# Diseño e implementación en Python de un sistema de identificación de segmentos musicales

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial,
Aeroespacial i Audiovisual de Terrassa

Director: Nogueiras Rodriguez, Albino

Alumno: Hedo Garcia, Jesús

Fecha de entrega: 28/09/2020

Grado: Ingeniería de Sistemas Audiovisuales

**1. Introducción** 3
1.1 Objeto 3
1.2 Alcance 3
1.3 Requerimientos 3
1.4 Justificación y utilidad 4
1.5 Objetivos 5

**2. Herramientas para el desarrollo del Software** 6
2.1. Introducción 7
2.2. Visual Studio Code 7
2.3 Python 3 9
2.3.1 Definición de Python 9
2.3.2 Módulos requeridos: 10
2.4 Base de datos 12
2.4.1 Introducción 12
2.4.2 Descarga 12
2.4.3 Ficheros para el reconocimiento 13
2.4.4 Adición del ruido 14

**3. Desarrollo del sistema de reconocimiento musical** 14
3.1. Introducción. 15
3.2. Representación tiempo-frecuencia de los segmentos de audio. 15
3.2.1. El espectrograma. 15
3.2.2. Espectrograma de segmentos de señal. 18
3.3. Banco de filtros en escala Mel. 20
3.3.1 Escala Mel. 21
3.3.2 Banco de filtros. 23
3.4. Decorrelación y disminución de la dimensionalidad. 26
3.5.Identificación musical basada en distancia euclídea y uso de árboles binarios. 28
3.5.1. Clasificación ciega de señales usando árboles binarios. 31
3.6. Construcción de la tabla de reconocimiento. 37
3.7. Puesta en marcha del sistema 38

**4. Software Dejavu** 44
4.1 Introducción 45
4.2. Primeros pasos 45
4.3. Música como una señal 45
4.4. Muestreo 46
4.5. Espectrogramas 47
4.6. Búsqueda de picos 48
4.7. Fingerprint hashing 49

4.8. Entrenando una canción 51
4.9. Tabla Fingerprints 51
4.10. Tabla canciones 53
4.11. Alineación de los fingerprints 54
4.13 Instalación DejaVu 56

**5. Resumen de los resultados** 58
5.1 Resultados PEQE 58
5.2 Resultados DejaVu 61
5.3 Conclusiones y programación para la continuidad 62
5.4 Bibliografía 65

# 1. Introducción

## 1.1 Objeto

Desarrollo e implementación de un sistema de reconocimiento musical a través de algoritmos basados en la clasificación de los segmentos de las señales musicales en árboles binarios y toma de decisiones para reconocer los mismos.

## 1.2 Alcance

Se llevarán a cabo las siguientes tareas:

- Estudiaremos técnicas de caracterización de segmentos musicales.
- Parametrización de las señales
- Entrenamiento del sistema creando el árbol binario para la clasificación y la búsqueda de las señales parametrizadas.
- Reconocimiento de un segmento, donde el tiempo de parametrización de la señal de entrada será la clave de nuestro sistema.
- Crear ficheros de prueba con distintos niveles de calidad a través del SNR para la evaluación del sistema.
- Evaluar la precisión del mismo.
- Búsqueda de un software desarrollado dónde podamos ejecutarlo con nuestra base de datos y comparar los resultados.

## 1.3 Requerimientos

- Una computadora con las siguientes prestaciones:
  - 8 GB Ram o más.
  - Disco SSD para la rápida lectura/escritura de los datos, y donde la capacidad dependerá del volumen de la base de datos.
  - Procesador i5 o mayor.

- Habilidad para el desarrollo de Software.
- Conocimientos en procesamiento digital de audio.

## 1.4 Justificación y utilidad

Hoy en día vivimos conectados en el mundo virtual a través de los dispositivos móviles. La tecnología *streaming* ha sido una revolución en estos últimos años, tanto a nivel de video como de música.
Atrás quedó lo de descargarse toda la música previamente de almacenarla en el dispositivo mp3. Existen grandes compañías que ofrecen el servicio de acceso inmediato a millones de canciones en un solo click. Pero, ¿cuántas veces hemos escuchado una canción y nos ha gustado, pero sin embargo no sabemos como se llama? Ahí es donde entramos nosotros. Esta tecnología no es nueva, hay varias aplicaciones que llevan a cabo el reconocimiento de las mismas con una grabación de x segundos, pero nosotros queremos desarrollar la nuestra y ver si somos capaces de como mínimo, reproducir éste objetivo, y en el futuro ser capaces de optimizarlo, e incluso, un posible modelo de negocio sería asociarse con Google o algún gigante de internet, y hacer big data y data science para una búsqueda mucho más rápida con un menor procesado de datos.

## 1.5 Objetivos

El objetivo del proyecto es implementar un sistema de identificación musical semejante a Shazam pero usando una tecnología diferente a la usada por esta empresa, ya que ésta es propietaria y protegida por distintas patentes.
En el proyecto, y como diferencias fundamentales frente a los detalles hechos públicos por los autores de Shazam, planteamos dos líneas de trabajo

1. Uso de una representación tiempo-frecuencia completa, en lugar de centrarse en los picos de la misma.
La detección de picos es una técnica complicada en sí misma ya que la propia definición de pico es discutible. ¿Cómo se resuelven los valores máximos próximos en el tiempo o en frecuencia? ¿Cuántos picos hemos de considerar?...
El fingerprinting basado en la posición de los picos pierde toda la información acerca de su amplitud. Ésta podría ser una característica interesante, ya que, como defienden sus autores, puede aportar robustez al sistema. Ahora bien, esa misma robustez se puede alcanzar por otros mecanismos, no tan drásticos como la reducción de toda la información de la ventana a un conjunto discreto de puntos.
Una elección desacertada del número de puntos a tomar, o algún error en la detección o no detección de alguno de ellos, puede llevar a dificultar mucho la identificación de la canción.

La caracterización de los segmentos a partir de los picos dificulta el establecimiento de medidas que indiquen cuán semejantes son dos de ellos. Pequeñas perturbaciones en la señal pueden provocar que segmentos semejantes queden representados por constelaciones de picos muy diferentes, o lo contrario.

2. Uso de árboles de decisión, en principio binarios, en lugar de tablas hash.

En una tabla hash, los datos se encasillan en posiciones concretas de la misma, perdiéndose toda la información de cuáles son las posiciones de los segmentos semejantes. En un árbol de decisión puede conseguirse un cierto nivel de estructura, de tal modo que se pueda tener un cierto conocimiento de qué otras ramas del árbol están cerca de una dada.

Pueden construirse estructuras semejantes a árboles de decisión, en las que distintas ramas de un mismo nodo puedan estar activadas simultáneamente, eliminando las fronteras duras de los árboles binarios clásicos o las tablas hash.

# 2. Herramientas para el desarrollo del Software

## 2.1. Introducción

Para el desarrollo de aplicaciones (móviles, web..) es importante contar con las herramientas adecuadas que nos asistan durante el proceso para ahorrar tiempo y esfuerzo sin descuidar la calidad del resultado final.

Un SDK (Software development kit), o kit de Desarrollo de software, es un conjunto de herramientas que ayudan al desarrollo de la aplicación para un entorno tecnológico particular.

## 2.2. Visual Studio Code

Los algoritmos están basados en instrucciones, y estas mismas pueden ser de diferente nivel. El formato del código es texto plano, y puede ser generado por editores de texto muy sencillos como el caso de Bloc de notas en los sistemas operativos Windows.

Cuando se trata de proyectos de gran envergadura, utilizar estos editores deja de ser práctico, así que para este proyecto hemos escogido **Visual Studio Code** (VS Code de aquí en adelante).

Este editor de código fuente admite muchos lenguajes de programación (Código abierto o bajo licencia) y muchas funcionalidades prácticas (nombramos las más importantes):

- Multiplataforma

Otra de las ventajas que obtenemos al usar VS Code es que se puede instalar en los 3 principales sistemas operativos: Windows, Linux y Mac OS, basta con entrar en la página web principal y descargar los archivos binarios correspondientes.

- IntelliSense

Denominamos *IntelliSense* la capacidad de predecir la instrucción que vamos a escribir. Esto nos da la ventaja de con el tabulador, autocompletar el texto y evitar errores de sintaxis y a la vez ser más productivos.

- Compilaciones

Trabajamos con el código y el compilador por separado, por lo que nos permite crear un código nuevo o editar el mismo.

- Depuración

La depuración con un editor de texto no podremos ver los cambios a tiempo real, habrá que finalizar la depuración, alterar el código y volver a compilarlo.

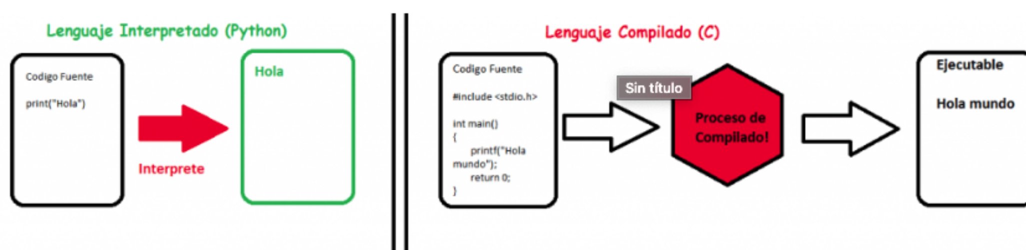
A todo esto, podemos ver que VS Code es un editor de código fuente multiplataforma con muchas bondades, ventajas, ligero y rápido. Nos permite trabajar con muchos lenguajes distintos y desarrollar todo tipo de aplicaciones.

## 2.3 Python 3

### 2.3.1 Definición de Python

> *Según la Wikipedia Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible. Y define este como un lenguaje multiparadigma, debido a que soporta orientación a objetos, programación imperativa y en menor medida programación funcional. Es interpretado de tipado dinámico y multiplataforma.*

Los lenguajes de programación se agrupan en **Interpretados** y **Compilados** según el modo en el que son traducidos a binario, lenguaje en el que se comunican las máquinas.




Los lenguajes Interpretados en los que el código es traducido por un intérprete y solo traduce lo necesario. Entre los más comunes encontramos a "Python", "Javascript"…

Ventajas:

- Nos ahorra tiempo en el desarrollo y prueba de la App.
- El código fuente puede ser ejecutado en cualquier software que disponga del intérprete.

Python como lenguaje de programación científico:

Python es un lenguaje de programación interpretado. Nosotros en la universidad hemos trabajado con Matlab para el procesado de señal, y la pregunta era si podíamos hacer con Python lo que hacemos con Matlab. La respuesta es **SÍ**, añadiendo las librerías correspondientes se puede convertir en un auténtico entorno científico.
La principal ventaja por la que nos decantamos por Python es que es un lenguaje *Open source* y no tenemos que pagar licencias para usarlo.

### 2.3.2 Módulos requeridos:

- **NumPy** : Es una librería de Python, que le agrega mayor soporte para vectores y matrices, constituyendo una biblioteca de funciones matemáticas de alto nivel para operar con los mismos.

  En el tratamiento de señal como trabajaremos con vectores 1D y 2D para los espectrogramas, nos dará mucho soporte a la hora de hacer cálculos.

- **Docopt**: Este módulo de descripción de interfaz de línea de comandos. Nos ayuda a definir una interfaz para una aplicación y ejecutarla a través del terminal y generando un analizador para ella.
  Nos da la posibilidad de mostrar mensajes de ayuda invocando la opción -*h* o -*help,* y a su vez, nos dará un fuerte control sobre nuestro sistema.
  Lo utilizaremos para el script final, donde podremos ejecutar el programa principal.

- **h5py:** El paquete h5py es una interfaz pitónica para el formato de datos binarios HDF5.
  Nos permite almacenar grandes cantidades de datos numéricos, y manipularlos fácilmente desde NumPy. Por ejemplo, nos permite cortar esos conjuntos de

datos de varios terabytes almacenados en el disco, como si fueran verdaderos vectores de NumPy, o pueden ser almacenados en un mismo archivo, categorizados y etiquetados como queramos.

H5py utiliza metáforas directas de NumPy y Python, como el diccionario y la sintaxis de la matriz NumPy. Podemos iterar sobre los conjuntos de datos del archivo, o comprobar los atributos *.shape* o *.dtype* de los conjuntos de datos, y no necesitamos saber nada especial sobre HDF5 para empezar.

- **Struct**: Este paquete incluye funciones para convertir en cadenas de bytes y tipos de datos nativos de Python, como números y cadenas.
  Éstos permiten empaquetar datos en cadena y desempaquetar los utilizando especificadores de formato compuestos por caracteres que representan el tipo de datos y los indicadores opcionales de conteo y extremidad.

- **Scipy**: Es una biblioteca libre y de código abierto para Python. Se compone de herramientas y algoritmos matemáticos. SciPy contiene módulos para optimización, álgebra lineal, integración, interpolación, funciones especiales, FFT, procesamiento de señales y de imagen, resolución de ODEs y otras tareas para la ciencia e ingeniería.

