200
MAX_POINTS = 1000  # Número máximo de puntos a mostrar
SAMPLE_RATE = 1000  # Hz (corresponde a 1000 us de período de muestreo)

class FilterAnalyzer:
    def __init__(self, root):
        self.root = root
        self.root.title("Analizador de Filtros FIR en Tiempo Real")
        self.root.geometry("1200x800")
```

```
        # Variables de configuración
        self.connected = False
        self.ser = None
        self.input_signal = np.zeros(MAX_POINTS)
        self.output_signal = np.zeros(MAX_POINTS)

        # Variables para el análisis
        self.times = np.linspace(0, MAX_POINTS/SAMPLE_RATE, MAX_POINTS)
        self.freqs = np.fft.fftfreq(MAX_POINTS, 1/SAMPLE_RATE)
        self.input_fft = np.zeros(MAX_POINTS//2)
        self.output_fft = np.zeros(MAX_POINTS//2)

        # Crear la interfaz
        self.create_widgets()

        # Iniciar actualización
        self.update_plot()

    def create_widgets(self):
        # Frame principal
        main_frame = ttk.Frame(self.root, padding="10")
        main_frame.pack(fill=tk.BOTH, expand=True)

        # Frame de control
        control_frame = ttk.Frame(main_frame, padding="5")
        control_frame.pack(fill=tk.X, pady=5)

        # Puerto serial
        ttk.Label(control_frame, text="Puerto
Serial:").pack(side=tk.LEFT, padx=5)
        self.port_entry = ttk.Entry(control_frame, width=10)
        self.port_entry.pack(side=tk.LEFT, padx=5)
```

```
        self.port_entry.insert(0, PUERTO_SERIE)

        # Botón de conexión
        self.connect_button = ttk.Button(control_frame, text="Conectar",
command=self.toggle_connection)
        self.connect_button.pack(side=tk.LEFT, padx=5)

        # Selector de vista
        ttk.Label(control_frame, text="Vista:").pack(side=tk.LEFT,
padx=5)
        self.view_var = tk.StringVar(value="Tiempo")
        view_combo = ttk.Combobox(control_frame,
textvariable=self.view_var,
                                  values=["Tiempo", "Frecuencia",
"Ambos"])
        view_combo.pack(side=tk.LEFT, padx=5)
        view_combo.bind("<<ComboboxSelected>>", self.change_view)

        # Frame para las gráficas
        plot_frame = ttk.Frame(main_frame)
        plot_frame.pack(fill=tk.BOTH, expand=True)

        # Crear figura y ejes
        self.fig = plt.figure(figsize=(10, 8))

        # Vista por defecto: tiempo
        self.ax1 = self.fig.add_subplot(211)
        self.ax2 = self.fig.add_subplot(212)

        # Líneas para gráficos de tiempo
        self.line_input, = self.ax1.plot(self.times, self.input_signal,
'r-', label='Entrada')
        self.line_output, = self.ax1.plot(self.times, self.output_signal,
'g-', label='Salida')
```

```
        self.ax1.set_xlim(0, MAX_POINTS/SAMPLE_RATE)
        self.ax1.set_ylim(0, 5)
        self.ax1.set_xlabel('Tiempo (s)')
        self.ax1.set_ylabel('Voltaje (V)')
        self.ax1.grid(True)
        self.ax1.legend()

        # Líneas para gráficos de frecuencia
        self.line_input_fft, =
self.ax2.semilogy(self.freqs[:MAX_POINTS//2], np.ones(MAX_POINTS//2), 'r-
', label='Entrada')
        self.line_output_fft, =
self.ax2.semilogy(self.freqs[:MAX_POINTS//2], np.ones(MAX_POINTS//2), 'g-
', label='Salida')
        self.ax2.set_xlim(0, SAMPLE_RATE/2)
        self.ax2.set_ylim(1e-3, 1e3)
        self.ax2.set_xlabel('Frecuencia (Hz)')
        self.ax2.set_ylabel('Magnitud')
        self.ax2.grid(True)
        self.ax2.legend()

        self.fig.tight_layout()

        # Añadir figura a la interfaz
        self.canvas = FigureCanvasTkAgg(self.fig, master=plot_frame)
        self.canvas.draw()
        self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

        # Área de log
        log_frame = ttk.Frame(main_frame, padding="5")
        log_frame.pack(fill=tk.X, pady=5)

        ttk.Label(log_frame, text="Log:").pack(side=tk.LEFT, padx=5)
```

```
        self.log_text = tk.Text(log_frame, height=5, width=80)
        self.log_text.pack(fill=tk.X, padx=5)

        # Mensaje inicial
        self.log("Inicializado. Conecte a un puerto serial para comenzar
el análisis.")

    def log(self, message):
        """Añadir mensaje al log con timestamp"""
        timestamp = time.strftime("%H:%M:%S")
        self.log_text.insert(tk.END, f"[{timestamp}] {message}\n")
        self.log_text.see(tk.END)

    def toggle_connection(self):
        """Conectar o desconectar del puerto serial"""
        if not self.connected:
            port = self.port_entry.get()
            try:
                self.ser = serial.Serial(port, BAUDRATE)
                time.sleep(2)  # Esperar inicialización
                self.connected = True
                self.connect_button.config(text="Desconectar")
                self.log(f"Conectado a {port} a {BAUDRATE} baudios")
            except Exception as e:
                self.log(f"Error al conectar: {e}")
        else:
            if self.ser and self.ser.is_open:
                self.ser.close()
            self.connected = False
            self.connect_button.config(text="Conectar")
            self.log("Desconectado")
```

```
    def change_view(self, event=None):
        """Cambiar la vista de las gráficas"""
        view = self.view_var.get()
        self.fig.clear()

        if view == "Tiempo":
            self.ax1 = self.fig.add_subplot(111)
            self.line_input, = self.ax1.plot(self.times,
self.input_signal, 'r-', label='Entrada')
            self.line_output, = self.ax1.plot(self.times,
self.output_signal, 'g-', label='Salida')
            self.ax1.set_xlim(0, MAX_POINTS/SAMPLE_RATE)
            self.ax1.set_ylim(0, 5)
            self.ax1.set_xlabel('Tiempo (s)')
            self.ax1.set_ylabel('Voltaje (V)')
            self.ax1.grid(True)
            self.ax1.legend()

        elif view == "Frecuencia":
            self.ax2 = self.fig.add_subplot(111)
            self.line_input_fft, =
self.ax2.semilogy(self.freqs[:MAX_POINTS//2], self.input_fft, 'r-',
label='Entrada')
            self.line_output_fft, =
self.ax2.semilogy(self.freqs[:MAX_POINTS//2], self.output_fft, 'g-',
label='Salida')
            self.ax2.set_xlim(0, SAMPLE_RATE/2)
            self.ax2.set_ylim(1e-3, 1e3)
            self.ax2.set_xlabel('Frecuencia (Hz)')
            self.ax2.set_ylabel('Magnitud')
            self.ax2.grid(True)
            self.ax2.legend()

