Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Desarrollo e implementación de algoritmos para escalado de melodías flamencas

Autor: Miguel García Carrasco

Tutor: José Miguel Díaz Báñez

Dpto. de matemática aplicada
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Desarrollo e implementación de algoritmos para escalado de melodías flamencas

Autor:
Miguel García Carrasco

Tutor:
José Miguel Díaz Báñez
Catedrático de universidad

Dpto. de matemática aplicada
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Grado: Desarrollo e implementación de algoritmos para escalado de melodías flamencas

Autor: Miguel García Carrasco
Tutor: José Miguel Díaz Báñez

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 tutor, el Dr. José Miguel Díaz-Báñez, por contagiar su pasión por fusionar las matemáticas, la computación y el flamenco.

A Fabio Rodríguez, por estar siempre dispuesto a echar un cable con python o con lo que haga falta.

A cada profesor de la escuela del que pude aprender y contribuyó en mi formación.

A cada compañero que me crucé en la carrera y los buenos momentos vividos durante estos años.

Por último, a mi familia y amigos por estar en el día a día haciendo esto posible.

Miguel García Carrasco
Sevilla, 2021

Resumen

El estudio de las medidas de la similitud melódica es de vital importancia en los sistema de recuperación de información musical. En este proyecto usamos técnicas de concordancia geométrica para medir la similitud entre dos melodías. Proponemos un algoritmo eficiente para un problema de optimización inspirado en la operación melódica del escalado lineal.

En el problema de escalado, una melodía de consulta es escalada hacia adelante hasta que la similitud melódica entre la consulta y la referencia sea mínima. La métrica de similitud melódica utilizada es el área comprendida entre dos contornos melódicos.

Abstract

Melodic similarity is of key importance in Music Information Retrieval. In this project, we use geometric matching techniques to measure the similarity between two melodies. We propose an efficient algorithm for an optimization problem inspired in the operation of linear scaling on melodies. In the scaling problem, an incoming query melody is scaled forward until the similarity measure between the query and the reference melody is minimized. The similarity measure used is the area between two melodic contours.

Índice Abreviado

Índice

1 Introducción

El presente documento corresponde a la memoria del trabajo de fin de grado del grado de ingeniería de las tecnologías industriales (GITI) cursado en la Universidad de Sevilla (US). En concreto está en el marco del proyecto COFLA (COMPUTATIONAL ANALYSIS OF FLAMENCO MUSIC), proyecto del que hablaremos a continuación.

1.1 Alcance y motivación

El flamenco es sin lugar a duda una de las manifestaciones culturales más importantes de nuestro país, así como el principal referente de nuestra cultura a nivel internacional. Con origen andalúz y desarrollado en esta tierra, siempre ha sido una expresión artística de tradición fundamentalmente oral, al tratarse en sus orígenes de una manifestación popular. Este hecho, unido a que los expertos en música flamenca no han seguido habitualmente una formación académica tradicional, ha provocado un escaso acercamiento del método científico a cualquier ámbito relacionado con el flamenco, por lo que la cantidad y calidad de la investigación existente en torno al flamenco es limitada.

Debido a la escasa investigación científica en la musicología del flamenco, se precisan diferentes fases de actuación para su puesta en valor en la comunidad científica, que abarcan desde la creación de un corpus de transcripciones o anotaciones musicales significativo, hasta el diseño de algoritmos eficientes para calcular distancias entre dos melodías. Entre los aspectos fundamentales que demandan avances en este área científica pueden destacarse: los orígenes del flamenco, evolución y relación entre los distintos estilos, propiedades de preferencia de estilos, influencia de músicas externas a Andalucía o búsqueda de estilos ancestrales.

La necesidad de un estudio riguroso y metódico de este arte a través de las matemáticas y las ciencias de la computación es el origen del nacimiento del proyecto COFLA[1], liderado por José Miguel Díaz Báñez y financiado por la junta de Andalucía, proyecto que ha cimentado el área de investigación *Teoría Computacional del Flamenco* y marco donde se desarrolla este proyecto, con la intención de aunar los maravillosos mundos de la música, y la ciencia, en concreto el flamenco con las matemáticas y las ciencias de la computación.

La industria de la música fue una de las primeras en verse completamente reestructurada por los avances de la tecnología digital, y hoy en día tenemos acceso a miles de canciones almacenadas en nuestros dispositivos móviles y a millones más a través de servicios en internet. Dada esta inmensa cantidad de música a nuestro alcance, necesitamos nuevas maneras de describir, indexar, buscar e interactuar con el contenido musical. Desgraciadamente, las aplicaciones informáticas existentes en la actualidad no suelen ser útiles para el contexto musical flamenco, por lo que es necesario adaptar y diseñar nuevos algoritmos que abran las puertas del mundo tecnológico a la música flamenca.

Las técnicas computacionales y modelos matemáticos utilizados en el área de investigación de la Teoría Computacional del flamenco, pertenecen al campo de la Music Information Retrieval (MIR), ciencia interdisciplinar que se dedica a la recuperación de información de la música. MIR es un campo de conocimiento en

[1] http://www.cofla-project.com/

crecimiento que involucra a distintas ramas de conocimiento como pueden ser: la psicología, la formación musical académica, tratamiento de señal, algoritmia, etc. Algunas de las aplicaciones del MIR son:

- Sistemas de recomendación musical.
- Separación y reconocimiento de instrumentación.
- Transcripción automática.
- Caracterización automática.

Desde los comienzos de MIR como campo de conocimiento, la mayoría de las técnicas y modelos han sido desarrollados para una corriente particular de la música tradicional, la conocida como música popular de tradición occidental. En los últimos años ha surgido un especial interés en aplicar las técnicas desarrolladas por el MIR para el estudio de otras músicas tradicionales, folk o música étnica (ISMIR)[2], ya que etnomusicólogos, musicólogos y otros profesionales de la música podrían beneficiarse considerablemente de esta investigación. Aunque las técnicas computacionales han demostrado ser de gran interés cuando se aplica a diferentes repertorios musicales, es evidente que se deben desarrollar técnicas y algoritmos matemáticos específicos para comprender, modelar y procesar diferentes repertorios musicales.

Debido a que el flamenco es una música de tradición y difusión oral, está fuertemente ligada a la cultura del entorno donde nace, en este caso la historia y cultura andaluza. Por ello, no debe realizarse una investigación científica ajena al contexto cultural y las implicaciones que esto supone en el desarrollo y sentido de la música flamenca. De hecho, existe un interés reciente por el estudio tecnológico interdisciplinar de músicas de tradición oral, lo que ha llevado a la consideración de una nueva área de investigación conocida como Etnomusicología Computacional. La Etnomusicología Computacional se remonta a 1978 [9] cuando Halmos, Köszegi y Mandler, dos matemáticos y un ingeniero proporcionaron una interesante reflexión sobre el papel de los ordenadores en las cinco principales áreas de investigación en Etnomusicología: recopilación de datos, administración, notación, selección y sistematización, y tratamiento científico. El término fue redefinido por Tzanetakis [20] en 2007 como "el diseño, el desarrollo y la utilización de herramientas informáticas que tienen el potencial de ayudar en la investigación etnomusicológica". Con este enfoque se ve a la etnomusicología como la base científica que se sirve de los avances en otras disciplinas de apoyo tales como Matemáticas Computacionales, Musicología o estudios de carácter cultural. Cuando nos acercamos a la Etnomusicología Computacional de esta manera, se adopta un nuevo marco mental que ayuda a reestructurar problemas y percibir las relaciones entre sus elementos bajo una perspectiva diferente.

1.2 Objetivo y aplicaciones

Muchos problemas relacionados con la teoría musical son fundamentalmente geométricos por naturaleza, esto es, miden alguna característica que contiene el corpus musical . Pongamos un ejemplo: dos melodías pueden ser representadas por polígonos ortogonales (funciones escalón) y una posible medida de similitud es el área comprendida entre las dos curvas correspondientes a cada canción (permitiendo traslaciones verticales u horizontales, compresiones o escalado de melodías, etc). De esta forma, el problema se traslada al campo de la computación, donde coincide con el problema de emparejamiento de formas poligonales [1].

Este proyecto toca un problema fundamental del Análisis Computacional de la Música: la similitud melódica. La trascendencia de la similitud melódica ha sido, y es, tal que numerosas disciplinas se han ocupado intensivamente de su estudio:

- En Etnomusicología, por ejemplo, para entender la lógica musical, para evaluar los estilos y sus características, para conocer los criterios de improvisación [2][19][10].
- En Análisis Musical, para construir modelos tanto teóricos como computacionales [13].
- En la resolución de conflictos de propiedad intelectual [6].
- En Tecnología Musical, para aplicar los modelos obtenidos tras el correspondiente análisis [10].
- En Psicología de la música, para comprender mejor el hecho musical, para aportar conocimiento a un análisis integral de la música [8].

[2] The International Society for Music Information Retrieval (www.ismir.net)

La investigación existente en similitud musical está centrada fundamentalmente en música occidental y en los sistemas actuales de recomendación es esta música la que se encuentra etiquetada. Sin embargo, existe un creciente interés en el campo para analizar y etiquetar músicas tradicionales y folclóricas.

En este proyecto trabajaremos en el contexto de la música tradicional más importante en Andalucía y, que posee además, gran interés por todo el mundo. La música flamenca es fundamentalmente de tradición y difusión oral, lo que dificulta, y hace más interesante a su vez, su estudio científico. La conservación de esta música de generación en generación hace que la melodía juegue un papel crucial en la evolución y clasificación de los distintos estilos o palos del flamenco, siendo éste uno de los problemas abiertos más apasionantes de este área de investigación. Puesto que el cante flamenco no ha sido llevado a partituras (al menos hasta ahora no existe consenso sobre la transcripción del cante flamenco), trabajaremos con una base de datos de melodías de cantes a capela (corpus tonás), que veremos con detalle en el capítulo 4.