  SciPy se basa en el objeto de matriz NumPy y es parte del conjunto NumPy, que incluye herramientas como Matplotlib, pandas y SymPy, y un conjunto en expansión de bibliotecas de computación científica. Este conjunto está dirigido al mismo tipo de usuarios que los de aplicaciones como MATLAB, GNU Octave, y Scilab. A veces también se hace referencia a este conjunto de herramientas y bibliotecas como SciPy.

## 2.4 Base de datos

### 2.4.1 Introducción

Para una correcta evaluación de nuestro  sistema de reconocimiento musical es necesario una amplia base de datos con miles de ficheros.
Uno de los principales problemas que hay que evitar es el de las licencias y derechos de autor de los mismos, así que hemos optado por una base de datos FMA (Free Music Archive) con decenas de miles de archivos incluyendo gran variedad de géneros. Podemos ver su contenido en este link.

### 2.4.2 Descarga

Para descargar la base de datos lo haremos a través del repositorio GitHub cuyo autor es Michaël Defferard.

En la pantalla principal del repositorio, encontramos el apartado **README.md** el cual deberemos situarnos en el apartado **Data** y descargarnos el fichero *fma_large.zip* que consta de 106.574 archivos de música de 30 seg por archivo, y un total de 161 géneros ocupando 93 GB de almacenamiento en el disco.

Al descomprimir el archivo vemos que los ficheros se reparten en 155 directorios.

Problemas

En nuestro caso, hemos tenido problemas de almacenamiento, ya que al trabajar con una base de datos tan grande, necesitamos un disco SSD donde la lectura y escritura de datos es mucho mayor al disco duro tradicional, y supone un coste elevado.
Trabajamos con un disco de 250 GB SSD y la base de datos ocupa casi la mitad del disco.

Soluciones

El primer intento de solución fue reubicar la base de datos a un disco externo, tanto HDD como SSD. Al hacerlo con el HDD vimos que las pruebas tardaban varios días tanto como para nuestro sistema como el DejaVu (Más adelante, veremos lo que es) y al trabajar con una computadora portátil, sufría recalentamiento, así que optamos por probar con un disco duro externo SSD, pero nos encontramos con otra barrera, y es que los Mac están limitados para los discos externos (política de la compañía) y la velocidad de transferencia de datos disminuye hasta comportarse como un HDD.

En nuestro segundo intento optamos por subir la base de datos a la nube, ya que en el Mac te da la opción de sincronizar el escritorio con iCloud, pero nos encontramos con otro inconveniente, y es que cuando trabajas con el directorio de la base de datos, lo descarga previamente en local antes de trabajar, y eso nos suponía un retroceso.

Finalmente, optamos por acotar la base de datos 10 veces menos, es decir trabajar con un total de 10 mil ficheros.

### 2.4.3 Ficheros para el reconocimiento

Para evaluar nuestro sistema, hemos clasificado las pruebas en distintos niveles de SNR:

$$SNR = \frac{Psignal}{Pnoise}$$

dónde el valor del SNR a medida que aumenta, más limpia es la señal. Tenemos mil ficheros para cada valor del SNR = [0,10,20,30,40].

### 2.4.4 Adición del ruido

Primero hemos obtenido señales de ruido, como por ejemplo grabaciones de tráfico, conversaciones de bares… Al final de lo que se trata es de añadir perturbaciones con degradación desconocida lo más ajustado posible a la realidad.

A través de un script, cogiendo las señales originales como *x*, y las señales de ruido como *n,* debemos amplificarlas por un factor de potencia de tal manera que la señal resultado serà:

$$y = \alpha x + \beta n$$

Donde $\alpha$ es la ganancia de la señal original y $\beta$la ganancia del ruido. Para variar el SNR debemos ajustar la amplitud del ruido hasta el valor deseado.

Finalmente, para evitar saturación, dividiremos cada muestra de la señal resultado por el máximo de la misma.

# 3. Desarrollo del sistema de reconocimiento musical

## 3.1. Introducción.

Habiendo explicado los objetivos del proyecto anteriormente en la memoria, a continuación explicaremos algunos conceptos sobre el procesado digital de audio, para entender a nivel científico el procedimiento de los algoritmos y técnicas en que nos basamos para el alcance del objetivo.

## 3.2. Representación tiempo-frecuencia de los segmentos de audio.

La base de todo sistema de reconocimiento automático es una representación de la señal que proporcione el máximo de información acerca de lo que se desea reconocer, pero, al mismo tiempo, permita eliminar el máximo de información irrelevante o redundante. Esta información irrelevante o redundante aumenta la varianza de la representación, constituyendo, por tanto, un ruido que enmascara lo realmente relevante.

Por ejemplo, en identificación musical no estamos interesados en mantener la referencia temporal, porque el segmento a reconocer puede estar colocado en cualquier posición dentro de la canción.
La señal temporal incorpora toda la información; tanto la que es relevante como la que no. Es por ello necesario acudir a otro tipo de representación en la que sea posible aprovechar el máximo de información útil, descartando el resto. Una alternativa clásica a la señal temporal es la representación tiempo-frecuencia o espectrograma.

### 3.2.1. El espectrograma.

El espectrograma de una señal consiste en el espectro de los segmentos de corta duración que forman una señal. De este modo se pretende caracterizar la quasi-estacionariedad de señales como la voz o la música, en las que la información está concentrada en la forma del espectro, pero éste varía con el tiempo.

Se calcula el espectrograma dividiendo la señal temporal en ventanas, también denominadas tramas; cada trama es multiplicada por una ventana adecuada para evitar o mitigar el desparrame frecuencial; para cada ventana de señal se obtiene su transformada discreta de Fourier, de la que se extrae el módulo al cuadrado (periodograma); finalmente, el espectrograma es el resultado de representar los *periodogramas* de cada ventana, habitualmente usando decibelios.

La elección de la longitud de la ventana resulta un parámetro crítico en la representación obtenida. Idealmente, la longitud debería ser tan larga como para realizar una buena caracterización en frecuencia ya que, cuanto más larga es la ventana, mayor es la resolución obtenida. Pero, al mismo tiempo, las señales de audio se componen de segmentos de espectro más o menos constante, pero distinto del de sus vecinos, así que la ventana debería ser lo suficientemente corta como para capturar el detalle temporal del sonido.

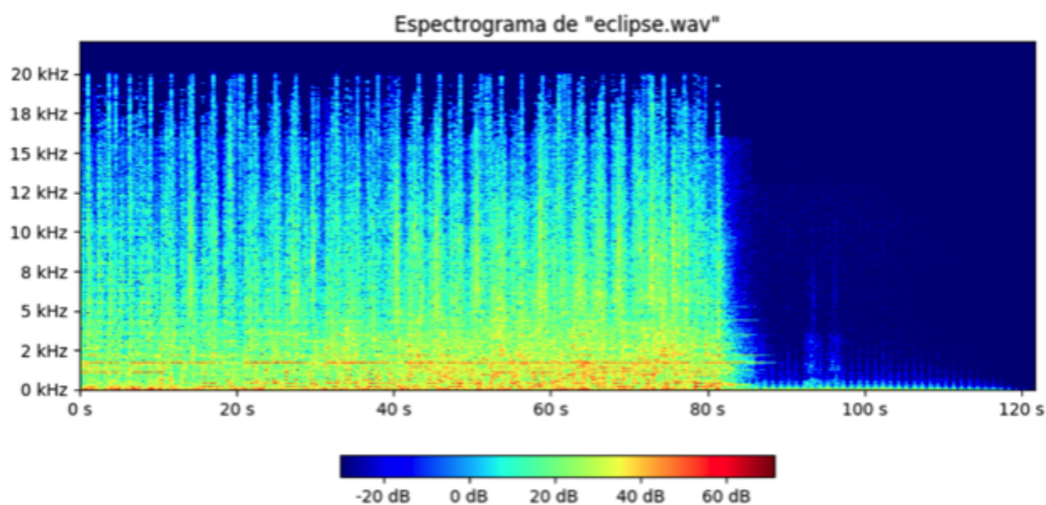
Ambos requisitos entran en contradicción ya que, dependiendo de la aplicación, el primero exigiría ventanas mucho más largas que unos 100 ms, mientras que el segundo requeriría ventanas mucho más cortas que unos 20 ms. El compromiso depende de la aplicación concreta para la que se va a usar el espectrograma. En aplicaciones de procesado de voz es habitual el uso de ventanas de entre 10 ms y 30 ms, en identificación musical suelen usarse ventanas de mayor duración, en torno a los 100 ms.

El desparrame frecuencial es el efecto por el que, al tomar un segmento de señal y calcular su espectro, aparecen componentes frecuenciales no presentes en la señal

completa; componentes que ensucian la estimación del resto de frecuencias. Este efecto depende de la ventana usada al extraer el segmento, ya que está asociado a la presencia de lóbulos secundarios en su transformada de Fourier.
El peor caso se da usando la ventana rectangular, pudiéndose mitigar su efecto usando ventanas que atenúen la señal en los extremos del segmento; como, por ejemplo, las ventanas de von Hann, Hamming o Kaiser.

El precio a pagar por el uso de una ventana es la pérdida de resolución frecuencial, esto es, la capacidad del estimador de diferenciar componentes frecuenciales próximas entre sí. En este caso, la ventana rectangular es la que mayor resolución proporciona, disminuyendo ésta conforme se escogen ventanas de menor desparrame.
Elecciones de compromiso para resolución y desparrame son la ventana de Hamming, la de Kaiser o la de Blackman, que es la que se usará en este trabajo.

El uso de una ventana distinta de la rectangular presenta un inconveniente añadido: al atenuar los extremos del segmento, éstos quedan peor representados que la parte central, en la que la ventana presenta un máximo. Para solucionar este problema es habitual usar un cierto solapamiento entre segmentos consecutivos. Es decir, el desplazamiento entre ventanas no es igual a su longitud, sino menor. Es típico usar desplazamientos de ventana que den lugar a solapamientos de entre el 50 % y el 75 %.



La figura anterior muestra el espectrograma de ’Eclipse’, el último tema del disco de Pink Floyd The Dark Side of the Moon. La duración de la ventana se ha fijado a **longVent** = 100 ms, con un desplazamiento de **despVent** = 25 ms; se ha aplicado la ventana de Blackman; y se representa el espectro de energía de cada ventana de señal usando decibelios. Sólo se usa la primera mitad de la DFT, descartando las componentes correspondientes a las frecuencias negativas.

### 3.2.2. Espectrograma de segmentos de señal.

En la identificación musical no estamos interesados en reconocer la canción a partir de la señal completa, sino a partir de segmentos de corta duración de la misma (en torno a 10 s). Para ello es conveniente dividir la señal a reconocer en **segmentos** de esa duración y caracterizar cada uno de ellos con su espectrograma.

Como se desconoce a priori cuál será la posición del segmento a reconocer dentro de la canción, es habitual introducir también aquí un solapamiento entre segmentos consecutivos. Típicamente, los segmentos tienen una duración **longSegm** ≈ 10 segundos, con un desplazamiento entre ellos de **despSegm** ≈ 1 segundo.

Es conveniente usar un valor del desplazamiento de segmento, **despSegm**, que sea múltiplo entero del de ventana, **despVent**. De este modo, las ventanas de señal son compartidas por distintos segmentos, evitando cálculos redundantes.

