Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Implementación de dispositivo de procesado de señal de audio y generación de excitación a LEDs individualmente direccionables

Autor: José Manuel Gilibert Valdés

Tutor: Juan Antonio Becerra González

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Departamento de Teoría de la Señal y Comunicaciones

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Implementación de dispositivo de procesado de señal de audio y generación de excitación a LEDs individualmente direccionables

Autor:
José Manuel Gilibert Valdés

Tutor:
Juan Antonio Becerra González
Profesor Sustituto Interino

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Trabajo Fin de Grado: Implementación de dispositivo de procesado de señal de audio y generación de excitación a LEDs individualmente direccionables

Autor: José Manuel Gilibert Valdés
Tutor: Juan Antonio Becerra González

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

A mi familia y amigos. A mi padre, que no pudo verme terminar lo que me ayudó a empezar. A mi tutor, por su paciencia y su ayuda. A Sevilla, por acogerme y hacerme sentir como en mi segunda casa.

José Manuel Gilibert Valdés

Resumen

Implementación en Python de un sistema sobre una Raspberry Pi capaz de controlar una matriz 8x8 de LED de colores RGB en función de un sonido de entrada, a través de un cable de audio, en tiempo real. Se intenta simular un efecto de sinestesia entre el sonido y la imagen.

Se introduce la base teórica del procesado digital de señales, con especial énfasis en su aplicación a las señales de audio. Se explican diferentes conceptos como el de los filtros mel, la frecuencia fundamental (*pitch*), esenciales en el funcionamiento básico del sistema a implementar. Al ser un trabajo interdisciplinar, también aparecerán conceptos musicales y de teoría del color.

La señal de audio de entrada se va enventanando y se procesa como un *stream* en el dominio de la frecuencia por bandas mediante un filtrado mel, calculándose la potencia por bandas y determinando el nivel de LEDs que se encienden en la matriz. Cada columna correspondería a una banda de frecuencia, correspondiéndose el nivel de potencia con las filas. Respecto a los colores, se dispone de un modo monofónico (modo *pitch*) basado en la detección de la frecuencia fundamental y un modo polifónico (modo *default*) basado en la tímbrica espectral, es decir, en las posiciones relativas de los centroides espectrales por banda.

Abstract

Python implementation of a system on a Raspberry Pi able of controlling an LED RGB colors matrix based on a real-time input sound through a speaker wire. An attempt is made to simulate a synesthesia effect between sound and image.

The theoretical basis of digital signal processing is introduced, with special emphasis on its application to audio signals. Different concepts are explained, such as mel filters, fundamental frequency (pitch), essential in the basic operation of the system to be implemented. As it's an interdisciplinary work, musical and color theory concepts will also appear.

The input audio signal is windowed and processed as a stream in the frequency domain by bands through a filter mel, calculating the power by bands and determining the level of LEDs that are lit in the matrix. Each column would correspond to a frequency band, with the power level corresponding to the rows. Regarding colours, there's a monophonic mode (pitch mode) based on the detection of the fundamental frequency and a polyphonic mode (default mode) based on the spectral timbre, that is, the relative positions of the spectral centroids per band.

Índice abreviado

Índice

1 Introducción

Para ver un mundo en un grano de arena y un paraíso en una flor silvestre, sostén el infinito en la palma de la mano y la eternidad en una hora.

William Blake

En una realidad cada vez más globalizada e interconectada, cada vez cobran mayor relevancia las interfaces, sistemas intermediarios encargados de comunicar elementos que, a priori, no podrían comunicarse.

Existen grandes e interesantes proyectos actuales relativos a las interfaces en el ámbito tecnológico, como el desarrollo de una interfaz cerebro-máquina. También otros no tan ambiciosos, aunque igualmente interesantes, como el reconocimiento y lectura de texto en imágenes para personas con dificultades visuales.

Sin embargo, las mayores interfaces de las que disponemos con la realidad que nos rodea son nuestros sentidos. Y, para algunas personas, estos se mezclan. Este es el fenómeno conocido como **sinestesia**, consistente en una intercomunicación de dos o más sentidos. Por ejemplo, visualizar colores concretos al escuchar notas musicales concretas o poder oír texturas visuales. También existen sinestesias más abstractas, como la asociación de un color a un día de la semana.

Si bien en el siglo XX el fenómeno de la sinestesia obtuvo relevancia debido a las vanguardias artísticas y al consumo de sustancias psicodélicas (entre sus efectos, destaca la sinestesia), muchas personas adquieren esta capacidad de forma innata. Aleksandr Skriabin (1872-1915), pianista y compositor ruso, fue uno de ellos. Skriabin tenía la capacidad de asociar notas musicales con colores y desarrolló un sistema de composición basado en esto, completamente alejado de los tradicionales.

Skriabin solo es un ejemplo de lo cerca que pueden estar los medios auditivos y los visuales. De hecho, en el presente hablamos constantemente de medios audiovisuales, por lo que no debería extrañarnos.

El objetivo de este trabajo es, tecnología mediante, simular una especie de sinestesia artificial, implementando un sistema capaz de controlar una matriz de LED de colores en función de un sonido de entrada en tiempo real. Concretamente, nos centraremos en la música, tratando de generar una respuesta visual y aproximarnos a una interfaz músico-visual.

Debido a la naturaleza interdisciplinar de este trabajo, se tocarán muchos temas, aunque su esencia teórica estará en el procesado digital de señales. Además, como la representación visual de la música es bastante subjetiva y compleja, el resultado obtenido podría no ser el esperado, ya que simplemente nos apoyaremos en algunas características objetivas y cuantificables de las señales de audio.

2 Hardware

En este capítulo se detallarán los elementos de hardware utilizados en el proyecto, incluyendo tanto las herramientas de trabajo como los elementos del sistema del proyecto en sí.

2.1 Raspberry Pi

La Raspberry Pi es un ordenador de bajo coste multipropósito ideado para el aprendizaje de programación y proyectos de electrónica.

Figura 2.1 Raspberry Pi 3B.

Su uso está bastante extendido debido a su coste, su versatilidad y su facilidad de uso. Según el modelo, sus características son diferentes, y en el trabajo que nos ocupa se va a emplear una Raspberry Pi 3B v1.2, representada en la Figura 2.1, la cual posee (entre otras) las siguientes características:

- CPU Broadcom BCM2837 ARM Cortex A53 Quad Core 1.2 GHz de 64 bits.
- 1 GB de RAM.
- 40 pines GPIO (General Purpose Input/Output).
- Puerto Ethernet 100Base-T (100 Mb/s).
- Tarjeta de conexión inalámbrica BCM43438 con 802.11 b/g/n y Bluetooth 4.2.
- 4 puertos USB 2.0.
- Puerto MicroSD para almacenar y cargar datos y el sistema operativo.

- Alimentado por MicroUSB con hasta 2.5 A.

El sistema operativo a emplear es Raspbian, una distribución de GNU/Linux basada en Debian Strech y adaptada específicamente a la Raspberry Pi.

Este sistema operativo permitirá, además de administrar de forma sencilla el sistema, ejecutar programas en lenguaje de alto nivel (especialmente Python) para procesar los datos y actuar sobre el hardware externo (en este caso, la matriz LED).

Asimismo, se conectará una interfaz de audio USB con entrada y salida *minijack* 3.5mm. Se empleará la entrada para conectar el audio en línea.

La comunicación con la Raspberry Pi, necesaria para transferir y ejecutar los programas, se hará con el ordenador de la Subsección 2.3.1 a través de SSH y, generalmente, vía Wi-Fi en la misma red local.

Respecto a los pines de la Rapberry, los cuales aparecen en la Figura 2.2, se utilizarán tres de ellos:

- **Pin 2/5 V**: Tensión de alimentación de 5 V.
- **Pin 6/Ground**: Tensión de referencia o tierra.
- **Pin 12/GPIO18**: Pin de tipo *General Purpose Input/Output* (Entrada/Salida de Propósito General). Concretamente, se trata de un pin para controlar la señal de reloj de PCM o, como en este caso, para PWM.

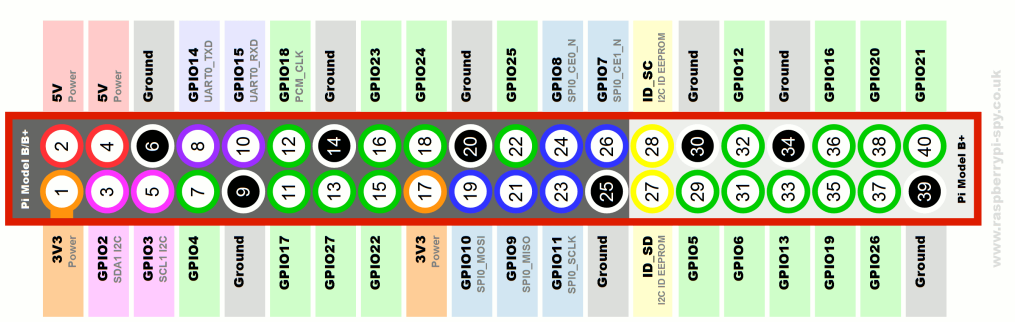

Figura 2.2 Pines disponibles en la Raspberry Pi 3B.

2.2 Matriz LED

Otro hardware que se va a emplear en este proyecto es la tira de LED 5050 WS2812 RGB, pero en forma de matriz *8x8*, como aparece en la Figura 2.3.

Entre sus características se encuentran:

- Un total de 64 LEDs RGB (8x8).
- 256 valores de brillo.
- Transmisión de la señal en cascada en una sola línea (pines DIN a DOUT).
- Envío de datos a velocidades de 800 Kb/s.
- Alimentado por 5 V.

Como se ha comentado, la matriz en realidad es una tira, solo que dispuesta de esa forma, por lo que solo tiene tres pines de conexión de entrada.

Dichos pines son:

- **DIN**: El pin que controla los datos de entrada. Típicamente se controla mediante PWM.

Figura 2.3 Matriz WS2812 8x8.

- **VCC/VDD**: El pin de alimentación. Necesita 5 V.
- **VSS/GND**: El pin de referencia de tensión o toma de tierra.

Cabe destacar que estos pines vienen sin soldar, por lo que el proceso de soldarlos formará parte del trabajo.

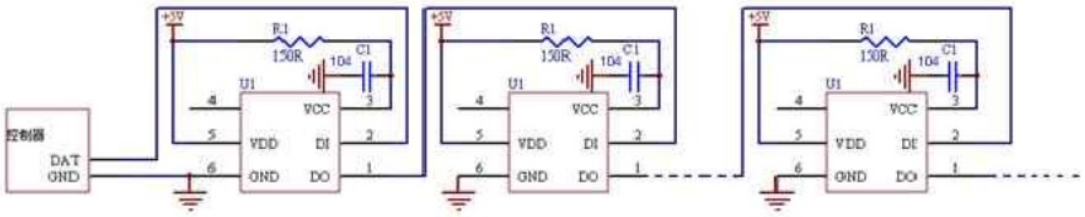

Figura 2.4 Ejemplo de conexionado en serie de varios WS2812.

Asimismo, el componente incluye otros tres pines similares a la salida, permitiendo combinar varias matrices conectándolas en serie como en el esquema de la Figura 2.4. Sin embargo, en este proyecto solo se trabajará con una.

2.2.1 Protocolo

Según la datasheet[28], con los valores de alimentación descritos la frecuencia típica con la que se trabaja con este componente es de 800 KHz, por lo que la tasa de transmisión máxima de dato es de 400 KHz (50% del *duty cycle*).

El protocolo que emplea la tira LED para saber qué color mostrar en cada LED es bastante sencillo. Utiliza una modulación PWM (modulación por ancho de pulso) con tres posibles símbolos (ver Figura 2.5):

- **1**: Pulso alto largo (más tiempo en nivel alto que en nivel bajo).
- **0**: Pulso bajo largo (más tiempo en nivel bajo que en nivel alto).
- **Reset**: Más de 50µs en nivel bajo.

Debido a que se trata de un sistema en cascada, los datos se van enviando en serie. Cada secuencia corresponde a la información asignada a un solo LED, constituida por 24 bits de información, 8

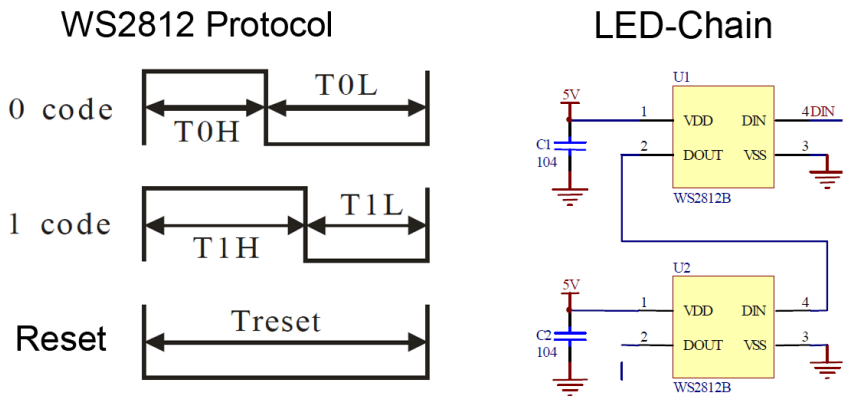

Figura 2.5 Esquema del protocolo del WS2812.

bits para cada canal de color (rojo, verde y azul). Sin embargo, debido a las especificaciones, los canales se envían en el orden G, R y B y los bits de más a menos significativo.

2.3 Otros componentes

Además de los componentes ya mencionados, en el proyecto se utilizarán otros componentes descritos a continuación en esta sección.

2.3.1 Ordenador empleado

El ordenador donde se ha desarrollado el proyecto (especialmente para la documentación, programación, pruebas y conexión con la Raspberry Pi) posee las siguientes características:

- 16 GB de RAM DDR4.
- Procesador Intel®Core™i7-7700HQ 2.81 GHz.
- SSD 256 GB.
- Disco duro de 1 TB.
- x64 bits.
- Tarjeta gráfica NVIDIA®GeForce GTX 1060.

2.3.2 Tarjeta de sonido USB

Si bien la Raspberry Pi posee de forma nativa hardware para el audio, como ya se comentó en la Sección 2.1, se ha optado por utilizar una tarjeta USB. Concretamente, la Sabrent Adaptador de USB externo para estéreo, visible en la Figura 2.6.

Específicamente, se dispondrá de su conector de entrada audio analógico estéreo 3.5mm, por el cual la Raspberry Pi recibirá el audio a procesar.

2.3.3 Cableado y otros

En el proyecto se emplean los siguientes cables:

- Cable de alimentación de la Raspberry Pi.
- Cable *minijack* estéreo de doble hembra a macho.
- Dos cables *minijack* estéreo macho a macho.
- Cables simples para la conexión de pines.
- Cable Ethernet RJ-45 para la conexión de red sin utilizar Wi-Fi.

Figura 2.6 Sabrent Adaptador de USB externo para estéreo [26].

También tendremos algunos equipos de audio:

- Un reproductor de audio con salida *minijack*.
- Unos altavoces con entrada *minijack*.

Los tres cables *minijack* conectarán el reproductor y los altavoces con la Raspberry Pi, de tal forma que sea posible escuchar el audio que está recibiendo la Raspberry Pi. Este esquema de conexión puede verse en la Figura 2.7.

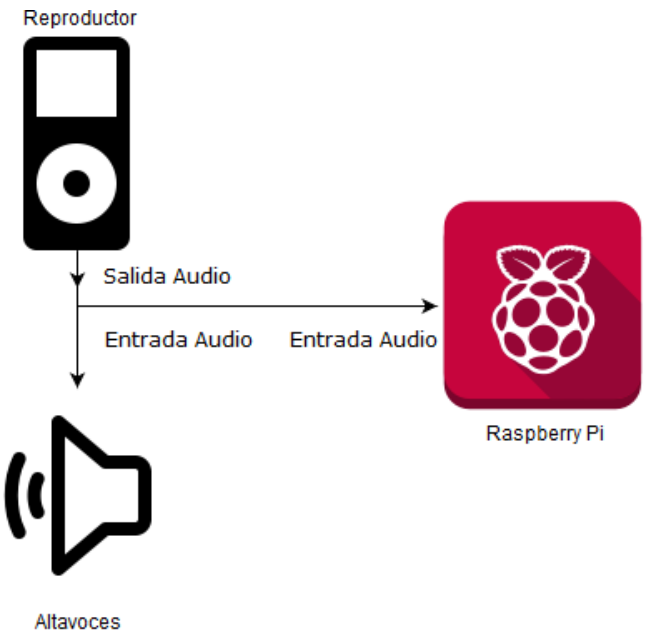

Figura 2.7 Conexión del hardware de audio.

3 Procesado digital de señales

En este capítulo se explicarán ciertos conceptos básicos relacionados con el ámbito de la teoría del procesamiento de señales digitales, así como su aplicación al audio digital.

No se pretende, ni mucho menos, que este apartado teórico se constituya como un referente integral donde se desarrollan estos conceptos, ya que existen referencias clásicas como [25] y [24]. También resultan imprescindibles ciertos conocimientos previos sobre señales y sistemas en tiempo continuo, pues se especificará relativo a tiempo discreto. Otro material interesante de consulta sobre este ámbito es [29].

Más bien, el objetivo de este capítulo es desarrollar el camino desde lo elemental de señales digitales hasta la extracción de características básicas de las señales de audio fundamentales para este trabajo.

3.1 Introducción

Una señal x en tiempo discreto consiste en una secuencia cuya variable independiente n (normalmente el número de la muestra) es discreta, por lo que dicha variable podrá tomar un número finito de valores, llamados muestras.

Un ejemplo sencillo de señal $x[n]$ en tiempo discreto es la de la Ecuación 3.1, donde se define la función escalón unitario en tiempo discreto.

$$u[n] \equiv \begin{cases} 0 & si \quad n < 0 \\ & \forall n \in \mathbb{Z} \\ 1 & si \quad n \geq 0 \end{cases} \tag{3.1}$$

Si la variable dependiente x también es discreta, se denomina señal digital. En este caso, la señal anterior es también digital, ya que x solo toma los valores $0, 1$. No obstante, con bastante frecuencia hay que convertir los valores continuos de la señal en valores discretos mediante algún tipo de redondeo. Este proceso se denomina cuantización.

Un ordenador posee un número finito de bits para almacenar la información, siendo todo digital, por lo que manejar este tipo de señales nos será bastante útil.

3.1.1 Muestreo y cuantización

En nuestra realidad (hasta que se demuestre lo contrario) las señales son de naturaleza continua, por lo que es esencial realizar algún tipo de transformación a señal digital para poder procesarla como tal.

Esta conversión se denomina *muestreo*, y consiste en tomar valores (equidistantes) de la señal continua cada vez que pasa un intervalo T_s, denominado periodo de muestreo. Cuanto menor sea

este valor, más rápido se tomarán las muestras, por lo que se asemejará más a la señal continua original.

