Proyecto 1: Sistema de Reconocimiento de Tonos DTMF

Objetivo General

Diseñar e implementar un sistema de reconocimiento de tonos DTMF (Dual-Tone Multi-Frequency) utilizado en la telefonía, mediante el uso de bancos de filtros digitales y análisis espectral.

Justificación Académica

Este proyecto integra conceptos fundamentales de procesamiento digital de señales como diseño de filtros pasabanda, análisis espectral, muestreo y aplicación práctica en sistemas de telecomunicaciones. Los tonos DTMF son la base de la marcación telefónica y su reconocimiento automático tiene aplicaciones en control remoto y sistemas interactivos.

Materiales Necesarios

- Arduino UNO o similar
- Módulo micrófono electret con amplificador (como KY-038)
- Protoboard y cables de conexión
- Altavoz o bocina para generar tonos
- Resistencias: 2x 10kΩ, 1x 4.7kΩ
- Capacitores: 1x 0.1µF
- LEDs (para indicar tonos detectados) y resistencias limitadoras
- Computadora con Python y Arduino IDE

Descripción del Proyecto

Los estudiantes diseñarán un sistema que capture audio a través del micrófono, identifique los tonos DTMF (correspondientes a las teclas de un teléfono) mediante filtros digitales y análisis espectral, y muestre en tiempo real qué tecla fue presionada.

Procedimiento

PARTE 1: Análisis teórico de los tonos DTMF

1. Investigar las frecuencias exactas utilizadas en el sistema DTMF:
    - Frecuencias bajas: 697Hz, 770Hz, 852Hz, 941Hz
    - Frecuencias altas: 1209Hz, 1336Hz, 1477Hz, 1633Hz
    - Cada tecla combina una frecuencia baja y una alta
2. Desarrollar un script en Python para generar tonos DTMF:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
```

```python
from scipy.signal import spectrogram
import sounddevice as sd

# Frecuencias DTMF
low_freq = [697, 770, 852, 941]
high_freq = [1209, 1336, 1477, 1633]

# Matriz de teclas
keys = [
    ['1', '2', '3', 'A'],
    ['4', '5', '6', 'B'],
    ['7', '8', '9', 'C'],
    ['*', '0', '#', 'D']
]

def generate_dtmf(key, duration=0.5, fs=8000):
    """
    Genera un tono DTMF para la tecla especificada
    """
    # Encontrar índices para la tecla
    row_idx = -1
    col_idx = -1
    for i in range(4):
        for j in range(4):
            if keys[i][j] == key:
                row_idx = i
                col_idx = j
                break
        if row_idx != -1:
            break

    if row_idx == -1 or col_idx == -1:
        raise ValueError(f"Tecla {key} no válida")
```

```
    # Generar tono
    t = np.linspace(0, duration, int(fs * duration), endpoint=False)
    row_tone = np.sin(2 * np.pi * low_freq[row_idx] * t)
    col_tone = np.sin(2 * np.pi * high_freq[col_idx] * t)

    # Combinar tonos
    dtmf_tone = 0.5 * row_tone + 0.5 * col_tone

    return dtmf_tone, fs

def plot_tone_analysis(tone, fs, key):
    """
    Analiza y muestra el tono DTMF en tiempo, frecuencia y espectrograma
    """
    # Crear figura
    plt.figure(figsize=(12, 10))

    # Señal en tiempo
    plt.subplot(3, 1, 1)
    t = np.linspace(0, len(tone)/fs, len(tone))
    plt.plot(t, tone)
    plt.title(f'Señal DTMF para tecla "{key}" en tiempo')
    plt.xlabel('Tiempo (s)')
    plt.ylabel('Amplitud')
    plt.grid(True)

    # Espectro de frecuencia (FFT)
    plt.subplot(3, 1, 2)
    n = len(tone)
    fft_result = np.fft.rfft(tone)
    freq = np.fft.rfftfreq(n, 1/fs)
    plt.plot(freq, np.abs(fft_result))
    plt.title(f'Espectro de frecuencia DTMF para tecla "{key}"')
    plt.xlabel('Frecuencia (Hz)')
```