Suponiendo que tomamos **despSegm = k * despVent**, con k entero, el algoritmo para realizar la conversión tiempo frecuencia de una señal de audio es:

Para cada ventana de señal

- Multiplicar la señal por una ventana de forma adecuada para evitar problemas de despa-

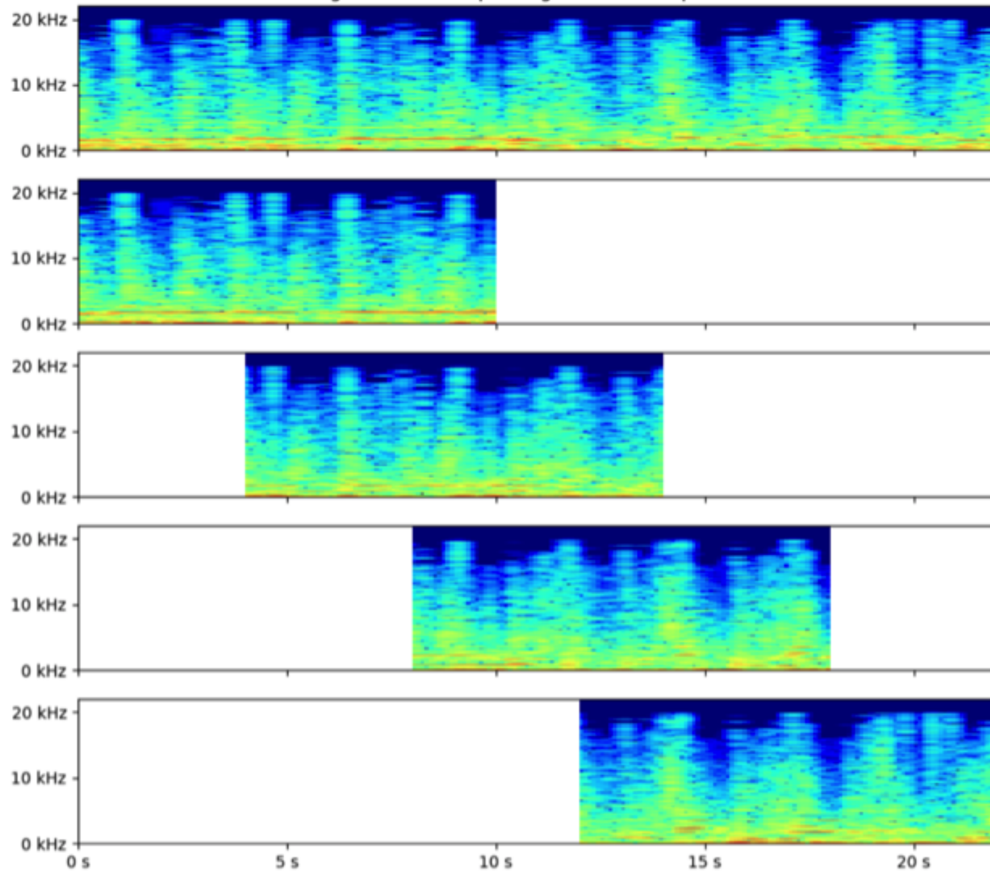
rrame frecuencial; por ejemplo, la de Hamming o la de Blackman.

• Cálculo de la transformada discreta de Fourier (DFT) con un número de puntos igual a
una potencia entera de dos, y mayor o igual que el doble de la duración de la ventana.

• Elevado al cuadrado del módulo de la DFT; las frecuencias negativas, correspondientes a la mitad superior de la DFT, se descartan.

Cada segmento de duración **longSegm** se modela agrupando las ventanas correspondientes para construir su *espectrograma*.

Por ejemplo, tomando segmentos de duración **longSegm** = 10 segundos, desplazados **despSegm** = 4 segundos, el inicio de la canción anterior queda representado de la manera siguiente:



Segmentos del espectrograma de "eclipse.wav"
20 kHz
10 kHz
0 kHz
0 s
5 s
10 s
15 s
20 s

## 3.3. Banco de filtros en escala Mel.

### 3.3.1. Escala Mel.

Un problema de la representación tiempo-frecuencia de la sección anterior es que trata todas las bandas frecuenciales del mismo modo, aunque en aplicaciones de audio no todas tienen la misma importancia. El comportamiento del oído en frecuencia no es lineal: aumentar en una misma cantidad la frecuencia de un sonido no tiene el mismo efecto sensorial a una frecuencia u otra.
Por ejemplo, el cambio de 100 Hz a 200 Hz es percibido como una variación muy importante de la frecuencia, mientras que pasar de
10000 Hz a 10100 Hz es apenas perceptible.

El comportamiento del oído frente a la frecuencia puede modelarse aproximadamente como logarítmico: multiplicar la frecuencia por un mismo factor conduce a la misma sensación de variación de frecuencia, con independencia de la frecuencia concreta en la que se realiza. Así, la sensación de octava (la distancia que va de una cierta nota a la siguiente con el mismo nombre) es equivalente a multiplicar por dos la frecuencia.

El espectrograma en escala lineal de frecuencias tiene el inconveniente de que sobre-representa las altas frecuencias. Así, en las gráficas de la sección anterior se puede observar como la mayor parte de la energía se concentra en mitad inferior, mientras que en la mitad superior predominan los tonos azules.
Sin embargo, la representación lineal otorga la misma relevancia a esta mitad superior, que representa una sola octava (de 10 kHz a 20 kHz), que a la otra mitad, que representa unas nueve octavas de banda audible (de unos 20 Hz a 10 kHz).

En una representación logarítmica, cada octava de frecuencia aportaría la misma información, con independencia de su posición en la escala. No obstante, el oído no tiene una respuesta perfectamente logarítmica. En bajas frecuencias el

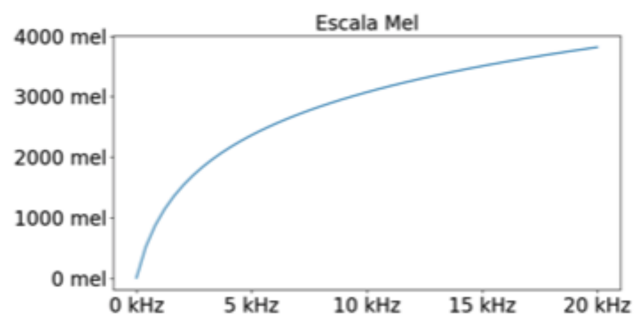
comportamiento se asemeja más al lineal, mientras que, a partir de unos 1000 Hz, el comportamiento se aproxima más al logarítmico.

La *escala Mel* (de melódica) fue propuesta en los años 30 del siglo XX para representar más adecuadamente el comportamiento del oído. Se utiliza convirtiendo los valores de frecuencia en hercios a unidades denominadas mel usando las fórmulas de conversión siguientes:

$$m = 1127 \ln\left(1 + \frac{f}{700}\right)$$

$$f = 700\left(e^{\frac{m}{1127}} - 1\right)$$

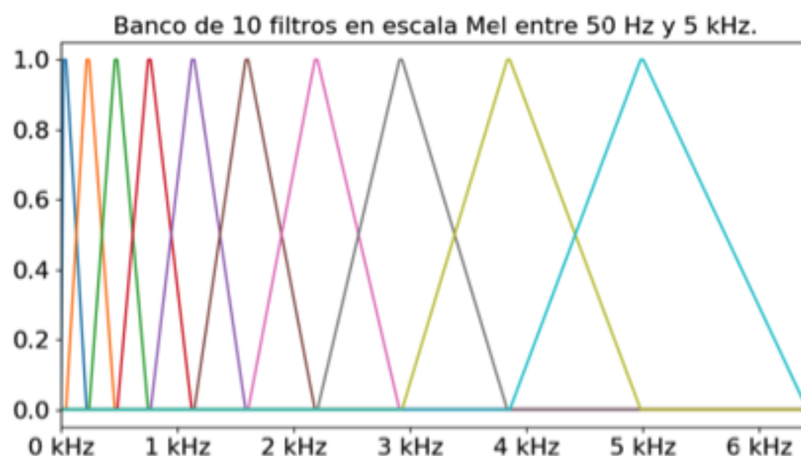
La curva siguiente muestra la forma de la curva en función de la frecuencia:



### 3.3.2. Banco de filtros.

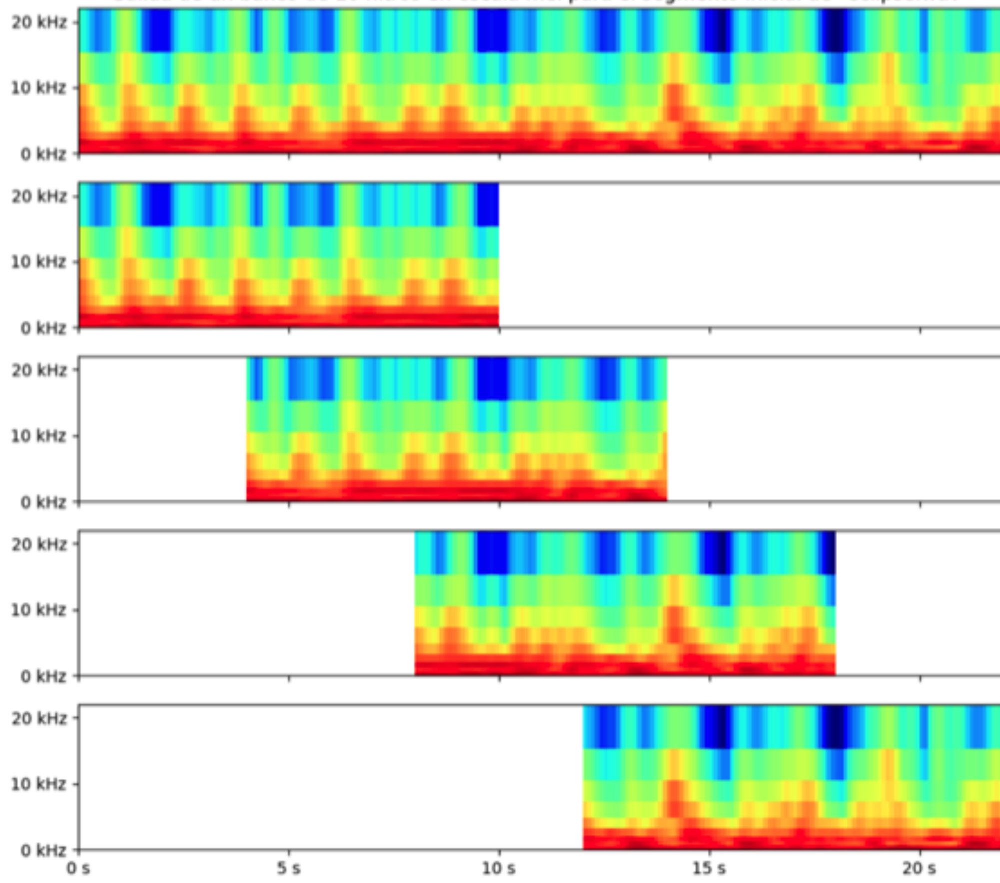
La escala Mel suele aplicarse usando un banco de filtros. La idea es integrar (sumar) la energía del espectrograma en bandas de frecuencia de anchura constante en la escala Mel. Haciéndolo de esta manera se pierde resolución frecuencial, pero esto es algo en lo que estamos interesados, ya que es un primer mecanismo para eliminar información irrelevante y/o redundante.

El banco de filtros se construye a partir de **numBnd** filtros pasa-banda, de anchura constante en escala mel, distribuidos en una banda de límites **frecMin** y **frecMax**. Típicamente se usan **frecMin ≈ 50 Hz, frecMax ≈ 5 kHz y numBnd ≈ 20**, con lo que la anchura de cada filtro es de, aproximadamente, un tercio de octava, que se corresponde con la resolución aproximada del oído interno. Además, y para evitar problemas en los bordes de las bandas, éstas suelen solaparse y tener los extremos atenuados.
Es habitual usar una forma triangular asimétrica, en la que el extremo inferior es igual a la frecuencia central de la banda inmediatamente inferior, y el extremo superior lo es a la de la inmediatamente superior.
La gráfica siguiente muestra la forma de un banco de 10 filtros distribuidos linealmente en escala Mel entre 50 Hz y 5000 Hz:



La salida de cada filtro del banco, definido en frecuencia, es simplemente el producto escalar de su respuesta en frecuencia por el periodograma de la ventana de señal. En python esta operación se puede efectuar de una manera muy eficiente usando multiplicación matricial.

En la gráfica se observa la representación tiempo-frecuencia obtenida para la canción del ejemplo usando un banco de 10 filtros equiespaciados en escala Mel entre 50 Hz y 20000 Hz:


Salida de un banco de 10 filtros en escala mel para el segmento inicial de "eclipse.wav"
20 kHz
10 kHz
0 kHz
0 s
5 s
10 s
15 s
20 s

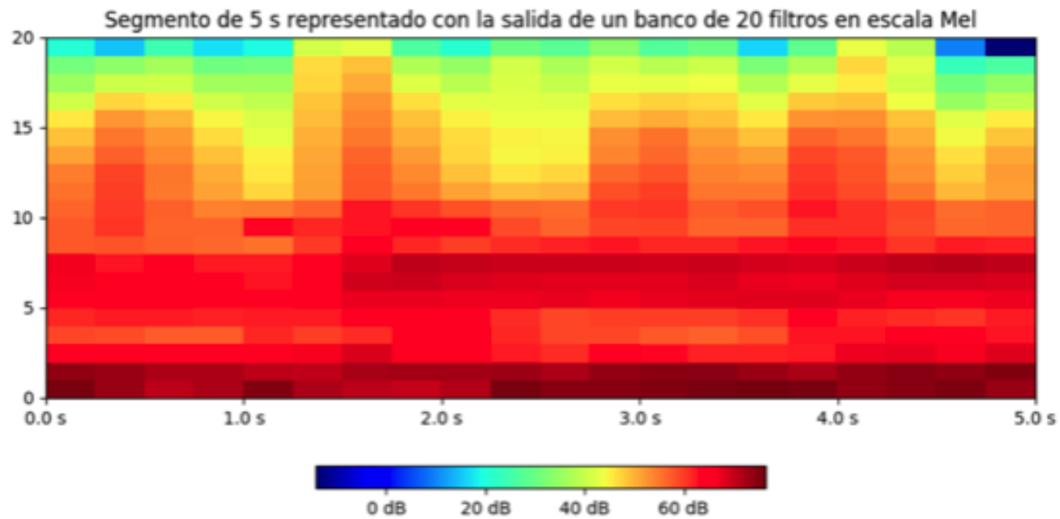
## 3.4. Decorrelación y disminución de la dimensionalidad.