En ocasiones, es más cómodo hablar de frecuencia de muestreo (muestras por segundo), definido como:

$$f_s \equiv \frac{1}{T_s} \tag{3.2}$$

Llegados a este punto, surge la cuestión de qué frecuencia de muestreo utilizar, de tal forma que se conserve la información. Es decir, lo óptimo es encontrar una frecuencia de muestreo tal que pueda reconstruirse de forma unívoca la señal original continua a partir de la señal digital.

La solución a este problema viene determinada por el teorema de Nyquist-Shannon, que establece que la frecuencia de muestreo a tomar debe ser dos veces la de su máxima componente de frecuencia. Por ejemplo, si tenemos una señal sinusoidal de 100 Hz, habría que aplicarle una frecuencia de muestreo de 200 Hz, o lo que es lo mismo, tomar una muestra cada 5 milisegundos.

Si esto último no ocurre, muestreándose a una frecuencia inferior y no pudiendo reconstruir la señal original, se producirá un efecto conocido como *aliasing*, distorsionando la información original.

En el caso concreto del audio, la frecuencia máxima audible por el ser humano es de 20 kHz. Por lo que, siguiendo este teorema, habría que aplicar una frecuencia de muestreo de 40 kHz. Se suelen tomar 44.1 kHz según el estándar más utilizado, popularizado por los CDs de audio. Sin embargo, es común ver también frecuencias de muestreo más altas para audio, como la de 48 kHz, justificadas por otras razones técnicas.

La cuantización, por el contrario, consiste en hacer un redondeo de los valores de la señal real. Es necesario definir cuánta precisión le queremos dar a la señal digital, es decir, cuánto rango de valores vamos a tener.

Como la información digital se codifica en bits, se suele emplear directamente el número de bits que ocupa una muestra, directamente relacionado con el rango de valores posibles. Como es obvio, a más bits, mayor rango y precisión. Siguiendo el ejemplo del CD estándar, que tiene una cuantización de 16 bits, habría $2^{16} = 65536$ valores posibles.

3.1.2 Periodicidad

Un tipo especial de señales muy interesante es el de las señales periódicas, las cuales se repiten cada intervalo T, denominado periodo. Matemáticamente, son las señales que cumplen:

$$x[n] = x[n+kT] \quad \forall k \in \mathbb{Z} \tag{3.3}$$

3.1.3 Energía y potencia

Un concepto fundamental relacionado con las señales es el de energía, definida generalmente como:

$$E \equiv \sum_{n=0}^{N-1} |x[n]|^2 \tag{3.4}$$

Siendo N el número de muestras de la señal. Por otro lado, como este valor es dependiente del número de muestras de la señal, es decir, de su duración, suele emplearse la potencia, definida para un intervalo de N muestras como:

$$P \equiv \frac{1}{N} \sum_{n=0}^{N-1} |x[n]|^2 \tag{3.5}$$

Sin embargo, debido a la naturaleza no lineal de la percepción de la energía del sonido de nuestros oídos, estas expresiones suelen utilizarse en el ámbito de audio en decibelios (dB). Estos poseen naturaleza logarítmica y nos permiten manejarnos en una escala más coherente con lo que escuchamos.

Los decibelios no son unidades absolutas, sino relativas, ya que se calculan a partir de un valor de referencia P_0. En el caso del sonido, suele tomarse este valor de referencia como la del umbral de audición de una señal sinusoidal con frecuencia de 1000 Hz, es decir, $P_0 = 10^{-12}\ W$.

$$L_w = 10 \log \frac{P}{P_0} \tag{3.6}$$

3.2 Discrete Fourier transform (DFT)

3.2.1 La señal sinusoidal

Una de las señales básicas más importantes (especialmente en lo relacionado con el audio) es la señal sinusoidal (también llamado tono puro). Esta señal, en su versión discreta, se define como:

$$x[n] = A_0 \cos(2\pi f_0 n + \phi) \tag{3.7}$$

Los elementos que aparecen en la Ecuación 3.8 son:

- A_0: Amplitud de la señal.
- f_0: Frecuencia.
- ϕ: Fase inicial (puede ser nula).
- n: Muestras. Tiene que ser un número entero.

No obstante, en muchos contextos (como en el caso del audio digital), resulta más interesante tener esta expresión en función del tiempo t en lugar de las muestras. Es aquí donde entra la frecuencia de muestreo de la señal, siendo $f_s = \frac{n}{t}$, la ecuación resulta:

$$x[t] = A_0 \cos(2\pi f_0 f_s t + \phi) \tag{3.8}$$

3.2.2 Componentes de frecuencia

El matemático y físico Joseph Fourier demostró que cualquier señal (y, por extensión, las digitales también) periódica puede descomponerse como una suma ponderada de sus armónicos, es decir, señales sinusoidales de distintas frecuencias. Esto es lo que se conoce como las **series de Fourier**, que nos revelan una fascinante propiedad de las señales periódicas.

$$x(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos \frac{2n\pi}{T} t + b_n \sin \frac{2n\pi}{T} t \right] \tag{3.9}$$

La expresión de la serie de Fourier para una señal $x(t)$ con periodo T es la de la Ecuación 3.9. Los coeficientes a_0, a_n y b_n son los que ponderan la suma, los que regulan cuánto *aporta* cada armónico a la construcción de la señal original. Es necesario remarcar que la expresión que aparece no está particularizada para señales digitales, sino que es la expresión general para señales continuas y que la n en este caso se refiere al índice del armónico o componente de frecuencia.

3.2.3 La transformada de Fourier

A la hora de tratar con señales, una herramienta muy útil es la transformada de Fourier, bautizada así por lo visto en la Subsección 3.2.2. Basada en las series de Fourier, nos permiten trasladar una señal del dominio del tiempo al dominio de la frecuencia, pudiendo analizar de una forma bastante sencilla sus armónicos o componentes de frecuencia (también llamadas componentes espectrales).

Se define la señal $X(f)$, transformada de Fourier de la señal $x(t)$, como:

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-j2\pi ft} dt \tag{3.10}$$

Si analizamos esta expresión, puede sorprendernos el elemento exponencial complejo. Este no es otra cosa que una señal sinusoidal compleja, que debido a la fórmula de Euler:

$$e^{-j2\pi ft} = \cos 2\pi ft - j \sin 2\pi ft$$

Esta señal es especial porque es la única que solo posee una componente de frecuencia (ella misma). Las señales sinusoidales no complejas, si bien por definición solo parecen tener una componente de frecuencia, en realidad son dos (una negativa y otra positiva) debido a la simetría de las funciones seno y coseno.

Así que, en realidad, esta expresión describe la proyección de la señal original, en el dominio del tiempo, sobre esta señal especial, la más elemental en el dominio de la frecuencia.

A su vez, la transformada de Fourier es reversible, ya que existe la trasformada inversa de Fourier, definida como:

$$x(t) = \int_{-\infty}^{\infty} X(f) e^{j2\pi ft} df \tag{3.11}$$

Esta última nos permite obtener la señal original a partir de la transformada de Fourier, permitiéndonos movernos continuamente entre el dominio de la frecuencia o del tiempo, según nos convenga.

En ocasiones, en lugar de trabajar directamente con la transformada de Fourier, se trabaja con el espectro de densidad de energía. Este representa cómo se reparte la energía de la señal respecto a la frecuencia, y se define como:

$$S_{xx}(f) = |X(f)|^2 \tag{3.12}$$

Puede calcularse la energía de la señal $x(t)$ a partir de su densidad espectral de energía:

$$E_x = \int_{-\infty}^{\infty} S_{xx}(f) df$$

Sin embargo, con estas expresiones seguimos generalizando para señales periódicas continuas... ¿Y qué pasa con las señales periódicas digitales?

3.2.4 La transformada discreta de Fourier (DFT)

Una particularización de la transformada de Fourier para señales discretas (y digitales) es la transformada discreta de Fourier (DFT). Esta transformada $X(f)$ de la señal discreta $x(n)$ con N muestras se define como:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{\frac{-j2\pi nk}{N}} \qquad k = 0, 1, \ldots, N-1 \tag{3.13}$$

De forma análoga a la Subsección 3.2.1, esta expresión puede escribirse en función del tiempo en lugar de las muestras, conociendo la frecuencia de muestreo.

Su tranformada inversa quedaría como:

$$X[n] = \frac{1}{N}\sum_{0}^{N-1} X[k]e^{\frac{j2\pi nk}{N}} \qquad n = 0,1,\ldots,N-1 \tag{3.14}$$

La expresión para el espectro densidad de energía S_{xx} resulta exactamente igual que en el caso de señales continuas. Una expresión importante es la relación de Parseval, que demuestra la conservación de la energía. Para una señal $x[n]$ de N muestras en el rango definidas $[N/2, N/2-1]$, con su correspondiente DFT $X[k]$, quedaría:

$$\sum_{-N/2}^{N/2-1} |x[n]|^2 = \frac{1}{N}\sum_{-N/2}^{N/2-1} |X[k]|^2 \tag{3.15}$$

Hay que destacar que, al igual que el resto de transformadas de Fourier, la DFT es una señal compleja, pudiendo descomponerse en magnitud y fase. Sin embargo, debido a sus propiedades, si la señal $x[n]$ original es real y de simetría par, la DFT resultante será real, pues posee nula, por lo que carecería de parte imaginaria.

3.2.5 La transformada rápida de Fourier (FFT)

Una implementación eficiente en forma de algoritmo del cálculo de la DFT es la transformada rápida de Fourier, más comúnmente conocida como FFT, por sus siglas en inglés.

Sin entrar demasiado en detalles, en su implementación más típica este algoritmo se basa en ir dividiendo la señal en dos partes (inicialmente una para las muestras de índice impar y otra para las de índice par) e ir dividiendo así el cálculo en pequeños cálculos de transformadas más pequeñas, y así sucesivamente, hasta obtener el resultado final.

Debido a que su eficiencia es computacionalmente bastante mayor que la del cálculo directo de la DFT, este algoritmo es ampliamente utilizado a la hora de implementar el cálculo de la transformada de Fourier en un computador.

Sin embargo, ya que la señal se irá dividiendo en dos grupos sucesivamente, es bastante útil emplear señales con una longitud igual a alguna potencia de 2, por ejemplo, 512 muestras.

¿Y qué ocurre si la señal cuya FFT queremos calcular no tiene ese número exacto de muestras? Típicamente la solución es añadir *ceros* al final, rellenando con tantos ceros como se requieran para llegar al número de muestras deseado, es decir, a la potencia de 2 más cercana.

Esta técnica se conoce como rellenado con ceros o *zero-padding*. Por ejemplo, si tenemos una señal $x(n)$ con 500 muestras, se añaden 12 muestras con ceros, llegando así a $512 = 2^9$ muestras.

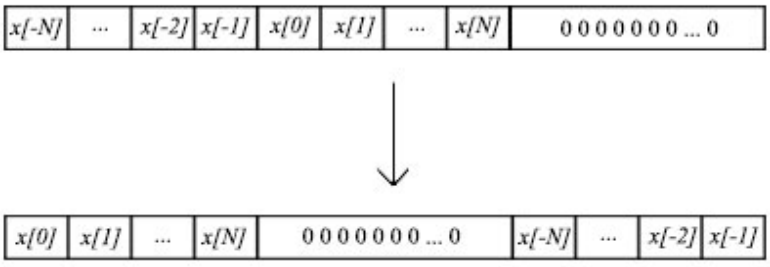

Figura 3.1 Ejemplificación esquemática del *zero-phase windowing*.

Además, cuantas más muestras tengamos, más muestras tendrá la FFT resultante, por lo que será mayor la resolución en el dominio de la frecuencia. Podemos aprovechar el *zero-padding* para esto y obtener así una interpolación de la señal en el dominio de la frecuencia, suavizando la curva. Por otro lado, hay que tener cuidado y llegar a un término medio, ya que se puede perder información si añadimos demasiados ceros.

Para no alterar la fase al realizar el *zero-padding* se puede realizar la técnica del *zero-phase windowing*, ejemplificada en la Figura 3.1. Esta consiste en, tras añadir los ceros al final, tomar la parte negativa de la señal ($t < 0$) y colocarla al final, tras los ceros.

3.3 El espectrograma y la short-time Fourier transform (STFT)

Si bien a partir de la DFT se puede analizar frecuencialmente una señal periódica, generalmente las señales de audio reales con las que trabajamos no lo son. Cualquier sonido real presenta, al menos, ciertas dinámicas debidas a transitorios y su potencia va variando, generando así una estructura bien definida en la envolvente de su nivel:

- **Attack**: El ataque o inicio de la señal de sonido. Se caracteriza por una subida del nivel mínimo de potencia hasta el máximo. Por ejemplo, es el momento justo en el que un pianista pulsa una tecla del piano.
- **Decay**: El decaimiento de la potencia de la señal de sonido. Suele aparecer tras el ataque, haciendo descender el nivel de la señal del máximo anterior hasta la altura que mantendrá. Es la transición entre el estado de *attack* y el de *sustain*.
- **Sustain**: El sostenimiento. Es la parte más estable de la señal, manteniéndose en su nivel de potencia. Por ejemplo, es el sonido sostenido de la tecla del piano que se mantiene pulsada.
- **Release**: Es lo que se conoce como la cola de la señal de sonido. En este estado la señal va perdiendo potencia hasta desvanecerse. Por ejemplo, es el sonido residual que progresivamente se va apagando cuando el pianista suelta la tecla del piano.

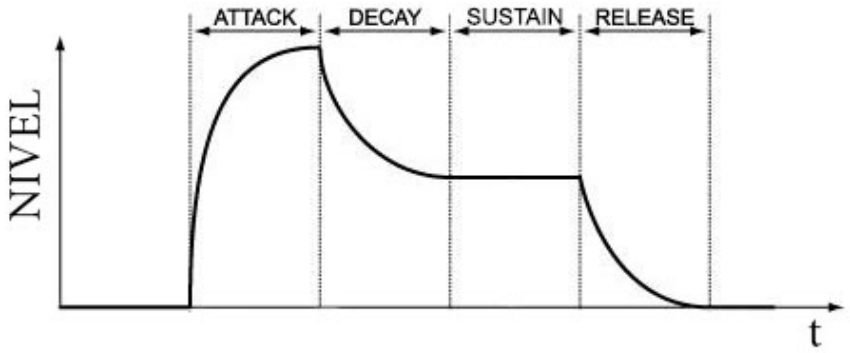

Figura 3.2 Estructura de las dinámicas de un sonido real.

Sin embargo, las duraciones de cada una de estas etapas de la evolución de la dinámica de un sonido real son bastante variables, ya que dependen de la fuente de sonido y de la propia acústica. Por ejemplo, normalmente un violín tiene un ataque de mayor duración que un piano, o en una catedral el *release* será mayor que en un estudio, debido a la mayor reverberación.

Además de todo esto, una señal de audio real normalmente no será un simple sonido, sino que será una sucesión de estos, por lo que rara vez una señal de audio será una señal periódica ideal.

3.3.1 Enventanado

En primer lugar, es necesario definir el concepto de enventanado. El enventanado es un proceso que consiste en *recortar* la señal en trozos sucesivos más pequeños, permitiendo así analizar en mayor detalle una sección concreta de la señal.

Existen diversas formas de realizar el enventanado, pero de forma genérica se hace tomando una serie de muestras sucesivas de la señal de longitud L (que denominaremos longitud de la ventana o *window length*) y multiplicarlo por una señal $w(l)$ (que denominaremos ventana o *window* de la misma longitud L. Así, dispondremos de la señal original dividida en pequeños trozos enventanados. A la señal enventanada la denotaremos como $x_w[n]$.

Además, este enventanado puede hacerse con solapamiento u *overlap*, no teniendo por qué ser disjuntos estos pequeños trozos enventanados.

Como es de esperar, el tamaño que elijamos de la ventana, el solapamiento o la elección de la propia ventana en sí condicionarán el resultado obtenido. Existen diferentes tipos de ventana, siendo la más básica la rectangular:

$$w_{rect}[l] = 1 \qquad l = 0, 1, \ldots, L-1 \tag{3.16}$$

Elección de la ventana

Una de las propiedades de la DFT es que una convolución entre dos señales en el dominio del tiempo se convierte en una multiplicación en el dominio de la frecuencia, y viceversa, por lo que, siendo $W[k]$ la DFT de la ventana $w[n]$:

$$x_w[n] = x[n] \cdot w[n] \leftrightarrow X_W[k] = X[k] * W[k] \tag{3.17}$$

Entonces, en el dominio de la frecuencia, la señal $X[k]$ es filtrada por la señal ventana $W[k]$. Esto implica que la elección de una ventana u otra nos dará una DFT distinta, siempre distorsionada respecto a la que idealmente sería la correcta.

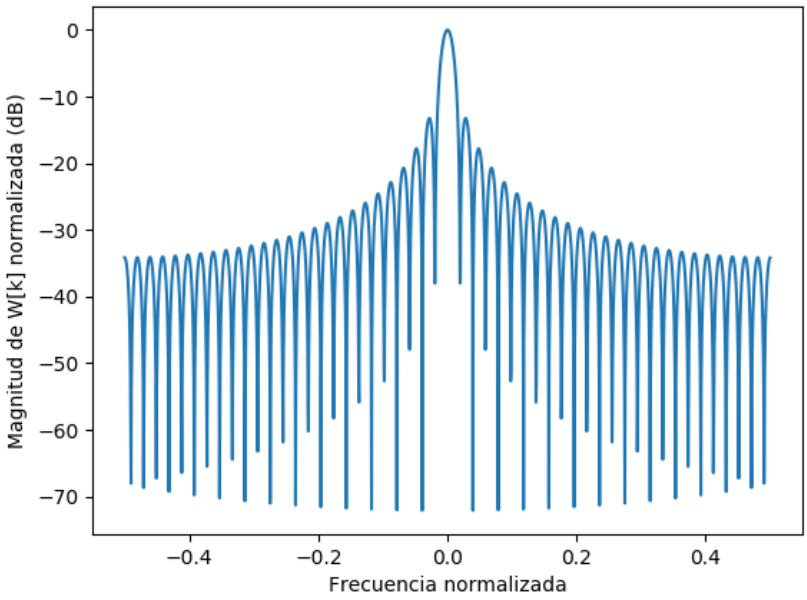

Figura 3.3 Magnitud en dB normalizada de la DFT de una ventana $w_{rect}[n]$.

En la Figura 3.3 puede apreciarse la magnitud en dB normalizada para una ventana $w_{rect}[n]$. En la parte central tenemos lo que se conoce como lóbulo principal, y a los lados los lóbulos laterales.

Cuanto menor sea el ancho de banda del lóbulo principal y menor sea la altura de los lóbulos laterales, menos se distorsionará la señal al enventanarla. Esto es debido a que queremos que la señal enventanada se modifique lo menos posible en frecuencia alrededor de sus picos.

En el caso de la $w_{rect}[n]$, se tiene un ancho de banda del lóbulo principal pequeño, pero sus lóbulos laterales son bastante pronunciados.