```
    plt.ylabel('Magnitud')
    plt.grid(True)
    plt.xlim(0, 2000)  # Limitamos a las frecuencias relevantes

    # Espectrograma
    plt.subplot(3, 1, 3)
    f, t, Sxx = spectrogram(tone, fs)
    plt.pcolormesh(t, f, 10 * np.log10(Sxx), shading='gouraud')
    plt.title(f'Espectrograma DTMF para tecla "{key}"')
    plt.ylabel('Frecuencia (Hz)')
    plt.xlabel('Tiempo (s)')
    plt.colorbar(label='Potencia/Frecuencia (dB/Hz)')
    plt.ylim(0, 2000)  # Limitamos a las frecuencias relevantes

    plt.tight_layout()
    plt.savefig(f'dtmf_analysis_{key}.png')
    plt.show()

# Ejemplo: generar y analizar tono para la tecla '5'
key = '5'
tone, fs = generate_dtmf(key)
plot_tone_analysis(tone, fs, key)

# Reproducir el tono
sd.play(tone, fs)
sd.wait()

# Guardar tono como archivo WAV
wavfile.write(f'dtmf_{key}.wav', fs, (tone * 32767).astype(np.int16))

# Generar todos los tonos para las pruebas
def generate_all_dtmf_tones():
    for row in range(4):
        for col in range(4):
```

```
            key = keys[row][col]
            tone, fs = generate_dtmf(key)
            wavfile.write(f'dtmf_{key}.wav', fs, (tone *
32767).astype(np.int16))
            print(f"Generado tono para tecla {key}")

generate_all_dtmf_tones()
```

PARTE 2: Diseño de filtros pasabanda

1. Diseñar ocho filtros pasabanda FIR usando el método de la ventana para cada una de las frecuencias DTMF:

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

def design_bandpass_filter(center_freq, bandwidth, fs, order=101):
    """
    Diseña un filtro pasabanda FIR para una frecuencia central específica

    Parámetros:
    center_freq: Frecuencia central del filtro en Hz
    bandwidth: Ancho de banda del filtro en Hz
    fs: Frecuencia de muestreo en Hz
    order: Orden del filtro (debe ser impar)

    Retorna:
    b: Coeficientes del filtro FIR
    """
    if order % 2 == 0:
        order += 1  # Asegurar que el orden sea impar

    # Normalizar frecuencias
    nyquist = fs / 2
    low_cutoff = (center_freq - bandwidth/2) / nyquist
```

```
    high_cutoff = (center_freq + bandwidth/2) / nyquist

    # Limitar a rango válido
    low_cutoff = max(0.001, min(0.999, low_cutoff))
    high_cutoff = max(0.001, min(0.999, high_cutoff))

    # Diseñar filtro usando ventana Hamming
    b = signal.firwin(order, [low_cutoff, high_cutoff], window='hamming',
pass_zero=False)

    return b

def analyze_filter(b, fs, center_freq):
    """
    Analiza y muestra la respuesta en frecuencia del filtro
    """
    # Calcular respuesta en frecuencia
    w, h = signal.freqz(b, 1, worN=8000)
    f = w * fs / (2 * np.pi)

    # Graficar respuesta en magnitud
    plt.figure(figsize=(10, 6))
    plt.subplot(2, 1, 1)
    plt.plot(f, np.abs(h))
    plt.title(f'Respuesta en magnitud del filtro para {center_freq} Hz')
    plt.xlabel('Frecuencia (Hz)')
    plt.ylabel('Ganancia')
    plt.grid(True)
    plt.axvline(center_freq, color='r', linestyle='--', alpha=0.5)
    plt.xlim(0, 2000)

    # Respuesta en dB
    plt.subplot(2, 1, 2)
    plt.plot(f, 20 * np.log10(np.abs(h) + 1e-10))
```

```
    plt.title(f'Respuesta en dB del filtro para {center_freq} Hz')
    plt.xlabel('Frecuencia (Hz)')
    plt.ylabel('Ganancia (dB)')
    plt.grid(True)
    plt.axvline(center_freq, color='r', linestyle='--', alpha=0.5)
    plt.xlim(0, 2000)
    plt.ylim(-80, 5)

    plt.tight_layout()
    plt.savefig(f'filter_response_{center_freq}Hz.png')
    plt.show()

# Parámetros
fs = 8000  # Frecuencia de muestreo
bandwidth = 30  # Ancho de banda de 30 Hz
filter_order = 251  # Orden del filtro

# Diseñar y analizar filtros para frecuencias bajas
for freq in [697, 770, 852, 941]:
    b = design_bandpass_filter(freq, bandwidth, fs, filter_order)
    analyze_filter(b, fs, freq)
    np.savetxt(f'dtmf_filter_coef_{freq}Hz.csv', b, delimiter=',')

# Diseñar y analizar filtros para frecuencias altas
for freq in [1209, 1336, 1477, 1633]:
    b = design_bandpass_filter(freq, bandwidth, fs, filter_order)
    analyze_filter(b, fs, freq)
    np.savetxt(f'dtmf_filter_coef_{freq}Hz.csv', b, delimiter=',')
```