La representación de los segmentos de audio usando el esquema tiempo-frecuencia de la sección anterior es, probablemente, excesivamente detallada. Suponiendo el uso de bancos de 100 filtros en escala Mel, y desplazamientos de ventana de 100 ms, cada ventana de 10 s requiere 100 × 100 = 10.000 coeficientes. Este exceso de información tiene dos repercusiones:

Almacenar toda esta información para un número elevado de canciones puede resultar descomunal. Por ejemplo, si generamos una ventana de estas cada segundo, y

consideramos un millón de canciones de doscientos segundos (algo más de tres minutos), a cuatro bytes por coeficiente real, necesitaremos unos 8 TB.

Por otro lado, como se comentó en la introducción, el exceso de información es contraproducente: aunque tendremos toda la necesaria para identificar la canción, también tendremos información innecesaria o redundante para hacerlo. Y esta información extra puede ser considerada como ruido que es conveniente eliminar.
Visualizando un segmento corto, podemos ver esta redundancia en la forma de correlación entre coeficientes vecinos, tanto en el tiempo como en frecuencia:




Puede observarse claramente cómo no hay grandes diferencias entre los valores cercanos en el tiempo y/o frecuencia. Una manera de reducir esta redundancia es la *decorrelación* de los datos.

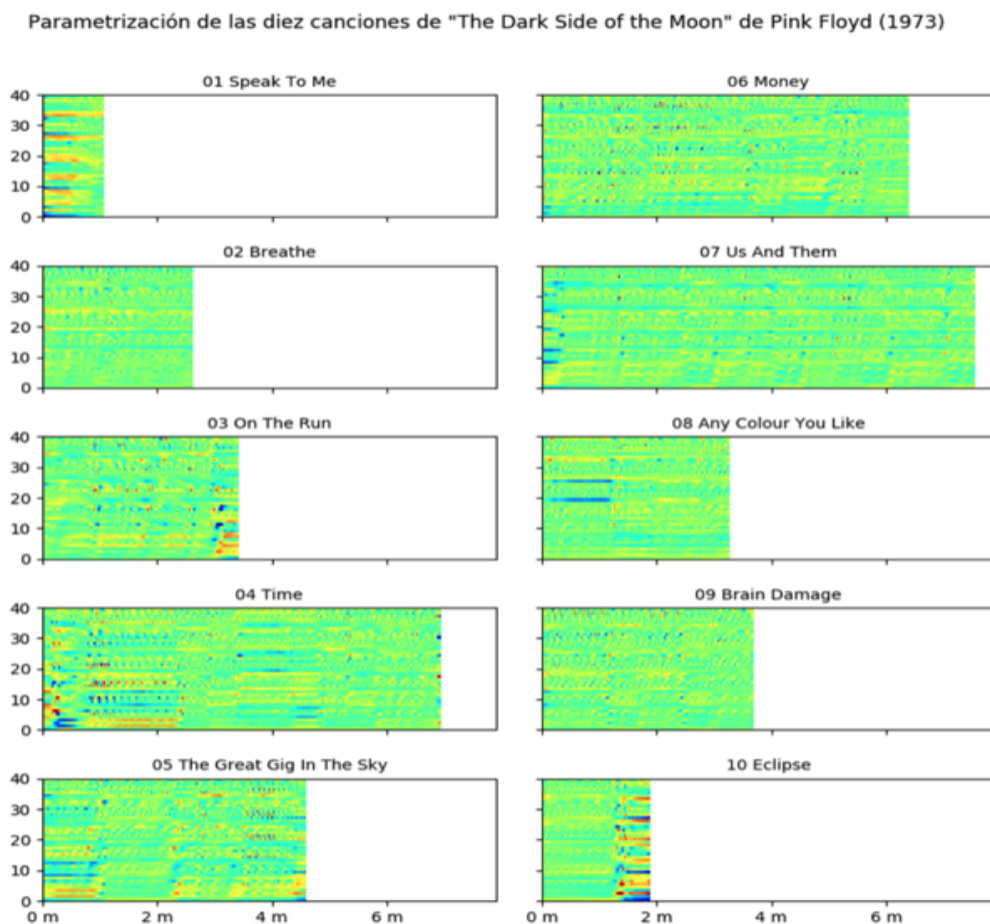
Tanto la **transformada de Fourier** como la **transformada coseno** proporcionan mecanismos para realizar esta *decorrelación*.

> La transformada coseno es útil cuando la posición relativa de la información es relevante. En la representación tiempo-frecuencia, este el caso en el eje frecuencial, ya que no es lo mismo una cierta forma en la evolución de la amplitud en una banda de frecuencias que en otra.
> En el eje temporal, la referencia absoluta es irrelevante, ya que puede representar un simple retardo en la captura de la señal. Por este motivo, usaremos el módulo de la transformada de Fourier.

Tanto una representación como la otra reflejan la forma a *grandes rasgos* en los coeficientes de orden más bajo, mientras que los de orden elevado reflejan el *pequeño detalle*.

Este detalle no sólo resulta irrelevante en la identificación de la canción sino que, además, es más susceptible al ruido. Con lo que la decorrelación nos aporta un mecanismo para reducir la dimensionalidad de la representación.

La gráfica siguiente muestra la parametrización de las 10 canciones del disco de Pink Floyd "The Dark Side of the Moon". Como se esperaba, se aprecia una cierta estructura en cada canción, pero, al mismo tiempo, cada una es distinta:



## 3.5.Identificación musical basada en distancia euclídea y uso de árboles binarios.

La identificación de canciones comparte elementos y características con otras tareas de reconocimiento automático, aunque también presenta diferencias importantes que hacen que técnicas habituales en este tipo de aplicación no le sean aplicables. En concreto, no parece muy apropiado ni el empleo de modelado estadístico ni el de redes neuronales:

> Aunque una canción puede ser considerada un proceso estocástico, en la que cada grabación o interpretación constituye una realización o muestra del mismo, en esta tarea nos centramos en realizaciones concretas.
> Podría plantearse, y hay sistemas experimentales que trabajan en este sentido, el reconocimiento de canciones de manera genérica, en la que se reconociera ésta con independencia de cuándo, cómo o, incluso, por quién se ha grabado. Pero en este proyecto nos centramos en la identificación de ediciones concretas de cada canción.
>
> Aunque es probable que técnicas de modelado neuronal y *deep learning* puedan encontrar aplicación en distintas partes del sistema de identificación, su uso directo como arquitectura del sistema de identificación parece problemática debido al elevado número de nodos terminales que sería necesario considerar (hay sistemas comerciales que permiten identificar millones de canciones) y la escasez de material de entrenamiento (de cada canción se dispone de una única realización.

En estas circunstancias, un planteamiento posible es el de comparación directa de la señal a reconocer y las canciones que forman el repertorio reconocible. Por ejemplo, puede considerarse la distancia (euclídea, de Mahalanobis o cualquier otra), como criterio de búsqueda.

Dado que, en ausencia de ruido o distorsión, la señal a identificar debe coincidir exactamente con una de las señales del repertorio, cualquier criterio de distancia que se minimice cuando ambas señales son idénticas sería útil para realizar la identificación. El objetivo de la extracción de características es el de inmunizar la identificación a la distorsión y el ruido, capturando los elementos esenciales de cada canción y eliminando, en la medida de lo posible, la influencias de éstos.

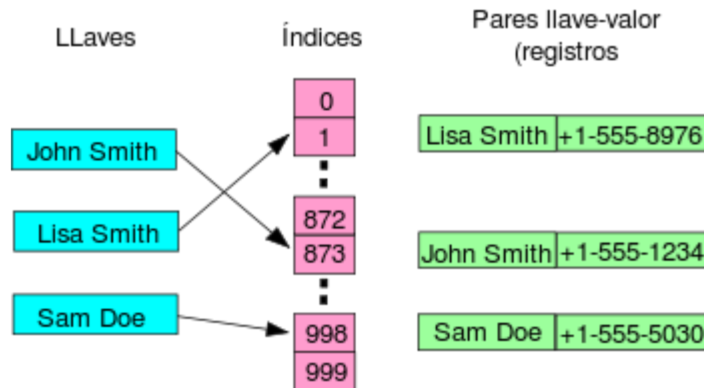
Un planteamiento de este tipo presenta el problema de su exagerado consumo computacional. Si el repertorio está compuesto, como en la parte experimental de este proyecto, por cien mil canciones, y cada canción contiene varias decenas o centenas de segmentos a reconocer, el sistema deberá realizar millones de comparaciones, a menudo muy costosas, para identificar cada segmento.

El modo habitual de reducir este coste computacional consiste en la generación de una clave hash a partir de la parametrización del segmento. Con ello conseguimos reducir la búsqueda a los segmentos del repertorio que comparten la clave con él. Esta estructura de búsqueda es lo que se denomina tabla hash.

Una **tabla hash** es una estructura de datos que asocia llaves o claves con valores. La operación principal que soporta de manera eficiente es la búsqueda: permite el acceso a los elementos almacenados a partir de una clave generada. Funciona transformando la clave con una **función hash** en un *hash*, un número que identifica la posición donde la tabla localiza el valor deseado.

Las **tablas hash** se suelen implementar sobre vectores de una dimensión, aunque se pueden hacer multidimensionales basadas en varias claves. Como en el caso de los *arrays*, las tablas hash proveen tiempo constante de búsqueda promedio sin importar el número de elementos en la tabla.

Estas tablas son más útiles cuando el volumen de información es realmente grande.




**3.5.1. Clasificación ciega de señales usando árboles binarios.**

El inconveniente del uso de claves y tablas hash es que los segmentos que comparten clave pueden ser muy desemejantes, mientras que segmentos casi idénticos pueden tener claves muy distintas. Esto implica que, una vez determinada la clave hash, el segmento asignado queda igualmente determinado, sin la posibilidad de aplicar técnicas que pudieran refinar la búsqueda.

En este proyecto se aplica un razonamiento distinto: en lugar de una tabla hash se usa un árbol de búsqueda binario. Este tipo de clasificador garantiza que los segmentos clasificados de igual modo son parecidos entre sí. Además, puede ser extendido fácilmente para localizar segmentos que, aún habiendo sido clasificados de manera diferente, son susceptibles de ser cercanos al segmento a reconocer.

Un **árbol binario** es una estructura de datos jerárquica, compuesta por nodos que permite clasificar una cierta población. Los nodos pueden ser de tres tipos: inicial, intermedios o terminales:

- Nodo inicial:

También denominado raíz, es el punto de entrada al árbol. Todos los elementos de la población pertenecen a él. Por lo demás, su estructura y comportamiento es idéntico al de los nodos intermedios.

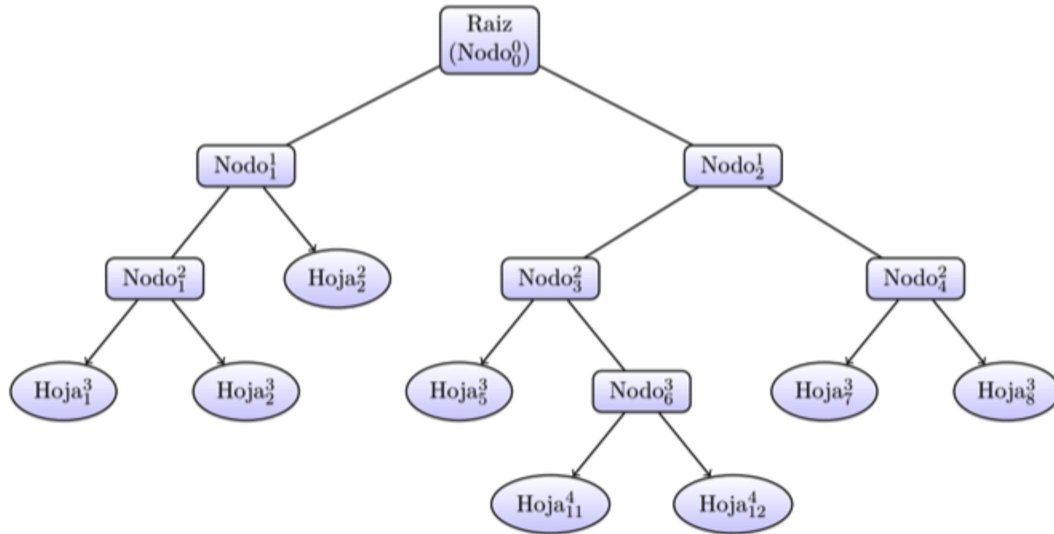
- Nodos intermedios:

Consisten en una regla de decisión, en función de la cual se divide la población del nodo en dos subconjuntos disjuntos que se asignan a otros dos posibles nodos, denominados *ramas*.