Teniendo en cuenta lo expuesto en los anteriores párrafos, en este trabajo de fin de grado pretendemos dar un nuevo paso que permita realizar una aproximación al flamenco a través del uso de técnicas computacionales con objeto de analizar con más profundidad y con rigor científico este arte milenario. El objetivo de este trabajo será la resolución e implementación del problema de escalado lineal de una melodía, basado en el algoritmo propuesto en el trabajo [4]. En concreto, vamos a considerar dos estilos flamencos que corresponden al grupo de cantes a capela del flamenco, las deblas y los martinetes. Habría que aclarar que aunque en este proyecto se realizan los experimentos computacionales con estos cantes a capella, podrían llevarse a cabo igualmente con otros estilos musicales.

2 Definición del problema

La musicología y los estudios computacionales de similitud rítmica y melódica han dado lugar a numerosos problemas geométricos. Una melodía se puede representar como una secuencia consecutiva de notas y cada nota se puede representar por un punto o una línea horizontal en el plano bidimensional. En este trabajo estudiamos uno de los problemas que surgen de la recuperación de información musical (MIR): el escalado lineal. Dada una melodía de referencia, una melodía de consulta y una medida de similitud, en el problema de escalado la melodía de consulta es escalada en la dirección horizontal para encontrar el mínimo de la medida de similitud entre las melodías de consulta y referencia. El estudio de medidas para la similitud melódica es de vital importancia en un sistema de recuperación de información musical. Técnicas basadas en medidas de similitud geométrica han sido ampliamente utilizadas en la literatura. La ventaja de utilizar concordancia geométrica es su alta tasa de acierto, mientras que su desventaja es que consume mucho tiempo de computación. En [15], la siguiente técnica de concordancia geométrica fue propuesta: Cada nota es representada por un segmento horizontal de manera que la secuencia de notas puede describir un contorno rectangular en un sistema de coordenadas de dos dimensiones, en el cual los ejes horizontal y vertical corresponden respectivamente al tiempo y la altura o tono. Entonces la medida de similitud es el área mínima que hay entre dos contornos melódicos.

2.1 Ingredientes del problema

En este trabajo, las melodías son representadas por una secuencia de notas en un plano bidimensional cuyos ejes son el tiempo (eje horizontal) y la altura o tono (eje vertical). Cada nota es descrita por una línea horizontal donde su altura y su longitud denotan el tono y la duración, respectivamente. En esta representación, la melodía es una secuencia de segmentos horizontales, es decir, una melodía se puede ver como una función de escalón en el tiempo.

Resaltar que gracias a la propiedad de invarianza de este tipo de representación, solo hay necesidad de mover la melodía de consulta en el eje del tiempo para el problema de escalado. Sean por tanto $R = (R_1, R_2, \ldots, R_n)$ y $Q = (Q_1, Q_2, \ldots, Q_m)$ las secuencias que representan las dos melodías, con $m < n$. R es la melodía de referencia de una base de datos y Q la melodía de consulta de la cual se quiere averiguar la concordancia. Los elementos de R y Q son segmentos horizontales.

Los ingredientes principales del problema, introducidos a continuación, son la medida geométrica que utilizamos para evaluar la similitud y operación que le realizamos a la melodía de consulta:

- **Área:** la región comprendida entre las dos melodías con la misma duración en la representación de línea puede ser dividida en rectángulos con bordes verticales soportados por líneas verticales que pasan por el final de los segmentos. El área entre las dos melodías se define como la suma de las áreas de cada región en las que han sido divididas. Para la melodía más corta, se extiende la última nota hasta igualar las longitudes. Ver figura 2.1
- **Escalado lineal:** consideremos la representación bidimensional de las melodías R y Q. Sean $X = (x_0 = 0, x_1, x_2, \ldots, x_n)$ y $T = (t_0 = 0, t_1, t_2, \ldots, t_m)$ las divisiones temporales de la representación de R y Q respectivamente. Dado $\varepsilon > 0$, se define el ε-escalado de la melodía de consulta Q, $S_Q(\varepsilon)$, como la operación de incrementar por ε la duración de cada segmento de Q pero manteniendo fijado

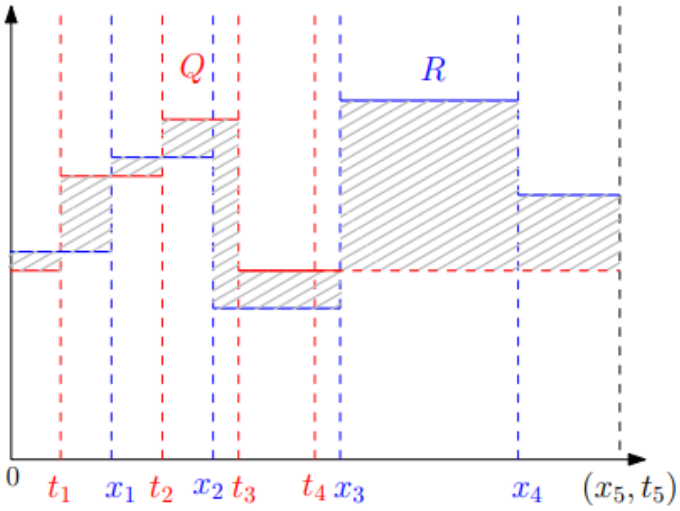


Figura 2.1 Área entre 2 melodías genéricas.

el inicio de la melodía Q. Entonces, tras un ε-escalado, X no se transforma y T es transformada a $T+\varepsilon = (t_0, t_1+\varepsilon, t_2+2\varepsilon, \ldots, t_m+m\varepsilon)$. La melodía de consulta puede ser escalada hasta que ambas melodías tengan la misma duración. Entonces, $0 \leq \varepsilon \leq \frac{x_n - t_m}{m}$.
Nótese que la altura de las notas no se ve afectada por el problema de escalado.

Definición del problema: **Dados dos contornos melódicos R y Q, calcular cuál es el valor de $\varepsilon > 0$ que minimiza el área entre R y $S_Q(\varepsilon)$**

2.2 Escalado usando el área como medida de la similitud

Dado que la duración de Q es menor que la de R, extendemos el último segmento de Q de forma que las duraciones de Q y R sean iguales. Así, el área entre R y Q es la suma de $O(m+n)$ rectángulos tal y como se ilustra en la figura 1. Denotamos $A_{RQ}(\varepsilon)$ al área entre R y $S_Q(\varepsilon)$ como función de ε. Obsérvese que tras el ε-escalado de la consulta para un ε lo suficientemente grande, al menos dos de los límites verticales coincidirán, esto es, $x_i = t_j(\varepsilon)$ para algunos i, j distintos. En este instante, hay un rectángulo que desaparece, y después de ese instante aparecerá uno nuevo. A este valor de ε se le llama *evento*. Hay que destacar también que, entre dos eventos, las áreas de algunos rectángulos se incrementan, otras disminuyen y otras no se ven afectadas. De hecho, el tipo de rectángulo puede ser determinado por sus límites verticales. Los rectángulos pueden ser clasificados en 4 tipos: Tipo C_0, con límites verticales $[x_i, x_{i+1}]$ (rectángulo estático, azul); tipo C_1, con límites verticales $[t_j, t_{j+1}]$ (rectángulo rojo en aumento); tipo C_2, con límites verticales $[x_i, t_j]$ (rectángulo en aumento)y tipo C_3, con límites verticales $[t_j, x_i]$ (rectángulo en disminución); La figura 2.2 ilustra cómo desaparece un rectángulo tipo C_3 al producirse un evento.

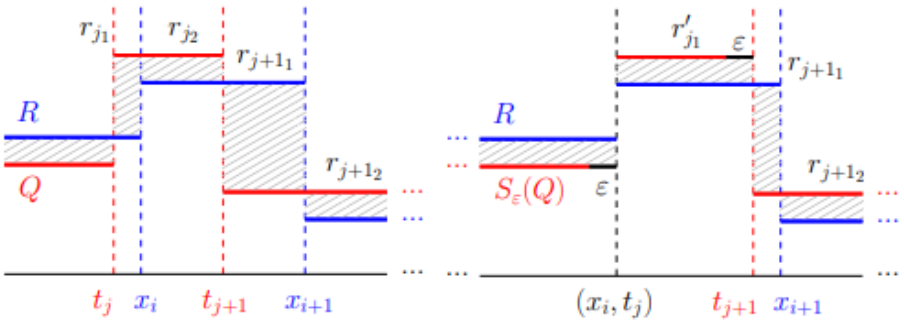


Figura 2.2 Tras realizar un escalado a Q, se produce un evento en el que desaparece un rectángulo tipo C3.

Los siguientes resultados nos permiten discretizar el problema:

Lemma 1. $A_{RQ}(\varepsilon)$ es una función lineal definida a trozos.