Existen otras ventanas más suaves, como la ventana Hann o hanning, cuyos lóbulos laterales son significamente menores, a costa de aumentar el ancho de banda del lóbulo principal. Esto puede verse comparando la Figura 3.5 con la Figura 3.3.

$$w_{hann}[n] = 0.5 - 0.5\cos\left(\frac{2\pi n}{L-1}\right) \tag{3.18}$$

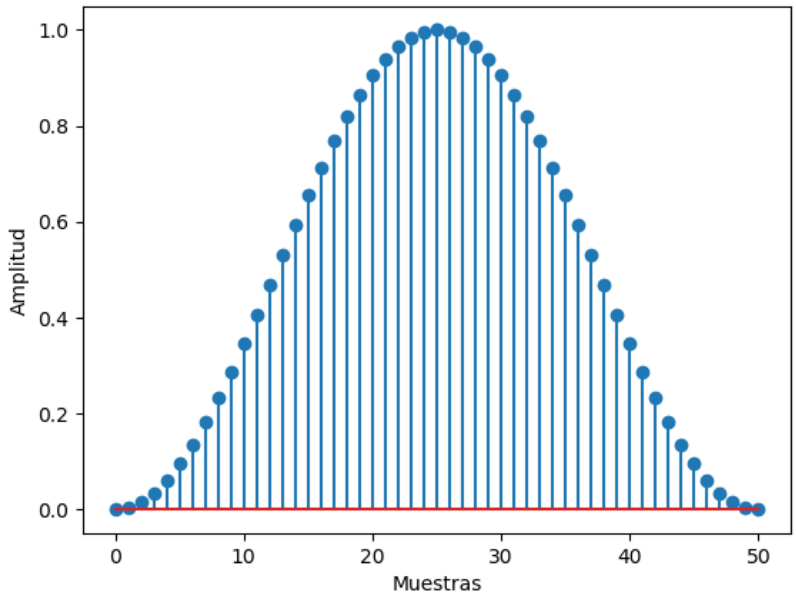

Figura 3.4 Ventana Hann $w_{hann}[n]$ con $L = 51$.

Otras ventanas bastante utilizadas en el ámbito de las señales de audio son la Hamming y la Blackman-Harris, entre otras. La ventana Hamming es muy similar a la Hann y reduce aún más el nivel de los lóbulos laterales respecto a ella, sin un coste asociado al aumento del ancho de banda, lo que la hace objetivamente mejor que su predecesora. Su expresión es:

$$w_{hamming}[n] \approx 0.54 - 0.46\cos\left(\frac{2\pi n}{L-1}\right) \tag{3.19}$$

La ventana Blackman-Harris va más allá con la reducción de los lóbulos laterales introduciendo nuevos términos sinusoidales. Sin embargo, esta mejora no es gratis, ya que aumenta también el ancho de banda del lóbulo principal. La más común es la de tres términos y su expresión es:

$$w_{bh}[n] \approx 0.36 - 0.49\cos\left(\frac{2\pi n}{N-1}\right) + 0.14\cos\left(\frac{4\pi n}{N-1}\right) - 0.01\cos\left(\frac{6\pi n}{N-1}\right) \tag{3.20}$$

3.3.2 Short-Time Fourier Transform (STFT)

La transformada de Fourier de tiempo reducido o, como se conoce en inglés, la *Short-Time Fourier Transform* (STFT) es una extensión de la DFT, ya descrita anteriormente en la Subsección 3.2.4.

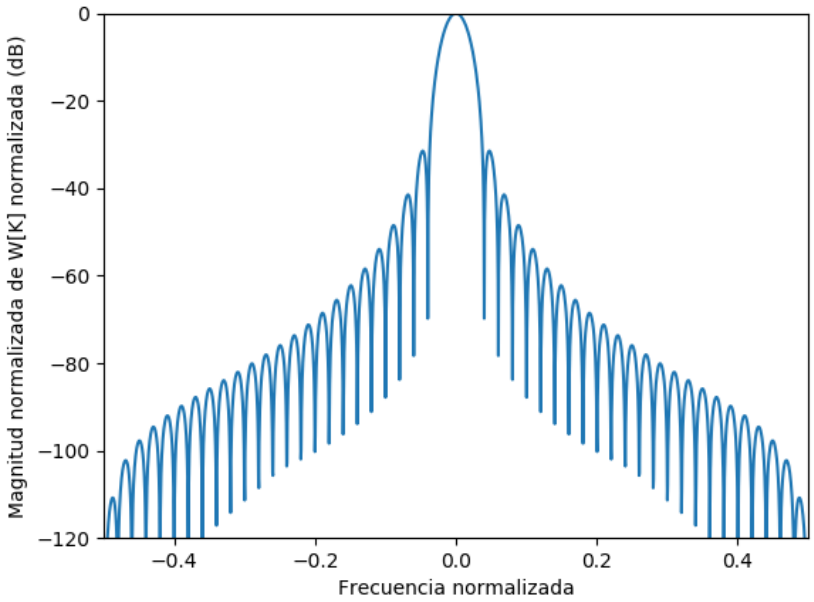

Figura 3.5 Magnitud en dB normalizada de la DFT de una ventana $w_{hann}[n]$.

Como ya se ha comentado, la DFT no permitía el análisis en el dominio de la frecuencia de señales no periódicas. Sin embargo, si hacemos *zoom* sobre una señal no periódica, tomamos un pequeño trozo de esta y la repetimos periódicamente, podremos calcular su DFT. De esta forma, se puede analizar esa pequeña parte de la señal original no periódica en el dominio de la frecuencia.

En esto se basa la STFT. Su expresión, en su versión discreta, es la siguiente:

$$X[m,k] = \sum_{n=0}^{N-1} x[n+mH]w[n]e^{\frac{-j2\pi nk}{N}} \qquad k = 0,1,\ldots,N-1 \tag{3.21}$$

Lo que se está haciendo realmente es calcular varias DFT sobre la señal $x[n]$ desplazada variablemente y enventanada con la señal $w[n]$. Así, dividiremos la señal en distintos trozos enventanados sucesivos (con solapamiento o no).

En esta ocasión no tendremos una sola transformada, sino que tendremos un conjunto m de transformadas, que denominaremos tramos o *frames*. La variable H indica el tamaño del paso o *hop size*, es decir, cuántas muestras se desplazan entre tramo y tramo.

El tamaño de cada tramo vendrá determinado por el tamaño L de la ventana elegida. El número de tramos resultantes dependerá de L, de H, de si elegimos solapamiento y del número de muestras N de la señal original $x[n]$.

Los valores que elijamos para H y L afectarán también a la resolución en frecuencia de la STFT resultante. Un tamaño de ventana muy pequeño nos da mejor resolución en el dominio del tiempo, pero peor en el de la frecuencia (podríamos perder información frecuencial), y viceversa, por lo que hay que llegar a un compromiso.

3.3.3 El espectrograma

El espectrograma es una representación gráfica bidimensional, normalmente en forma de mapa de calor, de la magnitud de la STFT $X[m,k]$.

En el eje horizontal se coloca el eje temporal, es decir, el índice m, mientras que en el eje vertical se coloca el eje frecuencial, es decir, el índice k. Los valores de la magnitud se suelen representar mediante alguna escala de color, de tal forma que se puedan distinguir visualmente con alto contraste entre sí los valores altos y los bajos.

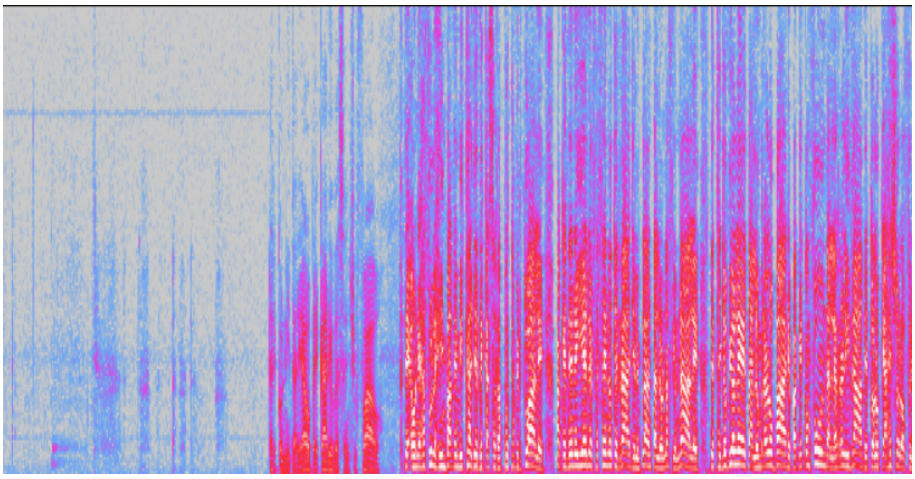

Figura 3.6 Espectrograma de una señal de voz real.

En la figura Figura 3.6, a modo de ejemplo, se muestra el espectrograma de un fragmento de una señal de voz real, obtenido con el software Audacity.

En aplicaciones de procesamiento de audio el espectrograma se utiliza, por ejemplo, para detectar los fonemas vocales de una señal de voz o para detectar el *pitch* (la frecuencia fundamental) de una nota musical.

Si bien en este proyecto no se empleará directamente el espectrograma o la STFT, nos dará una visión conceptual de cómo se procesará el sonido globalmente.

3.4 Filtros

Otro proceso elemental en el procesado de señales es el filtrado. En bastantes ocasiones es necesario mitigar, realzar o eliminar algunas componentes de las señales a tratar. Por ejemplo, muchas señales de audio presentan ruido que, al ser componentes indeseadas en la señal, se necesitan eliminar.

En ingeniería de sonido, a la hora de mezclar el audio, es también común el uso de filtros para realzar o disminuir ciertas frecuencias a conveniencia, normalmente para conseguir una mezcla más limpia.

En esta sección se explicarán los conceptos relativos al filtrado que, en el caso de este proyecto, se utilizarán filtros para extraer algunas características de las señales de audio. Se particularizará para el caso de sistemas con respuesta finita al impulso (FIR), los cuales se caracterizan por ofrecer una respuesta estable y finita a la entrada del sistema. Tampoco se ahondará en la caracterización de sistemas mediante la transformada Z, sino que se ofrecerá una visión superficial y necesaria para comprender los fundamentos de la implementación de este proyecto.

3.4.1 Respuesta al impulso y convolución

El filtrado consiste hacer pasar una (o varias) señal $x[n]$ por un sistema, obteniendo una señal $y[n]$. La forma que emplearemos para caracterizar el sistema es la de la respuesta de este al impulso.

La respuesta al impulso, que denominaremos $h[n]$, es la respuesta que ofrece el sistema para una señal de entrada $\delta[n]$, denominada delta de Dirac o impulso. Matemática y conceptualmente, esta señal se corresponde con la función $\delta(x)$, que se describe como:

$$\delta(x) \equiv \begin{cases} \infty & x = 0 \\ 0 & x \neq 0 \end{cases} \tag{3.22}$$

Figura 3.7 Filtrado de una señal x[n].

Figura 3.8 Respuesta al impulso h[n].

Resulta obvio que esta señal de la Ecuación 3.23 no puede existir en la realidad, ya que toma valores de magnitud infinita. Más bien, hay que verla como un concepto. Su característica particular es que es una señal que es nula en para todos sus valores excepto para uno, definido en un intervalo infinitamente pequeño, donde toma un valor infinitamente grande. Por ello, satisface la siguiente propiedad:

$$\int_{-\infty}^{\infty} \delta(x)dx = 1 \tag{3.23}$$

Esto se debe a que al calcular el área bajo la curva de esta función, el valor donde la función no es nula, el dx tiende a $1/\infty$, mientras que el valor de la función en ese punto tiende a ∞, por lo que se cancelan y la integral converge a la unidad.

Sin embargo, en el caso de señales digitales o discretas de N muestras, esta señal impulso puede aproximarse y satisfacer esta propiedad, acercándose a este concepto, de la siguiente forma:

$$\delta[n] \equiv \begin{cases} N & n = 0 \\ 0 & n \neq 0 \end{cases} \tag{3.24}$$

Puede comprobarse que esta señal δ[n] satisface la propiedad anteriormente mencionada calculando el equivalente a la integral:

$$\frac{1}{N}\sum_{-N/2}^{N/2-1} \delta[n] = \frac{1}{N} \cdot \delta[0] = \frac{1}{N} \cdot N = 1 \tag{3.25}$$

Una vez obtenida la respuesta al impulso $h[n]$ con N muestras, para una señal $x[n]$ con N muestras se calcula la salida $y[n]$ al sistema de la siguiente forma:

$$y[n] = x[n] * h[n] = \sum_{k=-N/2}^{N/2-1} x[k] \cdot h[n-k] \tag{3.26}$$

Si recordamos la propiedad vista en la Ecuación 3.17, definimos $H[k]$ como la DFT de la respuesta al impulso $h[h]$ y, siendo $X[k]$ la DFT de la señal $x[n]$:

$$y[n] = x[n] * h[n] \leftrightarrow Y[k] = X[k] \cdot H[k] \tag{3.27}$$

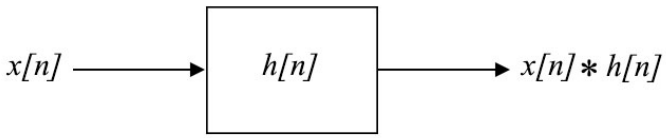

Figura 3.9 Respuesta del sistema ante señal de entrada x[n].

Así, se puede obtener de forma sencilla la DFT $Y[k]$ de la respuesta $y[n]$ del filtro ante la señal de entrada $x[n]$, simplemente obteniendo las DFT correspondientes. Solo quedaría calcular la transformada inversa para obtener finalmente la respuesta $y[n]$. A $H[k]$, DFT de $h[n]$, la denominaremos función de transferencia del sistema.

3.4.2 Filtrado en frecuencia

La función de transferencia $H[k]$ definida en la sección anterior puede interpretarse directamente en el dominio de la frecuencia.

Siendo $|H[k]|$ la magnitud o módulo de dicha función de transferencia, puede deducirse que, al ser una simple multiplicación, $|Y[k]|$ será nula para los valores k donde $|H[k]|$ sea nula.

Por tanto, si queremos eliminar algunas frecuencias de la señal $x[n]$, nos basta con filtrarla a través de un sistema cuya función de transferencia sea nula para dichas frecuencias. Esta es la idea fundamental a tener en mente para el diseño de filtros de frecuencia.

Idealmente, a la hora de filtrar frecuencias, la curva $|H[k]|$ no debería presentar ninguna pendiente. Sin embargo, en la realidad esto no siempre es así y depende de su implementación. A continuación se explican los tipos elementales de filtros de frecuencia:

- Un filtro paso bajo o *low-pass filter* (LP) es un filtro cuya función de transferencia $|H[k]|$ se hace nula a partir de una frecuencia f_H, denominada frecuencia de corte superior, dejando pasar solo las frecuencias menores a esta y eliminando las superiores.
- Un filtro paso alto o *high-pass filter* (LP) es un filtro cuya función de transferencia $|H[k]|$ se hace nula hasta una frecuencia f_L, denominada frecuencia de corte inferior, dejando pasar solo las frecuencias mayores a esta y eliminando las inferiores.
- Un filtro paso banda o *band-pass filter* (BP) es un filtro que puede verse como una combinación del LP y el HP. En este tipo de filtro la función de transferencia $|H[k]|$ solo no es nula dentro de la banda (rango de frecuencia) definida por la frecuencia inferior f_L y la frecuencia superior f_H. A la diferencia entre f_H y f_L se le denomina ancho de banda o *bandwith*. Es bastante útil para analizar el contenido energético de una señal $x[n]$ dentro de una banda concreta.

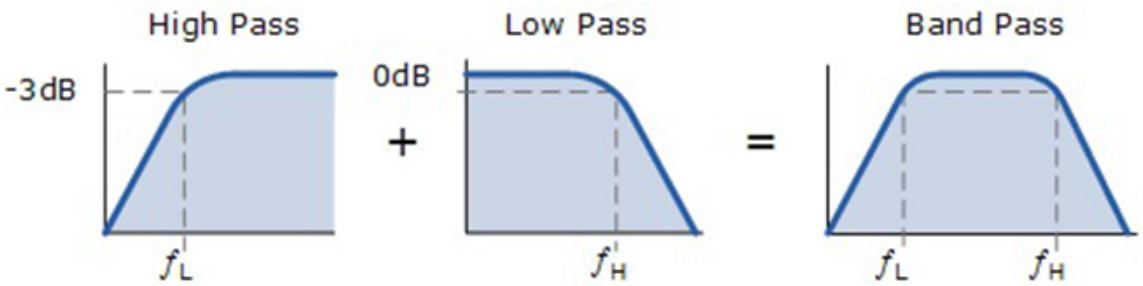

Figura 3.10 Filtros de frecuencia elementales.

4 Conceptos del sistema

En este capítulo, bastante heterogéneo, se explicarán algunos elementos teóricos fundamentales sobre la implementación de este trabajo. Algunas de las características extraídas del audio de este capítulo pueden consultarse en [23].

Al tratarse de un proyecto interdisciplinar, a caballo entre el procesamiento de audio y la representación de la imagen (el color), aparecerán conceptos en principio no tan relacionados entre sí y será necesario manejarlos y conectarlos.

Por ello, aunque pueda parecer una suerte de cajón de sastre, cada elemento es una pieza esencial del sistema.

Especialmente, nos centraremos en las características a extraer del audio que de verdad se emplean en el trabajo, dejando de lado muchas otras que podrían haberse implementado.

4.1 Bancos de filtros mel

El funcionamiento de nuestro sistema auditivo no es lineal. Ante estímulos externos relacionados con alguna magnitud física, si esta varía de forma geométrica (exponencial), nuestra percepción de ella será aritmética (lineal), tal y como establece la ley psicofísica de Weber-Fechner.

Por ejemplo, notaremos más la diferencia de peso al pasar de coger una carga de un gramo a otra de un kilogramo que al pasar de coger una carga de un kilogramo a otra de dos kilogramos.

Nuestra percepción de la frecuencia del sonido, así como su nivel de energía, no son una excepción, por lo que es de bastante utilidad trabajar con escalas logarítmicas, más coherentes con lo que escuchamos.

4.1.1 La escala mel

Una escala logarítmica típica para trabajar con valores de frecuencia es la escala mel. Esta es una escala que, perceptualmente, dispone la frecuencia en valores equidistantes.