2. Comparar filtros FIR e IIR para esta aplicación:

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

```python
def design_fir_bandpass(center_freq, bandwidth, fs, order=101):
    """Diseña filtro FIR pasabanda usando método de la ventana"""
    nyquist = fs / 2
    low_cutoff = (center_freq - bandwidth/2) / nyquist
    high_cutoff = (center_freq + bandwidth/2) / nyquist

    # Limitar a rango válido
    low_cutoff = max(0.001, min(0.999, low_cutoff))
    high_cutoff = max(0.001, min(0.999, high_cutoff))

    b = signal.firwin(order, [low_cutoff, high_cutoff], window='hamming',
pass_zero=False)
    return b, [1.0]  # b, a para formato compatible con IIR

def design_iir_bandpass(center_freq, bandwidth, fs, order=2):
    """Diseña filtro IIR pasabanda usando Butterworth"""
    nyquist = fs / 2
    low_cutoff = (center_freq - bandwidth/2) / nyquist
    high_cutoff = (center_freq + bandwidth/2) / nyquist

    # Limitar a rango válido
    low_cutoff = max(0.001, min(0.999, low_cutoff))
    high_cutoff = max(0.001, min(0.999, high_cutoff))

    b, a = signal.butter(order, [low_cutoff, high_cutoff], btype='band')
    return b, a

def compare_filters(center_freq, bandwidth, fs):
    """Compara filtros FIR e IIR para una frecuencia dada"""
    # Diseñar filtros
    b_fir, a_fir = design_fir_bandpass(center_freq, bandwidth, fs,
order=101)
    b_iir, a_iir = design_iir_bandpass(center_freq, bandwidth, fs,
order=2)
```

```
    # Calcular respuestas en frecuencia
    w, h_fir = signal.freqz(b_fir, a_fir, worN=8000)
    _, h_iir = signal.freqz(b_iir, a_iir, worN=8000)
    f = w * fs / (2 * np.pi)

    # Graficar respuestas en magnitud
    plt.figure(figsize=(12, 10))

    # Respuesta en magnitud
    plt.subplot(2, 1, 1)
    plt.plot(f, np.abs(h_fir), 'b-', label='FIR (Orden 101)')
    plt.plot(f, np.abs(h_iir), 'r--', label='IIR (Orden 2)')
    plt.title(f'Comparación de filtros para {center_freq} Hz')
    plt.xlabel('Frecuencia (Hz)')
    plt.ylabel('Ganancia')
    plt.grid(True)
    plt.axvline(center_freq, color='g', linestyle='--', alpha=0.5)
    plt.legend()
    plt.xlim(center_freq - 200, center_freq + 200)

    # Respuesta en dB
    plt.subplot(2, 1, 2)
    plt.plot(f, 20 * np.log10(np.abs(h_fir) + 1e-10), 'b-', label='FIR
(Orden 101)')
    plt.plot(f, 20 * np.log10(np.abs(h_iir) + 1e-10), 'r--', label='IIR
(Orden 2)')
    plt.title(f'Respuesta en dB para {center_freq} Hz')
    plt.xlabel('Frecuencia (Hz)')
    plt.ylabel('Ganancia (dB)')
    plt.grid(True)
    plt.axvline(center_freq, color='g', linestyle='--', alpha=0.5)
    plt.legend()
    plt.xlim(0, 2000)
    plt.ylim(-80, 5)
```

```
    plt.tight_layout()
    plt.savefig(f'filter_comparison_{center_freq}Hz.png')
    plt.show()

    # Analizar fase
    plt.figure(figsize=(10, 6))
    plt.plot(f, np.unwrap(np.angle(h_fir)), 'b-', label='FIR (Orden 101)')
    plt.plot(f, np.unwrap(np.angle(h_iir)), 'r--', label='IIR (Orden 2)')
    plt.title(f'Comparación de fase para {center_freq} Hz')
    plt.xlabel('Frecuencia (Hz)')
    plt.ylabel('Fase (radianes)')
    plt.grid(True)
    plt.axvline(center_freq, color='g', linestyle='--', alpha=0.5)
    plt.legend()
    plt.xlim(center_freq - 200, center_freq + 200)
    plt.savefig(f'phase_comparison_{center_freq}Hz.png')
    plt.show()

# Comparar filtros para una frecuencia baja y una alta
fs = 8000
bandwidth = 30
compare_filters(697, bandwidth, fs)  # Frecuencia baja
compare_filters(1209, bandwidth, fs)  # Frecuencia alta
```