Demostración. Sea r_{j_k} el k-ésimo rectángulo generado por el j-ésimo segmento de Q tal y como ilustra la figura 2.2. Dado que en la operación de escalado solo cambia el ancho de los rectángulos, denotemos por b_{j_k} y h_{j_k} la base y la altura de r_{j_k}, respectivamente. Usando esta notación, el área entre dos eventos consecutivos se puede expresar como:

$$A_{RQ}(\varepsilon) = \sum_{r_{j_k} \in C_0} b_{j_k}(\varepsilon) h_{j_k} + \sum_{r_{j_k} \in C_1} b_{j_k}(\varepsilon) h_{j_k} + \sum_{r_{j_k} \in C_2} b_{j_k}(\varepsilon) h_{j_k} + \sum_{r_{j_k} \in C_3} b_{j_k}(\varepsilon) h_{j_k}. \tag{2.1}$$

Ahora, sea $\varepsilon' < \varepsilon''$ dos movimientos de escalado entre dos eventos consecutivos. Solo cambia la anchura del rectángulo, y el área se ve modificada por:

$$A_{RQ}(\varepsilon'') = A_{RQ}(\varepsilon') + (\varepsilon'' - \varepsilon') \left(\sum_{r_{j_k} \in C_1} h_{j_k} + \sum_{r_{j_k} \in C_2} j h_{j_k} - \sum_{r_{j_k} \in C_3} (j-1) h_{j_k} \right). \tag{2.2}$$

De esta forma, el área entre las dos melodías es lineal entre dos eventos consecutivos. ■

Como consecuencia del Lemma 1, el área mínima se alcanza en los eventos y nuestro objetivo es reevaluar el área en cada evento (hay $O(nm)$ eventos en el peor de los casos) y quedarse con el mínimo valor.

Lemma 2. $A_{RQ}(\varepsilon)$ entre dos eventos consecutivos puede ser actualizada en tiempo $O(1)$.

Demostración. Sean ε_i, ε_{i+1} dos eventos consecutivos. Usando la ecuación (2.2) se puede calcular $A_{RQ}(\varepsilon_{i+1})$ a partir de $A_{RQ}(\varepsilon_i)$ actualizando el tipo de rectángulo modificado. Nótese que solo hay 4 cambios posibles para dos notas de Q involucradas en el evento:

- Un rectángulo C_2 cambia a un C_0 y crea otro C_2
- Un rectángulo C_1 cambia a un C_3 y crea otro C_2
- Un rectángulo C_0 cambia a un C_3 y crea otro C_3
- Un rectángulo C_2 cambia a un C_1 y borra un C_3 (ver figura 2.2) ■

Entonces, la suma en el evento dado por ε_{i+1} puede recalcularse actualizando la altura de los rectángulos modificados. Dado que como mucho solo 3 rectángulos habrán cambiado, se puede computar en $O(1)$ y el resultado está probado.

Estamos listos para resolver eficientemente el problema del escalado lineal.

Teorema 3. El problema de escalado puede ser resuelto en tiempo $O(nm \log m)$.

Demostración. Primero, para cada segmento de la consulta Q, se guardan sus eventos en una lista ordenada. A cada evento de la lista le corresponde a un valor de épsilon, una altura y el tipo de rectángulo del que se trata. Tenemos en total m listas. Dado que la entrada R es una secuencia ordenada de segmentos, cada lista puede ser computada en $O(n)$, así que nos toma $O(nm)$ para las m listas.

Por otra parte, dada un área para un $\varepsilon = 0$, $A_{RQ}(0)$, construimos un montículo (más comúnmente conocido por su anglicismo, *heap*) cuyos nodos son los m eventos, el primero para cada lista. Desplazando de izquierda a derecha la melodía de consulta usamos las listas para mantener en la pila el siguiente evento y actualizar el área de cada evento. El deslazamiento acaba cuando no quedan eventos en la lista y el área mínima se ha mantenido en el proceso. Dado que actualizar el área toma un tiempo $O(1)$ por el lemma 2, el algoritmo en total consume tiempo $O(mn \log m)$. ■

3 Algoritmo e Implementación

En este capítulo desarrollamos en detalle el algoritmo que proponemos para resolver el problema del escalado lineal y mostramos su implementación. En esencia el problema que queremos resolver es averiguar cuánto se parecen dos melodías entre sí, y en concreto qué modificación hay que hacerle a una de las dos melodías para que sea lo más parecida posible a la otra.

Nuestro algoritmo requiere en esencia dos entradas, que son las dos melodías Q y R, siendo respectivamente la melodía de la cual queremos extraer la información y la referencia con la que queremos contrastar nuestra consulta. Nuestro objetivo es por tanto obtener el valor de ε que minimiza el área entre las dos melodías, es decir el valor del escalado que hay que realizarle a la consulta Q para que se parezca lo máximo posible a la referencia R.

Una manera ampliamente extendida de representar gráficamente un algoritmo o proceso es mediante un diagrama de flujo. En la figura 3.1 podemos apreciar el diagrama de flujo de nuestro algoritmo, que ayuda a analizar las directrices generales que vamos a seguir a lo largo del algoritmo.

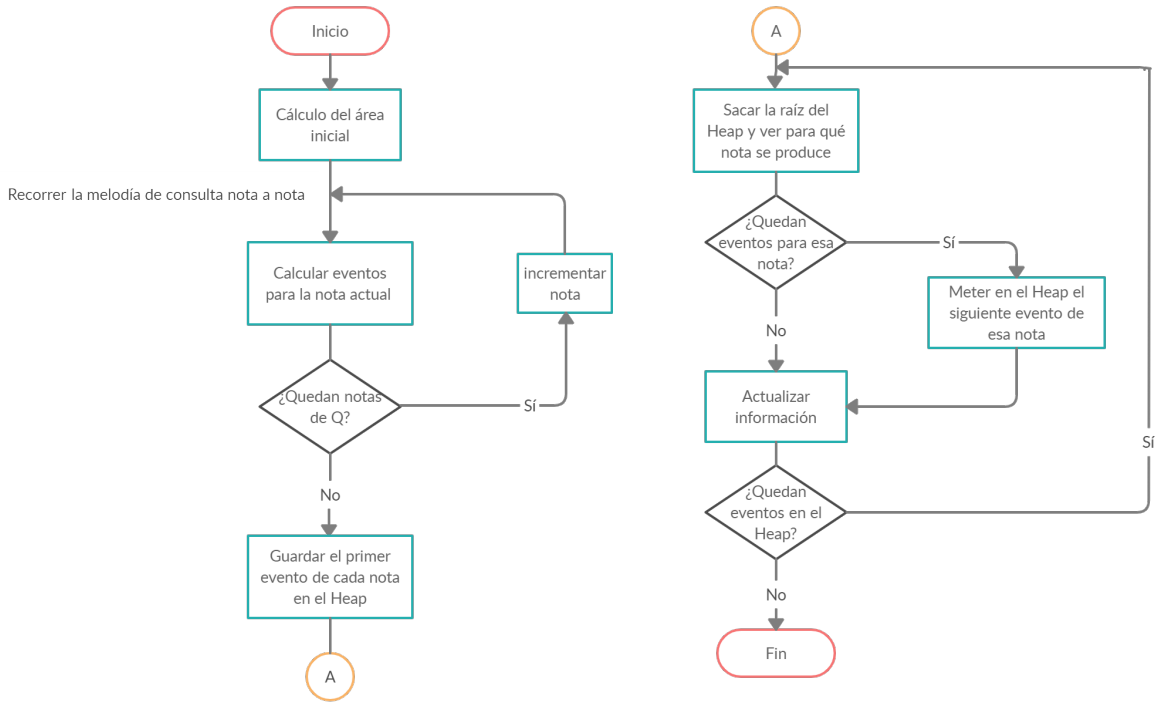


Figura 3.1 Diagrama de flujo general del algoritmo de escalado.

La estrategia que vamos a seguir para abordar el problema está dividida en 4 estapas o pasos principales, los cuales son independientes entre sí para su resolución individual, y que necesitan información secuencial del bloque que le precede.

- En primer lugar vamos a calcular el área encerrada por los dos contornos melódicos en el instante inicial, antes de realizarle ninguna modificación a la melodía de consulta Q. De esta manera obtendremos un estado inicial del cual partir para poder beneficiarnos del resultado obtenido en el lemma 2 definido en el capítulo 2.

- En segundo lugar procedemos a calcular todos los eventos que se pueden producir entre las melodías Q y R. Para ello vamos a ir iterando ordenadamente por cada una de las notas de Q, comprobando qué evento tiene cada nota de Q con cualquiera de las notas de R.

- En el tercer paso vamos a construir un heap en el cual vamos a guardar el primer evento que tenga cada nota de Q, en el cual el nodo raíz va a ser el evento más pequeño.

- En el cuarto y último paso vamos a obtener el resultado o salida que buscamos. Para ello vamos a ir actualizando el área de forma ordenada de menor a mayor valor de ε, por cómo está construido el heap. Cada vez que se saca un evento del heap hay que comprobar para qué nota se ha producido dicho evento y comprobar si a dicha nota le quedan aun más eventos. En caso afirmativo habrá que insertarlo en el heap. Con el heap reconstruido y el área actualizada podemos comprobar si esta nueva área es la menor hasta el momento. En caso afirmativo vamos actualizando el valor del menor área hasta el momento y su respectivo ε.

El lenguaje de programación escogido para la implementación ha sido Python. Python es un lenguaje de programación interpretado, multiparadigma, dinámico y multiplataforma cuya filosofía hace hincapié en la legibilidad de su código. Lo describimos:

- Lenguaje interpretado: el intérprete de comandos permite mayor facilidad a la hora de realizar pruebas y detectar errores a la hora de probar y depurar código frente a sistema compilados que necesitan del compilador para que haga la "traducción" de código fuente a código máquina
- Multiparadigma: permite tanto programación secuencia como OOP (programación orientada a objeto, por sus siglas en inglés), recursos de los cuales vamos a hacer uso en nuestro programa
- Dinámico: no es necesario indicar de qué tipo de dato se trata cada vez que se declara una nueva variable, lo cual facilita el desarrollo de software sin tener que estar lidiando con peculiaridades del lenguaje
- Multiplataforma: esta propiedad facilita mucho la portabilidad del código ente distintos sistemas operativos, pues existe intérpretes para Unix, Linux, Windows y sistemas Mac Os

Python fue desarrollado por Guido van Rossum a finales de los 80 de forma gratuita y su licencia pertenece a Python Software Foundation License, una organización sin ánimo de lucro.