> Es habitual referirse a las dos ramas de un nodo intermedio como hijos, y al nodo del cual es hijo como *padre*.
>
> Todo nodo, salvo los terminales u hojas, tiene a lo sumo dos hijos, que se suelen denominar *izquierdo* y *derecho*.
>
> > En este proyecto, un nodo sólo se subdivide si es posible construir las dos ramas hijas. Por tanto, todo nodo tiene 0 o 2 hijos y se trata de un árbol lleno.

- Nodos terminales:

Nodos finales del árbol que no se subdividen más. Son denominados, también, hojas, y es el resultado último de la clasificación de la población.




En el diagrama se puede ver una estructura típica en árbol. La sucesiva división de los nodos en ramas permite establecer una estratificación por niveles, en la que el nivel viene marcado por el número de comparaciones realizado hasta llegar al nodo u hoja. En la gráfica, este nivel se indica como el superíndice del nodo. Si todos los nodos de un cierto nivel se subdividen en dos ramas, el número total de ramas crece exponencialmente con el número de niveles.
El subíndice de los nodos del diagrama indica su posición teórica, suponiendo que todos los nodos del nivel superior se han dividido en dos.

El número de clases que permite separar un árbol es igual al número de hojas del mismo, H. En un árbol lleno, como el que se usa en el trabajo, este número está limitado por el número de niveles, V:

$$V \leq H \leq 2^{V}$$

El valor mínimo se obtiene cuando cada nodo intermedio, salvo el último, da lugar a una hoja y otro nodo intermedio. El máximo se produce cuando el árbol está perfectamente balanceado, y todos los nodos intermedios tienen dos hijos del mismo tipo, ya sean nodos u hojas. La efectividad del árbol depende de cuánto se acerca el

número de hojas a su valor máximo, $H = 2^V$. En general, mientras el número de niveles se mantiene en valores bajos, los árboles diseñados en este trabajo se han acercado a este valor máximo.

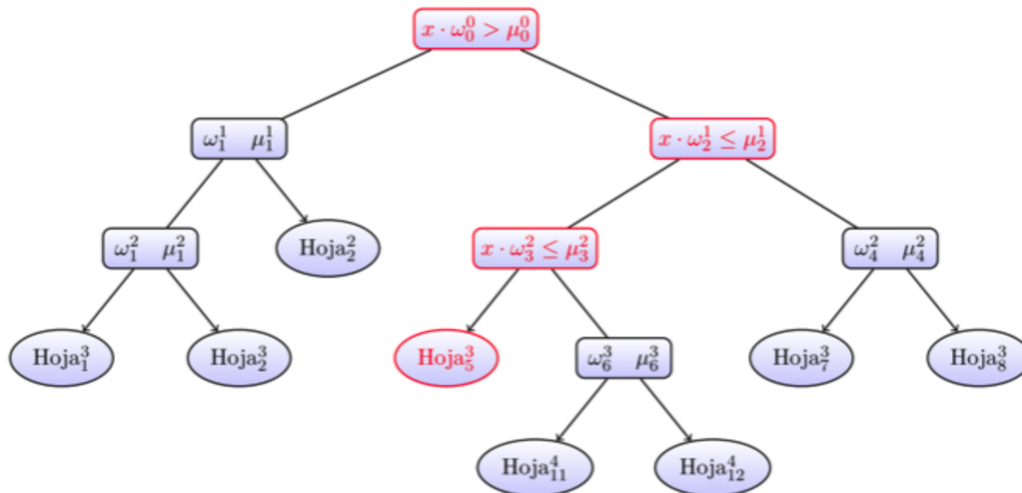
Para garantizar el máximo parecido de las poblaciones de las hojas del árbol, se ha optado por dividir los nodos utilizando un hiperplano caracterizado por el vector $\omega^j_i$ y el umbral $\mu^j_i$. Dada la señal $x$ y el nodo $Nodo^j_i$, la rama escogida en el nivel $j + 1$ se obtiene comparando el producto escalar de $x$ y $\omega^j_i$ con el umbral $\mu^j_i$:

$$k = xw_i^j \qquad \text{rama del nivel } j+1 = \begin{cases} \text{izquierda} & \text{si} \quad k \leq \mu_i^j \\ \text{derecha} & \text{si} \quad k > \mu_i^j \end{cases}$$

El procedimiento de clasificación es el siguiente algoritmo recursivo:

1. Inicialmente, el nodo asignado a cada elemento de la población es la raíz.
2. Mientras el nodo asignado no sea una hoja:
   a) Se calcula el resultado de la regla de decisión del nodo actual.
   b) Se asigna el elemento a la rama correspondiente.
3. El resultado es la hoja a la que queda asignado el elemento.

La gráfica siguiente muestra, en color rojo, este proceso para un segmento que acaba clasificado en la $Hoja^3_5$:




## 3.6. Construcción del árbol.

El criterio empleado para la construcción del árbol es el de mínima varianza o distancia euclídea: la población de cada nodo se divide en dos nuevas poblaciones de manera que la varianza total respecto a un representante, denominado centroide, sea mínima. Para ello se utiliza el algoritmo de Lloyd:

1. Dada una partición arbitraria de la población en dos subgrupos, el centroide óptimo de cada uno de ellos es igual a la media aritmética de sus poblaciones.

2. Dados un conjunto de centroides, la partición óptima es la que asigna cada elemento al subgrupo cuyo centroide está más cerca.

Como cada uno de estos dos pasos disminuye de manera independiente la varianza total, su aplicación iterativa conduce forzosamente a un mínimo local de la misma (estrategia toma daca).

Calculados los centroides de la rama izquierda y derecha, el vector $\omega^j_i$ es igual a un vector unitario en la dirección que los une y el umbral $\mu^j_i$ es igual al producto escalar de $\omega^j_i$ y centro de gravedad de la población del $Nodo^j_i$.

El algoritmo de Lloyd permite optimizar una partición arbitraria inicial. Para construir el árbol completo se utiliza una variante del algoritmo Linde-Buzo-Gray (LBG):

1. Inicialmente el árbol sólo contiene el nodo raíz, al que pertenecen todos los elementos de la población.
2. Mientras no se alcance el número de niveles predeterminado:
   a) Para todos los nodos terminales:
      1) Se realiza una partición inicial de la población del nodo usando como vector ω la dirección de máxima covarianza.
      2) Se reestima la partición con un cierto número de iteraciones del algoritmo de Lloyd.
   b) Se incrementa el número de niveles con las ramas calculadas en el paso anterior.

El entrenamiento del árbol se realiza con el programa **entrena.py**, que toma como argumentos las señales que participan en el entrenamiento y el nombre del árbol generado. Las opciones principales del programa son:

- **minTrm**: Número mínimo de señales para dividir una hoja.
- **numNiv**: Número máximo de niveles del árbol.

  Las hojas del árbol se dividirán en dos ramas en tanto su población. supere el umbral minTrm y el número de niveles sea menor que numNiv.

- **numIte**: Número de iteraciones del algoritmo de Lloyd.
-

Otros parámetros que gobiernan el comportamiento del programa son:

- **arbIni**: Árbol inicial: permite añadir niveles a un árbol previamente construido. Creciendo iterativamente el número de niveles del árbol es posible realizar experimentos sobre la influencia que este número tiene sobre el reconocimiento.

- **numCof**: Número de coeficientes a usar de las señales parametrizadas. Este parámetro resulta de utilidad al optimizar el número de coeficientes de la parametrización, ya que el mismo conjunto de señales parametrizadas, con un número de coeficientes suficiente, puede ser utilizado para distintos experimentos.

- **covMax**: Número máximo de coeficientes utilizados en la matriz de covarianza de las hojas para calcular la dirección inicial de partición de las hojas. En principio, este parámetro debería ser igual al número de coeficiente de las señales parametrizadas; pero, como el tamaño de la matriz es su cuadrado, el consumo de memoria puede ser excesivo. Este parámetro permite reducir estos requisitos de memoria.

## 3.7. Construcción de la tabla de reconocimiento.

Una vez determinada la hoja correspondiente a la señal a reconocer, es preciso comparar ésta con los segmentos de las canciones a reconocer. La información de qué segmentos están más cerca de cada hoja se organiza en forma de tabla con el script tabula.py.

En su primera implementación, este script simplemente almacena los segmentos cuantificados a cada rama del árbol. Un posible refinamiento del algoritmo podría incluir la localización de otros segmentos que, aún habiendo sido cuantificados a una rama distinta y debido al modo cómo éste se ha construido, pueden estar cerca de los que sí se han cuantificado a ella.

Las señales que deben participar en el cálculo de la tabla son todas las que forman el repertorio a reconocer. Sin embargo, no es necesario que sean las mismas que las que participan en el entrenamiento del árbol, que pueden ser un subconjunto del repertorio, para acelerar el entrenamiento, o un conjunto completamente distinto.

Los parámetros de tabula.py en esta primera versión son el árbol, el repertorio a incluir y el nombre del fichero con la tabla resultante.

## 3.8. Puesta en marcha del sistema

La arquitectura de nuestro sistema de reconocimiento musical consta de un directorio principal (de ahora en adelante, este directorio que lo abarca todo, se denominará **solución**).
Nuestra solución incluye los siguientes directorios:

- **Src**: Aquí tendremos todos los scripts de nuestro sistema, donde cada parte de él estará en un fichero distinto. Más adelante lo explicaremos más detalladamente.
- **Gui**: 2 ficheros denominados *fma_1k.gui* y *test_1k.gui* donde cada línea contiene el nombre de la señal con su directorio. El primero es para la parametrización de toda la base de datos y el segundo para los ficheros test del reconocimiento.
- **Sen:** 1 directorio denominado *fma_large* donde encontraremos el total de los ficheros de nuestra base de datos.
  Habrán 5 directorios más con el nombre *fma_test_qXXdB* donde *XX* será el valor del SNR de las señales.
- **todo.sh**: Fichero generado en Bash para poner nuestro sistema completo en funcionamiento a través del terminal con el soporte de nuestra librería **docopt**.
- **Log:** Fichero dónde almacenará todos los *outputs* después de cada reconocimiento, incluyendo errores.
- **Prm:** Semejante al directorio *Sen,* con la misma arquitectura salvo que las señales serán las parametrizadas, no las originales.

- **Arb:** En esta carpeta almacenaremos el árbol o los árboles generados para el reconocimiento junto a sus tablas con las señales clasificadas.
- **Res**: Para cada prueba, generamos un fichero resultado

Adentrémonos en nuestro directorio *src* y veamos su composición. A continuación explicaremos los ficheros más importantes:

- **arbol.py**: Tendremos las 3 clases: *Arbol*, *Nodo* y *Segmento*. Dentro de la clase *Arbol* tendremos los métodos para leer, escribir, buscar nodos (padres e hijos), buscar hojas (nodos de nivel máximo donde almacenamos las tramas).
  La clase *Nodo* nos dirá si es un padre o hijo, si está activo, su id y *maxcov*.
  Para los segmentos, almacenaremos la información del indice, la trama y el nodo al que pertenece.

- **entrena.py**: Entrena un modelo de árbol binario a partir de una base de datos de canciones en formato mp3.
  La función lee los ficheros de señal parametrizada indicados por el fichero guía 'guiSen' del directorio 'dirPrm' y construye un árbol binario de manera iterativa. Si se indica el argumento 'arbIni', el árbol inicial es el contenido en él; si no se indica, el árbol inicial es un único nodo con todas las tramas de todas las señales de entrenamiento.
  En cada iteración de entrenamiento sólo se usa una fracción aleatoria de las tramas de señal, determinada por la opción 'frcSen'. Si 'frcSen==1', se toman todas las tramas. Durante el entrenamiento, el árbol aumenta de tamaño en 'numNiv' niveles, en cada uno de los cuales el nodo terminal es dividido en dos nuevos nodos denominados hojas. Las hojas iniciales se determinan dividiendo el nodo original por la dirección de máxima varianza. Para aliviar las necesidades de cómputo y memoria, sólo se consideran las 'maxCov' primeras componentes en el cálculo de las matrices de covarianza. Sólo se dividen los nodos con más de 'minTrm' tramas.

  Las hojas creadas de este modo son reestimadas con 'numIte' iteraciones del algoritmo k-means El número de coeficientes de las señales parametrizadas puede limitarse a 'numCof' durante todo el proceso. De este modo, unos mismos fichero de señal parametrizada pueden ser usados para experimentos con distintos números de coeficientes efectivos.

- **parametriza.py**: Parametriza una base de datos de canciones en formato 'mp3'.