PARTE 3: Implementación de detector DTMF en Arduino

1. Código para Arduino que muestrea una señal de audio y aplica los filtros:

```
/*
 * Detector DTMF con Arduino
 * Este código implementa un detector de tonos DTMF utilizando filtros digitales
 */
```

```
// Configuración de pines
const int audioInPin = A0;  // Pin analógico para entrada de audio
const int sampleRate = 8000; // Frecuencia de muestreo en Hz
const int ledPins[] = {2, 3, 4, 5, 6, 7, 8, 9}; // Pines para LEDs indicadores

// Frecuencias DTMF
const float lowFreq[] = {697, 770, 852, 941};
const float highFreq[] = {1209, 1336, 1477, 1633};

// Matriz de teclas
const char keys[4][4] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};

// Buffer para muestras de audio
const int bufferSize = 256; // Tamaño del buffer
int audioBuffer[bufferSize];
int bufferIndex = 0;

// Coeficientes de filtros FIR (se cargarán desde Python)
// Aquí se declararán los arrays para los coeficientes

// Valores umbral para detección
const float detectionThreshold = 30.0; // Ajustar según pruebas

// Variables para control de tiempo
unsigned long lastSampleTime = 0;
const unsigned long samplePeriod = 1000000 / sampleRate; // en microsegundos
```

```
// Variables para detección
float filterOutputs[8]; // Salidas de los 8 filtros
int rowIndex = -1;
int colIndex = -1;
char lastDetectedKey = ' ';

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

  // Configurar pines LED
  for (int i = 0; i < 8; i++) {
    pinMode(ledPins[i], OUTPUT);
    digitalWrite(ledPins[i], LOW);
  }

  // Inicializar filtros y buffers
  initFilters();

  Serial.println("Detector DTMF iniciado");
  Serial.println("Esperando tonos...");
}

void loop() {
  // Tomar muestra a la frecuencia definida
  unsigned long currentTime = micros();
  if (currentTime - lastSampleTime >= samplePeriod) {
    lastSampleTime = currentTime;

    // Leer muestra de audio
    int audioSample = analogRead(audioInPin);

    // Normalizar a valor centrado en cero (-512 a 511)
    audioSample = audioSample - 512;
```

```
    // Guardar en buffer circular
    audioBuffer[bufferIndex] = audioSample;
    bufferIndex = (bufferIndex + 1) % bufferSize;

    // Procesar buffer completo cada N muestras
    if (bufferIndex == 0) {
      processBuffer();
    }
  }
}

void initFilters() {
  // Aquí se cargarían los coeficientes de los filtros
  // En una implementación real, estos se cargarían desde la EEPROM o
  // se definirían como constantes basadas en los resultados de Python

  Serial.println("Filtros inicializados");
}

void processBuffer() {
  // Aplicar los 8 filtros al buffer de audio
  for (int i = 0; i < 8; i++) {
    filterOutputs[i] = applyFilter(i);

    // Encender LED si la salida supera el umbral
    digitalWrite(ledPins[i], filterOutputs[i] > detectionThreshold ? HIGH
: LOW);