        elif view == "Ambos":
```

```
            self.ax1 = self.fig.add_subplot(211)
            self.line_input, = self.ax1.plot(self.times,
self.input_signal, 'r-', label='Entrada')
            self.line_output, = self.ax1.plot(self.times,
self.output_signal, 'g-', label='Salida')
            self.ax1.set_xlim(0, MAX_POINTS/SAMPLE_RATE)
            self.ax1.set_ylim(0, 5)
            self.ax1.set_xlabel('Tiempo (s)')
            self.ax1.set_ylabel('Voltaje (V)')
            self.ax1.grid(True)
            self.ax1.legend()

            self.ax2 = self.fig.add_subplot(212)
            self.line_input_fft, =
self.ax2.semilogy(self.freqs[:MAX_POINTS//2], self.input_fft, 'r-',
label='Entrada')
            self.line_output_fft, =
self.ax2.semilogy(self.freqs[:MAX_POINTS//2], self.output_fft, 'g-',
label='Salida')
            self.ax2.set_xlim(0, SAMPLE_RATE/2)
            self.ax2.set_ylim(1e-3, 1e3)
            self.ax2.set_xlabel('Frecuencia (Hz)')
            self.ax2.set_ylabel('Magnitud')
            self.ax2.grid(True)
            self.ax2.legend()

        self.fig.tight_layout()
        self.canvas.draw()

    def update_plot(self):
        """Actualizar las gráficas con nuevos datos"""
        if self.connected and self.ser and self.ser.is_open:
            if self.ser.in_waiting:
                try:
```

```
                    line = self.ser.readline().decode().strip()
                    if line.startswith("Input:") and "Output:" in line:
                        parts = line.split()
                        input_val = float(parts[1])
                        output_val = float(parts[3])

                        # Actualizar señales
                        self.input_signal = np.roll(self.input_signal, -
1)
                        self.input_signal[-1] = input_val

                        self.output_signal = np.roll(self.output_signal,
-1)
                        self.output_signal[-1] = output_val

                        # Calcular FFT
                        self.input_fft =
np.abs(np.fft.fft(self.input_signal))[:MAX_POINTS//2]
                        self.output_fft =
np.abs(np.fft.fft(self.output_signal))[:MAX_POINTS//2]

                        # Actualizar gráficos según la vista actual
                        view = self.view_var.get()

                        if view in ["Tiempo", "Ambos"]:
                            self.line_input.set_ydata(self.input_signal)

self.line_output.set_ydata(self.output_signal)

                        if view in ["Frecuencia", "Ambos"]:
                            self.line_input_fft.set_ydata(self.input_fft)

self.line_output_fft.set_ydata(self.output_fft)
```

```
                self.canvas.draw_idle()
            except Exception as e:
                self.log(f"Error procesando datos: {e}")

        # Programar próxima actualización
        self.root.after(50, self.update_plot)

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

2. Modificar el código Arduino de filtro para enviar datos de entrada y salida
    - Añadir el siguiente código dentro del loop principal, justo después de enviar al DAC:

```
// Enviar datos de entrada y salida para análisis
if (sampleCount % 10 == 0) {  // Enviar cada 10 muestras para no saturar
  Serial.print("Input: ");
  Serial.print(voltage);
  Serial.print(" Output: ");
  Serial.println(outputValue);
}
```

3. Experimento de Análisis en Tiempo Real
    - Ejecutar el script `real_time_analyzer.py`
    - Conectar al puerto serial correspondiente
    - Generar diferentes señales con el Arduino generador
    - Observar el análisis en tiempo y frecuencia
    - Identificar el efecto de los diferentes filtros FIR en el espectro

ANÁLISIS DE RESULTADOS

Para cada experimento, analice:

1. Respuesta en frecuencia del filtro:
    - Compare la respuesta teórica (simulación) con la respuesta real
    - Identifique discrepancias y explique sus posibles causas
    - Evalúe el ancho de la banda de transición y la atenuación en la banda de rechazo
2. Efecto del tipo de ventana:
    - ¿Qué ventana ofrece la mejor atenuación en la banda de rechazo?
    - ¿Qué ventana proporciona la banda de transición más estrecha?
    - ¿Cuál es el compromiso entre ancho de banda y atenuación?
3. Comparación entre tipos de filtros:
    - ¿Cómo se comporta cada tipo de filtro (paso-bajas, paso-altas, paso-banda)?
    - ¿Qué tipo de filtro es más eficiente en términos de procesamiento?
    - ¿Cuál es más efectivo para eliminar ruido?
4. Limitaciones del sistema:
    - ¿Cuál es el orden máximo del filtro que puede implementar Arduino sin problemas?
    - ¿Cómo afecta la frecuencia de muestreo al rendimiento del filtro?
    - ¿Qué mejoras podrían realizarse al sistema?

PREGUNTAS DE COMPRENSIÓN

1. ¿Qué es un filtro FIR y cuáles son sus principales ventajas respecto a los filtros IIR?
2. Explique el método de la ventana para el diseño de filtros FIR y sus cuatro pasos principales.
3. ¿Por qué es importante considerar la fase lineal en los filtros FIR? ¿Qué implicaciones tiene en aplicaciones de tiempo real?
4. Compare las ventanas Rectangular, Hanning, Hamming y Blackman en términos de:
    - Ancho del lóbulo principal
    - Atenuación de lóbulos secundarios
    - Aplicaciones recomendadas para cada una
5. ¿Cómo se relaciona el orden del filtro con:
    - La selectividad en frecuencia
    - El costo computacional
    - El retardo de grupo
6. Explique el fenómeno de las oscilaciones de Gibbs en los filtros FIR y cómo las ventanas ayudan a mitigarlo.
7. ¿Cómo afecta la causalidad al diseño de filtros FIR? ¿Qué estrategias pueden usarse para implementar filtros causales?
8. ¿Qué consideraciones deben tenerse al implementar filtros FIR en sistemas con recursos limitados como Arduino?

RECOMENDACIONES Y TIPS

- Verificar siempre las conexiones antes de energizar el circuito

- Asegurarse de que la señal de entrada no exceda el rango permitido por el ADC (0-5V)
- Exportar los coeficientes con suficiente precisión (al menos 8 decimales)
- Monitorear el tiempo de procesamiento para evitar problemas de sobrecarga
- Guardar las gráficas generadas para el informe
- Experimentar con diferentes órdenes y tipos de ventana para encontrar el mejor compromiso
- Utilizar funciones de la biblioteca scipy.signal para verificar resultados

ANEXO: IMPLEMENTACIÓN CON STM32F746G-DISCO

Descripción de la Plataforma STM32F746G-DISCO

La STM32F746G-DISCO es una placa de desarrollo basada en el microcontrolador STM32F746NGH6 de ST Microelectronics. Esta placa ofrece ventajas significativas para el procesamiento de señales digitales en comparación con Arduino:

- CPU ARM Cortex-M7 a 216 MHz (vs 16 MHz en Arduino UNO)
- 1MB de memoria Flash y 320KB de RAM (mucho mayor que Arduino)
- Conversor ADC de 12 bits (vs 10 bits en Arduino)
- Conversor DAC integrado de 12 bits (no requiere hardware externo)
- DSP optimizado para operaciones de procesamiento de señal
- Pantalla táctil LCD-TFT a color de 4.3" (para visualización directa)
- Múltiples interfaces de comunicación

