Práctica 4 - Procesamiento de Audio

Alejandro Bolaños García y David García Díaz

1 Introducción

La presente práctica tiene como objetivo principal el análisis y procesamiento de audio digital mediante Python. Para ello, se implementan funciones que permiten aplicar transformadas de Fourier para visualizar el espectro de frecuencias, identificar notas musicales en archivos de audio y aplicar filtros de paso de banda para mejorar la calidad de las señales. También se construye una interfaz gráfica que permite aplicar diferentes tipos de filtros sobre archivos de audio, visualizar las señales y guardar los resultados.

2 Ejercicio 1: Identificador de Notas Musicales

Este ejercicio desarrolla un identificador de notas musicales que permite reconocer la nota tocada en un archivo de audio. La implementación emplea técnicas de procesamiento de señales, incluyendo la Transformada Rápida de Fourier (FFT) para analizar el espectro de frecuencias, filtros de paso de banda para reducir ruido, y un diccionario de referencia que asocia cada frecuencia dominante con una nota musical. Este identificador está diseñado para funcionar con un único instrumento, lo que simplifica el análisis de la señal.

2.1 Funciones Principales

A continuación se describen las funciones clave del ejercicio:

- `fft_plot()`: Esta función carga un archivo de audio en formato `.wav` y aplica la Transformada Rápida de Fourier (FFT) para convertir la señal al dominio de frecuencias. Además, muestra el espectro de frecuencias, destacando los picos de frecuencias dominantes, y extrae la frecuencia fundamental, es decir, la frecuencia con la mayor amplitud.
- `notas_frecuencias`: Es un diccionario que asocia cada nota musical (como *Do*, *Re*, *Mi*, etc.) con sus frecuencias correspondientes en distintas octavas. Este diccionario sirve como referencia para identificar la nota que corresponde a la frecuencia dominante detectada.
- `identificador_nota(frecuencia)`: Recibe una frecuencia en Hz y determina la nota musical correspondiente utilizando el diccionario `notas_frecuencias`. Emplea una tolerancia del 5% para asegurar la identificación de la nota, incluso si la frecuencia tiene pequeñas desviaciones.
- `filtro_pasa_banda()`: Implementa un filtro pasa banda utilizando un filtro Butterworth para eliminar frecuencias fuera del rango de interés. Este filtro ayuda a reducir el ruido de fondo y mejora la claridad de la señal. Los límites de corte `lowcut` y `highcut` son ajustables para adaptarse al rango de frecuencias esperado.
- `get_frecuencia_sonido()`: Lee un archivo de audio y calcula la frecuencia fundamental. Si el archivo es estéreo, selecciona un solo canal para simplificar el procesamiento. Utiliza la FFT para analizar el espectro de frecuencias, identifica la frecuencia con la mayor amplitud, y llama a `identificador_nota` para determinar la nota correspondiente a esta frecuencia.

2.2 Problemas Encontrados y Soluciones

Durante el desarrollo del identificador de notas, se presentaron algunos problemas que fueron abordados de la siguiente manera:

- **Ruido en el Espectro de Frecuencias**: Al analizar el espectro de frecuencias, se observó que había ruido en frecuencias fuera del rango de interés. Para solucionarlo, se implementó la función `filtro_pasa_banda()`, que permite aplicar un filtro Butterworth de paso banda para limitar las

frecuencias al rango esperado para el instrumento. Esto mejoró la precisión en la identificación de la frecuencia fundamental.

- **Desviación en la Frecuencia de las Notas**: Durante las pruebas, se observaron ligeras desviaciones en la frecuencia de las notas, lo cual dificultaba su identificación precisa. Para resolver este problema, se introdujo una tolerancia del 5% en la función `identificador_nota`, permitiendo identificar una nota aunque la frecuencia medida no coincida exactamente con los valores teóricos del diccionario.
- **Diferencias en la Detección con `find_peaks`**: Se implementaron dos versiones para el procesamiento en tiempo real: una utilizando `find_peaks` de `scipy` para detectar picos significativos en el espectro de frecuencias, y otra sin esta función. Aunque ambas versiones funcionan correctamente, decidimos mantener ambas alternativas en el código para facilitar futuras pruebas con equipos de captura de audio de alta precisión, como micrófonos especializados. La función `find_peaks` podría ser particularmente útil si se quiere ampliar el código para detectar acordes.

Estas soluciones mejoraron significativamente la precisión y la funcionalidad del identificador de notas musicales, haciendo que sea más robusto frente a desviaciones de frecuencia y ruidos en la señal de audio. El filtro pasa banda, la tolerancia en la identificación de notas, y las dos versiones de procesamiento en tiempo real aseguran que el código sea versátil y adecuado para distintas configuraciones de audio.