El IDE (Integrated Development Enviroent o entorno de desarrollo integrado) escogido ha sido Spyder (Scientific Python Development Environment). Se trata de un potente entorno de desarrollo interactivo para el lenguaje Python proporcionado por Anaconda. Anaconda es una Suite de código abierto que abarca una serie de aplicaciones, librerías y conceptos diseñados para el desarrollo de la Ciencia de Datos.

3.1 Entradas y salidas

A la hora de afrontar un problema de programación es de vital importancia tener claro de qué datos de entrada vamos a disponer, y qué se espera recibir tras haber tratado estos datos, pues esto condicionará absolutamente el código a desarrollar.

En nuestro caso, sabemos que las entradas van a ser las dos melodías, Q y R. Cada una de ellas va a llegar como una lista de listas. Una lista en Python es un tipo de dato que nos permiten almacenar cualquier tipo de valor en ella como enteros, cadenas y hasta otras funciones. Por tanto, cada melodía vendrá representada como una lista, y cada elemento de esas listas corresponderá a cada una de las notas de la melodía, la cual es a su vez una lista.
Las notas quedan totalmente identificadas con 3 valores. El primero es el instante de comienzo de dicha nota, el segundo es el valor del instante en el que acaba la nota y 3 el tercero es su respectiva altura o tono. Por lo general estos valores van a ser tipo float. Con esta información disponemos de todo lo necesario para realizarle el escalado a la consulta Q, contrastarla con la referencia R y encontrar el valor de ε que minimiza el área entra ambas.

Como salidas vamos a proporcionar una lista que contiene una pareja de valores: el área mínima que existe entre las melodías tras escalar Q, y el respectivo valor de ε que minimiza el área.

Vemos a continuación en la tabla 3.1 un ejemplo de valores reales de entradas y salidas:

Tabla 3.1 Ejemplo de entradas y salidas para dos melodías R y Q.

Entradas	R	[[0, 1.5, 120], [1.5, 4, 500], ..., [21.876, 22.10, 700]]
	Q	[[0, 3.8, 300], [3.8, 7.25, 900], ..., [15.54, 18.32, 550]]
Salidas	*Result*	[3655.0, 0.5]

3.2 Pseudocódigo

Una manera de expresar los distintos pasos que va a seguir un programa o algoritmo es mediante pseudocódigo. El peseudocódigo corresponde a una etapa intermedia entre el diagrama de flujo y la codificación formal, por lo que es agnóstica del lenguaje de programación escogido para su implementación. Para ello se utiliza un lenguaje convencional (español, inglés) pero el problema se estructura de manera similar a un lenguaje de programación. Su principal función es la de representar paso a paso un algoritmo, de la forma más detallada posible, utilizando un lenguaje cercano al de programación.

Partiendo por tanto del diagrama de flujo del capítulo 3 (ver figura 3.1)), vamos a traducirlo a pseudocódigo con algo más de detalle (ver figura 3.2) antes de entrar en la programación e implementación en Python.

3.3 Complejidad y coste computacional

La teoría de la complejidad computacional o teoría de la complejidad informática es una rama de la teoría de la computación que se centra en la clasificación de los problemas computacionales de acuerdo con su dificultad inherente, y en la relación entre dichas clases de complejidad.

La catalogación de un problema como “más” o “menos” difícil no depende del algoritmo utilizado, si no de la cantidad de recursos computacionales que requiera para su implementación. La teoría de la complejidad computacional introduce los modelos matemáticos necesarios para para cuantificar el coste de estos recursos, como tiempo y memoria, y la relación entre ellos.

Según la notación de Landau, la función f pertenece a la clase de complejidad de g (en símbolos, $f \in O(g)$) si existe un c y un n_0 tales que para todo $n \geq n_0$ se tiene que $|f(n)| \leq c|g(n)|$, ignorando los factores constantes y considerando únicamente el término más importante o de mayor orden.

La propiedad más importante de la que vamos a hacer uso es de la siguiente:

- $O(f+g) = O(max(f,g))$

Lo cual supone que la complejidad total del problema viene impuesta por la mayor complejidad que tenga cada una de las funciones o partes del algoritmo. Por ello vamos a desgranar la complejidad computacional de cada una de las partes, las cuales dependen de los datos de entrada.

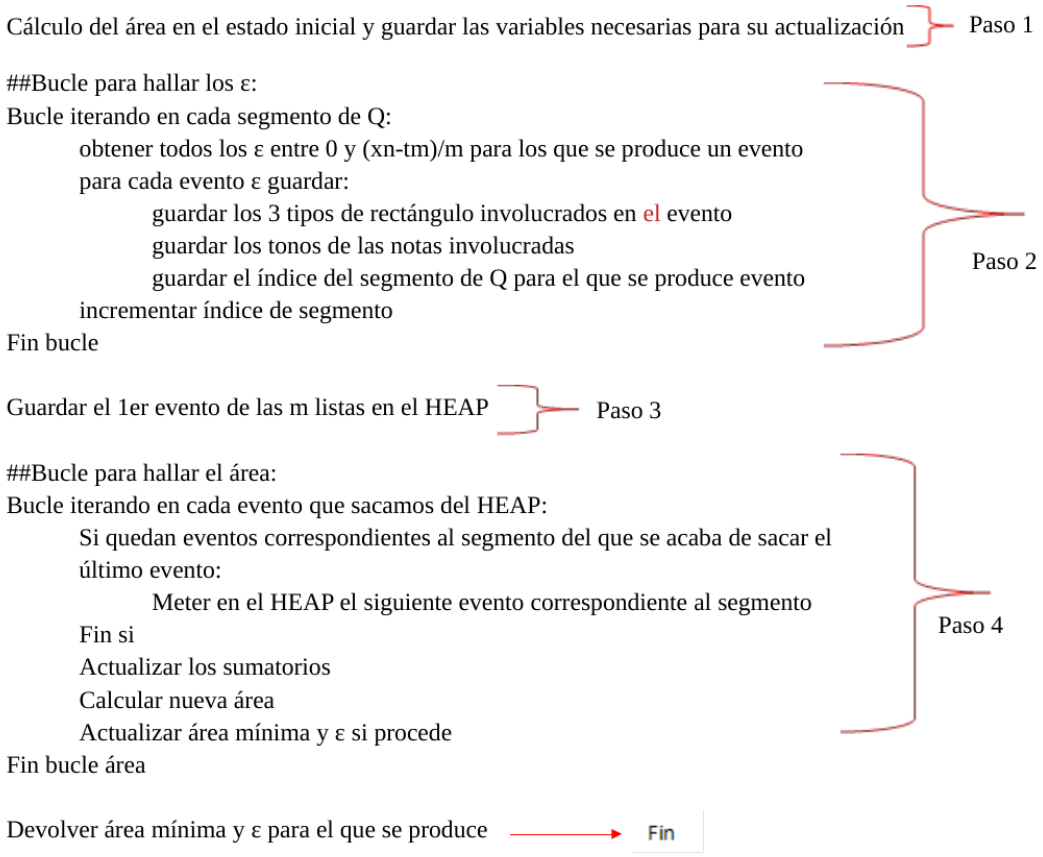


Figura 3.2 Pseudo-código de la implementcaión del algoritmo eficiente.

Nuestras entradas son dos listas, R y Q, de longitudes n y m respectivamente.

En el primer paso, para el cálculo del área, necesitamos recorrer ambas listas de forma que el orden es lineal de magnitud $n+m$, $O(n+m)$.

En el segundo paso necesitamos encontrar los eventos que tiene cada nota de Q con las notas de R. Esto supone recorrer la lista R tantas veces como notas tenga Q, es decir, $n*m$ veces como máximo. Deducimos por tanto que el orden del segundo paso es cuadrático ($O(nm)$).

En el apartado anterior hemos creado una lista que guarda los eventos que tiene cada nota de Q (con una longitud de m elementos por tanto). Esta lista es la que vamos a usar en el tercer paso para crear el heap. En este heap queremos introducir el primer evento que tiene cada una de las notas de Q. El coste de introducir o eliminar un elemento es un heap es de orden logarítmico ($O(log(m))$), por tratarse de un árbol binario. Como queremos insertar m elementos (el primer evento de cada nota) el tercer paso tiene un coste de $O(mlog(m))$.

En el último paso, en esencia, vamos a recorrer todos los eventos existentes (mn como máximo, tal y como hemos visto en el segundo paso), los cuales están almacenados en el heap. Concluimos entonces que el coste de esta parte es de $O(nmlog(m))$.

Concluimos por tanto que, dado que el último paso es el de mayor coste computacional, la complejidad computacional del algoritmo es de $O(nmlog(m))$.

3.4 Explicación detallada del código

3.4.1 Paso 1, cálculo del área inicial

En este paso vamos a extraer la información del estado de partida, la cual es fundamental para darle sentido al código que le sigue.

Para resolver este paso será necesaria una única función "area_inicial" ((código 3.1)) la cual necesita como entradas las dos melodías R y Q y cuyas salidas corresponden con la información que nos interesa del estado inicial $(area\_inicial, h11, h22, h33)$.