Lee las señales indicadas por el fichero guía 'guiSen' del directorio 'dirSen', las parametriza conforme a los argumentos de la función y almacena en el resultado en el directorio 'dirRes'.
Las señales son, primero, filtradas con un factor de preénfasis igual a 'preEnf'.
A continuación son divididas en ventanas de 'longVent' segundos tomadas cada 'despVent'. Cada ventana es multiplicada por la función indicada por 'window'; se calcula su
periodograma (módulo al cuadrado de su FFT); y se obtiene la salida de un banco de filtros equiespaciados en escala Mel de frecuencia mínima 'frecMin' y máxima 'frecMax', ambas en hercios.
La salida del banco de filtros de las ventanas está agrupada para conformar segmentos de duración 'longSegm' desplazados 'despSegm' segundos.

En cada segmento se aplica un 'umbral' de energía por banda y se extrae el logaritmo. Si 'sigma' es distinto de False, se convoluciona con la segunda derivada gaussiana (GSD) de parámetro 'sigma', que puede ser un escalar, que se aplicará a las dos dimensiones de la representación tiempo-frecuencia, o un iterable de dos elementos, en el que el primero se usará para la dimensión temporal y el segundo para la frecuencial.

Finalmente, el resultado se decorrela usando FFT en la dimensión temporal y IDCT en la frecuencial. La matriz formada por los 'cofVnt' primeros coeficientes temporales y los 'cofBnd' primeros coeficientes frecuenciales es recorrida en zig-zag, descartando el coeficiente [0, 0] para obtener 'numCof' coeficientes de salida.

- **tabula.py**: Construye una tabla con la correspondencia entre los índices del árbol y los segmentos de las canciones de entrenamiento.
  La función lee las señales del directorio 'dirPrm' indicadas en el fichero guía 'guiSen' y genera una tabla en la que, para cada nodo del árbol 'ficArb' se incluye la lista de tramas que le corresponde. Cada trama es representada mediante el nombre de la canción y el número de trama.

Este fichero se usa en el programa reconoce.py de dos maneras:
- Por un lado, permite localizar las tramas que comparten nodo del árbol con la trama a reconocer. La canción reconocida será aquélla que contiene la trama más cercana.
- Por otro, permite conocer el nombre de la canción reconocida, con lo que la propia función es capaz de calcular la exactitud del reconocimiento.

- **reconoce.py**: Reconoce la canción correspondiente a cada segmento a reconocer.

  'reconoce' toma del directorio 'dirPrmRec' cada segmento indicado por 'guiSen' y determina la hoja que mejor le representa en el arbol contenido en 'ficArb'. A continuación, determina el segmento de entrenamiento, dentro de la hoja y en el directorio 'dirPrmEnt', más cercano en distancia euclídea usando, para ello, la tabla 'tblEnt'.

  El resultado se debería almacenar en el fichero 'ficPra' para permitir su evaluación posterior, pero esta funcionalidad no se ha implementado todavía. Sin embargo, y usando la información contenida en 'tblEnt', la función devuelve, y muestra en pantalla, la tasa de exactitud del reconocimiento.
  Si 'verboso' es True, además, se muestra en pantalla el resultado de reconocimiento de cada trama de las señales de entrada.

- **selec_test.py**: Selecciona segmentos de una base de datos para su uso como material de evaluación.

  La función lee los ficheros '.mp3' del directorio 'dirSen', extrae de cada uno 'numSegm' segmentos de 'durSegm' segundos y los almacena en el directorio 'dirRes'.
  El nombre de los ficheros de resultado es el mismo que el de los segmentos de entrada pero acabados con la partícula '_#', donde '#' es el número de segmento.

Para evitar coincidencia temporal entre los segmentos de entrenamiento y los de evaluación, puede usarse el argumento 'despVent', que garantiza que la distancia entre ambos sea no menor que 'despVent / 5'.
Las señales se contaminan con una señal de ruido construida a partir de 'numSen' señales de música y/o voz. El fichero de las señales, la lista de señales, su extensión y el número a usar se indican con las opciones '--musica' y '--voz'.
En el caso de las señales de música, éstas se invierten temporalmente para evitar problemas al usar las mismas señales que en el entrenamiento. Las señales que participan en la construcción del ruido se mezclan con un margen dinámico aleatorio de excursión máxima 'margDin'.
El resultado se mezcla con la señal original con una relación señal a ruido igual a 'snr'. Los segmentos producidos están codificados con la 'calidad' indicada, en la que calidad=0 es la máxima calidad y calidad=10 es la más baja que proporciona FFMPEG. Es conveniente usa una calidad baja para evitar que los segmentos se parezcan demasiado a los utilizados para entrenar el modelo.

También habrá scripts de soporte al sistema donde tendremos funciones adicionales, como leer y escribir los ficheros, comprobaciones de correctos directorios, los *plots* para los resultados y las transformaciones de las unidades a escala Mel.

La implementación de nuestro sistema de reconocimiento musical lo haremos mediante el script *todo.sh* a través nuestro Terminal.
En el script, definiremos primero las variables de los directorios, y los parámetros de las funciones, como el número de bandas para los bancos de filtros, los coeficientes de de las ventanas y las bandas, longitud y desplazamiento de la ventana para la parametrización, longitud y desplazamiento de los segmentos, tipo de ventana, frecuencia máxima y mínima para los espectrogramas, umbral, sigma y tipo.

A partir de aquí, crearemos un bucle para cada valor de SNR parametrizar sus señales, y a continuación, darle distintos niveles al árbol y ver la precisión del reconocimiento según la variabilidad del mismo.

# 4. Software Dejavu

## 4.1 Introducción

Para evaluar los resultados de nuestro sistema de reconocimiento, es importante poder compararlo con otro software y determinar si nuestra precisión está en el camino correcto o lejos del éxito.

Para ello, hemos encontrado por internet un software llamado **dejavu** donde implementa un sistema de reconocimiento bastante completo.

## 4.2. Primeros pasos

Dejavu requiere una base de datos MySQL donde almacenaremos los audios y sus hashes (fingerprints) en formato binario. Este software nos permite reconocer tanto ficheros como grabaciones de audio con un mínimo de 5 seg.

A continuación, todo el conocimiento que necesitaremos para entender el reconocimiento y el fingerprint de la canción, empezando por lo básico.

## 4.3. Música como una señal

La música está codificada digitalmente como un gran listado de números, donde cada número representa el valor amplitud de la onda.
Un archivo **.wav** sin comprimir, tiene una frecuencia de muestreo de 44100 Hz, es decir, 44100 muestras por segundo por canal. Si queremos procesar una canción de 4 minutos:

3 min * 60 seg * 44100 muestras * 2 canales = 21.168.000 muestras

Un canal es una secuencia separada de muestras que una computadora o dispositivo electrónico puede reproducir.

Existen dos estándares comunes: **mono** y **estéreo**. El primero de ellos es cuando tenemos un único canal, y aunque usemos auriculares y tengamos dos salidas de audio, habrá redundancia ya que repetimos el mismo dos veces, mientras que en estéreo son dos canales distintos, dónde se puede dividir el sonido por frecuencias, por ejemplo, un canal para bajas frecuencias y el otro para las altas.

Hoy en día, existe la posibilidad de tener múltiples canales con los sistemas que tenemos. Claro ejemplo de ello podríamos tenerlo en casa: Home cinema.

## 4.4. Muestreo

¿Por qué 44,1 kHz por segundo? La elección es bastante arbitraria, pero está relacionado con el Teorema de Muestreo de Nyquist-Shannon, donde en una fórmula matemática larga de deducir hay un límite teórico en la frecuencia máxima a capturar con precisión. Esta frecuencia máxima se basa en la rapidez con la que muestreamos la señal.

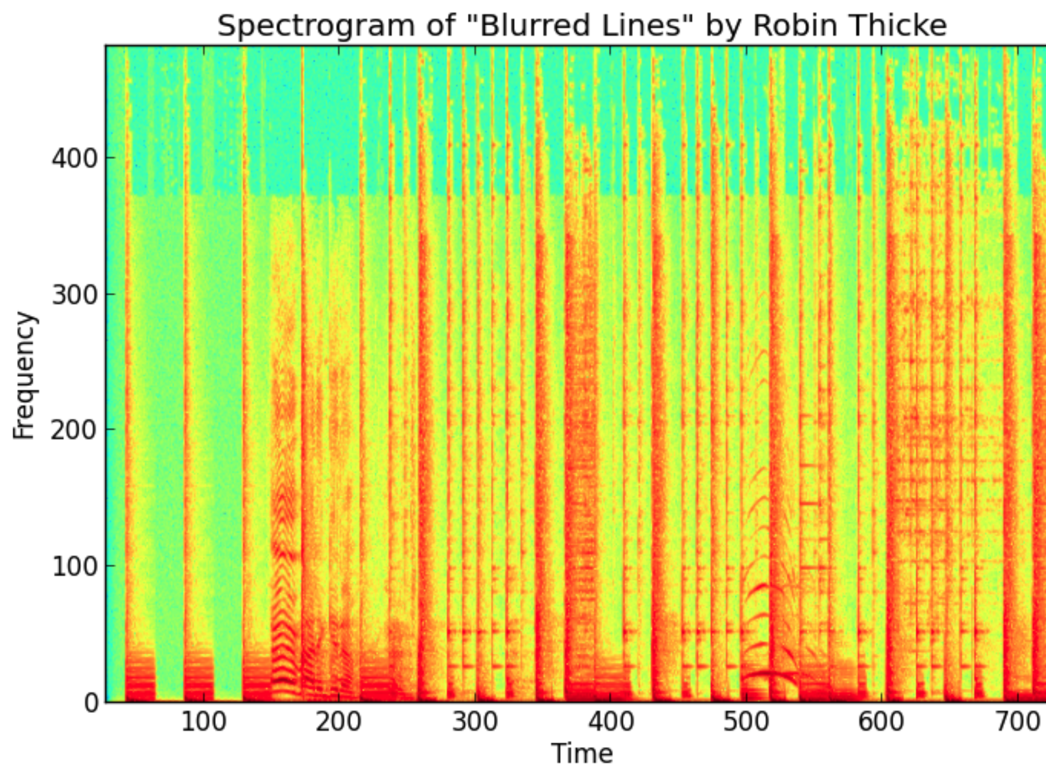
*El Teorema de Muestreo de Nyquist-Shannon* se reduce a que la frecuencia de muestreo debe ser igual o mayor al doble de la máxima frecuencia de la señal.

En este caso, la frecuencia máxima audible para el ser humano es de 20 kHz, cogemos los 22,5 kHz como frecuencia máxima, y el doble es 44.1 kHz, como hemos dicho es un poco arbitraria esta frecuencia de muestreo.

El formato MP3 lo comprime para ahorrar espacio en el disco duro, ya que un archivo en formato .wav es solo una lista de números enteros de 16 bits, es decir que la onda abarca en valores de amplitud del -32768 al 32767.

## 4.5. Espectrogramas

Dado que la señal es un conjunto de muestras, podremos usar su FFT repetidamente sobre pequeñas ventanas de tiempo para crear espectrogramas y ver dónde se concentra la energía en el dominio frecuencial a lo largo del tiempo, fíjese en la figura:




El espectrograma está compuesto por vectores 2D, en cada punto (x,y) tenemos el valor del tiempo en el eje de abscisas y el valor de la energía en la frecuencia en el eje de ordenadas.

El color verde representa la ausencia de energía para ese instante de tiempo y esa frecuencia en particular, mientras que el color rojo sería el nivel máximo de la misma.

Entonces, ¿En qué ayuda esto a reconocer el audio? Usaremos el espectrograma para reconocer el audio de manera única, y crearemos N espectrogramas para N ventanas a lo largo del segmento de la canción. También nos podemos encontrar con

contaminación acústica en nuestros registros, denominados *ruidos*, y es entonces donde entran en juego los fingerprints.

## 4.6. Búsqueda de picos

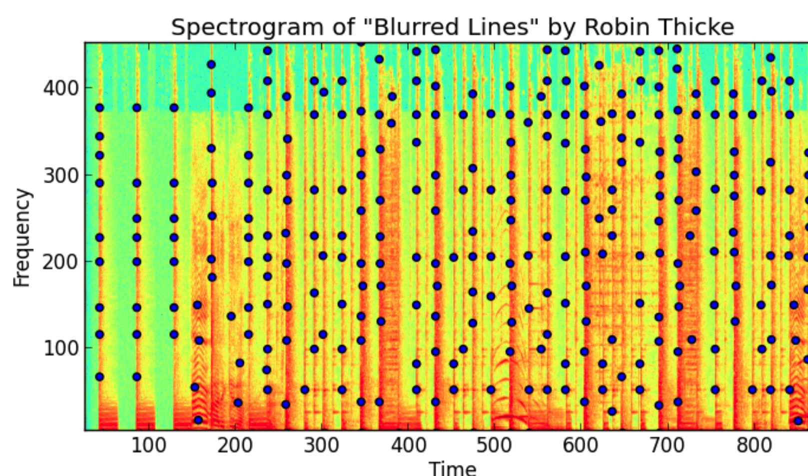
Una vez tenemos almacenado el espectrograma de la trama de la canción que queremos reconocer, es necesario encontrar los picos de amplitud.