  }

  // Encontrar las frecuencias dominantes
  int maxLowIndex = findMaxIndex(filterOutputs, 0, 3);
  int maxHighIndex = findMaxIndex(filterOutputs, 4, 7);
```

```
// Verificar si superan el umbral
if (filterOutputs[maxLowIndex] > detectionThreshold &&
    filterOutputs[maxHighIndex] > detectionThreshold) {

  // Calcular índices de fila y columna
  rowIndex = maxLowIndex;
  colIndex = maxHighIndex - 4;

  // Determinar la tecla
  char detectedKey = keys[rowIndex][colIndex];

  // Reportar solo si es una tecla nueva
  if (detectedKey != lastDetectedKey) {
    lastDetectedKey = detectedKey;

    // Mostrar resultados
    Serial.print("Tecla detectada: ");
    Serial.println(detectedKey);

    // Mostrar energía de cada filtro
    Serial.println("Energía de filtros:");
    for (int i = 0; i < 8; i++) {
      Serial.print(i < 4 ? lowFreq[i] : highFreq[i-4]);
      Serial.print(" Hz: ");
      Serial.println(filterOutputs[i]);
    }
    Serial.println();
  }
} else {
  // Si no se detecta nada, resetear
  if (lastDetectedKey != ' ') {
    lastDetectedKey = ' ';
    Serial.println("No se detecta tono");
  }
```

```
  }
}

float applyFilter(int filterIndex) {
  // Aplicar un filtro FIR al buffer de audio
  // Esta es una implementación simplificada que se debe adaptar según
  // los coeficientes específicos de cada filtro

  // En una implementación real, se usarían los coeficientes específicos
  // para cada una de las 8 frecuencias DTMF

  // Este es un ejemplo de cómo podría implementarse
  float sum = 0;
  int startIdx = bufferIndex;

  // Simplificación: usar Goertzel en lugar de FIR completo
  // para eficiencia en Arduino
  float targetFreq = filterIndex < 4 ? lowFreq[filterIndex] :
highFreq[filterIndex-4];
  sum = goertzelAlgorithm(audioBuffer, bufferSize, targetFreq,
sampleRate);

  return sum;
}

int findMaxIndex(float array[], int startIdx, int endIdx) {
  // Encuentra el índice del valor máximo en un rango del array
  int maxIndex = startIdx;
  float maxValue = array[startIdx];

  for (int i = startIdx + 1; i <= endIdx; i++) {
    if (array[i] > maxValue) {
      maxValue = array[i];
      maxIndex = i;
```

```
    }
  }

  return maxIndex;
}

float goertzelAlgorithm(int samples[], int numSamples, float targetFreq,
float sampleRate) {
  // Implementación del algoritmo de Goertzel para detección eficiente de
frecuencias
  float omega = 2.0 * PI * targetFreq / sampleRate;
  float sine = sin(omega);
  float cosine = cos(omega);
  float coeff = 2.0 * cosine;

  float q0 = 0;
  float q1 = 0;
  float q2 = 0;

  // Procesar todas las muestras
  for (int i = 0; i < numSamples; i++) {
    q0 = coeff * q1 - q2 + samples[(bufferIndex + i) % numSamples];
    q2 = q1;
    q1 = q0;
  }

  // Calcular magnitud
  float result = sqrt(q1*q1 + q2*q2 - q1*q2*coeff);
  return result;
}
```