Preparación del Entorno de Desarrollo

1. Instalación del Software Necesario
   - Descargar e instalar STM32CubeIDE desde st.com
   - Instalar STM32CubeMX (incluido en STM32CubeIDE)
   - Instalar el paquete de firmware STM32F7xx
2. Configuración Inicial del Proyecto
   - Abrir STM32CubeMX
   - Crear un nuevo proyecto seleccionando la placa STM32F746G-DISCO
   - Configurar los periféricos necesarios:
     - ADC1 para adquisición de señales
     - DAC1 para generación de señales
     - UART para comunicación con PC
     - LCD para visualización

Implementación Paso a Paso

PARTE 1: Configuración del Hardware

1. Configuración de Periféricos con STM32CubeMX
   - Abrir STM32CubeMX y seleccionar "File > New Project"
   - En la pestaña "Board Selector", buscar "STM32F746G-DISCO" y seleccionarlo
   - En la vista de configuración de pines, configurar:

A. Configuración de ADC:

   - Seleccionar ADC1 en PA0 (CN8-Pin 1, entrada analógica externa)
   - Configurar para conversión continua
   - Resolución: 12 bits
   - Modo de escaneo: Habilitado
   - Modo DMA: Circular

B. Configuración de DAC:

- Seleccionar DAC en PA4 (salida analógica)
- Modo de trigger: Mediante software
- Resolución: 12 bits

C. Configuración de UART para depuración:

- Habilitar USART1
- Modo: Asíncrono
- Velocidad: 115200 bps
- Word Length: 8 bits
- Paridad: Ninguna
- Stop Bits: 1

D. Configuración del LCD:

- Habilitar LTDC
- Modo: RGB565

2. Configuración de Timers para Muestreo
   - Habilitar TIM6 para el disparo del ADC
   - Configurar la frecuencia del timer para 1000 Hz (1 ms)
3. Configuración del DMA
   - Configurar DMA para ADC1 en modo circular
   - Configurar DMA para DAC1
4. Configuración del Reloj
   - Configurar PLL para 216 MHz (frecuencia máxima)
5. Generar el Código Inicial
   - En STM32CubeMX, ir a "Project > Generate Code"
   - Seleccionar STM32CubeIDE como toolchain
   - Dar un nombre al proyecto y ubicación
   - Hacer clic en "Generate"

PARTE 2: Desarrollo del Software

1. Estructura del Proyecto
   - Abrir el proyecto generado en STM32CubeIDE
   - Organizar la estructura de archivos:
     - `main.c`: Contiene la función principal
     - `signal_processing.c`: Funciones de procesamiento de señales
     - `filter_coeffs.h`: Coeficientes de los filtros

- `display.c`: Funciones para la visualización

2. Definición de Coeficientes de Filtros
   - Crear el archivo `filter_coeffs.h` con los siguientes contenidos:

```
/**
 * @file filter_coeffs.h
 * @brief Coeficientes para diferentes tipos de filtros FIR
 */

#ifndef FILTER_COEFFS_H
#define FILTER_COEFFS_H

// Definiciones de tamaño de filtros
#define FILTER_ORDER_LOW  31  // Orden bajo para pruebas básicas
#define FILTER_ORDER_MED  63  // Orden medio para mejor rendimiento
#define FILTER_ORDER_HIGH 127 // Orden alto para alta selectividad

// Filtro Paso-Bajas (fc = 150Hz, fs = 1000Hz, ventana Hamming)
static const float LP_FILTER_COEFFS[FILTER_ORDER_MED + 1] = {
  // Aquí se insertan los coeficientes generados por Python
  // Los estudiantes deben generar estos coeficientes usando el código Python proporcionado
  0.0010072326, 0.0011642754, 0.0016174478, 0.0023609131, 0.0033889611,
  0.0046968487, 0.0062755863, 0.0081111661, 0.0101869913, 0.0124837891,
  0.0149793453, 0.0176488733, 0.0204651506, 0.0233998768, 0.0264240255,
  0.0295081456, 0.0326237162, 0.0357429174, 0.0388393474, 0.0418887962,
  0.0448694726, 0.0477619097, 0.0505495499, 0.0532183307, 0.0557563137,
  0.0581534016, 0.0604020333, 0.0624967055, 0.0644336069, 0.0662102845,
  0.0678258901, 0.0692817276, 0.0705803072, 0.0717254173, 0.0727217981,
  0.0735751103, 0.0742918108, 0.0748793153, 0.0753456493, 0.0756893174,
  0.0759093676, 0.0760056778, 0.0759782333, 0.0758273047, 0.0755535293,
  0.0751578834, 0.0746417599, 0.0740068329, 0.0732550319, 0.0723885187,
  0.0714096653, 0.0703210268, 0.0691253165, 0.0678254918, 0.0664247396,
```

```
  0.0649264563, 0.0633342198, 0.0616517702, 0.0598830865, 0.0580322665,
  0.0561035019, 0.0541010647, 0.0520292918, 0.0498924665, 0.0476948972
  // Continuar con todos los coeficientes...
};

// Filtro Paso-Altas (fc = 200Hz, fs = 1000Hz, ventana Hamming)
static const float HP_FILTER_COEFFS[FILTER_ORDER_MED + 1] = {
  // Aquí se insertan los coeficientes generados por Python
  // Los estudiantes deben generar estos coeficientes usando el código
Python proporcionado
  -0.0005467342, -0.0007358114, -0.0011243527, -0.0017123452, -
0.0024963388,
  -0.0034696248, -0.0046225584, -0.0059420613, -0.0074111414, -
0.0090084775,
  -0.0107086531, -0.0124828764, -0.0143000564, -0.0161266673, -
0.0179274639,
  -0.0196661196, -0.0213055761, -0.0228088036, -0.0241392797, -
0.0252614997,
  -0.0261421795, -0.0267508971, -0.0270607673, -0.0270493276, -
0.0266996299,
  -0.0259995057, -0.0249413184, -0.0235218156, -0.0217425742, -
0.0196096487,
  -0.0171330333, 0.9586354142, -0.0171330333, -0.0196096487, -
0.0217425742,
  -0.0235218156, -0.0249413184, -0.0259995057, -0.0266996299, -
0.0270493276,
  -0.0270607673, -0.0267508971, -0.0261421795, -0.0252614997, -
0.0241392797,
  -0.0228088036, -0.0213055761, -0.0196661196, -0.0179274639, -
0.0161266673,
  -0.0143000564, -0.0124828764, -0.0107086531, -0.0090084775, -
0.0074111414,
  -0.0059420613, -0.0046225584, -0.0034696248, -0.0024963388, -
0.0017123452,
  -0.0011243527, -0.0007358114, -0.0005467342
  // Continuar con todos los coeficientes...
```

```
};

// Filtro Paso-Banda (fc1 = 150Hz, fc2 = 250Hz, fs = 1000Hz, ventana Hamming)
static const float BP_FILTER_COEFFS[FILTER_ORDER_MED + 1] = {
  // Aquí se insertan los coeficientes generados por Python
  // Los estudiantes deben generar estos coeficientes usando el código Python proporcionado
  0.0003784615, 0.0005364735, 0.0007751174, 0.0010905473, 0.0014732131,
  0.0019082894, 0.0023757827, 0.0028514245, 0.0033073452, 0.0037135551,
  0.0040380647, 0.0042485693, 0.0043135563, 0.0042039159, 0.0038947693,
  0.0033670764, 0.0026101877, 0.0016226072, 0.0004128299, -0.0010036927,
  -0.0026049491, -0.0043611471, -0.0062343284, -0.0081778515, -0.0101390553,
  -0.0120622962, -0.0138901861, -0.0155653807, -0.0170323437, -0.0182391666,
  -0.0191389342, -0.0196900871, -0.0198578276, -0.0196144069, -0.0189387688,
  -0.0178161387, -0.0162388333, -0.0142061818, -0.0117252578, -0.0088106903,
  -0.0054844845, -0.0017741694, 0.0022991035, 0.0067107857, 0.0114331975,
  0.0164350333, 0.0216814493, 0.0271346561, 0.0327539566, 0.0384968317,
  0.0443195764, 0.0501782842, 0.0560292175, 0.0618292302, 0.0675366465,
  0.0731114617, 0.0785161077, 0.0837151095, 0.0886752172, 0.0933652511,
  0.0977565478, 0.1018227095, 0.1055395953, 0.1088855293, 0.1118414972
  // Continuar con todos los coeficientes...
};

// Filtro Rechaza-Banda (fc1 = 150Hz, fc2 = 250Hz, fs = 1000Hz, ventana Hamming)
static const float BS_FILTER_COEFFS[FILTER_ORDER_MED + 1] = {
  // Aquí se insertan los coeficientes generados por Python
  // deben generar estos coeficientes usando el código Python proporcionado
```

```
    // ...
};

// Función para seleccionar el conjunto de coeficientes según el tipo de filtro
const float* get_filter_coeffs(int filter_type, int* length) {
    switch (filter_type) {
        case 0: // Paso-bajas
            *length = FILTER_ORDER_MED + 1;
            return LP_FILTER_COEFFS;
        case 1: // Paso-altas
            *length = FILTER_ORDER_MED + 1;
            return HP_FILTER_COEFFS;
        case 2: // Paso-banda
            *length = FILTER_ORDER_MED + 1;
            return BP_FILTER_COEFFS;
        case 3: // Rechaza-banda
            *length = FILTER_ORDER_MED + 1;
            return BS_FILTER_COEFFS;
        default:
            *length = 0;
            return NULL;
    }
}

#endif /* FILTER_COEFFS_H */
```