Como su nombre indica, "area_inicial" corresponde con el área en el estado inicial según lo definido en la sección 2.1.

$h11, h22, h33$ son 3 variables en la que nos vamos a apoyar para poder actualizar el área entre eventos consecutivos.

Este cálculo inicial es de vital importancia para poder actualizar posteriormente el área en cada instante en el que se produzca un evento. Para ello hay que recorrer al completo las 2 listas que representan las melodías R y Q y calcular el área de cada rectángulo que forman cada par de notas de R y Q, siendo el área total de la melodía la suma de las áreas de todas las parejas de rectángulos.

En esencia, este paso se resuelve con un bucle *while* que contiene un conjunto de secuencias $if/else$ anidadas. En él, se recorren las melodías mientras queden notas y se guarda el valor del área del rectángulo que forman cada par de notas y en función de qué tipo de rectángulo formen se guardan también las variables $h11, h22$ y $h33$. Estas variables corresponden a los sumatorios de la (ec. 2) del lemma 1, que son la suma de las alturas de cada rectángulo ponderadas por una cantidad que depende del índice j (número de la nota de Q de la que se trata).

Código 3.1 Función para el cálculo del área inicial.

```
# Cálculo del área entre dos melodías
# Dadas dos melodías devuelve el área inicial y el sumatorio de las alturas

def initial_area(R, Q):
    n = len(R)
    i = j = 0
    c0, h0 = [], []
    c1, h1 = [], []
    c2, h2 = [], []
    c3, h3 = [], []

    while i < n:
        # caso inicial(hay que tratarlo a parte)
        if i == 0 and j == 0:
            if R[i][1] <= Q[j][1]:
                h0.append(abs(R[i][2]-Q[j][2]))
                c0.append(h0[-1]*(R[i][1]-R[i][0]))
                i += 1

            else:
                h = abs(R[i][2]-Q[j][2])
                h2.append((j+1)*h)
                c2.append(h*(Q[j][1]-Q[j][0]))
                j += 1
```

```
        # caso especial en el que 2 notas empiezan a la vez y no es la primera
            nota
        elif R[i][0] == Q[j][0] and (i != 0 or j != 0):
            if R[i][1] <= Q[j][1]:
                h = abs(R[i][2]-Q[j][2])
                h3.append(j*h)
                c3.append(h*(R[i][1]-Q[j][0]))
                i += 1

            else:
                h1.append(abs(R[i][2]-Q[j][2]))
                c1.append(h1[-1]*(Q[j][1]-Q[j][0]))
                j += 1

        # caso habitual
        elif R[i][1] <= Q[j][1]:
            if R[i][0] > Q[j][0]: # azul contenido en rojo
                h0.append(abs(R[i][2]-Q[j][2]))
                c0.append(h0[-1]*(R[i][1]-R[i][0]))
                i += 1

            else: # rectangulo en disminucion
                h = abs(R[i][2]-Q[j][2])
                h3.append(j*h)
                c3.append(h*(R[i][1]-Q[j][0]))
                i += 1

        else:
            if Q[j][0] >= R[i][0]:
                h1.append(abs(R[i][2]-Q[j][2]))
                c1.append(h1[-1]*(Q[j][1]-Q[j][0]))
                j += 1

            else: # rectangulo en aumento
                h = abs(R[i][2]-Q[j][2])
                h2.append((j+1)*h)
                c2.append(h*(Q[j][1]-R[i][0]))
                j += 1


    # Calculo del area inicial:
    c00 = sum(c0)
    c11 = sum(c1)
    c22 = sum(c2)
    c33 = sum(c3)
    areainicial = c00+c11+c22+c33

    h11 = sum(h1)
    h22 = sum(h2)
    h33 = sum(h3)

    return (areainicial, h11, h22, h33)
```

3.4.2 Paso 2, cálculo de eventos

Este paso va a ser el encargado de encontrar los eventos y calcular su valor, así como de guardar la información restante que será necesaria para la actualización del área entre eventos consecutivos.

Para ello tenemos el archivo *calculoeventos*, el cual contiene la definición de la clase que nos permite crear objetos *Event* (código 3.2), y las funciones *calculaeventos_main* (código 3.3) y *get_epsilons* (código 3.4).

Código 3.2 Objeto Event para gestionar información sobre un evento.

```
class Event():
    def __init__(self):
        self.eps = [] # esta lista tendrá tantos elementos como eventos tenga
            dicha nota de Q con R. Cada uno de estos elementos será otra lista
            que guarda
        # el valor de epsilon en el indice 0, la topología en el indice 1 y la
            nota de R con la que se produce el evento en el indice 2 y las
        # 4 alturas de los 6 rectangulos involucrados. El ultimo elemento es el
             numero del evento del que se trata
        self.index = 0
        self.pitch = 0
```

Código 3.3 Función para el cálculo de eventos.

```
def calculaeventos_main(R, Q, maxeps):
    q = [] # esta variable es una lista de objetos events
    for j in range(len(Q)-1):
        epsilon = Event()
        get_epsilons(epsilon, Q[j], Q[j+1], R, maxeps, j)
        epsilon.index = j
        epsilon.pitch = Q[j][2]
        q.append(epsilon)

    # la ultima nota posee un "evento especial" que hay que tratar a parte
    # este ultimo rectángulo será siempre un c3(rectangulo en disminucion)
    epsilon = Event()
    epsilon.eps.append([maxeps, "final", "final", 0])
    epsilon.index = j+1
    epsilon.pitch = -1
    q.append(epsilon)

    return q
```

Código 3.4 Función para calculo el valor de ε y su topología.

```
def get_epsilons(epsilon, notaactual, notasiguiente, R, maxeps, j):
    """Función para obtener los eventos y la topología"""
    n = 0
    for i in range(len(R)):
        # calculamos el valor de epsilon
        e = (R[i][1]-notaactual[1])/(j+1)
```

```
        if R[i][1] > notaactual[1] and e < maxeps:
            if j == 0:
                notaactualdesplazada = (
                    0, notaactual[1]+e*(j+1), notaactual[2])
                notasiguientedesplazada = (
                    notasiguiente[0]+e*(j+1), notasiguiente[1] + e*(j+2),
                        notasiguiente[2])
            else:
                notaactualdesplazada = (
                    notaactual[0]+e*(j), notaactual[1]+e*(j+1), notaactual[2])
                notasiguientedesplazada = (
                    notasiguiente[0]+e*(j+1), notasiguiente[1] + e*(j+2),
                        notasiguiente[2])

            # caculo de las 4 alturas existentes:
            # h1 es la altura del primer rectangulo que forman las 2 notas
            h1 = abs(notaactualdesplazada[2]-R[i][2])
            # h2 es la altura del rectanglo c3 que siempre desaparace en un
                evento
            h2 = abs(notasiguientedesplazada[2]-R[i][2])
            # h3 es la altura del rectangulo c2 que siempre aparace en un evento
            h3 = abs(notaactualdesplazada[2]-R[i+1][2])
            # h4 es la altura del rectangulo que forman las 2 siguientes notas
                de un evento
            h4 = abs(notasiguientedesplazada[2]-R[i+1][2])

            # caso 0: de c2,c3,c0 a c0,c2,c3
            #       antes del evento: c2 en aumento en la nota actual y un azul
                contenido en la nota siguiente
            #       Despues del evento: azul contenido en la nota actual y un c3
                en disminucion en la nota siguiente
            if notaactualdesplazada[0] <= R[i][0] and R[i+1][1] <=
                notasiguientedesplazada[1]:
                epsilon.eps.append([e, 0, i, h1, h2, h3, h4, n])
            # caso 1: de c1,c3,c2 a c3,c2,c1
            #       antes del evento: rojo contenido en aumento en la nota actual
                 y un c2 en aumento en la nota siguiente
            #       Despues del evento: c3 en disminucion en la nota actual y un
                rojo contenido en la nota siguiente
            elif R[i][0] <= notaactualdesplazada[0] and notasiguientedesplazada
                [1] < R[i+1][1]:
                epsilon.eps.append([e, 1, i, h1, h2, h3, h4, n])

            # caso 2: de c2,c3,c2 a c0,c2,c1
            #       antes del evento: c2 en aumento en la nota actual y un c2 en
                aumento en la nota siguiente
            #       Despues del evento: azul contenido en la nota actual y un
                rojo contenido en la nota siguiente
            elif notaactualdesplazada[0] < R[i][0] and notasiguientedesplazada
                [1] < R[i+1][1]:
                epsilon.eps.append([e, 2, i, h1, h2, h3, h4, n])
            # caso 3: de c1,c3,c0 a c3,c2,c3
            #       antes del evento: rojo contenido en aumento en la nota actual
                 y un azul contenido en la nota siguiente
            #       Despues del evento: c3 en disminucion en la nota actual y un
                c3 en disminucion en la nota siguiente
```

```
            elif R[i][0] < notaactualdesplazada[0] and R[i+1][1] <=
                notasiguientedesplazada[1]:
                epsilon.eps.append([e, 3, i, h1, h2, h3, h4, n])
            n += 1
```

En la clase *Event* se definen 3 atributos, *.eps*, *.index* y *.pitch*. La idea consiste en crear un objeto *Event* para cada nota de Q. El atributo *.eps* es una lista que contiene los eventos que tiene la nota de Q a la cual pertenece el objeto *Event* creado. En el peor de los casos esta lista tendrá una longitud de m eventos, que sería el caso en el que la nota de Q tuviera un evento con las m notas de la melodía de referencia R. Como es evidente, todos los eventos de una misma nota Q tienen el mismo atributo *.index* y *.pitch*, pues corresponden respectivamente al número de la nota en cuestión y a su altura.