La conversión de hertzios a mels viene dada por la siguiente expresión:

$$f_{mel} = 2595 \cdot \log_{10}(1 + f_{hertz}/700) \tag{4.1}$$

Por tanto, la conversión inversa viene dada por:

$$f_{hertz} = 700 \cdot (10^{f_{mel}/2595} - 1) \tag{4.2}$$

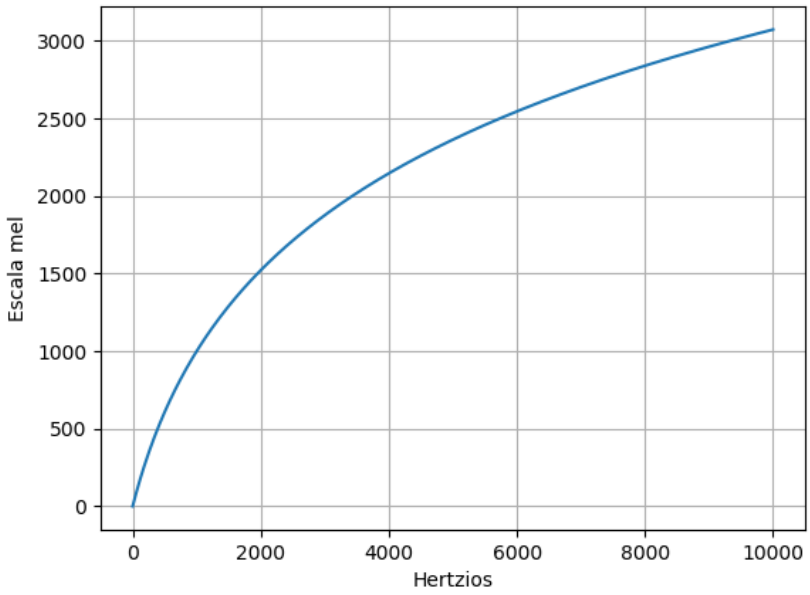

Figura 4.1 Gráfica de relación entre hertzios y mel.

4.1.2 Banco de filtros

Los bancos de filtros mel son un conjunto de filtros paso banda cuyo ancho de banda es proporcional a la escala mel. Es decir, cuanto mayor sea la frecuencia donde se coloque el filtro, mayor será su ancho de banda, creciendo exponencialmente.

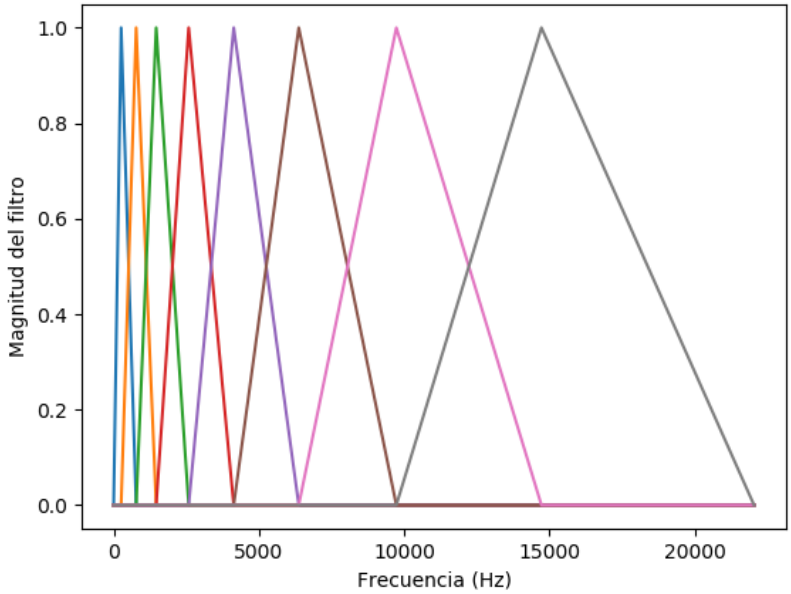

Figura 4.2 Banco de 8 filtros mel en escala Hz.

Hay que destacar que estos filtros no son excluyentes entre sí y existe un solape entre ellos, por lo que dos filtros sucesivos comparten frecuencias. Esto se hace porque, a priori, las bandas no son independientes entre sí, por lo que es necesario que, a la hora de analizarlas, los filtros compartan

información. En concreto, para el filtro siguiente la frecuencia inferior de corte f_L suele colocarse en la frecuencia central del filtro anterior.

Además, son filtros triangulares, alcanzando el máximo en la frecuencia central del filtro. En este trabajo se emplearán constantemente 8 filtros mel, ya que cada uno de ellos recogerá información de energía de una banda de frecuencia concreta, la cual será representada en las 8 columnas de la matriz LED.

Considerando una frecuencia de muestreo f_s igual a 44100 Hz, según lo visto en la Subsección 3.1.1, se tiene una frecuencia máxima de 22050 Hz. Dividiendo este rango en 8 filtros mel, su representación quedaría como en la Figura 4.2.

Si bien se han representado los filtros con la escala del eje de frecuencia en Hz, destacando el ancho de banda proporcionalmente creciente. Si se representa en escala mel, quedaría:

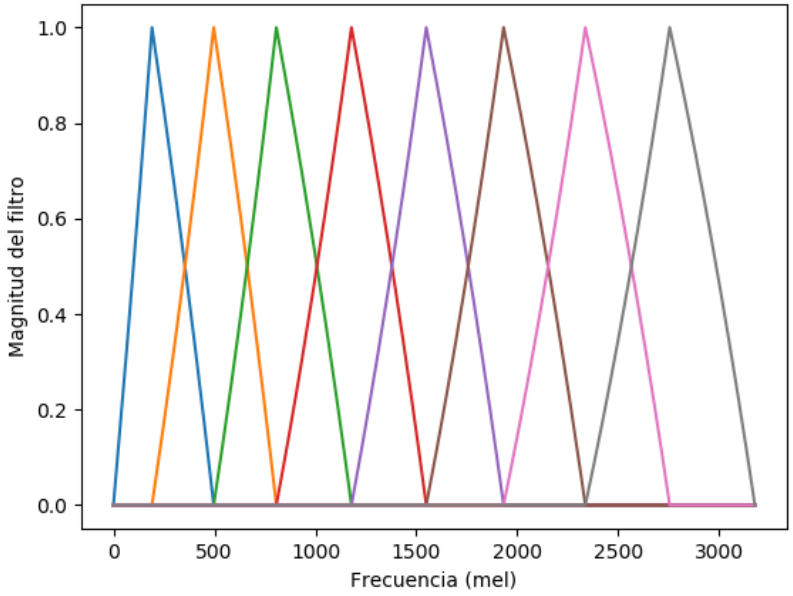

Figura 4.3 Banco de 8 filtros mel en escala mel.

4.2 Notas musicales y frecuencia fundamental

Otra característica del sonido, especialmente en lo relativo a lo musical, es la frecuencia fundamental o *pitch*. Musicalmente, característica está relacionado con el tono. Su extracción es un problema fundamental del área del procesado de señales de audio, con aplicaciones tales como el reconocimiento de melodías y acordes, permitiendo la transcripción automática de música.

Si bien este último es un campo de investigación en evolución, sobre todo en lo que a polifonías se refiere, existen diversos algoritmos bien asentados que nos permitirán conseguir una aproximación incluso en tiempo real.

4.2.1 Frecuencia fundamental

Lo que comúnmente denominamos nota musical no es más que la frecuencia fundamental o *pitch* de un sonido.

La frecuencia fundamental de un sonido es la frecuencia cuya presencia es mucho más intensa que la de las otras. Sin embargo, su detección puede ser engañosa, ya que en los sonidos de la vida real podemos encontrar armónicos de esta frecuencia fundamental distribuidos a lo largo del espectro.

Por ejemplo, analicemos un sonido real de una cuerda de guitarra dando la nota musical La. El sonido puede encontrarse en [21].

Mediante el software libre de análisis y edición de audio Audacity, obtenemos su espectrograma, configurado con un tamaño de ventana de 2048 muestras, un rango de frecuencias de 0 a 2000 Hz y un enventanado Blackman-Harris.

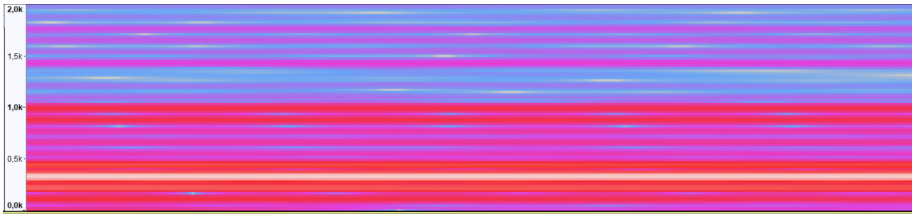

Figura 4.4 Espectrograma del sonido de cuerda de guitarra.

En la Figura 4.4 se tiene un *zoom* de dicho espectrograma, representado con colores de mapa de color. Pueden observarse algunas líneas paralelas de color rojo, las cuales son los armónicos del sonido. Pero, ¿cuál de las líneas rojas es la frecuencia fundamental? Sigamos haciendo *zoom* y, además, aumentemos el tamaño de la ventana a 4096 muestras para obtener mayor resolución.

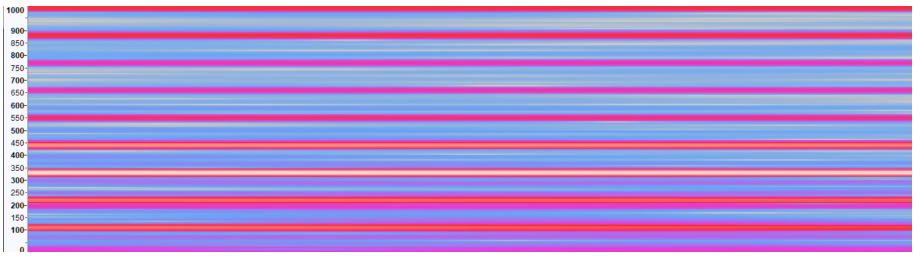

Figura 4.5 Armónicos del sonido de cuerda de guitarra.

En la Figura 4.5 puede verse que la primera línea horizontal roja se encuentra cerca de los 100 Hz. Se trata de la frecuencia fundamental, que si siguiéramos aproximándonos todavía más y ajustando la precisión, veríamos que es exactamente de 110 Hz, frecuencia correspondiente a la nota La_1 (A_2 en el cifrado americano).

Cuando un sonido posee un único *pitch* a la vez en el tiempo, se le denomina sonido monofónico. Por ejemplo, cuando tocamos una melodía simple en el piano, tocando notas individuales sucesivas, una detrás de otra.

Por otro lado, cuando existen temporalmente diferentes *pitch* en un sonido, se le denomina sonido polifónico. Por ejemplo, cuando tocamos algún acorde o suenan diferentes instrumentos melódicos a la vez.

4.2.2 Detección del pitch

Existen diversas formas de detectar el pitch. La más sencilla es utilizando la autocorrelación de la señal. Considerando un i-ésimo *frame* $x_i[n]$ de la señal $x[n]$, calculamos su autocorrelación como:

$$r_{xx}[k] = \sum_n x_i[n]x_i[n-k] \tag{4.3}$$

Esta autocorrelación tiene la propiedad de que, además de ser simétrica respecto al origen (donde alcanza su máximo), posee picos máximos donde la señal x_i se repite y, por tanto, nos da información sobre su periodicidad y frecuencia. De hecho, la distancia entre dos máximos consecutivos nos da información sobre la frecuencia fundamental.

Por ejemplo, volviendo al sonido de la Subsección 4.2.1, calculamos y representamos la parte positiva ($k > 0$) de su autocorrelación, ya que es redundante representarla completa debido a su simetría.

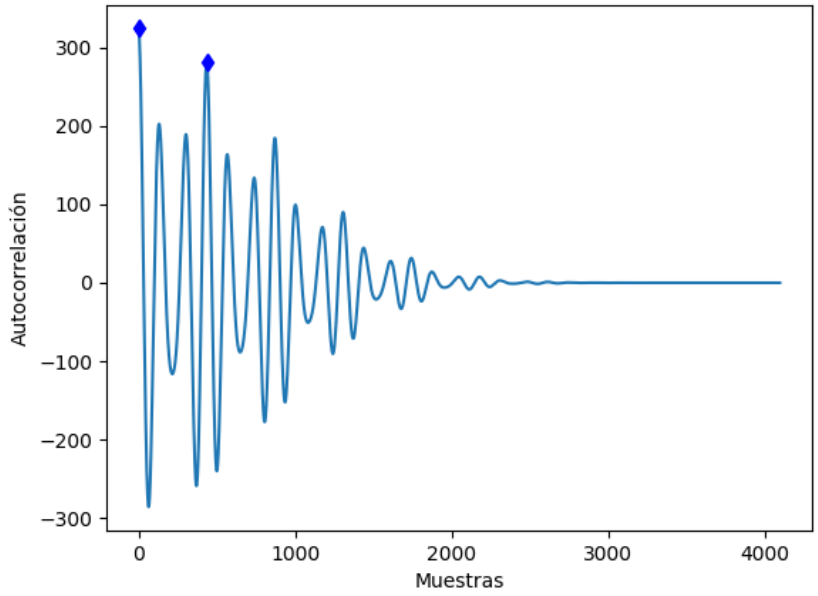

Figura 4.6 Autocorrelación de un *frame* del sonido de guitarra.

Además, en la Figura 4.6 se han representado los dos primeros máximos: el del origen ($k_0 = 0$) y el sucesivo ($k_1 = 436$). La diferencia de distancia entre ambos máximos es, obviamente, de 436 muestras. Conociendo que este sonido tiene una frecuencia de muestreo f_s igual a 48 kHz, puede calcularse la frecuencia fundamental a partir de esta distancia de 436 muestras:

$$f_0 = \frac{48000\,muestras/s}{436\,muestras} \approx 110.34\,\text{Hz}$$

Si bien esta aproximación es bastante buena, tiene sus carencias a la hora de automatizarla y generalizarla, por lo que existen algoritmos que parten de esta idea de utilizar la autocorrelación. Uno de ellos es el algoritmo YIN [17], el cual se empleará en este trabajo ya implementado.

4.2.3 Cifrado de notas

Si bien se ha explicado el concepto de *pitch*, faltan algunas cosas por aclarar respecto a su relación con las notas musicales. La afinación musical estándar que utilizamos hoy en día utiliza como referencia la nota La_4 (A_3 en cifrado americano), cuya frecuencia es de 440 Hz.

A partir de esta nota de referencia, se obtienen las siguientes multiplicando por el factor $2^{k/12}$, siendo k el número de semitonos a desplazarse (positiva o negativamente). Por ejemplo, si quisiéramos obtener la frecuencia del Do_5, que está 3 semitonos por encima del La_4:

$$f_0 = 2^{3/12} \cdot 440 \approx 523.25\,\text{Hz}$$

Obviamente, también se pueden calcular notas más graves. Por ejemplo, si quisiéramos obtener la frecuencia del Re_4, que está 7 semitonos por debajo:

$$f_0 = 2^{-7/12} \cdot 440 \approx 293.67\,\text{Hz}$$

Sin embargo, en la realidad rara vez se detecta el *pitch* con tal precisión o un instrumento está perfectamente afinado, por lo que en la práctica para un *pitch* detectado f_0, le asignaremos la nota más cercana, es decir, la nota cuya frecuencia fundamental estandarizada sea más cercana a f_0.

Croma

Un concepto relacionado es el de croma, que consiste en ignorar a qué octava pertenece la nota y quedarnos con la nota en sí. Por ejemplo, un Mi_3 tendrá el mismo croma que un Mi_5, pese a ser frecuencias diferentes. Por tanto, todas las notas de nuestro sistema musical podrían clasificarse en 12 posibles cromas, típicamente numerados del 0 al 11, aunque también se utiliza la el cifrado americano sin especificar octava.

En el caso de este trabajo, se empleará la detección del *pitch* para obtener su croma y trasladarlo al plano visual, donde tenemos su análogo cromático: el color.

4.3 Centroides espectrales

Las anteriores características nos permiten caracterizar lo que en términos musicales son el tono y la altura, carecemos de un medio para medir la tímbrica de un instrumento.

El timbre es lo que nos permite distinguir un sonido de otro, un instrumento de otro, es su ”textura”, y está determinado por la forma de su espectro. Si observamos el espectro de un sonido de un violín tocando la nota Mi, veremos que su forma difiere del espectro de un piano tocando exactamente lo mismo.

Esta diferencia viene establecida por los armónicos que componen el sonido total, así como por la altura de estos. En la Figura 4.7 se ejemplifica una comparación de dos sonidos, uno de un sintetizador (representado en naranja) y uno de un piano (representado en azul). Aunque ambos dan la misma nota (un A_2 o La_3), puede comprobarse que son diferentes.

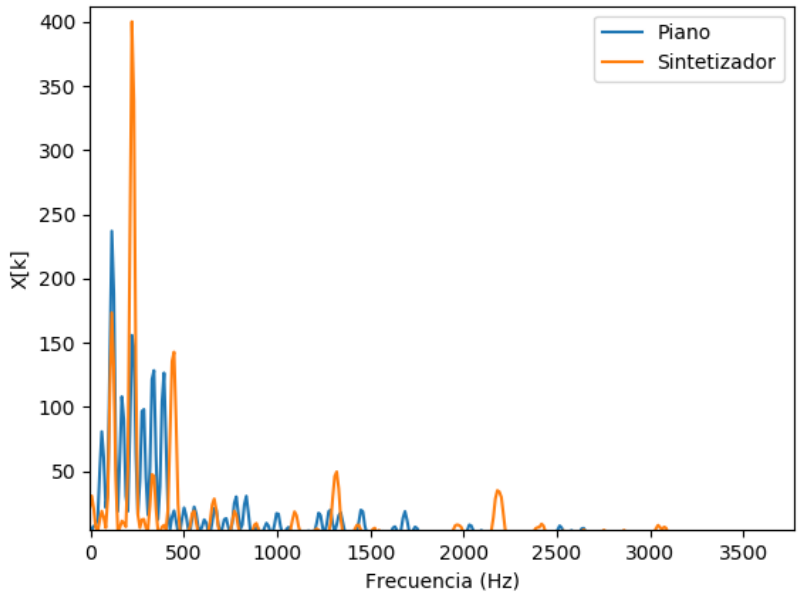

Figura 4.7 Comparativa del espectro de un piano y el de un sintetizador.

Si bien existen muchas formas de caracterizar el timbre, nos quedaremos con la más sencilla: el centroide espectral.

El centroide espectral nos da una idea del ”centro de masa” del espectro respecto al eje de frecuencias. Siendo $X[k]$ la DFT de la señal x[n], L el tamaño de la DFT y k la k-ésima frecuencia en muestras, calcula como:

$$\mathrm{CS} = \frac{\sum_{k=0}^{L-1} kX[k]}{\sum_{k=0}^{L-1} X[k]} \tag{4.4}$$

4.3.1 Centroides por bandas

Esta idea del centroide espectral puede extenderse por bandas. Si se consideran, como en este trabajo, 8 bandas de frecuencia, pueden considerarse cada una de ellas por separado como una DFT independiente.

De esta forma, se podrían calcular los centroides espectrales correspondientes de cada banda, dándonos una noción del contenido tímbrico de cada banda.

Además, este cálculo puede tener otra interpretación, ya que al ser una media ponderada, el centroide espectral de cada banda realmente está indicando la ”frecuencia media” de cada banda. Debido a esto, se podría interpretar que cada centroide por banda caracteriza qué frecuencia posee mayor presencia en cada banda.