Definimos un pico como el punto máximo, en tiempo y frecuencia, entre todos los de su "barrio" local (traducción del inglés, los *neighborhoods*). Los puntos vecinos, serán de energía menor, por lo tanto la probabilidad de que se puedan identificar entre el ruido es menor, así que seleccionamos los máximos.

Para encontrarlos, trataremos el espectrograma como una imagen y con el paquete Scipy, un módulo de Python comentado anteriormente, los hallaremos en calidad de puntos identificativos de la canción.

Nos quedaremos con esos valores para crear los *fingerprints* y el resto ya no lo necesitaremos, optimizando así la memoria de almacenamiento y el tiempo de procesamiento de datos.

Fijémonos en la siguiente figura:



Al haber tantos puntos, decenas de miles por canción, habremos discretizado los valores en un vector 2D de tiempo-frecuencia.

Al discretizar el espectrograma, hemos reducido la información de picos de infinito a finito, lo que significa que los picos de una canción podrían colisionar, emitiendo pares como picos de otras canciones, y así coincidir en distintas canciones.

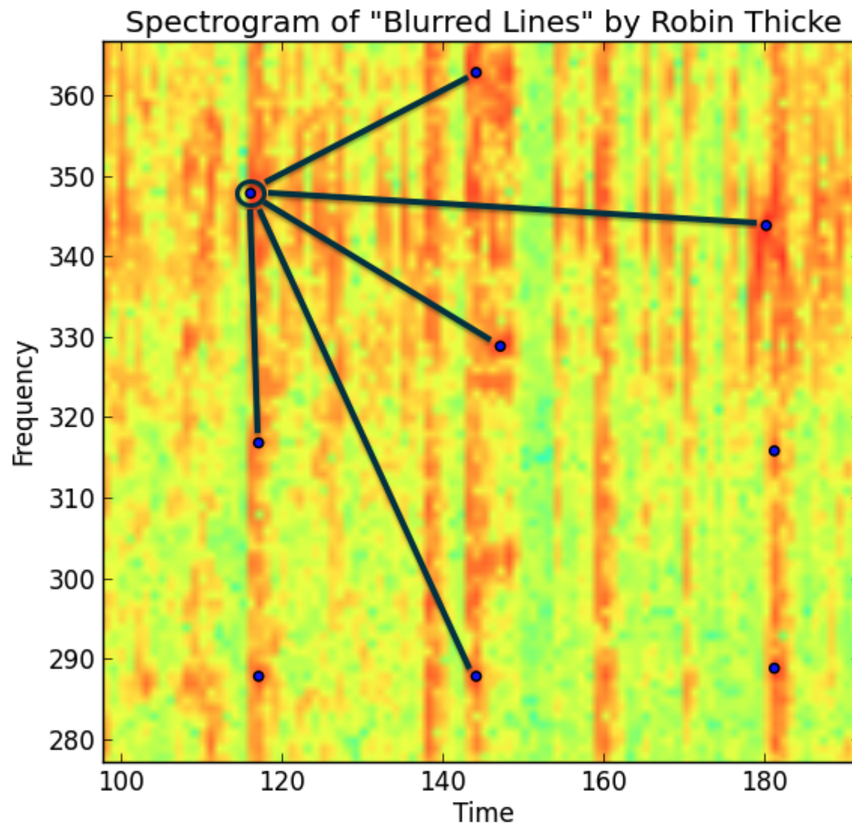
## 4.7. Fingerprint hashing

Para solucionar el problema de la similitud de picos en distintas canciones, combinaremos los máximos con sus fingerprints, usando una función **Hash**.

Observando los picos de nuestro espectrograma y combinando las frecuencias máximas junto con su diferencia de tiempo entre ellas, podremos crear un hash, que representa un único fingerprint para ese instante de la canción.

**Hash(freq preaks, time difference between peaks) = fingerprint hash value**

En nuestro caso, al tener en cuenta más valores que un sólo pico, crearemos fingerprints con más entropía, es decir, con más información que hará único al fingerprint y con mayor probabilidad de reconocimiento, y menos de colisión.




Shazam compara estos grupos de picos como una constelación de la señales discretas que se usan en la codificación digital de canal (BPSK, N-PAM, entre otras)

Cuantos más picos tenga el fingerprint, más única será por lo tanto más fácil de reconocer, pero también significa menos robusta frente al ruido.

## 4.8. Entrenando una canción

Este sistema tiene principalmente dos tareas:

1) Entrenar nuevas canciones registrando sus fingerprints
2) Reconocer canciones desconocidas buscándolas en nuestra base de datos de canciones ya parametrizadas y entrenadas.

Para ello, usaremos MySQL como base de datos y tendremos dos tablas: Canciones y Fingerprints.

## 4.9. Tabla Fingerprints

La tabla fingerprints tendrá los siguientes campos:

```
CREATE TABLE fingerprints (
     hash binary(10) not null,
     song_id mediumint unsigned not null,
     offset int unsigned not null,
     INDEX(hash),
     UNIQUE(song_id, offset, hash)
);
```

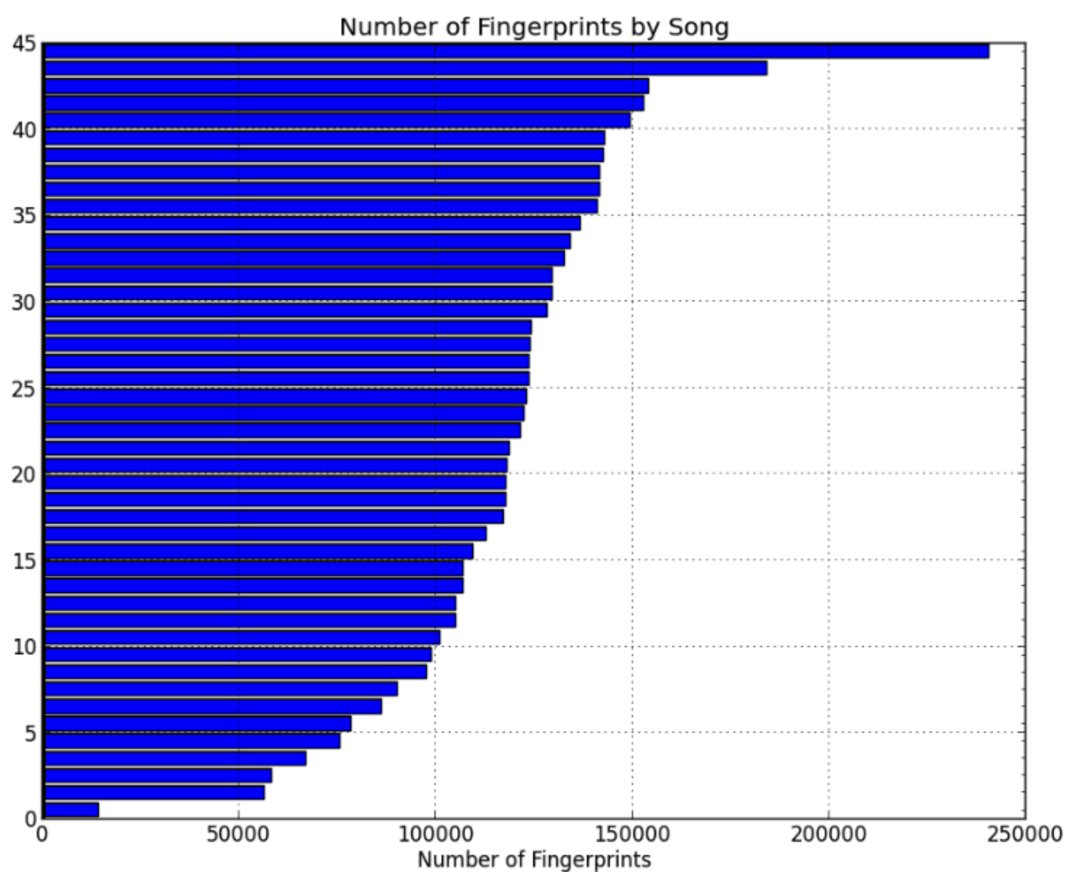
Además de almacenar el hash y el Id de la canción, también tendremos un *offset*. Este valor corresponde a la ventana de tiempo del espectrograma donde se creó el hash.

El valor se usará como filtro cuando queramos reconocer a través de nuestros hashes. Sólo cuando éstos se alinean serán una señal que queremos identificar.

En segundo lugar, hemos hecho un Índice en nuestro hash, ya que nuestras consultas tendrán que coincidir con eso y necesitaremos una recuperación muy rápida.

El Índice hará una función similar al del Id, evitar duplicados.

El campo binario(10) es para el hash y tendremos muchos de estos y deberemos reducir el espacio.




En la parte delantera de la canción que hemos tomado el espectrograma, *Mirrors by Justin Timberlake*, tiene más de 240 mil fingerprints.

Con esta cantidad de fingerprints necesitamos reducir el almacenamiento innecesario en el disco a nivel de valores hash. Para ello, empezaremos usando un hash **SHA-1** (El SHA es una familia de funciones hash publicadas por el Instituto Nacional de Normas y Tecnología, INNT de Estados Unidos) y luego reduciremos a la mitad de su tamaño (sólo los primeros 20 caracteres).
Esto reduce nuestro uso de Bytes por hash a la mitad:

char(40) => char(20) goes from 40 bytes to 20 bytes

Acto seguido, tomaremos esta codificación hexagonal y la convertimos en binario, una vez más reduciendo el espacio considerablemente.

char(20) => binary(10) goes from 20 bytes to 10 bytes

Ahora hemos pasado de 320 bits a 80 bits para el campo de hash, reduciéndolo un 75 %.

### 4.10. Tabla canciones

La tabla *canciones* la usaremos para guardar información seleccionada de las canciones. La necesitaremos para emparejar un identificador de canción con su nombre.

```
CREATE TABLE songs (
    song_id mediumint unsigned not null auto_increment,
    song_name varchar(250) not null,
    fingerprinted tinyint default 0,
    PRIMARY KEY (song_id),
    UNIQUE KEY song_id (song_id)
);
```

El flag del fingerprint será usado internamente para decidir si tomar o no los fingerprints de un archivo. Ponemos el bit a 0 inicialmente y se pondrá a 1 después que el proceso de fingerprints digitales se hayan completado.

## 4.11. Alineación de los fingerprints

Ahora que hemos escuchado una pista de audio, realizado diversas FFT en ventanas superpuestas a lo largo de la canción, extraído los picos y formado sus fingerprints, tocará la alineación de los últimos para el reconocimiento.

Asumiremos que ya tenemos nuestra base de datos parametrizada, nuestro pseudocódigo sería:

```
channels = capture_audio()

fingerprints_matching = [ ]
for channel_samples in channels
    hashes = process_audio(channel_samples)
    fingerprints_matching += find_database_matches(hashes)

predicted_song = align_matches(fingerprints_matching)
```

Para entender el concepto de alineación de los hashes, hay que ser conscientes que cuando registramos un subsegmento de la canción original, los hashes extraídos tendrán un desplazamiento relativo al inicio de la muestra.

En la base de datos, tendremos los hashes de la canción original con el desplazamiento absoluto, si nosotros no grabamos desde el inicio de la canción que queremos reconocer,
esos hashes nunca coincidirían.

Aunque no sean iguales, los almacenados a los que queremos reconocer, ya que podemos tener perturbaciones aditivas de degradación desconocida, sabemos que esas compensaciones relativas estarán a la misma distancia. Para ello debemos suponer que ambas señales de audio (original y a reconocer) están grabadas y muestreadas a una misma velocidad.

Bajo esta suposición, para cada coincidencia calculamos una diferencia entre las compensaciones:

difference = database offset from original track - sample offset from recording

La diferencia siempre será un entero positivo, ya que el seguimiento de la base de datos será por lo menos de la longitud de la muestra.

Todas las verdaderas coincidencias contienen esta misma diferencia, así que, nuestras coincidencias de la base de datos serán alteradas para que es vean así:

(song_id, difference)

Ahora, es mirar las coincidencias y predecir el ID de la canción para el cual cae un mayor número de una diferencia.

## 4.13 Instalación DejaVu

Para instalar el software DejaVu en nuestra máquina debemos ir al repositorio y clonarlo o descargarlo en el directorio que queramos.

Para ello necesitaremos tener MySQL server y Workbench también instalados en nuestra computadora junto a Python 2.X.

Una vez tengamos ambos requerimientos instalados, abriremos el fichero **INSTALLATION.md** y seguiremos los pasos para instalar los paquetes que requiere el sistema.