3. Implementación de Procesamiento de Señales
    - Crear archivo `signal_processing.h`:

```
/**
 * @file signal_processing.h
 * @brief Funciones para procesamiento de señales
 */
```

```c
#ifndef SIGNAL_PROCESSING_H
#define SIGNAL_PROCESSING_H

#include <stdint.h>

// Definiciones para tipos de filtros
#define FILTER_TYPE_LOWPASS    0
#define FILTER_TYPE_HIGHPASS   1
#define FILTER_TYPE_BANDPASS   2
#define FILTER_TYPE_BANDSTOP   3

// Funciones de inicialización
void SignalProcessing_Init(void);

// Funciones de procesamiento
void FIR_Filter_Init(int filter_type);
float FIR_Filter_Process(float input);

// Funciones de generación de señales
float Generate_Sine_Wave(float frequency, float amplitude, float offset);
float Generate_Square_Wave(float frequency, float amplitude, float
offset);
float Generate_Triangle_Wave(float frequency, float amplitude, float
offset);
float Generate_Sawtooth_Wave(float frequency, float amplitude, float
offset);
float Generate_Multi_Tone(float f1, float f2, float f3, float a1, float
a2, float a3, float offset);

// Funciones de análisis
void Calculate_FFT(float* input, float* output, int length);
float Calculate_SNR(float* signal, float* noise, int length);
```

```
#endif /* SIGNAL_PROCESSING_H */
```

- Crear archivo `signal_processing.c`:

```
/**
 * @file signal_processing.c
 * @brief Implementación de funciones para procesamiento de señales
 */

#include "signal_processing.h"
#include "filter_coeffs.h"
#include <math.h>
#include <string.h>

// Variables para filtro FIR
static float buffer[128];  // Buffer circular, tamaño máximo para el filtro más grande
static int buffer_index = 0;
static const float* current_coeffs = NULL;
static int current_filter_length = 0;
static int current_filter_type = -1;

// Inicialización del módulo de procesamiento de señales
void SignalProcessing_Init(void) {
    // Inicializar buffer
    memset(buffer, 0, sizeof(buffer));
    buffer_index = 0;

    // Inicializar con filtro paso-bajas por defecto
    FIR_Filter_Init(FILTER_TYPE_LOWPASS);
}
```

```
// Inicialización del filtro FIR
void FIR_Filter_Init(int filter_type) {
    if (filter_type != current_filter_type) {
        current_filter_type = filter_type;
        current_coeffs = get_filter_coeffs(filter_type,
&current_filter_length);

        // Reiniciar buffer
        memset(buffer, 0, sizeof(buffer));
        buffer_index = 0;
    }
}

// Procesamiento de una muestra con filtro FIR
float FIR_Filter_Process(float input) {
    if (current_coeffs == NULL || current_filter_length == 0) {
        return input;  // Sin filtro, pasar señal directamente
    }

    // Almacenar muestra en buffer circular
    buffer[buffer_index] = input;

    // Aplicar filtro FIR
    float output = 0.0f;
    int idx = buffer_index;

    for (int i = 0; i < current_filter_length; i++) {
        output += current_coeffs[i] * buffer[idx];
        idx = (idx > 0) ? idx - 1 : current_filter_length - 1;
    }

    // Actualizar índice del buffer
```

```
    buffer_index = (buffer_index + 1) % current_filter_length;

    return output;
}

// Generación de onda senoidal
float Generate_Sine_Wave(float frequency, float amplitude, float offset)
{
    static float phase = 0.0f;
    float value = offset + amplitude * sinf(phase);

    // Actualizar fase para la próxima muestra
    phase += 2.0f * M_PI * frequency / 1000.0f;  // Asumiendo fs = 1000Hz
    if (phase >= 2.0f * M_PI) {
        phase -= 2.0f * M_PI;  // Mantener fase en [0, 2π)
    }

    return value;
}

// Generación de onda cuadrada
float Generate_Square_Wave(float frequency, float amplitude, float
offset) {
    static float phase = 0.0f;
    float value = offset + amplitude * (sinf(phase) >= 0.0f ? 1.0f : -
1.0f);

    // Actualizar fase para la próxima muestra
    phase += 2.0f * M_PI * frequency / 1000.0f;  // Asumiendo fs = 1000Hz
    if (phase >= 2.0f * M_PI) {
        phase -= 2.0f * M_PI;  // Mantener fase en [0, 2π)
    }
```

```
    return value;
}

// Generación de onda triangular
float Generate_Triangle_Wave(float frequency, float amplitude, float
offset) {
    static float phase = 0.0f;
    float normalized_phase = phase / (2.0f * M_PI);
    float triangle = 2.0f * fabsf(2.0f * (normalized_phase -
floorf(normalized_phase + 0.5f))) - 1.0f;
    float value = offset + amplitude * triangle;

    // Actualizar fase para la próxima muestra
    phase += 2.0f * M_PI * frequency / 1000.0f;  // Asumiendo fs = 1000Hz
    if (phase >= 2.0f * M_PI) {
        phase -= 2.0f * M_PI;  // Mantener fase en [0, 2π)
    }

    return value;
}

// Generación de onda de sierra
float Generate_Sawtooth_Wave(float frequency, float amplitude, float
offset) {
    static float phase = 0.0f;
    float normalized_phase = phase / (2.0f * M_PI);
    float sawtooth = 2.0f * (normalized_phase - floorf(normalized_phase +
0.5f));
    float value = offset + amplitude * sawtooth;

    // Actualizar fase para la próxima muestra
    phase += 2.0f * M_PI * frequency / 1000.0f;  // Asumiendo fs = 1000Hz
    if (phase >= 2.0f * M_PI) {
```

```
        phase -= 2.0f * M_PI;  // Mantener fase en [0, 2π)
    }

    return value;
}

// Generación de señal multitono
float Generate_Multi_Tone(float f1, float f2, float f3, float a1, float
a2, float a3, float offset) {
    static float phase1 = 0.0f, phase2 = 0.0f, phase3 = 0.0f;

    // Calcular valor combinado
    float value = offset +
                  a1 * sinf(phase1) +
                  a2 * sinf(phase2) +
                  a3 * sinf(phase3);

    // Actualizar fases
    phase1 += 2.0f * M_PI * f1 / 1000.0f;
    if (phase1 >= 2.0f * M_PI) phase1 -= 2.0f * M_PI;

    phase2 += 2.0f * M_PI * f2 / 1000.0f;
    if (phase2 >= 2.0f * M_PI) phase2 -= 2.0f * M_PI;

    phase3 += 2.0f * M_PI * f3 / 1000.0f;
    if (phase3 >= 2.0f * M_PI) phase3 -= 2.0f * M_PI;

    return value;
}

// Nota: Las funciones de análisis FFT y SNR requieren la implementación
// de CMSIS-DSP que está disponible en STM32F7
```