Esta es la interpretación que se utilizará en este proyecto, calculando los centroides espectrales sobre cada banda obtenida según lo comentado en la Subsección 4.1.2.

4.4 Fundamentos del color

Aunque quizás de sobra conocido, el color es la percepción que tenemos de las distintas frecuencias pertenecientes al espectro visible de la luz. Es decir, es la representación mental de nuestro cerebro de las ondas electromagnéticas que captan nuestros ojos, con periodos de onda entre los 400 y 700 nanómetros.

Sin embargo, a la hora de trabajar con el color digitalmente no se emplea este modelo de señales electromagnéticas, debido a su complejidad tanto a la hora de obtenerlas mediante sensores como a la hora de procesarlas.

En su lugar, se trabaja con modelos más simples basados en cómo percibimos el color ya que, recordemos, es una percepción y no una magnitud física.

4.4.1 El modelo RGB

Basado en el funcionamiento de la retina del ojo humano, el cual posee tres tipos de conos (células fotorreceptoras que funcionan en condiciones de mucha luz y son sensibles a la frecuencia). Cada uno de estos tres tipos de cono están orientados a absorber luz electromagnética de una longitud de onda determinada y sus adyacentes, cada uno correspondiente al rojo, verde y azul respectivamente.

De esta forma, nuestros ojos recogen información en tres componentes: uno rojo, otro verde y otro azul. Esta información viaja a nuestro cerebro y es procesada, sintetizándose un color.

Así, surge el modelo RGB (Red, Green, Blue) mediante el cual podemos describir un color como la suma de cuánto lleva de cada componente. Este proceso se denomina síntesis aditiva (ver Figura 4.9).

Habitualmente estas tres componentes se representan en un vector de tres elementos de la forma $[R,G,B]$ cuyos valores son discretos y van del 0 al 255 (o decimales del 0 al 1 si se normaliza). Esto último es debido a que para cada componente de color en su representación digital suelen tomarse 8 bits.

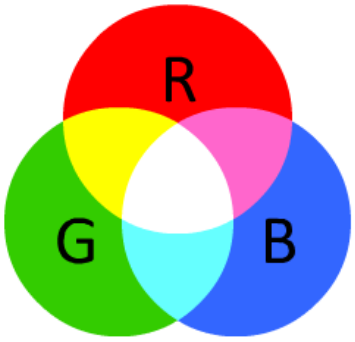

Figura 4.8 Síntesis aditiva RGB.

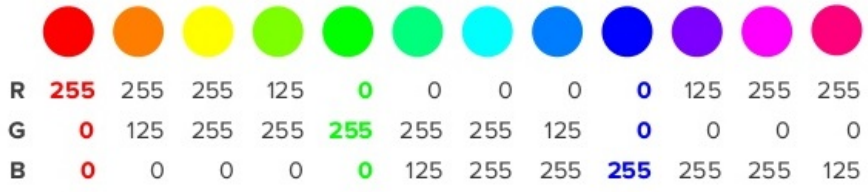

Figura 4.9 Combinaciones básicas de las componentes RGB.

4.4.2 El modelo HSV

Si bien mediante el modelo RGB podemos representar los colores, resulta poco intuitivo a la hora de situarlos. Si tenemos, por ejemplo, un naranja y queremos conseguir un color un poco más oscuro, no bastará con moverse en una coordenada de color, sino que tendríamos que jugar con las mezclas de las componentes RGB.

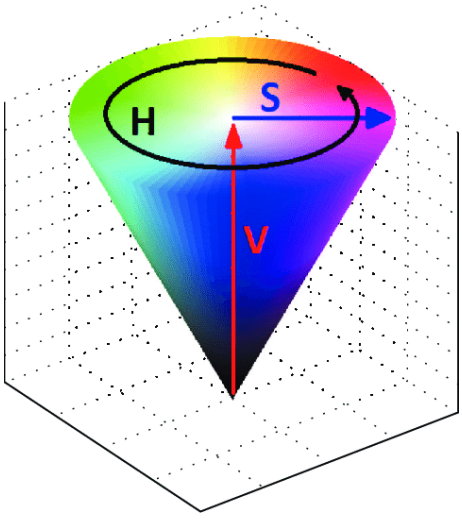

Figura 4.10 Espacio de color HSV.

Debido a esto, surge el modelo HSV, altamente utilizado por diseñadores gráficos, precisamente debido a lo intuitivo que resulta.

El modelo HSV es un modelo espacial de tres componentes, como el RGB, pero en lugar de ser cartesiano es cilíndrico. Sus componentes principales son:

- **H**ue: El matiz, tono o croma del color. Relacionado con la longitud de onda. En el espacio HSV corresponde a la base del cilindro, por lo que sus valores varían en un rango circular.
- **S**aturation: La saturación del color o cómo de puro es. A un nivel máximo de saturación se tiene el color puro y, conforme este nivel baja, se va haciendo más pastel, hasta llegar al color blanco con un nivel de saturación nulo.
- **V**alue: El valor, brillo, intensidad o luminosidad del color. A un nivel máximo de valor se tiene el color puro y, conforme este nivel baja, se va haciendo más grisáceo, hasta llegar al color negro con un nivel de valor nulo.

4.4.3 Mapeo cromático

La ventaja de disponer de diferentes espacios de color radica en que podremos pasar a trabajar entre uno u otro, aprovechando lo que nos ofrece cada uno.

Según lo visto en la Subsección 2.2.1, la matriz LED de la que disponemos trabaja en GRB, que no es otra cosa que RGB en otro orden. Por tanto, por lo general se trabajará en el espacio de color RGB.

Sin embargo, como el *hue* es al color lo que la altura al sonido (diferentes longitudes de onda), resulta interesante aprovechar esta equivalencia para realizar un mapeo y lograr un efecto de sinestesia, obteniendo un color a partir de una frecuencia de un sonido. Para ello, habría que trabajar en el espacio de color HSV, realizar el mapeo y convertir el color obtenido a RGB para enviarlo a la matriz LED.

Por ejemplo, si tenemos una banda de frecuencia definida por las frecuencias $[f_1, f_2]$ en escala mel, puede hacerse un mapeo lineal a *hue* (definido en los enteros en el rango $[0, 255]$) como el siguiente:

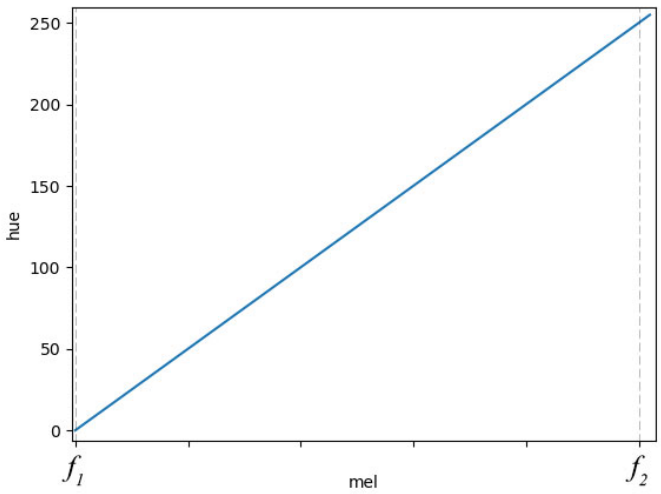

Figura 4.11 Mapeo lineal mel-hue.

Pero, ¿a qué frecuencias podría aplicarse este mapeo? Por ejemplo, según lo visto en la Subsección 4.3.1, pueden mapearse estos centroides por bandas, obteniendo así un color por banda. En esto estará basado el modo normal de funcionamiento de este proyecto.

Además, el proyecto dispondrá de otro modo, el modo Skriabin, denominado así por el conocido compositor con sinestesia y su sistema cromático que relaciona colores y notas musicales. En este modo se utilizará el *pitch*, visto en la Sección 4.2, aunque obviamente estará limitado a sonidos monofónicos.

5 Configuración del entorno

A continuación, en este capítulo se explicará cómo configurar el entorno necesario para el proyecto, incluyendo el software, librerías, la conexión y configuración de la matriz LED y la conexión de red con la Raspberry Pi.

5.1 Entorno software

En esta sección se comentará lo necesario para instalar el entorno básico de software de este proyecto, comenzando con la preparación de la Raspberry Pi.

Además de lo mencionado a continuación, se ha empleado `Sublime Text 3.1.1` para el desarrollo del código necesario.

5.1.1 Raspberry Pi

En la Raspberry Pi es necesario instalar algún sistema operativo. El más utilizado es `Raspbian`, una versión de Debian oficialmente adaptada para la Raspbery Pi, que incluye prácticamente todo lo necesario para comenzar a trabajar con ella.

Raspbian puede descargarse desde su página oficial en *https://www.raspberrypi.org/downloads/raspbian/*.

Una vez obtenido el archivo *.img* de Raspbian, lo siguiente será flashearlo (es decir, instalarlo) en la tarjeta microSD de la que disponemos. Para realizar esto existe una amplia variedad de software. En este caso, se utilizó la herramienta *open source* `balenaEtcher`, disponible en *https://www.balena.io/etcher/*.

Tras esto, puede insertarse la tarjeta microSD en la Raspberry Pi y comenzar a utilizarla.

5.1.2 Python

Python es el lenguaje de programación empleado en este trabajo. Caracterizado por ser de alto nivel, interpretado (script), dinámico, orientado a objetos y de propósito general, es un lenguaje cada vez más usado tanto en el ámbito industrial como en el científico, teniendo una gran relevancia actualmente en 2019.

Sin embargo, hay que tener en cuenta que se utilizan dos versiones que, incluso en la sintaxis, son algo incompatibles entre sí: la versión 2 y la versión 3. Aunque Raspbian trae preinstalada la versión 2 de Python, en este trabajo se optará por la versión 3.

Para las pruebas en Windows se utilizará la distribución `Anaconda3`, que es lo recomendable. En concreto, la build 3.10.5, con la versión **3.6.5** de Python, disponible en *https://www.anaconda.com/distribution/#download-section*.

Para instalar la versión 3 de Python en la Raspberry Pi (se utiliza la versión 3.5.3, ya que es la última disponible en el repositorio) habrá que ejecutar en la terminal el siguiente comando:

```
sudo apt-get install python3-dev
```

5.1.3 Librerías de Python

En el proyecto se emplean las siguientes librerías de Python. Por lo general, estas pueden instalarse mediante `pip3`, el gestor de paquetes de Python 3 o mediante `conda` si estamos en Windows.

Por ejemplo, para instalar la librería *numpy* en la Raspberry Pi:

```
pip3 install numpy
```

numpy

Esencial, bastante empleada en el ámbito científico. Permite trabajar con matrices de la misma forma que en Matlab. [9]

matplotlib

Necesaria para crear gráficas similares a las de Matlab. Compatible con *arrays* numpy. También permite crear animaciones y graficar en tiempo real. [8]

scikit-image

Algoritmos de procesamiento de imagen. En concreto, se utiliza su módulo *color* para las conversiones entre RGB y HSV. [13]

scipy

Gran variedad de algoritmos matemáticos. En el proyecto, se utiliza para el procesamiento de señales, especialmente para lo relativo a enventanado y FFT. [14]

sounddevice

Para la utilización de los dispositivos de entrada y salida de audio. Permite utilizar la entrada de audio como un *stream* para procesarlo en tiempo real como *arrays* numpy. [11]

pydub

Librerías relacionadas con audio. Se utiliza el paquete *AudioSegment* para cargar archivos MP3 en forma de señal de audio para algunas pruebas. [10]

aubio

Herramientas relacionadas con la extracción de características de audio. Se utiliza para la detección del pitch con el algoritmo YIN y para mapear los identificadores de las notas musicales de MIDI. Hay que instalarlo descargándolo de la web oficial. [3]

rpi_ws281x

Funciones para controlar la tira LED. [6].

5.2 Configuración de los LEDs

Una vez instalada la librería de `rpi_ws281x`, descrita en la sección anterior, hay que conectar la matriz LED según las siguientes figuras (realizadas con Fritzing, disponible en [5]):

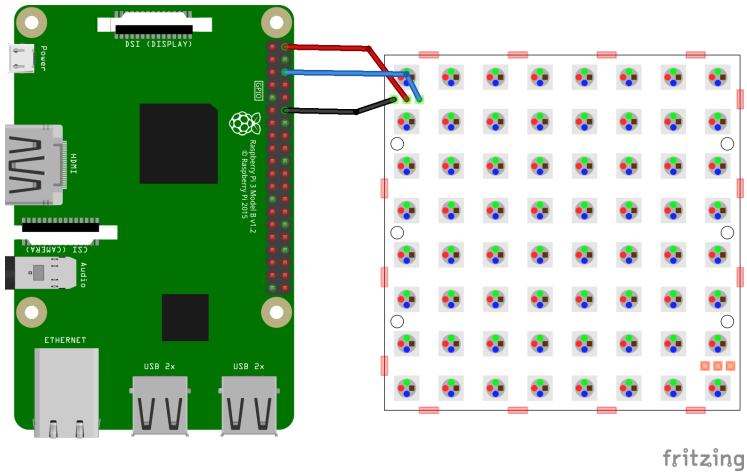

Figura 5.1 Conexión física de la Raspberry Pi con la matriz LED.

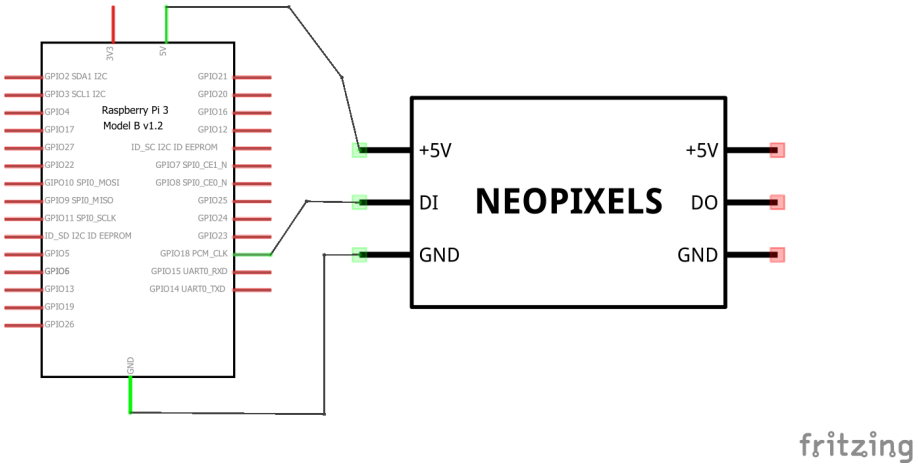

Figura 5.2 Esquemático de la conexión de la Raspberry Pi con la matriz LED.

Básicamente, hay que conectar la alimentación de 5 V y la tierra con los pines correspondientes, además de conectar la entrada de datos *DI* de la matriz LED con el pin *GPIO18* de la Raspberry Pi, como se anticipó en la Sección 2.1.

A continuación, pueden probarse los ejemplos de [7], especialmente el de `strandtest.py`. Hay que ejecutarlos con permisos *root* (es decir, utilizando el comando *sudo*), ya que son necesarios para acceder a controlar los pines de la Raspberry Pi.

```
sudo python3 strandtest.py
```

Sin embargo, la librería posee una serie de parámetros a configurar, que en el archivo mencionado vienen como:

Código 5.1 Parámetros de configuración de la librería rpi-ws281x.

```
# LED strip configuration:
LED_COUNT = 16          # Number of LED pixels.
```

```
LED_PIN = 18          # GPIO pin connected to the pixels (18
    uses PWM!).
# LED_PIN = 10        # GPIO pin connected to the pixels (10
    uses SPI /dev/spidev0.0).
LED_FREQ_HZ = 800000  # LED signal frequency in hertz (usually
    800khz)
LED_DMA = 10          # DMA channel to use for generating
    signal (try 10)
LED_BRIGHTNESS = 255  # Set to 0 for darkest and 255 for
    brightest
LED_INVERT = False    # True to invert the signal (when using
    NPN transistor level shift)
LED_CHANNEL = 0       # set to '1' for GPIOs 13, 19, 41, 45 or
    53
```

Al menos, es necesario cambiar el parámetro `LED_COUNT` a 64, ya que la matriz tiene 64 LEDs en total. Se recomienda disminuir el valor del parámetro `LED_BRIGHTNESS` debido a que, en 255 (el máximo), el brillo de la luz puede ser bastante molesto para la vista. El resto de valores de los parámetros se corresponden con lo deseado.

5.3 Conexiones de red

La Raspberry Pi puede utilizarse como un ordenador normal, conectándole teclado, ratón y pantalla. Sin embargo, resulta más interesante controlarla remotamente desde el ordenador. También, para poder tener el código en la Raspberry, se requiere transferir archivos.

Aunque la Raspberry tenga conexión WiFi, se va a emplear conexión cableada por RJ45, ya que facilita enormemente la portabilidad. Virtualmente, esta conexión corresponde en la Raspberry a la interfaz *eth0*. Se configurará automáticamente mediante DHCP, por lo que será necesario un servidor DHCP.

Si conectamos directamente la Raspberry al ordenador mediante el cable, se necesitará el servidor DHCP en el ordenador para asignarle una IP a la Raspberry. También puede conectarse directamente la Raspberry a un router al que conectemos también el ordenador (ya sea mediante WiFi o mediante cable).

Una vez todo correctamente conectado (puede comprobarse la conectividad con la Raspberry haciéndole *ping*), podemos acceder a la terminal de comandos mediante SSH. El programa recomendable para ello en Windows es `PuTTY`.

Para la transferencia de archivos empleamos SCP. El programa recomendable para ello en Windows es `WinSCP`.

6 Desarrollo e implementación

Una vez explicados los conceptos teóricos y establecida la configuración básica del entorno de trabajo, es el momento de implementarlo.

Comenzaremos con el planteamiento del sistema, donde se explicará su funcionamiento general y los parámetros básicos de configuración. Después, continuaremos con una explicación detallada sobre el código del proyecto, cuyo orden estará basado en el flujo que experimentan los datos por el sistema.

Se recomienda un entendimiento mínimo de la librería *numpy*, pues es fundamental para comprender todo el código.

Todo lo que se explicará a continuación se implementará en la Raspberry, ejecutando comandos o código mediante SSH y transfiriendo archivos mediante SCP, como se comentó en la Sección 5.3.

6.1 Planteamiento del sistema

En primer lugar, hay que comentar que, en función de la salida de colores a obtener, se han desarrollado dos modos de funcionamiento principales de este sistema:

- **Modo normal o *default*:** Es el modo por defecto. Se asigna un color por cada columna de la matriz LED en función de los centroides por bandas.
- **Modo *pitch* o Scriabin:** Mediante la estimación del *pitch*, trata de emular un efecto de sinestesia haciendo un mapeo cromático. Todas las columnas de la matriz muestran el mismo color. Ideado para sonidos monofónicos.