2. Script Python para procesar y visualizar los datos recibidos desde Arduino:

```
import serial
import numpy as np
import matplotlib.pyplot as plt
```

```python
from matplotlib.animation import FuncAnimation
from scipy import signal
import time

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

# Frecuencias DTMF
low_freq = [697, 770, 852, 941]
high_freq = [1209, 1336, 1477, 1633]

# Matriz de teclas
keys = [
    ['1', '2', '3', 'A'],
    ['4', '5', '6', 'B'],
    ['7', '8', '9', 'C'],
    ['*', '0', '#', 'D']
]

# 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, axs = plt.subplots(2, 1, figsize=(12, 10))
```

```
fig.suptitle('Análisis de Tonos DTMF en Tiempo Real')

# Inicializar datos
filter_energies = np.zeros(8)
bar_positions = np.arange(8)
bar_labels = [f"{f} Hz" for f in low_freq + high_freq]

# Crear gráficos de barras para energía de los filtros
bars = axs[0].bar(bar_positions, filter_energies, color='cyan')
axs[0].set_xticks(bar_positions)
axs[0].set_xticklabels(bar_labels, rotation=45)
axs[0].set_ylabel('Energía')
axs[0].set_title('Energía por Filtro')
axs[0].grid(True, axis='y')

# Visualización de tecla detectada
detected_key_text = axs[1].text(0.5, 0.5, "", fontsize=120,
                                ha='center', va='center')
axs[1].set_title('Tecla Detectada')
axs[1].set_xticks([])
axs[1].set_yticks([])

# Función para actualizar gráficos
def update_plot(frame):
    global filter_energies

    # Leer datos del puerto serie
    if ser.in_waiting:
        try:
            line = ser.readline().decode().strip()

            # Procesar datos de energía
            if line.startswith("Energía de filtros:"):
                # Leer las siguientes 8 líneas con valores de energía
```

```
                for i in range(8):
                    energy_line = ser.readline().decode().strip()
                    try:
                        # Extraer valor numérico
                        energy_value = float(energy_line.split(": ")[1])
                        filter_energies[i] = energy_value
                    except:
                        pass

                # Actualizar gráfico de barras
                for i, bar in enumerate(bars):
                    bar.set_height(filter_energies[i])

                # Determinar tecla detectada
                max_low_idx = np.argmax(filter_energies[:4])
                max_high_idx = np.argmax(filter_energies[4:]) + 4

                # Si ambos superan un umbral, mostrar tecla
                threshold = 10.0  # Ajustar según datos reales
                if filter_energies[max_low_idx] > threshold and
filter_energies[max_high_idx] > threshold:
                    row_idx = max_low_idx
                    col_idx = max_high_idx - 4
                    key = keys[row_idx][col_idx]
                    detected_key_text.set_text(key)
                    detected_key_text.set_color('lime')
                else:
                    detected_key_text.set_text("")

            # Mostrar en consola información de tecla detectada
            if line.startswith("Tecla detectada:"):
                key = line.split(": ")[1]
                print(f"Tecla detectada: {key}")
```

```
            except Exception as e:
                print(f"Error procesando datos: {e}")

    return bars + [detected_key_text]

# Configurar animación
ani = FuncAnimation(fig, update_plot, interval=100, blit=True)

plt.tight_layout()
plt.subplots_adjust(hspace=0.3)

try:
    plt.show()
except KeyboardInterrupt:
    print("Análisis finalizado")
finally:
    if ser is not None and ser.is_open:
        ser.close()
    print("Puerto serial cerrado")
```

PARTE 4: Pruebas y optimización

1. Generar tonos de prueba con diferentes duraciones
2. Evaluar precisión del detector con diferentes niveles de ruido
3. Optimizar los parámetros de los filtros y umbrales de detección
4. Comparar rendimiento de filtros FIR vs. algoritmo de Goertzel

PARTE 5: Expansión del proyecto

Los estudiantes pueden expandir el proyecto en varias direcciones:

1. Implementar un decodificador de comandos basado en secuencias DTMF
2. Añadir un control remoto para dispositivos usando tonos DTMF
3. Diseñar un sistema de acceso basado en códigos DTMF
4. Comparar el rendimiento con implementaciones basadas en FFT

Entregables

1. Informe técnico que incluya:
    - Fundamentación teórica de los tonos DTMF

- Diseño de filtros digitales (FIR e IIR)
- Análisis espectral de los tonos
- Comparativa de métodos de detección
- Resultados experimentales y discusión