4. Implementación de la Interfaz de Usuario y Visualización
    - Crear `display.h`:

```
/**
 * @file display.h
 * @brief Funciones para visualización en pantalla
 */

#ifndef DISPLAY_H
#define DISPLAY_H

#include <stdint.h>

// Inicialización de la pantalla
void Display_Init(void);

// Dibujar señal en tiempo
void Display_DrawSignal(float* signal, int length, uint32_t color);

// Dibujar espectro de frecuencia
void Display_DrawSpectrum(float* spectrum, int length, uint32_t color);

// Mostrar información textual
void Display_ShowText(const char* text, int x, int y, uint32_t color);

// Actualizar la pantalla
void Display_Update(void);

// Manejar interacción táctil
void Display_HandleTouch(void);

#endif /* DISPLAY_H */
```

5. Implementación del Programa Principal
    - Modificar `main.c` para incluir la implementación completa:

```
/* USER CODE BEGIN Header */
/**

******************************************************************************
  * @file           : main.c
  * @brief          : Main program body

******************************************************************************
  */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "adc.h"
#include "dac.h"
#include "dma.h"
#include "ltdc.h"
#include "tim.h"
#include "usart.h"
#include "gpio.h"
#include "fmc.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "signal_processing.h"
#include "display.h"
#include <stdio.h>
#include <string.h>
```

```
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
#define ADC_BUF_SIZE 1000
#define DAC_BUF_SIZE 1000
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
/* USER CODE BEGIN PV */
// Buffers para ADC y DAC
uint16_t adc_buffer[ADC_BUF_SIZE];
uint16_t dac_buffer[DAC_BUF_SIZE];

// Buffers para procesamiento
float input_signal[ADC_BUF_SIZE];
float output_signal[ADC_BUF_SIZE];
float input_spectrum[ADC_BUF_SIZE/2];
float output_spectrum[ADC_BUF_SIZE/2];
```

```
// Variables de control
volatile uint8_t adc_complete_flag = 0;
volatile uint32_t sample_count = 0;
volatile uint8_t current_filter_type = 0;  // 0: LP, 1: HP, 2: BP, 3: BS
volatile uint8_t current_signal_type = 0;  // 0: External, 1: Sine, 2: Square, 3: Triangle, 4: Multi

// Parámetros de señal
float signal_freq = 100.0f;
float signal_amplitude = 2.0f;
float signal_offset = 2.0f;
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------*/
void SystemClock_Config(void);
/* USER CODE BEGIN PFP */
void Process_Signal(void);
void Update_Display(void);
void Handle_UART_Commands(void);
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
```

```c
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_ADC1_Init();
  MX_DAC_Init();
  MX_FMC_Init();
  MX_LTDC_Init();
  MX_TIM6_Init();
```

```
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */

  // Inicializar módulos de aplicación
  SignalProcessing_Init();
  Display_Init();

  // Iniciar ADC en modo DMA
  HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buffer, ADC_BUF_SIZE);

  // Iniciar DAC en modo DMA
  HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_1, (uint32_t*)dac_buffer,
DAC_BUF_SIZE, DAC_ALIGN_12B_R);

  // Iniciar timer para muestreo
  HAL_TIM_Base_Start(&htim6);

  // Mostrar pantalla de bienvenida
  Display_ShowText("STM32F7 Filter Lab", 10, 10, 0xFFFFFFFF);
  Display_ShowText("Inicializando...", 10, 30, 0xFFFFFFFF);
  Display_Update();
  HAL_Delay(1000);

  // Mensaje inicial UART
  printf("STM32F7 Filtros FIR - Inicializado\r\n");
  printf("Comandos disponibles:\r\n");
  printf("  'l': Filtro paso-bajas\r\n");
  printf("  'h': Filtro paso-altas\r\n");
  printf("  'b': Filtro paso-banda\r\n");
  printf("  's': Filtro rechaza-banda\r\n");
  printf("  'e': Señal externa (ADC)\r\n");
  printf("  'i': Señal senoidal interna\r\n");
```

```
  printf("  'q': Señal cuadrada interna\r\n");
  printf("  't': Señal triangular interna\r\n");
  printf("  'm': Señal multitono interna\r\n");
  printf("  '+': Aumentar frecuencia\r\n");
  printf("  '-': Disminuir frecuencia\r\n");

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    // Verificar si hay datos ADC completos
    if (adc_complete_flag) {
      Process_Signal();
      Update_Display();
      adc_complete_flag = 0;
    }

    // Verificar comandos UART
    Handle_UART_Commands();

    // Manejar interacción táctil
    Display_HandleTouch();

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

```
/* USER CODE BEGIN 4 */
// Callback de ADC completo
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) {
  adc_complete_flag = 1;
  sample_count++;
}

// Procesamiento de señal
void Process_Signal(void) {
  // Convertir datos ADC a valores flotantes
  for (int i = 0; i < ADC_BUF_SIZE; i++) {
    // Normalizar a voltaje (0-3.3V)
    if (current_signal_type == 0) {
      // Señal externa del ADC
      input_signal[i] = (float)adc_buffer[i] * 3.3f / 4096.0f;
    } else {
      // Señal generada internamente
      switch (current_signal_type) {
        case 1: // Seno
          input_signal[i] = Generate_Sine_Wave(signal_freq,
signal_amplitude, signal_offset);
          break;
        case 2: // Cuadrada
          input_signal[i] = Generate_Square_Wave(signal_freq,
signal_amplitude, signal_offset);
          break;
        case 3: // Triangular
          input_signal[i] = Generate_Triangle_Wave(signal_freq,
signal_amplitude, signal_offset);
          break;
        case 4: // Multitono
          input_signal[i] = Generate_Multi_Tone(50.0f, 150.0f, 350.0f,
1.0f, 0.5f, 0.25f, signal_offset);
```