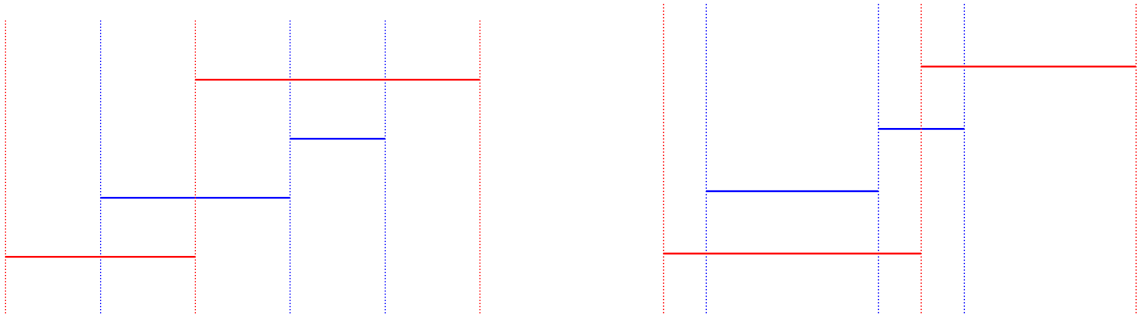
La función *calculaeventos_main* precisa como entradas las melodías R y Q y el valor máximo que puede tener un evento, *maxeps* según lo definido en la sección 2.1 y retorna como resultado una lista q en la que cada elemento de la lista es un objeto *Event* que contienen todos los eventos de cada nota de Q.

En esta función un bucle for itera a lo largo de cada una de las notas de Q. En cada iteración se crea su objeto *Event* correspondiente y se rellenan los atributos. Los atributos *.index* y *.pitch* se rellenan tal cual, pero para rellenar el atributo *.eps* es para lo que existe la función "get_epsilons", explicada a continuación.

La función *get_epsilons* precisa de los siguientes parámetros de entrada: *epsilon* (objeto *Event* para la nota de Q correspondiente), notaactual(es la nota de Q para la cual se produce el evento), notasiguiente(nota siguiente para la cual se produce el evento, también necesaria para calcular el evento), R (al completo), *maxeps*, j (número de la nota de Q para la que se produce el evento). Esta función no tiene ningún parámetro de retorno, pero guarda en el atributo *.eps* la información necesaria para cada objeto épsilon que recibe como entrada. Recordemos que esta función se ejecuta en un for para cada nota de Q. A su vez, en esta función, se itera en un for anidado para cada nota R. Lo primero que haremos es calcular el valor de ε al cual llamaremos e en el código. Este valor se calcula restando el valor de tiempo del final la nota de cada una de las notas de R con el final de la nota de Q de esa iteración. A continuación tenemos que contrastar si es un resultado valido, es decir, que sea mayor que cero y menor que *maxeps*. En caso de tratarse de un resultado válido, se escalan las notas actual y siguiente de Q según lo definido en la sección 2.1

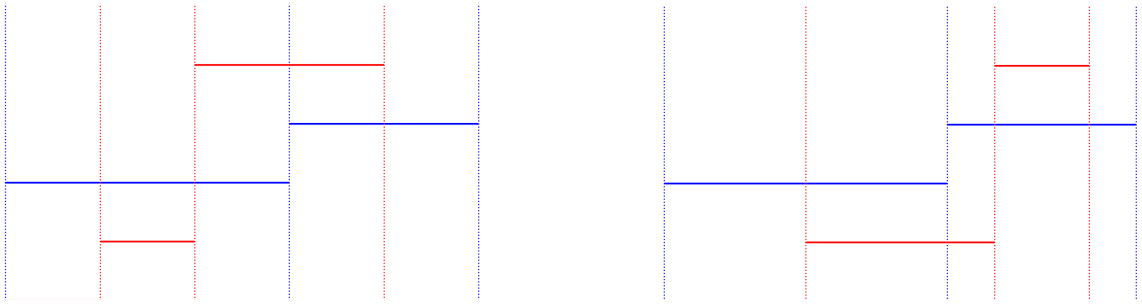
En el transcurso de un evento entre dos notas hay en juego 4 alturas correspondientes a los 6 rectángulos involucrados en el evento (3 antes del evento y 3 después). El cálculo de estas alturas es clave para actualizar los sumatorios $h11$, $h22$ y $h33$ que se calcularon en el paso anterior, y por eso es el siguiente paso de esta función.
Por último tenemos en esta función un conjunto de sentencias $if/elif$ para discernir entre los 4 pasibles casos correspondientes a los 4 tipos de eventos existentes, que se representan cualitativamente en las siguientes imágenes.

- Caso 0 (figura 3.3): de c_2, c_3, c_0 a c_0, c_2, c_3.
  Antes del evento: c_2 en aumento en la nota actual y un azul contenido en la nota siguiente.
  Después del evento: azul contenido en la nota actual y un c_3 en disminución en la nota siguiente
- Caso 1 (figura 3.4): de c_1, c_3, c_2 a c_3, c_2, c_1
  Antes del evento: rojo contenido en aumento en la nota actual y un c_2 en aumento en la nota siguiente.
  Después del evento: c_3 en disminución en la nota actual y un rojo contenido en la nota siguiente
- Caso 2 (figura 3.5): de c_2, c_3, c_2 a c_0, c_2, c_1
  Antes del evento: $c2$ en aumento en la nota actual y un c_2 en aumento en la nota siguiente
  Después del evento: azul contenido en la nota actual y un rojo contenido en la nota siguiente
- Caso 3 (figura 3.6): de c_1, c_3, c_0 a c_3, c_2, c_3
  Antes del evento: rojo contenido en aumento en la nota actual y un azul contenido en la nota siguiente
  Después del evento: c_3 en disminución en la nota actual y un c_3 en disminución en la nota siguiente.



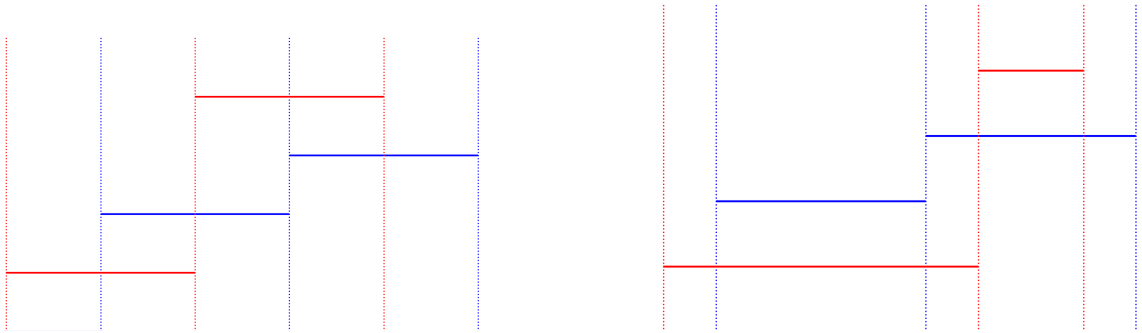
(a) Antes del evento. **(b)** Después del evento 0.

Figura 3.3 Evento de tipo 0.



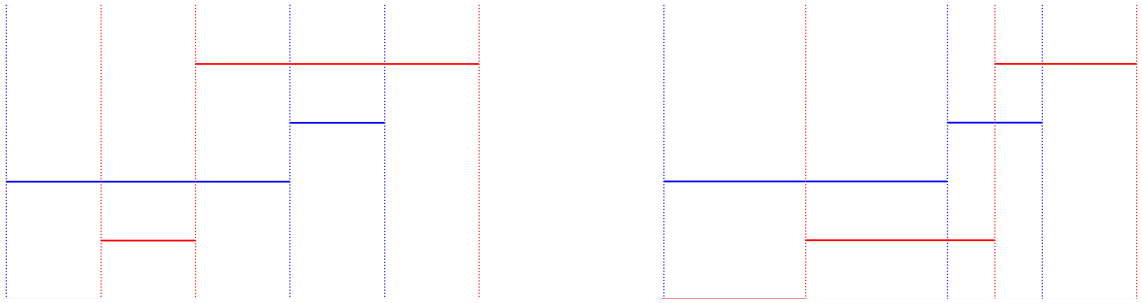
(a) Antes del evento. **(b)** Después del evento.

Figura 3.4 Evento de tipo 1.



(a) Antes del evento. **(b)** Después del evento.

Figura 3.5 Evento de tipo 2.



(a) Antes del evento. **(b)** Después del evento.

Figura 3.6 Evento de tipo 3.

Una vez averiguado en qué caso nos encontramos ya disponemos de toda la información necesarias para crear un nuevo elemento en el atributo *.eps*. Cada uno de estos elementos es una lista el cual contiene el valor *e*, la topología del evento del que se trata, el número de la nota de *R* con la que se ha producido el evento, las 4 alturas antes descritas ($h1$, $h2$, $h3$ y $h4$) y por último el número de evento del que se trata para esa nota.

La función display no tiene un objetivo funcional para la resolución del algoritmo, si no un apoyo para visualizar si se están realizando correctamente los cálculos.

3.4.3 Paso 3, construcción del heap

Este paso se resuelve con una única función llamada *heap* (código 3.5)en la cual se crea un heap en el que se guarda el primer evento de cada segmento de *Q*. Tiene como parámetro de entrada la lista *q* de objetos Evenets creadas en el paso anterior y devuelve como parámetro de salida el heap, *h*.