La forma de seleccionar el modo será cambiando el valor del parámetro *FX*. Para este parámetro y para el resto, dispondremos de un archivo de configuración llamado `config_lib.py`, cuyo contenido es el siguiente:

Código 6.1 config_lib.py - Parámetros de configuración del sistema.

```
# Configuraciones del proyecto
FX = "default"            # Tipo de efecto deseado: "default" =
    centroides; "pitch" = modo Scriabin
P_TOL = 0.7               # Tolerancia para la estimación del
    pitch

# Configuracion del audio
WIN_LENGTH = 500          # Tamaño de la ventana en muestras
```

```
WIN_TYPE = "hamming"      # Tipo de ventana a utilizar de Scipy.
FFT_SIZE = 512            # Tamaño de la FFT en muestras
MIN_DB = -20              # Mínimo dB de potencia por banda a
    tener en cuenta
MAX_DB = 100              # Mínimo dB de potencia por banda a
    tener en cuenta
MIN_FREQ = 20             # Mínima frecuencia (Hz) de los
    filtros
MAX_FREQ = 10000          # Máxima frecuencia (Hz) de los
    filtros
DOWNSAMPLE = 1            # Tamaño del downsample

# Configuracion de la tira LED
LED_COUNT      = 64       # Number of LED pixels.
LED_PIN        = 18       # GPIO pin connected to the pixels (18
     uses PWM!).
LED_FREQ_HZ    = 800000   # LED signal frequency in hertz (
    usually 800khz)
LED_DMA        = 10       # DMA channel to use for generating
    signal (try 10)
LED_BRIGHTNESS = 100      # Set to 0 for darkest and 255 for
    brightest
LED_INVERT     = False    # True to invert the signal (when
    using NPN transistor level shift)
LED_CHANNEL    = 0        # set to '1' for GPIOs 13, 19, 41, 45
    or 53
```

Además de los parámetros específicos del proyecto, en este archivo de configuración se han incluido los necesarios para configurar la tira LED, referenciados en el Código 5.1.

Aunque en el código se incluyen comentarios explicando qué es cada parámetro del proyecto y relativo al audio, haremos hincapié en ellos:

- **FX**: Selección del modo a emplear. Admite los valores *"default"* o *"pitch"*.
- **P_TOL**: Tolerancia del cálculo del valor del pitch. Es un parámetro del método del algoritmo YIN, por lo que solo se usa en el modo *pitch*.
- **WIN_LENGTH**: Tamaño en muestras de la ventana a utilizar. También es el tamaño de muestras de cada *frame* de audio. Se ha elegido el valor 500 porque en este contexto ha demostrado funcionar correctamente.
- **WIN_TYPE**: Tipo de la ventana a utilizar. Se establece la ventana *hamming*, explicada en Subsección 3.3.1. Corresponde al parámetro *window* del método *get_window* de la librería scipy. Para conocer los valores admisibles, puede consultarse la documentación en *https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.get_window.html*.
- **FFT_SIZE**: Tamaño en muestras de la FFT total calculada. Se establece como 512 por ser la potencia de 2 más cercana al tamaño de la ventana escogido, según lo visto en Subsección 3.2.5.
- **MIN_DB**: Mínima potencia por banda obtenida en dB a tener en cuenta. En el mapeo con los niveles en la matriz LED corresponde al 0. Valor establecido a -20 experimentalmente.
- **MAX_DB**: Máxima potencia por banda obtenida en dB a tener en cuenta. En el mapeo con los niveles en la matriz LED corresponde al 8. Valor establecido a 100 experimentalmente.

- **MIN_FREQ**: Mínima frecuencia en Hz a tener en cuenta. Se corresponde con la primera frecuencia de corte inferior de los filtros Mel.
- **MIN_FREQ**: Máxima frecuencia en Hz a tener en cuenta. Se corresponde con la última frecuencia de corte superior de los filtros Mel.

6.1.1 Funcionamiento general

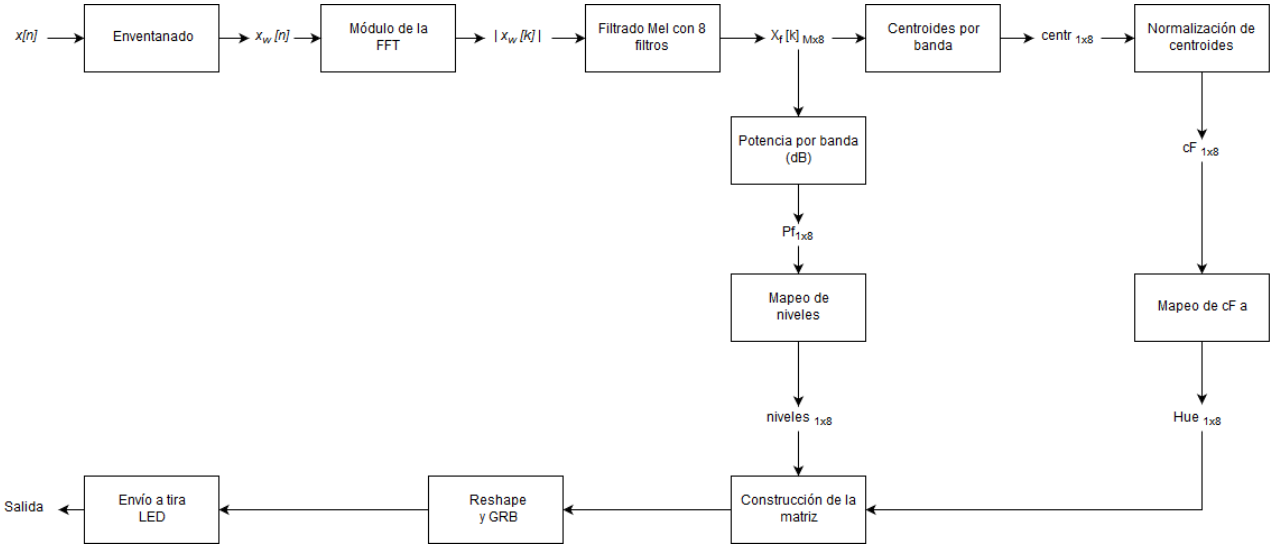

Figura 6.1 Diagrama de bloques simplificado del sistema en modo *default*.

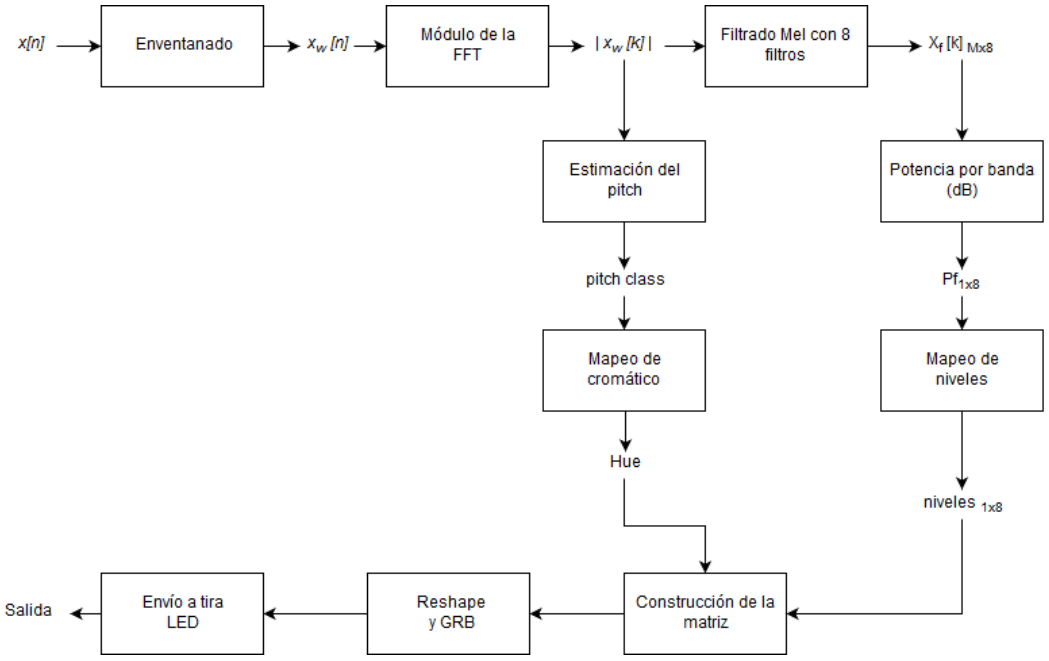

Figura 6.2 Diagrama de bloques simplificado del sistema en modo *pitch*.

En la Figura 6.1 se representa el diagrama de bloques simplificado del sistema en modo *default*, mientras que en la Figura 6.2 la del modo *pitch*. Sin embargo, en la práctica el sistema recorre todo

el flujo del modo *default*, siendo la diferencia fundamental la forma en que se mapea el color (hue) en la matriz final a mostrar.

En términos generales, el sistema hace lo siguiente:

1. Llegan datos de audio de entrada. Cuando son suficientes muestras (es decir, cuya cantidad sea mayor o igual a la ventana), se continúa procesando el *frame*. Si no, se espera hasta acumular más datos.
2. Si el *frame* tuviese más muestras que el tamaño de la ventana, se recorta hasta que coincidan. Posteriormente, se enventana.
3. Se calcula la FFT del *frame* enventanado. Nos quedamos con el módulo, desechando la fase, y con la parte positiva ($f \geq 0$). A este espectro resultante le denominaremos $|X_W[k]|$, que será un *array* de M muestras.
4. Se construyen 8 filtros mel equiespaciados (en escala mel) para el rango de frecuencias definido en la configuración.
5. Se filtra el espectro por cada filtro, obteniendo una matriz $X_f[k]_{Mx8}$. Cada columna corresponde al espectro filtrado por el filtro correspondiente, es decir, corresponde a una banda de frecuencia.
6. Por un lado, a partir de $X_f[k]_{Mx8}$, se calculan los centroides relativos a cada banda, normalizándolos según la frecuencia central del filtro correspondiente y pasándolos a escala mel. A lo resultante le denominaremos cF_{1x8}.
7. Por otro lado, se calcula la potencia en dB medida en cada banda a partir de $X_f[k]_{Mx8}$. Le denominaremos Pf_{1x8}.
8. Se mapea Pf_{1x8} con los niveles mediante una relación lineal y teniendo en cuenta el rango de dB definido en la configuración. Los niveles son cuántos cuadros deben encenderse en cada columna de la matriz LED, por lo que el valor mínimo de dB se correspondería con el nivel 0 (ningún cuadro de la columna encendido) y el nivel máximo con el nivel 8 (todos los cuadros de la columna encendidos).
9. En el modo *default*, se mapean los centroides normalizados cF_{1x8} para cada banda con el *hue* del sistema de color HSV, según lo visto en la Subsección 4.4.2 y en la Subsección 4.4.3. La forma concreta en la que lo haremos se detallará posteriormente. En el modo *pitch* se estima la frecuencia fundamental del *frame*, obteniéndose el *pitch class* y mapeándose con el *hue* mediante el círculo de quintas. Se detallará posteriormente.
10. Teniendo ya los niveles y el *hue*, se construye la matriz $8x8$ a mostrar, en el espacio de color HSV. Se convierte al espacio de color RGB.
11. Se hace un reescalado de la matriz resultante (la librería de la tira LED trabaja con secuencias de datos, no matrices) y se envía en orden GRB.

6.2 Mostrar matrices en el LED

Si bien en el flujo del sistema este es el último paso, conviene conocer cómo se comunica la Raspberry con la matriz LED antes de continuar con la implementación del sistema en sí.

En primer lugar, debemos importar la librería `rpi_ws281x` e inicializar el objeto *Adafruit_NeoPixel*, que será nuestro controlador de la matriz LED.

También importaremos el archivo de configuración definido en Código 6.1, ya que contiene los parámetros de configuración del constructor del *Adafruit_NeoPixel*. Una vez creado el controlador, utilizamos su método *begin()* para inicializar el control de la matriz LED.

Por tanto, el código sería el siguiente:

Código 6.2 Inicialización del control de la matriz LED.

```
from neopixel import Adafruit_NeoPixel  # Importa el
    controlador de la matriz LED
import config_lib as conf # Importa el archivo de configuración

# Inicialización del controlador/handler con todos los
    parámetros configurados
strip = Adafruit_NeoPixel(conf.LED_COUNT, conf.LED_PIN, conf.
    LED_FREQ_HZ, conf.LED_DMA, conf.LED_INVERT, conf.
    LED_BRIGHTNESS, conf.LED_CHANNEL)
# Inicializa la tira
strip.begin()
```

Después, definiremos un método para mostrar matrices *numpy* de 8*x*8 dimensiones en la matriz LED, ya que los métodos disponibles no permiten hacer esto directamente:

Código 6.3 Método *showMat()*.

```
# Muestra la matriz 8x8 dada como parametro en la tira LED con
    el handler strip.
def showMat(strip, mat):
  # Transforma las matrices en vectores
  a_R = np.reshape(mat[:, :, 0], mat[:, :, 0].size)
  a_G = np.reshape(mat[:, :, 1], mat[:, :, 1].size)
  a_B = np.reshape(mat[:, :, 2], mat[:, :, 2].size)
  # Debe enviarse en orden GRB
  for k in range(strip.numPixels()):
    strip.setPixelColor(k, Color(int(a_G[k]), int(a_R[k]), int(
        a_B[k])))
  strip.show()
```

La matriz *mat* posee 3 dimensiones: altura, anchura y canal de color. Por ejemplo, haciendo *mat[:, :, 0]* estamos extrayendo el canal R de la imagen, ya que van en orden RGB y es el primero. Mediante el método *reshape()* de la librería *numpy*, transformamos la matriz en un vector.

Todo esto se hace porque el método *setPixelColor()* del controlador de la tira requiere el envío de los canales de color por separado, en orden GRB y pixel a pixel. Debido a esto, es más sencillo recorrer un *array* mediante el bucle *for* e ir enviando cada píxel que hacerlo sobre una matriz.

Cuando se hayan enviado todos los píxeles de la matriz, llamamos al método *show* del controlador para que se muestre en la matriz LED.

Este código definido en el Código 6.3 irá dentro de un nuevo archivo llamado `dsp_lib.py`. En este archivo irán todos los métodos adicionales del programa, aunque no todos sean de procesado digital de señales (DSP), sí la mayoría.

6.3 Obtención de los datos de audio

En esta sección comenzaremos a explicar cómo implementar el sistema. Lo primero de todo es la obtención de los datos de audio que queremos procesar.

Como se explicó en la Subsección 2.3.3, tenemos una tarjeta de sonido USB con entrada y salida de audio. Utilizaremos la librería `sounddevice` para gestionar estos datos. Sin embargo, por cuestiones de practicidad, solamente se va a utilizar la entrada de audio, por lo que para poder escuchar lo que está llegando a esta, necesitaremos un cable *minijack* doble, tal y como adelantamos.

6.3.1 Identificación y configuración del dispositivo de audio

La librería `sounddevice` nos permite obtener en Python en tiempo real la señal de entrada de audio. Para ello, en primer lugar tenemos que configurarla con el dispositivo deseado. La librería incluye una función `query_devices()`, que nos proporciona los dispositivos de audio disponibles. A continuación, un ejemplo:

Código 6.4 Obtención de los dispositivos de audio mediante *sounddevice*.

```
pi@raspberrypi:~ $ sudo python3
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sounddevice as sd
>>> sd.query\_devices()
  0 bcm2835 ALSA: - (hw:0,0), ALSA (0 in, 2 out)
  1 bcm2835 ALSA: IEC958/HDMI (hw:0,1), ALSA (0 in, 2 out)
> 2 USB Audio Device: - (hw:1,0), ALSA (1 in, 2 out)
  3 sysdefault, ALSA (0 in, 128 out)
< 4 default, ALSA (0 in, 128 out)
  5 dmix, ALSA (0 in, 2 out)
>>>
```

En este caso, el dispositivo deseado tiene el identificador 2, información que nos será necesaria. Podemos comprobar que posee una entrada (*1 in*) y dos salidas, que en realidad es una porque es estéreo. Por tanto, la entrada de audio que tendremos será mono, así que nos ahorramos la conversión para procesarla cómodamente.

Además, mediante la información *(hw:0,1)* se nos indica que el canal de entrada es el 1.

6.3.2 Cola de datos

Para poder tratar de una forma correcta el flujo de datos que nos llega, acumularlos y agruparlos para poder montar un *array*, podemos utilizar una cola. Python ya posee de serie un módulo que nos permite utilizar colas, el módulo `queue`, cuyo uso es bastante sencillo.

Aunque posee diversos métodos y configuraciones, simplemente la vamos a utilizar para acumular datos y sacarlos. Su documentación está disponible en [12].

Un ejemplo de utilización básico de este módulo es el siguiente:

Código 6.5 Ejemplo de uso del módulo *queue*.

```
import queue # Importa la librería queue
```

```
import numpy as np # Importa numpy para manejar arrays

q = queue.Queue() # Inicializa el objeto de tipo Queue
datos = np.arange(0, 10) # Crea un array secuencia de los
    números del 0 al 9
q.put(datos) # Añade los datos a la cola
datosNuevos = q.get_nowait() # Extrae los datos de la cola
print(datosNuevos) # Imprimimos por consola lo que se ha
    extraido
```

Si ejecutamos el código, debería sacar por pantalla [0 1 2 3 4 5 6 7 8 9], que es precisamente el *array* que hemos guardado en la cola. Sin embargo, si volvemos a ejecutar *q.get_nowait()*, ocurrirá lo siguiente:

Código 6.6 Excepción de cola vacía.

```
>>> datosNuevos = q.get_nowait()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.5/queue.py", line 192, in get_nowait
    return self.get(block=False)
  File "/usr/lib/python3.5/queue.py", line 161, in get
    raise Empty
queue.Empty
```

Python lanza una excepción *queue.Empty*, ya que la cola se ha vaciado. Cuando extraemos datos, estos datos salen de la cola, por lo que podemos controlar esta excepción para saber cuándo la cola está vacía o contiene datos. Debido a esto, se recomienda extraer los datos de la cola dentro de un bloque *try* acompañado de un bloque *except queue.Empty:*. De todas formas, volveremos a esto más adelante.

6.3.3 Stream de datos

En adición a lo anterior, explicaremos cómo iniciar el flujo de datos y añadirlos a la cola. Para ello utilizaremos el método *InputStream* de la librería `sounddevice`. Este método requiere los siguientes parámetros:

- **device**: El dispositivo de entrada a utilizar. En este caso, corresponde al 2 que obtuvimos en la Subsección 6.3.1.
- **channels**: Array con los canales a utilizar. En este caso, corresponde al 1 que obtuvimos en la Subsección 6.3.1, aunque deberemos pasárselo como [1].
- **samplerate**: Frecuencia de muestreo de los datos de audio.
- **callback**: Método que se ejecuta en paralelo cada vez que se reciben datos. Será el lugar donde añadamos los datos a la cola y debemos definirlo aparte. La documentación sobre esto está disponible en [2].