```
            break;
          default:
            input_signal[i] = signal_offset;
        }
      }

      // Procesar cada muestra con el filtro FIR seleccionado
      output_signal[i] = FIR_Filter_Process(input_signal[i]);

      // Convertir a valores para el DAC (0-4095)
      dac_buffer[i] = (uint16_t)(output_signal[i] * 4095.0f / 3.3f);
    }

    // Calcular FFT para visualización espectral
    Calculate_FFT(input_signal, input_spectrum, ADC_BUF_SIZE);
    Calculate_FFT(output_signal, output_spectrum, ADC_BUF_SIZE);
}

// Actualización de la pantalla
void Update_Display(void) {
    static uint32_t last_update = 0;
    uint32_t current_time = HAL_GetTick();

    // Actualizar la pantalla cada 100ms para evitar parpadeo
    if (current_time - last_update > 100) {
      // Limpiar pantalla
      Display_Clear();

      // Mostrar título e información del filtro
      char title[50];
      const char* filter_name;
      switch (current_filter_type) {
```

```
        case 0: filter_name = "Paso-Bajas"; break;
        case 1: filter_name = "Paso-Altas"; break;
        case 2: filter_name = "Paso-Banda"; break;
        case 3: filter_name = "Rechaza-Banda"; break;
        default: filter_name = "Desconocido";
    }

    const char* signal_name;
    switch (current_signal_type) {
        case 0: signal_name = "Externa"; break;
        case 1: signal_name = "Senoidal"; break;
        case 2: signal_name = "Cuadrada"; break;
        case 3: signal_name = "Triangular"; break;
        case 4: signal_name = "Multitono"; break;
        default: signal_name = "Desconocida";
    }

    sprintf(title, "Filtro: %s, Señal: %s, Frec: %.1fHz", filter_name,
signal_name, signal_freq);
    Display_ShowText(title, 10, 10, 0xFFFFFFFF);

    // Dibujar señal de entrada y salida
    Display_DrawSignal(input_signal, 200, 0xFF0000FF);  // Rojo
    Display_DrawSignal(output_signal, 200, 0x00FF00FF); // Verde

    // Dibujar espectro
    Display_DrawSpectrum(input_spectrum, ADC_BUF_SIZE/2, 0xFF0000FF);
    Display_DrawSpectrum(output_spectrum, ADC_BUF_SIZE/2, 0x00FF00FF);

    // Mostrar estadísticas
    char stats[50];
    sprintf(stats, "Muestras: %lu", sample_count);
```

```
    Display_ShowText(stats, 10, 270, 0xFFFFFFFF);

    // Actualizar pantalla
    Display_Update();

    last_update = current_time;
  }
}

// Manejo de comandos UART
void Handle_UART_Commands(void) {
  uint8_t rx_data;

  if (HAL_UART_Receive(&huart1, &rx_data, 1, 0) == HAL_OK) {
    switch (rx_data) {
      // Selección de filtro
      case 'l': // Paso-bajas
        current_filter_type = 0;
        FIR_Filter_Init(FILTER_TYPE_LOWPASS);
        printf("Filtro cambiado a Paso-Bajas\r\n");
        break;

      case 'h': // Paso-altas
        current_filter_type = 1;
        FIR_Filter_Init(FILTER_TYPE_HIGHPASS);
        printf("Filtro cambiado a Paso-Altas\r\n");
        break;

      case 'b': // Paso-banda
        current_filter_type = 2;
        FIR_Filter_Init(FILTER_TYPE_BANDPASS);
        printf("Filtro cambiado a Paso-Banda\r\n");
```

```
    break;

  case 's': // Rechaza-banda (stop-band)
    current_filter_type = 3;
    FIR_Filter_Init(FILTER_TYPE_BANDSTOP);
    printf("Filtro cambiado a Rechaza-Banda\r\n");
    break;

  // Selección de tipo de señal
  case 'e': // Externa (ADC)
    current_signal_type = 0;
    printf("Señal cambiada a entrada externa\r\n");
    break;

  case 'i': // Senoidal interna
    current_signal_type = 1;
    printf("Señal cambiada a senoidal interna\r\n");
    break;

  case 'q': // Cuadrada interna
    current_signal_type = 2;
    printf("Señal cambiada a cuadrada interna\r\n");
    break;

  case 't': // Triangular interna
    current_signal_type = 3;
    printf("Señal cambiada a triangular interna\r\n");
    break;

  case 'm': // Multitono interna
    current_signal_type = 4;
    printf("Señal cambiada a multitono interna\r\n");
```

```
        break;

      // Control de frecuencia
      case '+': // Aumentar frecuencia
        if (signal_freq < 450.0f) {
          signal_freq += 10.0f;
          printf("Frecuencia: %.1f Hz\r\n", signal_freq);
        }
        break;

      case '-': // Disminuir frecuencia
        if (signal_freq > 10.0f) {
          signal_freq -= 10.0f;
          printf("Frecuencia: %.1f Hz\r\n", signal_freq);
        }
        break;

      default:
        break;
    }
  }
}

/* USER CODE END 4 */
```

Guía Paso a Paso para Usar STM32F746G-DISCO

1. Configuración del Hardware

   A. Conexiones externas:

   - Conectar una fuente de señal analógica a PA0 (CN8-Pin 1) para entrada externa
   - Conectar la salida PA4 al osciloscopio para visualizar la señal filtrada

- Conectar el puerto USB ST-LINK (CN14) al PC para programación y depuración

2. Programación del Dispositivo

   A. Compilación y carga:

   - Abrir el proyecto en STM32CubeIDE
   - Compilar el proyecto (Build)
   - Conectar la placa vía USB
   - Programar la placa (Run)

3. Interacción con el Sistema

   A. Mediante la pantalla táctil:

   - Tocar los botones en pantalla para seleccionar diferentes filtros
   - Deslizar para ajustar frecuencia y ver resultados en tiempo real
   - Visualizar señales en el dominio del tiempo y frecuencia

   B. Mediante terminal serial:

   - Conectar a través de puerto serie virtual (115200 baudios)
   - Enviar comandos de una letra para control
   - Observar respuestas y datos de rendimiento

4. Experimentos Específicos

   A. Comparación de rendimiento:

   - Medir el tiempo de procesamiento para diferentes órdenes de filtro
   - Comparar con implementación Arduino
   - Evaluar la capacidad de procesamiento en tiempo real

   B. Visualización avanzada:

   - Utilizar la pantalla para mostrar simultáneamente:
     - Señal original vs. filtrada
     - Espectro de frecuencia
     - Respuesta del filtro
     - Controles interactivos

   C. Experimento con señales complejas:

   - Generar señales combinadas con múltiples frecuencias
   - Aplicar diferentes filtros para extraer componentes específicos
   - Medir la selectividad y eficiencia de los filtros

5. Generación y Captura de Señales Externas

A. Conexión de hardware externo:

- Conectar el generador de funciones a PA0
- Ajustar a diferentes formas de onda y frecuencias
- Comparar señales generadas externamente con las internas

Comparación Entre Arduino y STM32F746G-DISCO

Característica	Arduino UNO	STM32F746G-DISCO	Ventaja
CPU	16 MHz, 8-bit	216 MHz, 32-bit	STM32 (13.5x más rápido)
RAM	2 KB	320 KB	STM32 (160x más capacidad)
Flash	32 KB	1 MB	STM32 (32x más capacidad)
ADC	10-bit	12-bit	STM32 (4x más resolución)
DAC	No incluido	12-bit incluido	STM32 (no requiere hardware adicional)
Visualización	No incluida	LCD-TFT 4.3"	STM32 (visualización integrada)
Orden máximo práctico de filtro FIR	~31	~255	STM32 (8x más coeficientes)
Frecuencia de muestreo máxima	~5 kHz	~50 kHz	STM32 (10x más rápido)
Instrucciones DSP	No	Sí (ARM Cortex-M7)	STM32 (instrucciones optimizadas)
Costo	$25	$50-70	Arduino (más económico)
Facilidad de uso	Alta	Media	Arduino (curva de aprendizaje menor)

Implementación de la Pantalla de Visualización

Para completar la implementación de la visualización, se necesita definir las funciones de la interfaz gráfica. A continuación se muestra un ejemplo de implementación básica:

```
/**
 * @file display.c
 * @brief Implementación de funciones para visualización en pantalla
 */
```

```c
#include "display.h"
#include "ltdc.h"
#include <stdio.h>
#include <string.h>
#include <math.h>

// Definiciones para la pantalla
#define SCREEN_WIDTH   480
#define SCREEN_HEIGHT  272
#define SIGNAL_AREA_HEIGHT 100
#define SPECTRUM_AREA_HEIGHT 100
#define SPECTRUM_AREA_Y 140

// Buffer para la pantalla
static uint32_t *framebuffer = (uint32_t*)0xC0000000;  // Dirección de memoria externa

// Fuente básica 8x8
extern const uint8_t font8x8[128][8];

// Inicialización de la pantalla
void Display_Init(void) {
    // Limpiar pantalla
    for (int i = 0; i < SCREEN_WIDTH * SCREEN_HEIGHT; i++) {
        framebuffer[i] = 0xFF000000;  // Fondo negro
    }

    // Dibujar áreas para señales y espectros
    for (int x = 0; x < SCREEN_WIDTH; x++) {
        // Líneas separadoras
        framebuffer[x + 30 * SCREEN_WIDTH] = 0xFF404040;
```

```
        framebuffer[x + (30 + SIGNAL_AREA_HEIGHT) * SCREEN_WIDTH] =
0xFF404040;
        framebuffer[x + SPECTRUM_AREA_Y * SCREEN_WIDTH] = 0xFF404040;
        framebuffer[x + (SPECTRUM_AREA_Y + SPECTRUM_AREA_HEIGHT) *
SCREEN_WIDTH] = 0xFF404040;
    }

    // Texto estático
    Display_ShowText("Señal en Tiempo", 10, 35, 0xFFAAAAAA);
    Display_ShowText("Espectro de Frecuencia", 10, 145, 0xFFAAAAAA);

    // Mostrar título inicial
    Display_ShowText("STM32F7 Análisis de Filtros FIR", 10, 5,
0xFFFFFFFF);
}

// Limpiar pantalla manteniendo el marco
void Display_Clear(void) {
    // Limpiar área de señales
    for (int y = 31; y < 30 + SIGNAL_AREA_HEIGHT; y++) {
        for (int x = 0; x < SCREEN_WIDTH; x++) {
            framebuffer[x + y * SCREEN_WIDTH] = 0xFF000000;
        }
    }

    // Limpiar área de espectro
    for (int y = SPECTRUM_AREA_Y + 1; y < SPECTRUM_AREA_Y +
SPECTRUM_AREA_HEIGHT; y++) {
        for (int x = 0; x < SCREEN_WIDTH; x++) {
            framebuffer[x + y * SCREEN_WIDTH] = 0xFF000000;
        }
    }
}
```

```
// Dibujar señal en tiempo
void Display_DrawSignal(float* signal, int length, uint32_t color) {
    int signal_offset = 30 + SIGNAL_AREA_HEIGHT/2;
    int signal_scale = SIGNAL_AREA_HEIGHT/2 - 5;

    // Limitar a ancho de pantalla
    length = (length > SCREEN_WIDTH) ? SCREEN_WIDTH : length;

    // Dibujar línea de referencia (0V)
    for (int x = 0; x < SCREEN_WIDTH; x++) {
        framebuffer[x + signal_offset * SCREEN_WIDTH] = 0xFF202020;
    }

    // Dibujar señal
    for (int i = 0; i < length-1; i++) {
        int x1 = i;
        int y1 = signal_offset - (int)(signal[i] * signal_scale / 3.3f);
        int x2 = i + 1;
        int y2 = signal_offset - (int)(signal[i+1] * signal_scale /
3.3f);

        // Limitar a área visible
        y1 = (y1 < 31) ? 31 : (y1 >= 30 + SIGNAL_AREA_HEIGHT) ? 30 +
SIGNAL_AREA_HEIGHT - 1 : y1;
        y2 = (y2 < 31) ? 31 : (y2 >= 30 + SIGNAL_AREA_HEIGHT) ? 30 +
SIGNAL_AREA_HEIGHT - 1 : y2;

        // Dibujar línea
        DrawLine(x1, y1, x2, y2, color);
    }
}
```

```
// Dibujar espectro de frecuencia
void Display_DrawSpectrum(float* spectrum, int length, uint32_t color) {
    int spectrum_offset = SPECTRUM_AREA_Y + SPECTRUM_AREA_HEIGHT;
    int spectrum_scale = SPECTRUM_AREA_HEIGHT - 5;

    // Limitar a ancho de pantalla y mitad del espectro (frecuencias positivas)
    length = (length > SCREEN_WIDTH) ? SCREEN_WIDTH : length;
    length = (length > SCREEN_WIDTH/2) ? SCREEN_WIDTH/2 : length;

    // Encontrar máximo para normalización
    float max_val = 0.1f;  // Valor mínimo para evitar división por cero
    for (int i = 0; i < length; i++) {
        if (spectrum[i] > max_val) max_val = spectrum[i];
    }

    // Dibujar espectro
    for (int i = 0; i < length; i++) {
        int x = i * 2;  // Escalar para mostrar mitad del espectro
        int height = (int)((spectrum[i] / max_val) * spectrum_scale);
        height = (height > spectrum_scale) ? spectrum_scale : height;

        // Dibujar línea vertical
        for (int y = 0; y < height; y++) {
            framebuffer[x + (spectrum_offset - y) * SCREEN_WIDTH] = color;
        }
    }

    // Dibujar marcadores de frecuencia cada 100Hz
    for (int f = 0; f <= 500; f += 100) {
        int x = f * SCREEN_WIDTH / 1000;  // Asumiendo fs = 1000Hz
```

```
        for (int y = SPECTRUM_AREA_Y; y < spectrum_offset; y++) {
            framebuffer[x + y * SCREEN_WIDTH] = 0xFF303030;
        }

        char freq_text[10];
        sprintf(freq_text, "%d", f);
        Display_ShowText(freq_text, x - 10, spectrum_offset + 5,
0xFFAAAAAA);
    }
}

// Mostrar texto
void Display_ShowText(const char* text, int x, int y, uint32_t color) {
    int len = strlen(text);

    for (int i = 0; i < len; i++) {
        char c = text[i];