Código 3.5 Función para la construcción del heap.

```
def heap(q):
    """Función que crea un heap en el que se guarda el primer evento de cada
        segmento de Q"""
    h = []
    for j in q:
        if len(j.eps) > 0:
            e = j.eps[0]
            heapq.heappush(h, [e[0], j.index, e[-1]])

    return h
```

Con un único bucle for iteramos para cada elemento de *q*. En cada iteración añadimos un elemento al heap. Cada elemento es una lista de 3 elementos, los cuales son el valor del primer épsilon de cada atributo *.eps* del objeto *Event*, el segundo es el índice de la nota de *Q* para el que se produce el evento y por último el número de evento del que se trata para esa nota de *Q*.

3.4.4 Paso 4, actualización y seguimiento del área

En esta última parte del algoritmo obtendremos el resultado deseado con la ayuda de la información recabada en las partes anteriores del algoritmo. La función "actualizar" del archivo "actualizacionarea" (código 3.6) será la encargada de esta misión. Esta función tiene como parámetros de salida el área mínima entre las dos canciones y el valor de épsilon para el que se produce dicha área mínima. Como entradas precisa del heap (*h*), del vector de objetos *Event* (*q*), el áreainicial (*areainicial*), de los sumatorios ponderados ($h11$, $h22$ y $h33$), y del valor máximo de épsilon (*maxeps*).

En líneas generales, en esta función, vamos a ir sacando y metiendo elementos en el heap. Un bucle *while* va extrayendo el nodo raíz del heap, averiguamos a qué nota de *Q* pertenece el evento que acabamos de sacar y metemos en el heap el siguiente evento de esa misma nota de *Q* (si queda alguna). Notar que en cada iteración se saca un evento del heap, pero no en todas se mete uno nuevo, de forma que la longitud del heap va disminuyendo y por ello la condición de fin del bucle es que no queden eventos en el heap.

Antes de comenzar el bucle inicializamos las dos variables que vamos a entregar como salidas. *areamin* la inicializamos con el valor del área inicial, y en cada iteración del bucle se comparará esta variable con el área actualizada tras cada iteración (*nuevaarea*). *epsmin* comienza inicializado en cero, pues es el valor de épsilon para el área inicial. Se irá sobreescribiendo cuando corresponda según dicte la comparación entre *areamin* y *nuevaarea*.

En cada iteración lo primero que haremos será sacar el nodo raíz del heap. Después calculamos el área para el valor de épsilon del evento sacado con la fórmula (2.1) del lemma 1. Con este cálculo estamos en

condiciones de realizar la comparación descrita en el párrafo anterior, llevando así el registro del área mínima y su respectivo épsilon. A continuación, tenemos que averiguar a qué nota de la melodía Q pertenece el épsilon con el que acabamos de trabajar y comprobar si a dicha nota tiene más eventos. En caso afirmativo se mete el siguiente evento en el heap. Por último, queda preparar las variables para la siguiente ejecución del bucle. Esto es, actualizar los sumatorios ponderados por el índice j ($h11$, $h22$ y $h33$) y las variables *areaanterior* y *epsilonanterior* de las que se hace uso en la ecuación (2.1).

Código 3.6 Archivo para la actualización y seguimiento de área entre eventos.

```
# Actualizacion y calculo de área

import heapq

# inicializamos el área mínima como área inicial

# while (mientras queden elementos en el heap):
#     sacar el nodo raíz del heap,
#     mirar a que segmento de q pertenece el evento que acabamos de sacar,
#     ver si para dicho segmento quedan eventos que meter en el heap,
#     en caso afirmativo, meter dicho evento en el heap

#     para el evento sacado:
#     recalcular el área
#     comparar la nueva área con el área mínima
#     si la nueva es menor sobreescribimos areamin con este nuevo valor
#     actualizamos las alturas de los rectángulos involucrados en el evento
    para el próximo evento


def actualizar(h, q, areainicial, h11, h22, h33, maxeps):
    areamin = areainicial
    areaanterior = areainicial
    epsilonanterior = 0
    epsmin = 0
    i = 0
    ayuda = []
    while(h):
        # recupera es una lista de 3 elementos. el 0 es un valor de epsilon, el
             1 es la nota de Q para la que se produce dicho epsilon y el 2 el nº
             de evento para esa nota
        recupera = heapq.heappop(h)
        nuevoepsilon = recupera[0]
        j = recupera[1]
        n = recupera[2]
        # en esta lista hemos guardado toda la info relativa al próximo evento
            que se producirá en el tiempo(osea, el menos evento siguiente)
        nuevoevento = q[j].eps[n]

        nuevaarea = areaanterior+(nuevoepsilon-epsilonanterior)*(h11+h22-h33)
        ayuda.append([nuevaarea, nuevoevento[1]])
        if nuevaarea < areamin:
            areamin = nuevaarea
            epsmin = nuevoepsilon

        if nuevoepsilon == maxeps:
```

```
            break

        # si aún quedan eventos para el segmento del que acabamos de sacar
            nuevoevento:
        if len(q[j].eps) > (n+1):
            heapq.heappush(h, [q[j].eps[n+1][0], q[j].index, n+1])

        # preparamos los datos para la siguiente ejecución:
        areaanterior = nuevaarea
        epsilonanterior = nuevoepsilon

        # chequeamos topología para saber como actualizar las alturas
        if nuevoevento[1] == 0:
            h11 = h11
            h22 = h22-(j+1)*nuevoevento[3]+(j+1)*nuevoevento[5]
            h33 = h33-(j+1)*nuevoevento[4]+(j+1)*nuevoevento[6]

        elif nuevoevento[1] == 1:
            h11 = h11-nuevoevento[3]+nuevoevento[6]
            h22 = h22-(j+2)*nuevoevento[6]+(j+1)*nuevoevento[5]
            h33 = h33-(j+1)*nuevoevento[4]+(j)*nuevoevento[3]

        elif nuevoevento[1] == 2:
            h11 = h11+nuevoevento[6]
            h22 = h22-(j+1)*nuevoevento[3]-(j+2) * \
                nuevoevento[6]+(j+1)*nuevoevento[5]
            h33 = h33-(j+1)*nuevoevento[4]

        elif nuevoevento[1] == 3:
            h11 = h11-nuevoevento[3]
            h22 = h22+(j+1)*nuevoevento[5]
            h33 = h33-(j+1)*nuevoevento[4]+(j) * \
                nuevoevento[3]+(j+1)*nuevoevento[6]

        i += 1

    return areamin, epsmin, ayuda
```

4 Base de datos: Tonás

El corpus objeto de estudio en este proyecto está formado por un grupo grabaciones de dos palos flamencos: las deblas y los martinetes, pertenecientes al grupo de las tonás. A continuación, se va a realizar un breve estudio descriptivo sobre dicho grupo y sobre los palos que se van a analizar.

El flamenco se nutre de fuentes muy variadas entre las que destaca el repertorio de romances, tonás y otros cantes "a palo seco" (sin acompañamiento de guitarra). La debla y el martinete pertenecen al grupo de las tonás. Toná proviene del término español tonada, que significa “cantable” o “fragmento melódico”. Se cree que muchas de estas tonadas (tonás en andaluz) provienen de antiguos romances y son partes desgajadas de los mismos que se conservan en la memoria del pueblo. Otros estilos que pertenecen al grupo de las tonás son las carceleras, la debla y la matriz, la gama de las tonás.

El concepto de tonás no fue bien definido en el cante flamenco hasta la obra de Ricardo Molina y Antonio Mairena “Mundo y formas del Cante Flamenco” [16]. Hasta ese momento, estos cantes se denominaban como martinetes, en el caso de Jerez, Cádiz o Málaga; como tonás; indistintamente como martinetes o tonás en Triana y El Puerto; o reservando el nombre de martinete a alguna de sus variedades y llamando tonás al resto.

A principios del s. XX el nombre de toná cae en desuso y en los discos de pizarra no aparece la palabra toná, pero sí martinete y, a veces, carcelera. En este contexto, el prestigio del que goza Antonio Mairena influye en que se adopte su propuesta relativa al concepto y la denominación de toná de modo generalizado.

Molina y Mairena definen la toná como coplas de cuatro versos octosílabos, rimados los pares de forma asonante e interpretadas a capella, sin acompañamiento de guitarra y sin marcar externamente el compás con palmas, bastón o pié. Algunos de los cantes con estas características tienen una denominación específica, como es el caso de los martinetes y las deblas. El nombre de carceleras, en cambio, puede aplicarse igual, a un estilo determinado de toná que a martinetes con letras de contenido alusivo a la vida en prisión. Para Molina y Mairena el cante por tonás es el género, y los martinetes y deblas las variedades o especies de tonás.

En la actualidad se conservan varios tipos de martinetes, destacando el conocido como “martinete trianero”, desarrollado en los ambientes fragueros del barrio de Triana, Sevilla, que será el estilo elegido aquí para su estudio. Al contrario que las tonás, que usan como tonalidad la cadencia andaluza, los martinetes se realizan en tono mayor. Con respecto al ritmo, se cantan, como todas las tonás, sin un compás determinado. Sin embargo, Antonio Ruiz El Bailarín realizó una versión bailable eligiendo el compás de la seguiriya para realizarlo y desde entonces se ha generalizado la interpretación del martinete sobre este metro rítmico. Finalmente, respecto a la copla, es una cuarteta octosílaba (romance) y se canta sin repetición de ningún verso o con repeticiones cuando se hace redoblado. Suele iniciarse el martinete con un onomatopéyico trantran, imitando el sonido del martillo sobre el yunque.