6.3.4 Programa principal

Juntando todo lo anterior, podemos comenzar a escribir la lógica del programa principal, que se hará en el archivo `main.py`:

Código 6.7 main.py - Programa principal.

```
import numpy as np, sounddevice as sd, queue
from neopixel import Adafruit_NeoPixel
from dsp_lib import update
import config_lib as conf

# Programa principal
if __name__ == '__main__':
  # Creación del handler de la tira LED
  strip = Adafruit_NeoPixel(conf.LED_COUNT, conf.LED_PIN, conf.
     LED_FREQ_HZ, conf.LED_DMA, conf.LED_INVERT, conf.
     LED_BRIGHTNESS, conf.LED_CHANNEL)
  # Inicialización de la tira
  strip.begin()
  # Selección del dispositivo con identificador 2
  device = 2
  # Obtención de la información del dispositivo deseado
  device_info = sd.query_devices(device, 'input')
  # Obtención de la frecuencia de muestreo de la señal de
     entrada del dispositivos
  fs = device_info['default_samplerate']
  # Especificación de los canales de audio
  channels = [1]
  # Inicialización de la cola de datos
  q = queue.Queue()

  # Callback a ejecutar para el flujo de datos
  def audio_callback(indata, frames, time, status):
    # Se define la cola q como global para acceder a ella desde
       este método
    global q
    # Añade los datos de entrada a la cola teniendo en cuenta
       el downsample
    q.put(indata[::conf.DOWNSAMPLE])

  # Mapeo de los datos de entrada de audio al stream
  stream = sd.InputStream(device=device, channels=max(channels)
     ,
  samplerate=fs, callback=audio_callback)
  with stream:
  # Cuando se inicie el stream:
    while True:
      update(strip, fs, q)
```

Esencialmente, se junta todo lo que se ha explicado anteriormente. La frecuencia de muestreo se obtiene de la forma que aparece en el código, mediante el método *query_devices()* y los parámetros indicados obtenemos la información del dispositivo con identificador 2. Ese método devuelve un objeto del cual nos interesa la propiedad *default_samplerate*, correspondiente a la frecuencia de muestreo.

Tras la inicialización del *stream*, se llama en bucle al método *update()* definido en el archivo `dsp_lib.py`, del que hablaremos más adelante en la próxima sección.

6.4 Procesamiento de los datos

Ahora implementaremos los métodos para procesar la señal. Todos estos métodos se definirán en el archivo `dsp_lib.py`. Debido a que Python es un lenguaje de *scripting*, los métodos deben definirse en el archivo antes de la línea donde se llaman, así que el orden de aparición en esta explicación no tiene por qué coincidir con la del archivo.

En primer lugar, debemos importar las librerías a utilizar:

Código 6.8 Librerías a importar para dsp_lib.py.

```
import numpy as np
import config_lib as conf
import queue
from scipy.signal import get_window
from scipy.fftpack import fft
from neopixel import *
import skimage.color as sc
from aubio import pitch, midi2note
```

El método *update()* mencionado en la sección anterior es el encargado de llamar al procesamiento de datos. Recibirá el controlador de la tira *strip*, la frecuencia de muestreo *fs* y la cola *q*.

Código 6.9 Método update().

```
def update(strip, fs, q, review=False, data_old=[]):
  # Se inicializa el hue
  prevHue = 0
  while True:
    try:
      # Extracción de los datos de la cola
      data = q.get_nowait()
      # No hubo suficientes datos para rellenar un frame
      if review:
        # Se añaden los datos nuevos a los anteriores
        data = np.concatenate((data, data_old))
    # Si la cola esta vacia
    except queue.Empty:
      break
    # Se llama a procesar
    [procesado, hue] = procesar(data, strip, fs, prevHue)
    if not procesado:
      # No hubo suficientes datos para rellenar un frame
      # Se llama recursivamente teniendo en cuenta los datos
         anteriores
      update(strip, fs, q, review=True, data_old=data)
```

```
    else:
      # Se almacena el hue para la siguiente iteración
      prevHue = hue
```

Básicamente es el método encargado de ir extrayendo los datos de la cola. En ocasiones, a la cola pueden no llegar los suficientes datos para construir un *frame*, por lo que el programa tiene que esperar hasta que los haya. Para esto, se utilizan los argumentos opcionales del método *review* y *data_old*.

El método, tras inicializar la variable *prevHue*, sobre el que hablaremos posteriormente, realiza un bucle infinito donde va leyendo los datos de la cola y almacenándolos en la variable *data*. También controla la excepción de la cola vacía, cortando el bucle de lectura si esta se produce.

El método *procesar()*, que definiremos después, será el siguiente encargado del procesamiento de los datos. Devolverá un booleano *procesado*, que vendrá como *false* si no hubieron suficientes datos, y el *hue* calculado.

Si *procesado* tiene el valor *false*, llamaremos recursivamente al mismo método, activando el booleano *review* para que sepa que se está llamando recursivamente, y pasándole los datos extraidos en esa iteración. En esa nueva iteración se concatenan los datos anteriores, obteniendo así más datos, quizás suficientes esta vez para montar el *frame* y procesarlo.

Si *procesado* tiene el valor *true*, se almacena el *hue* obtenido en *prevHue*, dándonos la posibilidad de conservar el Hue anteriormente calculado si algo sale mal.

6.4.1 Procesando los datos

El método *procesar()* continúa con el procesamiento de los datos. Se comprueba que el parámetro de entrada *data* tiene las suficientes muestras como para construir el *frame*, es decir, la cantidad sea mayor o igual al tamaño de la ventana. En este caso se devuelve *false*, como vimos anteriormente.

Código 6.10 Método procesar().

```
# Procesa los datos de entrada
def procesar(data, strip, fs, prevHue):
  # Si se han recibido suficientes datos para rellenar un frame
  if len(data) >= conf.WIN_LENGTH:
    # Obtencion de la ventana
    w = get_window(conf.WIN_TYPE, conf.WIN_LENGTH, False)
    # Enventanado y preparacion del frame
    xs = np.multiply(np.reshape(data[:conf.WIN_LENGTH], conf.
       WIN_LENGTH), w)
    # Calculo del modulo de la DFT mediante la FFT
    xf = np.abs(fft(xs, conf.FFT_SIZE, axis=0))
    # Lado positivo (f > 0) en la primera mitad del array
    xf = xf[:int( len(xf) / 2) + 1]
      # Obtencion de la matriz y del hue calculado
    [mat, hue] = get_mat(fs, xf, xs, conf.FX, prevHue)
    # Muestra el resultado
    showMat(strip, mat)
    return True, hue
  else:
    # Notifica que no se han recibido suficientes datos. Seguir
        acumulando.
```

```
    return False, 0
```

Mediante el método *get_window()* de la librería `scipy.signal` se obtiene la ventana configurada. El parámetro pasado como *False* es para que devuelva una sola ventana simétrica en lugar de periódica.

Luego, según lo visto en la Subsección 3.3.2, se enventanan y recortan los datos recibidos, teniendo así un *frame xs* del tamaño de la ventana. Posteriormente, se calcula la DFT utilizando el método *fft()* de la librería `scipy.fftpack` con el tamaño de FFT configurado (512 en este caso) y se le halla la magnitud mediante el método *abs()* de la librería `numpy`.

Según lo visto en la Sección 3.2, al ser *xs* una señal real, la DFT resultante tendrá simetría par, así que nos quedamos solo con las frecuencias positivas ($f \geq 0$). Como el método devuelve la DFT en un array de la forma $[xf(f > 0)\ xf(0)\ xf(f < 0)]$, nos quedamos con la primera mitad del array (incluyendo el valor para la frecuencia 0).

A continuación, se llama al siguiente método *get_mat()*, pasándose la frecuencia de muestreo *fs*, la magnitud de las frecuencias positivas de la DFT *xf* (*array* de tamaño $1x257$, es decir, la mitad del tamaño de la FFT más uno), el *frame xs*, el modo configurado *conf.FX* y el *prevHue* (para propagarlo al siguiente método).

Esto último devuelve ya la matriz resultante *mat* y el *hue* calculado, el cual podrá ser un escalar (en el modo *pitch*) o un vector (en el modo *default*). Este *hue* se devuelve, para que el método *procesar()* visto anteriormente pueda recibirlo.

La matriz resultante *mat* se envía al método *show_mat()*, descrito en el Código 6.3, junto al controlador de la tira *strip*.

Luego, se implementa el método *get_mat()*. En primer lugar, se tendrá la llamada al siguiente método *get_melfilts()*, que recibirá como parámetros la frecuencia de muestreo *fs*, el número de filtros *nfilt* (que le pasaremos el valor 8) y la magnitud de las frecuencias positivas de la DFT *xf*.

Código 6.11 Método get_mat() - Parte 1.

```
# Obtiene la matriz a mostrar en cada iteración
def get_mat(fs, xf, xs, fx="default", prevHue=0):
  # Filtrado mel y centroides
  pf, cF, cFmin, cFmax = get_melfilts(fs, 8, xf)
```

Obviamente el método *get_mat()* no se acaba ahí, por lo que continuaremos con la explicación de este método más adelante.

6.4.2 Filtrado por bancos mel

Antes de proseguir con el filtrado en sí, desarrollaremos un par de métodos que harán más legible y estructurado el código.

Se trata de los métodos *mel2hz()* y *hz2mel()* que, implementando las fórmulas vistas en la Subsección 4.1.1, se encargan de la conversión entre escala Mel y Hertzios.

Código 6.12 Métodos de conversión entre Mel y Hz.

```
# Convierte la frecuencia de entrada de Mel a Hertzios
def mel2hz(f):
  return (700 * (10**(f / 2595) - 1))
```

```
# Convierte la frecuencia de entrada de Hertzios a Mel
def hz2mel(f):
  return (2595 * np.log10(1 + f / 700))
```

Basándonos en la teoría vista en la Subsección 4.1.2, procedemos a construir los filtros mel en el método *get_melfilts()*, ya mencionado en la sección anterior.

A partir del rango de frecuencias definido en la configuración en los parámetros *MIN_FREQ* y *MAX_FREQ*, convertidos a la escala mel, se genera un vector de puntos de frecuencias mel equiespaciados de dimensión 10, debido a que se necesitan $nfilt + 2$ puntos.

Una vez obtenidos los puntos equiespaciados en mel, se pasan a Hz y se obtienen los bins de frecuencia (es decir, los puntos de frecuencia en muestras). Con esto, se construyen los filtros según la teoría. Serán filtros triangulares paso banda con solape y cuyo ancho de banda será creciente. Nos quedará una matriz *fbank* con 8 columnas, una para cada filtro, y $FFT\_SIZE/2 + 1$ filas, el tamaño de *xf*.

Código 6.13 Método get_melfilts().

```
# Filtra la magnitud de la DFT con nfilt filtros mel y devuelve
    la potencia por banda filtrada en dB.
# Tambien devuelve los centroides normalizados y su rango,
   necesarios para el mapeo a Hue.
def get_melfilts(fs, nfilt, xf):
  # Definicion del rango de frecuencias en mel
  low_freq_mel = hz2mel(conf.MIN_FREQ)
  high_freq_mel = hz2mel(conf.MAX_FREQ)
  # Puntos equiespaciados en escala mel
  mel_points = np.linspace(low_freq_mel, high_freq_mel, nfilt +
      2)
  # Conversión de dichos puntos a Hertzios
  hz_points = mel2hz(mel_points)
  # Calculo de los bins de frecuencia
  bins = np.floor(hz_points * len(xf) * 2 / fs)
  bin_vec = np.arange(0, len(xf))
  # Construccion de los filtros
  fbank = np.zeros((nfilt, int(conf.FFT_SIZE / 2 + 1)))
  for m in range(1, nfilt + 1):
    # Se construyen los puntos de frecuencia (bins) de cada
       filtro
    f_m_minus = int(bins[m - 1])   # lado izquierdo
    f_m = int(bins[m])             # centro
    f_m_plus = int(bins[m + 1])    # lado derecho

    for k in range(f_m_minus, f_m):
      fbank[m - 1, k] = (k - bins[m - 1]) / (bins[m] - bins[m -
          1])
    for k in range(f_m, f_m_plus):
      fbank[m - 1, k] = (bins[m + 1] - k) / (bins[m + 1] - bins
         [m])
  # Aplicacion del filtro
```

```
xf_filt = fbank*xf*xf.T
# Calculo de centroides (en bins)
centr = np.divide(np.dot(xf_filt, bin_vec), np.sum(xf_filt,
   axis=1))
# Conversión a Hertzios y posteriormente a Mel
centr = hz2mel(centr * conf.MAX_FREQ / bins[-1])
# Obtencion de los cF: centroides normalizados y su rango
cFmin, cFmax, cF = obtienecF(centr, mel_points, nfilt)
# Cálculo de la potencia
xf_filt = xf_filt ** 2 // xf_filt.shape[1]
# Sustitución del 0 por eps por estabilidad numérica
xf_filt = np.where(xf_filt == 0, np.finfo(float).eps, xf_filt
   )
# Conversión a dB
p_dB = 10 * np.log10(np.sum(xf_filt, axis=1))
print("=================================================")
print("Niveles de potencia por banda (dB):\n" + str(p_dB))
return p_dB, cF, cFmin, cFmax
```

Después, se filtra *xf* con los filtros generados. Se hace mediante operaciones matriciales de tal forma que el resultado *xf_filt* tenga las mismas dimensiones que la matriz *fbank* (8x257), separando por columnas las bandas.

Tras esto, obtenemos los centroides por cada banda según vimos en la Sección 4.3, cuyo resultado será un vector *centr* de 8 elementos, con los centroides por cada banda. Estos centroides estarán en muestras, de ahí que luego se convierta a Hz (mediante regla de 3) y esto a Mel. Lo relativo al método *obtienecF()* se explicará en la siguiente sección.

Una vez obtenida la matriz *xf_filt*, elevamos cada elemento al cuadrado y dividimos entre el número de muestras por banda (en este caso, 257) para obtener la potencia.

Antes de sumar las muestras por banda para obtener la potencia total y convertirlo a decibelios, realizamos una sustitución de los valores 0 por *eps* (el valor float más pequeño) para evitar problemas de estabilidad numérica en el cálculo del logaritmo.

Una vez obtenida la potencia por banda en dB, almacenada en el vector de 8 elementos *p_dB*, imprimimos por consola los valores obtenidos. Se devuelve este último vector y los centroides normalizados y su rango, explicados a continuación.

6.4.3 Cálculo de los centroides normalizados

A continuación, se definen los métodos encargados de la obtención de la normalización de los centroides. El método que realmente se utiliza es el de *obtienecF()*, el cual ya se encarga de llamar a los otros. Este método devuelve los vectores *cF_min*, *cF_max* y *cF*, en ese orden.

Los argumentos de entrada son:

- **centr**: Vector 1x8 de centroides en escala mel calculados para cada banda.
- **mel**: Vector 1x10 de los puntos de frecuencia en escala mel donde se han definido los filtros.
- **nfilt**: Número de filtros, 8 en este caso.

Código 6.14 Obtención de los centroides normalizados y su rango.

```
# Divide el punto mel 'k-1' por el punto mel 'k'.
```

```
# Devuelve el array de los centroides normalizados minimos para
    cada 'k'.
def obtieneCfMin(mel, nfilt):
  return np.divide(mel, np.roll(mel, -1))[:nfilt]

# Divide el punto mel 'k+1' por el punto mel 'k'.
# Devuelve el array de los centroides normalizados maximos para
    cada 'k'.
def obtieneCfMax(mel, nfilt):
  return np.divide(np.roll(mel, -2), np.roll(mel, -1))[:nfilt]

# Normaliza cada centroide 'k' en mel dividiendolo por la
   frecuencia central en mel del filtro 'k'
def normalizaCentroides(centr, mel):
  return np.divide(centr, np.roll(mel, -1)[:len(centr)])

# Obtiene los centroides normalizados junto a sus rangos.
   Necesario para construir el mapeo.
# Devuelve cFmin, cFMax, cF
def obtienecF(centr, mel, nfilt):
  return obtieneCfMin(mel, nfilt), obtieneCfMax(mel, nfilt),
     normalizaCentroides(centr, mel)
```

La idea es dividir cada centroide k por la frecuencia central del filtro k, normalizándolo y obteniéndose cF. También se divide la frecuencia central del filtro k con las frecuencia centrales de los filtros $k-1$ y $k+1$, obteniéndose así *cF_min*, y *cF_max*, respectivamente.

6.4.4 Mapeo de los centroides

Conociendo los vectores *cF*, *cF_min* y *cF_max*, se puede proceder al mapeo de los centroides normalizados con el Hue.

Se hará, basándonos en la teoría vista en Subsección 4.4.3, mediante la siguiente relación:

$$Hue \equiv \begin{cases} \frac{255}{1-cF_{min}}(1-cF) & si \quad cF < 1 \\ \\ \frac{255}{cF_{max}-1}(cF-1) & si \quad cF \geq 1 \end{cases} \tag{6.1}$$

En código, será el método *mapeaCf2Hue()*, definido como:

Código 6.15 mapeaCf2Hue() - Mapeo de centroides normalizados a Hue.

```
# Realiza el mapeo de centroides normalizados en rango [cFmin,
   cFmax] a Hue [0, 255]
def mapeaCf2Hue(cF, cFmin, cFmax, prevHue):
  # Inicializacion del array de Hue
  hue = np.zeros(cF.shape)
  for k in range(0, len(cF)):
    if cF[k] < 1:
      hue[k] = int(np.round(255 * (1 - cF[k]) // (1 - cFmin[k])
         ))
    elif cF[k] >= 1:
```

```
      hue[k] = int(np.round(255 * (cF[k] - 1) // (cFmax[k] - 1)
          ))
    else:
      hue[k] = prevHue[k]
  print("Centroides normalizados cF: " + str(cF))
  print("Hues obtenidos: " + str(hue))
  return hue
```

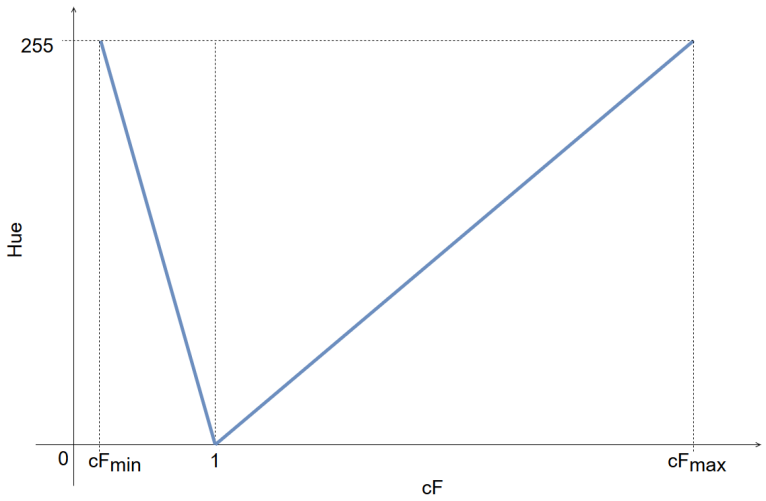

Figura 6.3 Mapeo del Hue con los centroides normalizados.