        // Solo caracteres ASCII imprimibles
        if (c >= 32 && c < 128) {
            // Dibujar cada carácter de 8x8
            for (int row = 0; row < 8; row++) {
                uint8_t row_data = font8x8[c - 32][row];
                for (int col = 0; col < 8; col++) {
                    if (row_data & (1 << col)) {
                        int pixel_x = x + i*8 + col;
                        int pixel_y = y + row;

                        // Verificar límites de la pantalla
                        if (pixel_x >= 0 && pixel_x < SCREEN_WIDTH &&
                            pixel_y >= 0 && pixel_y < SCREEN_HEIGHT) {
```

```
                framebuffer[pixel_x + pixel_y * SCREEN_WIDTH]
= color;
                        }
                    }
                }
            }
        }
    }
}

// Actualizar pantalla
void Display_Update(void) {
    // La pantalla se actualiza automáticamente con doble buffer en LTDC
}

// Función auxiliar para dibujar línea (algoritmo de Bresenham)
void DrawLine(int x1, int y1, int x2, int y2, uint32_t color) {
    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);
    int sx = (x1 < x2) ? 1 : -1;
    int sy = (y1 < y2) ? 1 : -1;
    int err = dx - dy;

    while (1) {
        // Dibujar pixel si está dentro de los límites
        if (x1 >= 0 && x1 < SCREEN_WIDTH && y1 >= 0 && y1 <
SCREEN_HEIGHT) {
            framebuffer[x1 + y1 * SCREEN_WIDTH] = color;
        }

        if (x1 == x2 && y1 == y2) break;
```

```
        int e2 = 2 * err;
        if (e2 > -dy) {
            err -= dy;
            x1 += sx;
        }
        if (e2 < dx) {
            err += dx;
            y1 += sy;
        }
    }
}

// Manejar interacción táctil
void Display_HandleTouch(void) {
    // Implementación básica de manejo táctil
    // Requiere configuración adicional del controlador táctil
    // No incluido en esta versión por simplicidad
}

// Definición de fuente básica 8x8 (incluir o declarar en otro archivo)
// Ejemplo:
const uint8_t font8x8[128][8] = {
    {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}, // Espacio
    {0x18, 0x3C, 0x3C, 0x18, 0x18, 0x00, 0x18, 0x00}, // !
    // ... (definir resto de caracteres)
};
```

Guía de Uso para Estudiantes con STM32F746G-DISCO

1. Preparación del Entorno

1. Instalación de Software:
   - Descargar e instalar STM32CubeIDE desde el sitio web de ST
   - Instalar los paquetes de firmware para STM32F7

- Conectar la placa STM32F746G-DISCO al ordenador mediante USB

2. Importación del Proyecto:
   - Abrir STM32CubeIDE
   - Seleccionar "File > Import > Existing Projects into Workspace"
   - Buscar y seleccionar la carpeta del proyecto proporcionada
   - Hacer clic en "Finish" para importar

2. Compilación y Carga

1. Compilar el Proyecto:
   - Hacer clic derecho en el proyecto en el explorador de proyectos
   - Seleccionar "Build Project"
   - Verificar que no haya errores en la consola
2. Conectar la Placa:
   - Conectar el cable USB al conector ST-LINK (CN14) de la placa
   - Verificar que el LED verde "PWR" se encienda
   - Esperar a que la placa sea reconocida por el sistema
3. Cargar el Programa:
   - Hacer clic derecho en el proyecto en el explorador de proyectos
   - Seleccionar "Run As > STM32 MCU Application"
   - Seleccionar la configuración de depuración si se solicita
   - Esperar a que el programa se cargue

3. Realización de los Experimentos

1. Experimento 1: Comparación de Filtros
   - Conectar un generador de funciones a PA0 (si se desea usar señal externa)
   - Utilizar la interfaz táctil o comandos UART para seleccionar diferentes filtros
   - Observar en la pantalla LCD las diferencias entre filtros
   - Medir y comparar el rendimiento y características de cada filtro

2. Experimento 2: Análisis de Ventanas
   - Modificar los coeficientes del filtro para usar diferentes ventanas
   - Recompilar y cargar el programa
   - Comparar el rendimiento y características de cada ventana
   - Documentar las observaciones
3. Experimento 3: Señales Complejas
   - Utilizar el generador interno para crear señales complejas
   - Aplicar diferentes filtros y observar sus efectos
   - Analizar la capacidad del sistema para procesar señales complejas en tiempo real

4. Análisis y Documentación

1. Comparación con Arduino:
   - Repetir los mismos experimentos en la plataforma Arduino
   - Comparar resultados, rendimiento y limitaciones
   - Documentar las ventajas y desventajas de cada plataforma
2. Análisis de Resultados:
   - Capturar imágenes de la pantalla para documentar resultados
   - Analizar el rendimiento de procesamiento (tiempo de ejecución)
   - Evaluar la precisión de los filtros implementados
3. Informe Final:
   - Incluir comparativas entre Arduino y STM32
   - Analizar las limitaciones y ventajas de cada plataforma
   - Documentar el proceso de aprendizaje y conclusiones

Información Adicional sobre STM32F746G-DISCO

1. Recursos de Aprendizaje:
   - Sitio oficial de ST para STM32F7
   - Documentación de STM32CubeF7
   - Manual de usuario de la placa STM32F746G-DISCO
2. Características Avanzadas:
   - Aceleración de hardware para DSP
   - Instrucciones SIMD para procesamiento paralelo
   - Cache de instrucciones y datos para mejor rendimiento
   - Varias interfaces de comunicación para conectar con otros dispositivos
3. Consideraciones Importantes:
   - Mayor complejidad de programación que Arduino
   - Requiere conocimientos de programación en C a nivel intermedio
   - Mayor potencia pero también mayor curva de aprendizaje
   - Herramientas de depuración más potentes pero más complejas

REFERENCIAS Y RECURSOS ADICIONALES

1. Documentación de Arduino:
   - Referencia AnalogRead
   - Wire Library (I2C)
2. Documentación de Python:
   - NumPy
   - SciPy Signal Processing
   - Matplotlib
3. Teoría de Filtros Digitales:
   - Filtros FIR
   - Ventanas en Procesamiento de Señales

- Oppenheim, A. V., & Schafer, R. W. (2010). Discrete-time signal processing. Pearson.
- Smith, S. W. (1997). The scientist and engineer's guide to digital signal processing.

4. Recursos sobre STM32:
   - STM32F7 Series Documentation
   - CMSIS-DSP Library Documentation
   - STM32 DSP with CMSIS Guide

CRITERIOS DE EVALUACIÓN

1. Implementación correcta de los diferentes tipos de filtros FIR (20%)
2. Análisis del efecto de las ventanas y el orden del filtro (20%)
3. Implementación funcional en Arduino (20%)
4. Experimentos con señales reales y análisis de resultados (20%)
5. Implementación funcional en STM32F746G-DISCO (bonus: +10%)
6. Respuestas a las preguntas de comprensión (20%)

OBSERVACIONES:

- Tomar fotografías del montaje experimental
- Guardar las gráficas generadas en Python
- Documentar cualquier variación observada entre plataformas
- Mantener respaldo digital del código utilizado
- Comparar el rendimiento entre Arduino y STM32 para mismos filtros