3 Ejercicio 2: Interfaz gráfica para filtrar audios

Este programa en Python implementa una aplicación de filtrado de audio que permite cargar archivos de sonido en formato `.wav`, aplicar distintos tipos de filtros, reproducir el audio procesado y visualizar tanto la señal original como la filtrada en el dominio del tiempo y de la frecuencia. La interfaz gráfica está construida con la biblioteca `Tkinter`, y los filtros se implementan mediante funciones de `scipy.signal`.

3.1 Funciones Principales

A continuación, se describen las funciones más importantes del programa:

- `open_file()`: Permite al usuario seleccionar un archivo de audio y lo carga en el programa. Si el archivo es estéreo, convierte la señal a mono tomando un solo canal.
- `low_pass_filter()`, `high_pass_filter()`, `filtro_pasa_banda()`, `filtro_rechaza_banda()`: Implementan los diferentes tipos de filtros de audio. Utilizan `butter` de `scipy.signal` para calcular los coeficientes de un filtro Butterworth y `filtfilt` para aplicar el filtro bidireccionalmente, minimizando el desplazamiento de fase. Cada filtro recibe la señal y la frecuencia de corte (o rangos de corte) y devuelve la señal filtrada, normalizada para asegurar que no supere el rango permitido.
- `apply_filter()`: Aplica el filtro seleccionado por el usuario. Valida que el filtro esté seleccionado, aplica el filtro correspondiente y reproduce el audio filtrado. Además, visualiza la señal filtrada en el dominio del tiempo y en el dominio de la frecuencia.
- `play_audio()` **y** `stop_audio()`: Permiten reproducir y detener el audio cargado o filtrado mediante la biblioteca `sounddevice`.
- `save_filtered_audio()`: Guarda la señal de audio filtrada como un nuevo archivo `.wav`. Escala la señal para asegurar que esté dentro del rango de amplitud adecuado antes de guardarla.

3.2 Aportaciones Adicionales

Además de la funcionalidad básica de filtrado de audio, se implementaron varias mejoras para mejorar la experiencia del usuario:

- **Botones de Reproducción y Detención del Audio**: Se añadieron dos botones, `Play Original Audio` y `Stop Audio`, que permiten al usuario reproducir el archivo de audio original y detener la reproducción en cualquier momento. Esto facilita escuchar la señal antes y después de aplicar los filtros, y permite al usuario comparar la calidad del audio original con el filtrado.

- **Opción para Guardar el Audio Filtrado**: La función `save_filtered_audio()` permite guardar el audio filtrado en formato `.wav`. Esto es útil para preservar el resultado después de aplicar un filtro. La función normaliza y escala la señal filtrada para asegurar que esté dentro del rango de amplitud adecuado para almacenamiento en un archivo de audio.
- **Etiqueta de Nombre de Archivo**: Se agregó una etiqueta en la interfaz gráfica que muestra el nombre del archivo de audio actualmente cargado. La etiqueta proporciona una referencia visual clara para el usuario, permitiéndole confirmar qué archivo está abierto en cada momento, lo cual es especialmente útil cuando se trabaja con múltiples archivos de audio.

3.3 Problemas Encontrados y Soluciones

Durante el desarrollo del código, se presentaron algunos problemas que fueron abordados de la siguiente manera:

- **Error al intentar graficar una señal filtrada no generada**: En un inicio, si el usuario no seleccionaba un filtro válido, se intentaba graficar una señal `None`, causando un error de valor nulo. Este problema se resolvió verificando que el filtro esté seleccionado antes de aplicar cualquier filtrado, y mostrando un mensaje de advertencia si no se selecciona un filtro válido.
- **Frecuencias de corte incorrectas en filtros pasa banda y rechaza banda**: Cuando se establecía una frecuencia de corte inferior mayor o igual a la frecuencia de corte superior, el filtro generaba un error. Esto se solucionó con una comprobación en las funciones `filtro_pasa_banda()` y `filtro_rechaza_banda()`, que advierte al usuario si las frecuencias de corte están en el orden incorrecto.
- **Normalización de la señal filtrada**: Para evitar distorsión en la reproducción y el almacenamiento de la señal filtrada, se implementó una normalización en cada función de filtrado. Esto asegura que la señal esté dentro del rango permitido para reproducción y almacenamiento en el formato de audio `.wav`.
- **Verificación de la señal filtrada antes de guardarla**: Para evitar intentos de guardar una señal no filtrada, se añadió una comprobación en `save_filtered_audio()` que verifica si `filtered_data` no es `None`. Si no se ha aplicado ningún filtro, se muestra un mensaje de advertencia al usuario.

4 Conclusión

Esta práctica nos ha permitido adentrarnos en el fascinante mundo del procesamiento de audio digital, explorando tanto el análisis de señales como el desarrollo de herramientas interactivas en Python. A través del identificador de notas musicales y la interfaz gráfica de filtrado de audio, hemos aplicado conceptos como la Transformada Rápida de Fourier (FFT) y el uso de filtros para mejorar la claridad de las señales. Este conocimiento sienta una buena base para futuros proyectos en los que queramos combinar análisis de señales con aplicaciones prácticas y visuales.