Lo que se intenta representar con esta relación es cuánto se aleja el centroide de cada banda filtrada respecto a la frecuencia central del filtro correspondiente. Así, cuando $cF = 1$, el centroide coincide con su frecuencia central, asignándole el valor del Hue a 0 (rojo).

Si el centroide se va alejando de ese valor, se le asigna linealmente el valor correspondiente, hasta llegar a cF_{max} o a cF_{min}. En esos valores, el centroide coincide con la frecuencia central de alguno de los filtros adyacentes donde, en teoría, no debería estar el centroide, de ahí que conformen el rango. Para esos valores, se obtiene un Hue de 255 que, ya que es un valor circular, vuelve a ser el rojo.

6.4.5 Mapeo de niveles

El método a continuación se utilizará para mapear los niveles de potencia en dB y los niveles de la matriz (es decir, cuántos LED deben encenderse por columna). Mediante una relación lineal, utilizando los parámetros de configuración *MIN_DB* y *MAX_DB*, se calculan los niveles de la matriz. Estos niveles obviamente son números enteros, de ahí el redondeo.

Código 6.16 Método convertLevels().

```
# Convierte los niveles dB a la escala entera [0, 8]
def convertLevels(pf):
  # Mapeo de dB minimo configurado a 0 y dB maximo configurado
     al 8
  lvl=np.round(8 * (pf - conf.MIN_DB) / (conf.MAX_DB - conf.
     MIN_DB))
  # Control de saturación y de rango dinámico
```

```
  lvl[np.where(lvl > 8)] = 8
  lvl[np.where(lvl < 0)] = 0
  return lvl
```

Debido a que pueden venir niveles de potencia menores o mayores al rango configurado, se ajusta una saturación donde los valores superiores a 8 se queden en 8 y los menores a 0, en 0. Finalmente se devuelve el vector *lvl* de 8 elementos, indicando cuántos LED encender por columna de la matriz.

6.4.6 Obtención de la matriz en modo default

En este método será donde se haga la distinción entre el modo *pitch* y el modo *default*. Por ahora, nos centraremos en este último. Tras obtener el vector de potencia por banda en dB pf y los centroides normalizados y su rango, se llama a la función *convertLevels()*, que realizará el mapeo entre los niveles.

Como el valor *prevHue* puede venir inicializado como 0 (es un *int*), en este modo se inicializa como un vector de 8 elementos (que coincide con el *shape* de los centroides normalizados).

Se mapea el Hue a partir de los centroides utilizando el método *mapeaCf2Hue()* del Código 6.15.

Código 6.17 Método get_mat() - Parte 2.

```
# Obtiene la matriz a mostrar en cada iteración
def get_mat(fs, xf, xs, fx="default", prevHue=0):
  # Filtrado mel y centroides
  pf, cF, cFmin, cFmax = get_melfilts(fs, 8, xf)
  filt = convertLevels(pf)
  # Inicialización de la matriz
  mat = np.zeros((8, 8, 3))
  # Modo default
  if fx == "default":
    if type(prevHue) is int:
      # En este modo, prevHue debe ser un array. Se inicializa
         como tal.
      prevHue = np.zeros(cF.shape)
    hue = mapeaCf2Hue(cF, cFmin, cFmax, prevHue)
    for k in range(0, 8):
      # Value con degradado
      mat[(8-int(filt[k])):, k, 2] = np.linspace(240, 255, int(
         filt[k]))
      # Saturation con degradado
      mat[(8-int(filt[k])):, k, 1] = np.flip(np.linspace(210,
         255, int(filt[k])))
      # Hue en función de lo mapeado
      mat[(8-int(filt[k])):, k, 0] = hue[k]*np.ones([1, int(
         filt[k])])
  # Normalización y Conversión a RGB
  mat_rgb = np.floor(255*sc.hsv2rgb(mat/255.0))
  return mat_rgb, hue
```

Se construye la matriz de dimensiones $8x8x3$ en el espacio de color HSV con cada componente por separado (value, saturation y hue). Lo que hacemos es iniciarla como una matriz de ceros y

rellenar solo lo que nos interese. Para cada columna k, rellenaremos tantos píxeles como sea el valor de $filt[k]$, es decir, el nivel de la matriz calculado para esa columna.

Mediante el método *linspace()* se trató de hacer un degradado para el value y la saturation. Experimentalmente se ha comprobado que el value no varía en la matriz LED de la que disponemos. El hue se rellena con los valores calculados por el método *mapeaCf2Hue()*.

Por último, normalizamos la matriz en el rango $[0,\ 1]$ para convertirla a RGB mediante el método *hsv2rgb()* de la librería `skimage.color`. Se vuelve a convertir al rango $[0,\ 255]$ y se devuelve como *mat_rgb* junto al hue calculado.

6.5 Modo pitch o Scriabin

Todo lo visto anteriormente describe la implementación para el modo *default*, pero sigue faltando código para poder seleccionar el modo *pitch*. La diferencia radica en el método *get_mat()*, donde si se selecciona este modo, se construya la matriz de otra forma.

Justo tras el bloque `if` del Código 6.17, añadiremos un `elif` tal que:

Código 6.18 Método get_mat() - Parte 3.

```
# Modo pitch o Scriabin
elif fx == "pitch":
  # Size de cada frame
  size = int(np.ceil(conf.WIN_LENGTH//conf.DOWNSAMPLE))
  # Calculo del pitch mediante el algoritmo YIN
  pitch_o = pitch("yin", size, size, int(fs))
  pitch_o.set_tolerance(conf.P_TOL)
  pitch_o.set_unit("midi")
  # Obtencion del pitch en nota MIDI
  midi_note = int(np.round(pitch_o(np.float32(xs[:size]))[0])
     )
  if not midi_note > 127:
    # En este caso se ha obtenido un pitch correcto
    note = midi2note(midi_note)[0]
    print("Nota obtenida: " + note)
    hue = pclass2color(note2pclass(note))
  else:
    # Si no se ha obtenido un pitch correcto, se mantiene el
       anterior
    hue = prevHue
  # Calculo de la matriz
  for k in range(0,8):
    # Value
    mat[(8-int(filt[k])):, k, 2] = np.linspace(240, 255, int(
       filt[k]))
    # Saturation
    mat[(8-int(filt[k])):, k, 1] = np.flip(np.linspace(210,
       255, int(filt[k])))
    # Hue
    mat[(8-int(filt[k])):, k, 0] = hue*np.ones([1, int(filt[k
       ])])
```

Primero, se calcula el *size* del *frame* teniendo en cuenta el tamaño de la ventana y el *downsample*. Este parámetro es necesario para el método *pitch()* de la librería `aubio`. Instanciamos el objeto *pitch* con el parámetro *size* y la frecuencia de muestreo, indicándole que se calcule mediante el algoritmo YIN.

A continuación, le establecemos la tolerancia configurada en el parámetro *P_TOL* y le pedimos que nos devuelva como unidades notas MIDI. Manejaremos estas unidades por la facilidad con la que resulta obtener el *pitch class* a partir de estas. Luego, se obtiene el *pitch* en nota MIDI, la cual no puede ser mayor de 127. En caso contrario, se recupera el hue anterior *prevHue*.

Note	Octave										
	-1	0	1	2	3	4	5	6	7	8	9
C	0	12	24	36	48	**60**	72	84	96	108	120
C#	1	13	25	37	49	**61**	73	85	97	109	121
D	2	14	26	38	50	**62**	74	86	98	110	122
D#	3	15	27	39	51	**63**	75	87	99	111	123
E	4	16	28	40	52	**64**	76	88	100	112	124
F	5	17	29	41	53	**65**	77	89	101	113	125
F#	6	18	30	42	54	**66**	78	90	102	114	126
G	7	19	31	43	55	**67**	79	91	103	115	127
G#	8	20	32	44	56	**68**	80	92	104	116	
A	9	21	33	45	57	**69**	81	93	105	117	
A#	10	22	34	46	58	**70**	82	94	106	118	
B	11	23	35	47	59	**71**	83	95	107	119	

Figura 6.4 Tabla de notas MIDI.

Luego, utilizamos el método *midi2note* de la librería `aubio`, que nos devolverá el *string* de la nota asociada a esa nota MIDI. Por ejemplo, como puede verse en la Figura 6.4, el valor 64 nos devolvería 1*E*4.

Mediante los métodos *note2pclass* y *pclass2color*, implementados posteriormente, obtenemos el *pitch class* de la nota y el hue asociado, respectivamente.

Tras esto, se construye la matriz de forma análoga a la Subsección 6.4.6, con la diferencia de que todas las columnas compartirán el mismo hue.

6.5.1 Mapeo cromático

Ahora implementaremos el método *note2pclass()*, mencionado anteriormente, que simplemente elimina los números del string, quedándose con lo demás:

Código 6.19 Método note2pclass().

```
# Método para extraer el pitch class de un string nota. Ej: A4
    -> A
def note2pclass(note):
```

```
  pclass = []
  for i in note:
    if not i.isdigit():
      pclass.append(i)
  return ''.join(pclass)
```

El mapeo cromático, es decir, el convertir de *pitch class* a hue, lo hará el método *pclass2color()*. Este método será simplemente un diccionario de Python con las equivalencias entre *pitch class* y hue, y estará basado en el círculo de quintas del ámbito musical y en el sistema sinestético propuesto por Scriabin.

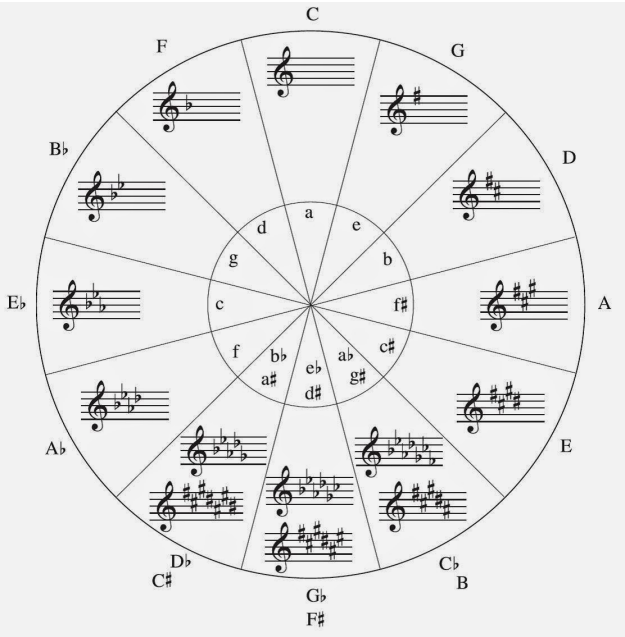

Figura 6.5 Círculo de quintas.

Código 6.20 Método pclass2color().

```
def pclass2color(note):
  # Círculo de quintas con paso de hue 22
  scriabin = {
    "C": 0,
    "G": 22,
    "D": 44,
    "A": 66,
    "E": 88,
    "B": 110,
    "F#": 132,
    "Gb": 132,
    "Db": 154,
    "C#": 154,
    "Ab": 176,
    "G#": 176,
    "Eb": 198,
```

```
    "D#": 198,
    "Bb": 220,
    "A#": 220,
    "F": 242
  }
  return scriabin[note]
```

Por cada salto en el círculo de quintas, se le añade un valor de 22 al hue. Este número se obtiene dividiendo 256 (el total de valores posibles de hue) entre 12 (el número total de *pitch classes*). Obviamente, algunos valores estarán repetidos, porque son *pitch classes* equivalentes. Por ejemplo, *G#* y *Ab*.

7 Conclusiones y propuestas de mejora

El resultado obtenido de la implementación del proyecto se ha incluído en un vídeo, compuesto por unas cuantas demostraciones del modo *default* y *pitch*. Sin embargo, no pueden apreciarse del todo bien la dinámica de las luces y sus colores, por limitaciones técnicas de la cámara e iluminación. De todas formas,

En el vídeo se emplean para el modo *default* fragmentos del ”Bolero” de Ravel y ”I Dovregubbens hall” de Edvard Grieg, ya que ambas composiciones poseen de grandes variaciones dinámicas.

Para la demostración del modo *pitch*, se ha utilizado una melodía simple y un fragmento de una interpretación del ”Clair de Lune” de Debussy al theremín por Grégoire Blanc, la cual puede verse en [1].

Si bien este último no es sonido monofónico propiamente dicho, ya que tiene instrumentación de fondo, la energía del theremín destaca sobre el resto de instrumentos, siendo en la práctica monofónico.

El vídeo de demostración está disponible en [4].

7.1 Problemas detectados

Aunque el resultado es bastante satisfactorio y ha sido avalado por todas las personas que lo han visto funcionar en directo, presenta algnas deficiencias, las cuales podrían dividirse en dos ámbitos: el hardware y el software.

El hardware empleado, especialmente la matriz LED, es bastante económico y con una calidad justa, por lo que su precisión con el color no es la mejor. Además, en ocasiones se sobrecalienta y comienza a fallar. Simplemente conectándola a la Raspberry Pi, sin ejecutar nada, se espera que se enciendan todos los LED de color blanco. Cuando se sobrecalienta, algunos LED comienzan a apagarse o a volverse de color amarillo. Se soluciona desconectándola y dejándola enfriar un tiempo.

En lo que se refiere al software, se puede ver un molesto parpadeo constante en la matriz. Se cree que es por los intervalos de tiempo en los que no hay suficientes datos en el *stream* de audio para formar un *frame*, mostrando una matriz vacía. Esto podría mejorarse reduciendo la frecuencia con la que se toman los datos y manteniendo una relación la matriz del intervalo anterior, haciendo los cambios más suaves, aunque el sistema perdería bastante instantaneidad. También, en ocasiones, el programa no logra ejecutarse a la primera y muestra un mensaje de *input overflow*. Se cree que esto es debido a que el *stream* tarda en arrancar y llegan muchos datos de golpe y la librería de *sounddevice* muestra este mensaje. No se logra discernir un método claro para reproducir este bug.

7.2 Propuestas de mejora

Además de pulir los problemas detectados de los que hemos hablado anteriormente, existen otras líneas de trabajo que podrían continuarse para mejorar todavía más este proyecto:

- Escalar el sistema, teniendo una matriz resultante de más de 8x8 LEDs, adaptándolo todo a las nuevas dimensiones.
- Implementar muchos más modos y efectos, basándose en distintos descriptores del sonido.
- Implementar una forma de iniciar y controlar el programa remotamente con una interfaz gráfica de usuario, que permitiera seleccionar y cambiar entre los distintos modos.
- Implementar un detector de sonidos monofónicos y polifónicos para cambiar el modo de forma automática.

Índice de figuras

Índice de códigos

Bibliografía

[1] *Debussy - « Clair de Lune » on the theremin*, 2013, *https://www.youtube.com/watch?v=PjnaciNT-wQ*, [Online, accedido el 24/08/2019].

[2] *Usage — python-sounddevice, version 0.3.12*, 2018, *https://python-sounddevice.readthedocs.io/en/0.3.12/usage.html#callback-streams*, [Online, accedido el 07/08/2019].

[3] *Aubio*, 2019, *https://aubio.org/*, [Online, accedido el 31/08/2019].

[4] *Demo tfg*, 2019, *https://www.youtube.com/watch?v=sTIzAAgnnxA*, [Online, accedido el 24/08/2019].

[5] *Fritzing*, 2019, *https://fritzing.org/home/*, [Online, accedido el 02/08/2019].

[6] *Github rpi ws281x python*, 2019, *https://github.com/rpi-ws281x/rpi-ws281x-python*, [Online, accedido el 31/08/2019].

[7] *Github rpi ws281x python - ejemplos*, 2019, *https://github.com/rpi-ws281x/rpi-ws281x-python/tree/master/examples*, [Online, accedido el 31/08/2019].

[8] *Matplotlib*, 2019, *https://matplotlib.org/*, [Online, accedido el 31/08/2019].

[9] *Numpy*, 2019, *https://numpy.org*, [Online, accedido el 31/08/2019].

[10] *Pydub*, 2019, *https://pydub.com/*, [Online, accedido el 31/08/2019].

[11] *Python sounddevice*, 2019, *https://python-sounddevice.readthedocs.io*, [Online, accedido el 31/08/2019].

[12] *queue — A synchronized queue class — Python 3.7.4 documentation*, 2019, *https://docs.python.org/3/library/queue.html*, [Online, accedido el 07/08/2019].

[13] *Scikit-image: Image processing in python*, 2019, *https://scikit-image.org/*, [Online, accedido el 31/08/2019].

[14] *Scipy*, 2019, *https://www.scipy.org/*, [Online, accedido el 31/08/2019].

[15] *scipy.signal.get_window — SciPy v1.3.0 Reference Guide*, 2019, *https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.get_window.html*, [Online, accedido el 07/08/2019].

[16] Phillip Burgess, *The magic of neopixels, adafruit neopixel uberguide*, 2017, *https://learn.adafruit.com/adafruit-neopixel-uberguide*, [Online, accedido el 12/01/2019].

[17] H. de Cheveigné, A. y Kawahara, *Yin, a fundamental frequency estimator for speech and music*, J Acoust Soc Am **111** (2002), 1917–1930.

[18] Haytham Fayek, *Speech processing for machine learning: Filter banks, mel-frequency cepstral coefficients (mfccs) and what's in-between*, 2016, *https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html/*, [Online, accedido el 09/06/2019].

[19] Raspberry Pi Foundation, *Raspberry pi documentation*, 2019, *https://www.raspberrypi.org/documentation/*, [Online, accedido el 12/01/2019].

[20] Paul Gries, *Practical programming: a introduction to computer science using python 3*, second edition ed., The Pragmatic Bookshelf, Dallas, Texas, 2013 (eng).

[21] Kyster, *Freesound - "a open string.wav"*, 2011, *https://freesound.org/people/Kyster/sounds/117673/*, [Online, accedido el 09/06/2019].

[22] Simon Monk, *Programar la raspberry pi con python*, Anaya, Madrid, 2018 (spa).

[23] Meinard Muller, *Fundamentals of music processing : audio, analysis, algorithms, applications*, Springer, 2016 (eng).

[24] R. W. Oppenheim, A. V. y Schafer, *Discrete-time signal processing*, Prentice Hall, 1989.

[25] J.G. Proakis and D.G. Manolakis, *Tratamiento digital de señales*, Pearson Educación, 2007.

[26] Sabrent, *Adaptador de usb externo para estéreo, amazon*, 2019, *https://www.amazon.es/gp/product/B00IRVQ0F8*, [Online, accedido el 11/01/2019].

[27] A. Spanias, T. Painter, and V. Atti, *Audio signal processing and coding*, Wiley, 2006.

[28] Worldsemi, *Intelligent control led integrated light source*, c2018.

[29] Udo Zolzer, *Digital audio signal processing*, New York, Chichester, 1998 (eng).