4.14. Puesta en marcha DejaVu

1) Abrimos el terminal e introduciremos los siguientes comandos:

```
$ mysql -u root -p
Enter password: **********
mysql> CREATE DATABASE IF NOT EXISTS dejavu;
```

2) En el archivo dejavu.cnf.SAMPLE introduciremos los datos de nuestra instancia en MySQL en los parámetros: *host*, *user*, *passwd* y *db*.
3) Abrimos el archivo *example.py*, buscamos donde carga la instancia *djv = Dejavu(config),* y a partir de ahí, debemos crear una variable 'path' donde contenga el path en forma de **string** donde hemos almacenado los ficheros del directorio **Sen** de **PEQE**.
   Creamos un bucle del 000 al 155, que son los directorios que hay dentro, y para cada uno de esos directorios en la variable 'fullpath' se lo pasamos a la instancia cargada anteriormente: *djv.fingerprint_directory(fullpath,[".mp3"])*
   Con este paso, hemos introducido nuestra base de datos en las tablas *fingerprint* y *songs* de nuestra base de datos.
4) Nos creamos un script que recorra los directorios test de los SNR, y para cada uno de los ficheros, lo volvemos a pasar como parametro a la instancia

previamente cargada de la misma forma: *djv.recognize(FileRecognizer,fullpath)* donde *fullpath* es el directorio + nombre archivo + extensión.

Podemos ir almacenando los resultados en un fichero para ver la precisión del mismo.

# 5. Resumen de los resultados

## 5.1 Resultados de nuestro propio sistema

Resultados para la señal limpia:

| SNR (DB) | PRECISIÓN (%) |
|---|---|
| **00** | 9.10 |
| **10** | 65.5 |
| **20** | 92.90 |
| **30** | 97.50 |
| **40** | 99.40 |

Resultados con el sistema entrenado para la señal ruidosa:

| SNR (DB) | PRECISIÓN (%) |
|---|---|
| **00** | 23.20 |
| **10** | 73.60 |
| **20** | 93.90 |
| **30** | 97.70 |
| **40** | 98.90 |

Variación de los niveles del árbol:

8 niveles:

| SNR (DB) | PRECISIÓN (%) | DURACIÓN (SEG) |
|---|---|---|
| **00** | 26.70 | 25.59 |
| **10** | 72.50 | 23.90 |
| **20** | 92.30 | 23.81 |
| **30** | 97.30 | 23.14 |

| **40** | 98.60 | 23.25 |
|---|---|---|

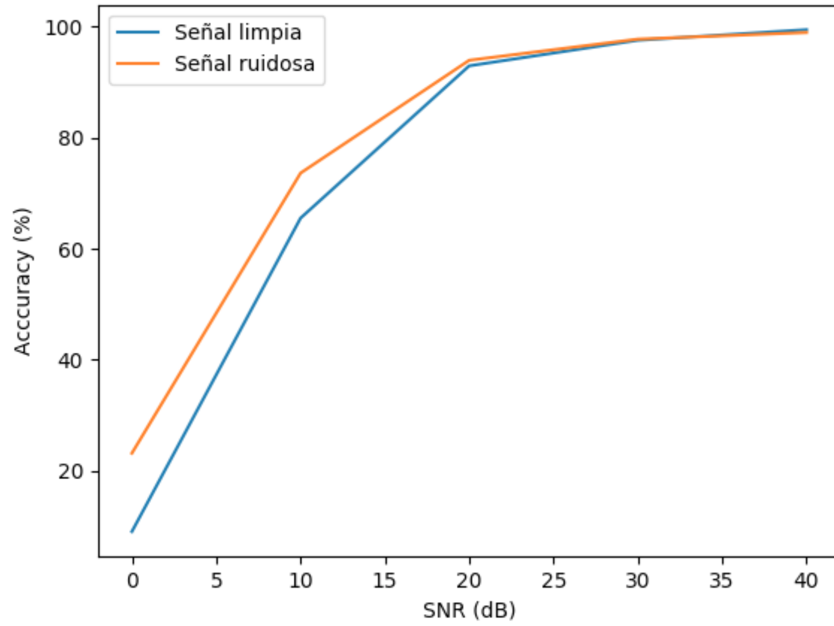
10 niveles:

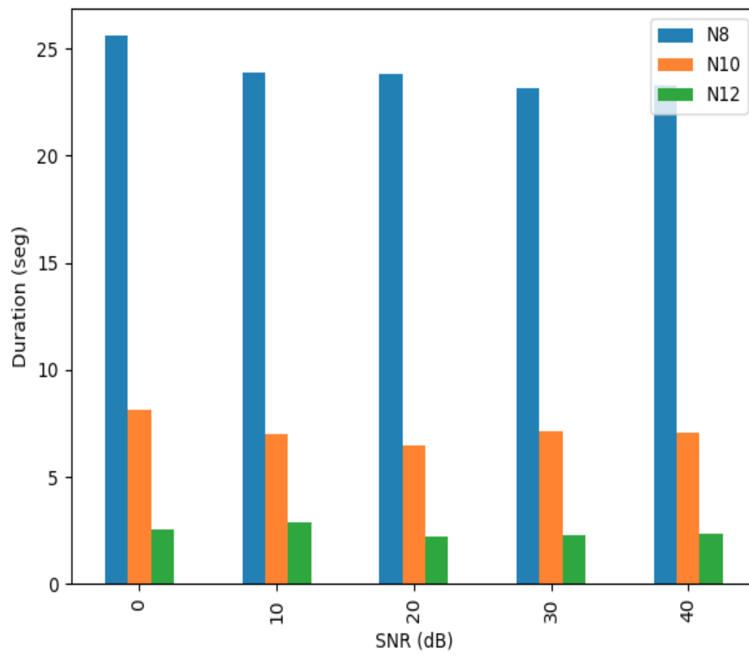
| **SNR (DB)** | **PRECISIÓN (%)** | **DURACIÓN (SEG)** |
|---|---|---|
| **00** | 23.00 | 8.102 |
| **10** | 69.40 | 6.975 |
| **20** | 91.90 | 6.470 |
| **30** | 97.30 | 7.142 |
| **40** | 98.40 | 7.031 |

12 niveles:

| **SNR (DB)** | **PRECISIÓN (%)** | **DURACIÓN (SEG)** |
|---|---|---|
| **00** | 21.20 | 2.523 |
| **10** | 67.20 | 2.901 |
| **20** | 91.40 | 2.232 |
| **30** | 97.10 | 2.297 |
| **40** | 98.20 | 2.325 |

Comparativa de la precisión con su duración:



## 5.2 Resultados DejaVu

Resultados reconocimiento:

| SNR (DB) | PRECISIÓN (%) | DURACIÓN (MIN:SEG) |
|---|---|---|
| **00** | 84.80 | 5:33 |
| **10** | 97.40 | 5:07 |
| **20** | 99.40 | 5:10 |
| **30** | 99.40 | 5:08 |
| **40** | 99.80 | 5:00 |

## 5.3 Conclusiones y programación para la continuidad

1) Árbol binario

En las pruebas realizadas para 3 distintos niveles de árbol, podemos observar como los resultados de precisión y de duración del reconocimiento son fuertemente influidos por la profundidad y la anchura del árbol.
La profundidad implica en mayor toma de decisiones a la hora de clasificar las señales parametrizadas, por lo que un error acarrea menores consecuencias que un árbol menos profundo pero más ancho.
En los resultados compartidos, vemos que la tasa de precisión disminuye considerablemente para las señales con alto nivel de ruido, pero para un nivel alto del SNR ambas funcionan con éxito.
Los tiempos de duración del reconocimiento son muy distintos con tan sólo 4 niveles más, sería conveniente hacer más pruebas con más niveles para alcanzar el nivel óptimo del árbol.

2) Resultados señal limpia vs señal ruidosa

Nuestro sistema si comparamos los datos con *DejaVu*, cuando se trata de señal limpia funcionan ambos con una precisión de casi el 100 %.
Cuando se trata de señal ruidosa, DejaVu sigue trabajando con una alta tasa de precisión mientras que el nuestro cae en picado, el plan de mejora irá en esta dirección.
DejaVu, una de las causas por las que sigue funcionando, es por su algoritmo de trabajar con máximos de amplitud en el espectro, ya que estos son los que mayor probabilidad tienen de sobrevivir al ruido. Si por ejemplo, de 100 picos, 80 están contaminados y 20 han sobrevivido, estos 20 serán suficientes para poder encontrar las coincidencias y alinearlos correctamente.

3) Plan de mejora para las señales ruidosas de nuestro propio sistema de reconocimiento

Una idea de mejora para las señales ruidosas, sería hacer la siguiente clasificación:
En el espectro, tenemos **N** bandas de señal, por **M** tramas dentro, que en total serían **NxM** valores. Estos valores se repartirán de manera aleatoria en 100 grupos (de momento el número de grupos es un número arbitrario, pero habría que investigar si existe el número de grupos óptimo). En cada uno de los grupos podríamos hacer una especie de *distancia euclidia*, osea que en total tendríamos 100 distancias euclidias. Tendríamos 100 reconocimientos incompletos, y la esperanza estaría en que la verdadera coincidencia será la canción que aparezca en más grupos.
Veamos el diagrama en la siguiente figura:

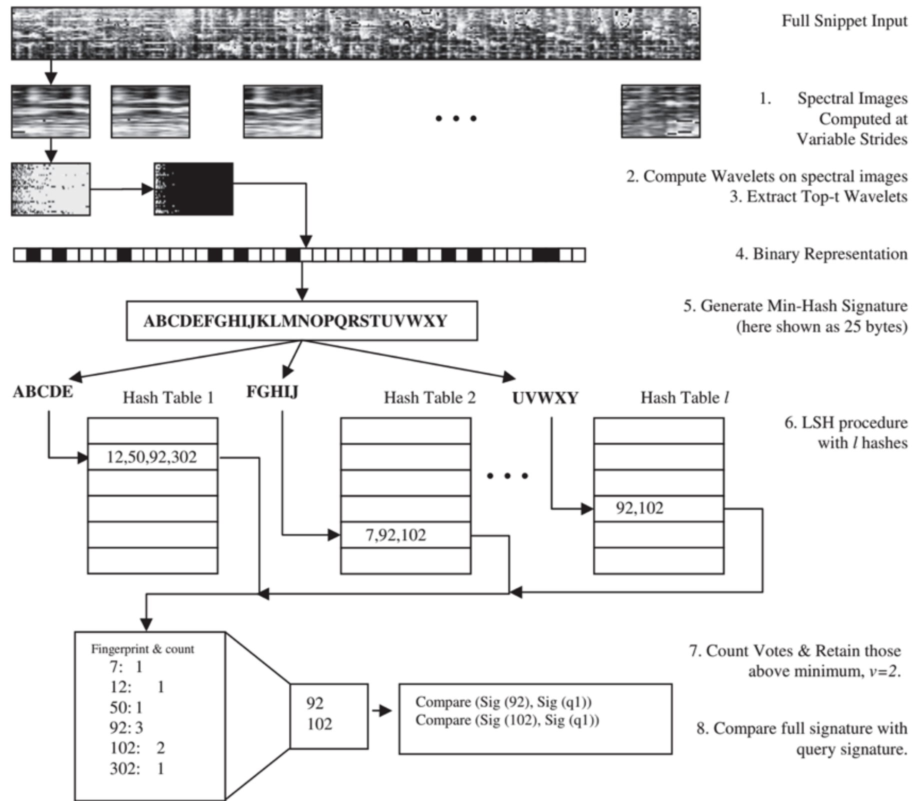



**Fig. 5.** Overall architecture of the retrieval process. Dynamic-time warping not shown.

## 5.4 Bibliografía

[1] Documentación oficial de Microsoft, disponible: "https://code.visualstudio.com/docs"

[2] Wikipedia acerca de Python 3: "https://es.wikipedia.org/wiki/Python"

[3] Módulo NumPy: "https://es.wikipedia.org/wiki/NumPy"

[4] Documentación oficial para el módulo docopt: "https://pypi.org/project/docopt/"

[5] Documentación oficial para el módulo h5py: "https://www.h5py.org"

[6] Módulo struct, **struct - Estructuras binarias de datos**, por Ernesto Rico Schmidt, 2018 - 2019 "https://rico-schmidt.name/pymotw-3/struct/"

[7] Módulo SciPy: "https://es.wikipedia.org/wiki/SciPy"

[8] Base de datos: "https://freemusicarchive.org/genre/Electronic" powered by Tribe of Noise Johan Huizingalaan.

[9] Dataset FMA: "https://arxiv.org/abs/1612.01840v1" por Kirell Benzi, Michaël Defferrard, Pierre Vandergheynst, Xavier Bresson el 06 de Diciembre de 2016.

[10] GitHub base de datos por Michaël Defferrard: "https://github.com/mdeff/fma", dónde la última actualización ha sido realizada en Junio de 2020.

[11] Escala de Mel: "https://es.wikipedia.org/wiki/Escala_Mel"

[12] Artículo DejaVu, con Will Drevo como autor, 5 Noviembre 2013: "https://willdrevo.com/fingerprinting-and-audio-recognition-with-python/"

[13] GitHub DejaVu: "https://github.com/worldveil/dejavu" propiedad de Will Drevo.