Para una descripción más profunda de algunos cantes del grupo de tonás, se pueden consultar los trabajos [12], [5] y [18].

4.1 Los cantes por tonás para un estudio computacional

Quizá sean los cantes por tonás de los cantes que ofrecen más dificultades a la hora de estudiarlos computacionalmente, debido a que las diferentes variedades existentes dependen más de la idiosincrasia del cantaor que de variables locales o de estructuras musicológicas bien definidas. La inexistencia del armazón armónico proporcionado por la guitarra y las irregularidades de afinación de muchos intérpretes entorpecen la identificación de los modos musicales y otro tanto cabe decir de una estructura rítmica que va ad libitum el intérprete puede variar el tempo, la velocidad a la que interpreta el cante, como lo desee. Por todo ello, muchos aficionados consideran que representan la esencia del cante flamenco.

Las tonás son cantes a capella, sin instrumentación, en algunos casos con alguna percusión tales como nudillos o golpes de martillo en el caso de martinetes, en el argot flamenco son denominados cantes a palo seco. Tradicionalmente se les ha considerado cantes de las fraguas. En efecto, algunos intérpretes históricos han sido herreros, y ciertas letras aluden a las labores de fragua. Lo mismo ocurre con las letras de tema carcelario, que justifican la denominación de carceleras que reciben algunas de las tonás.

La casi total ausencia de grabaciones de tonás en cilindros y discos de pizarra influye en las dificultades para su estudio. De los 5.500 títulos de la discografía en 78 rpm recogida en el Diccionario Enciclopédico Ilustrado del Flamenco de Blas Vega y Ríos Ruiz solo hay 6 grabaciones de tonás, realizadas entre 1922 y 1933 por Centeno, Tenazas, Cepero, El Gloria, Mazaco y El Cuacua. Esto confirma lo dicho por Demófilo [14] sobre el olvido en que habían caído estos cantes.

Sobre el número de tipos diferentes de tonás existentes se ha hablado mucho, y existen diferentes versiones. Por citar un ejemplo, Demófilo [14] enumera treinta y un tipos diferentes. Pierre Lefranc [12], sin embargo, identifica catorce. El repertorio de tonás diferentes merece un estudio independiente, que permita una clasificación basada más en criterios musicológicos que de filiación, local o de atribución, como hasta ahora se ha hecho.

A grandes rasgos, dentro del género de las tonás pueden distinguirse los siguientes cantes diferenciados:

- Martinetes
- Carceleras
- Deblas
- Saetas
- Tonás propiamente dichas

Conviene indicar que el grupo de las saetas presenta características que pueden aconsejar estudiarlas de manera independiente al resto de tonás. Para un estudio musicológico de las saetas se puede consultar [11], [3], [17]. Respecto al grupo de tonás propiamente dichas, se define por exclusión: pertenecen a él los cantes que no se incluyen dentro de algunos de los grupos restantes.

4.2 Descripción y características musicales: debla y martinete

Para nuestro estudio se van a seleccionar dos estilos dentro del grupo tonás: la debla que popularizó Tomás Pavón y el martinete trianero conocido como “martinete de Juan Pelao de Triana”. Un estudio musicológico de estos cantes se puede consultar en el trabajo [18]. Hacemos a continuación un breve resumen de dicho artículo.

- DEBLAS: La debla es una variante o especie de las tonás. En general, se caracteriza por su gran ornamentación melismática, más abrupta que el resto de las canciones de este estilo, que hace muy exclusiva su melodía. Las deblas se caracterizan por un contorno melódico en particular.

  La versión actual de la debla se debe a una aportación de Tomás Pabón en 1948, que la había aprendido de su suegro Antonio El Baboso. Es indudable que se trata de una toná trianera, pero algunos clásicos dudan que fuera la verdadera debla de la que hablaba Demófilo [14].

La letra que graba Pabón y que se ha difundido es insustancial. Ha sido modificada primero por Antonio Mairena y después por otros cantaores para hacer alusiones a persecuciones sufridas por los gitanos o a figuras trianeras legendarias.

Como otras tonás, se trata de un cante de cuatro versos que sigue el esquema 8-/8a/8- /8a. Seguramente por influjo de las características vocales de Pabón, este cante se presta a la aparición de melismas abundantes y prolongados, ejecutados con pocas respiraciones.

Las variables musicales que caracterizan las diferentes variantes dentro del estilo debla son las siguientes según el trabajo [18]:

I **Comienzo por la palabra ¡Ay!:** ¡Ay! es una interjección de dolor, característica de fuerte idiosincrasia en la música flamenca.

II **Vinculación de ¡Ay! con el resto de los versos:** El ¡Ay! inicial puede ser unido con el resto de los versos o simplemente separado de ellos.

III **La nota inicial:** Hace referencia a la primera nota del terceto. Normalmente es el sexto grado de la escala (VI), aunque también puede aparecer el quinto (V).

IV **Tendencia de la melodía en el primer hemistiquio:** (un hemistiquio es la mitad o el fragmento de un verso, dividido por una pausa en la entonación) La tendencia puede ser ascendente (appoggiatura rápida en V, a continuación, la progresión VI-IV), simétrica (III-VI-IV) o ascendente.

V **Repetición del primer hemistiquio:** La repetición puede ser del hemistiquio completo o de parte de él.

VI **Cesura:** La cesura (del latín caesura: cortadura) es el espacio o pausa dentro de un verso separando dos partes.

VII **Tendencia de la melodía en el segundo hemistiquio:** La tendencia se define igual que el primer hemistiquio.

VIII **Mayor grado en el segundo hemistiquio:** Es el grado más alto encontrado en el segundo hemistiquio. Por lo general, se alcanza el séptimo grado, aunque pueden aparecer quinto y sexto grado.

- MARTINETES: Como se ha visto antes, los martinetes también son considerados una variante de las tonás. Se diferencian de las deblas en la temática de sus letras y en su estructura melódica que siempre termina en modo mayor. La temática de las deblas es usualmente triste, cantadas sin acompañamiento de guitarra, como el resto de tonás. Sin embargo, los martinetes se suelen acompañar con el sonido de un martillo golpeando un yunque.

  Los aspectos musicales que caracterizan el martinete trianero [18] son:

  I **Repetición del primer hemistiquio:** Como en las deblas, la repetición puede ser completa o parcial.

  II **Clivis/flexa (movimiento alto/bajo) en el final del primer hemistiquio:** Normalmente, caída IV-III o IV-IIIb. El final más común para un terceto es el cuarto grado, cuyo sonido es sostenido hasta alcanzar la cesura. Algunos cantes utilizan a modo de cierre el III o IIIb.

  III **Uso del mayor grado en los dos hemistiquios:** La práctica habitual es llegar hasta el cuarto grado, y algunos cantantes llegan al quinto grado.

  IV **Nota final del segundo hemistiquio:** El segundo hemistiquio termina cayendo al segundo grado.

Se eligen estos dos estilos porque tienen un contorno melódico claramente diferenciado como se puede ver en la transcripción de la Figura 7. Por tanto, estamos abordando un estudio inter-estilo, esto es, definiendo una medida de similitud que discrimina estilos diferentes de cantes. En otros contextos, estudios intra-estilo, podríamos estudiar las distintas variantes de martinetes que interpretan distintos cantaores.

Una alternativa sería diseñar medidas de similitud combinadas en las que se involucra además del contorno melódico otras otros parámetros de carácter musical, como se propone en el artículo [18].

4.3 El Corpus Tonás. Un metadato de acceso libre

El corpus Tonás fue creado en el contexto de un estudio sobre similitud y clasificación de estilos de cantes flamencos a capella (tonás) por el grupo COFLA. Los 72 fragmentos son monofónicos y su duración media es de unos 30 segundos. También se ofrece en el corpus una transcripción melódica manual, como se explica a continuación:

TRANSCRIPCIÓN MELÓDICA SEMIAUTOMÁTICA:

Tres sujetos participaron en el proceso de transcripción: un músico con un conocimiento limitado de la música flamenca, un experto en la música flamenca y un cantaor flamenco. El músico realizó primero anotaciones manuales detalladas. Como su conocimiento de la música flamenco era muy limitado, se esperaba que no usara el conocimiento implícito en el estilo, para así no condicionara la transcripción manual y la realizara de una forma objetiva. Algunos ejemplos fueron corregidos por un experto en flamenco para establecer un criterio que se aplicó además a refinar manual de transcripciones. Por último, el cantaor verificaría de forma independientemente las transcripciones manuales.

A fin de reunir las anotaciones manuales, se dotó a los sujetos de un interfaz para visualizar la forma de onda y la frecuencia fundamental ($f0$) en centésimas de tono (en una representación de teclado de piano). Los sujetos podían escuchar la forma de onda y la transcripción sintetizada, durante la edición de los datos melódicos hasta que quedaron satisfechos con la transcripción. Se observó que todavía había un grado de subjetividad en cuanto a la diferenciación entre los ornamentos y los cambios de tono.

Los criterios para la transcripción manual de la colección de tonás se referencian en [7].

ARCHIVOS INCLUIDOS:

- Archivos de audio: 72 archivos de audio monofónicos en 16 bit. Formato WAV y frecuencia de muestreo 44.1 kHz.
- Estructura de datos: TONAS - _____.txt: Nombre del archivo (Primera columna), título (segunda columna) y cantaos (tercera columna) cada archive en formato Unicode UTF-8 para preservar los acentos y la ñ.
- Transcripción melódica: 3x72 archivos. Por cada uno de los 72 archivos de audio se crean los siguientes archivos de transcripción:
    - Frecuencia fundamental (nombre del archivo.f0.Corrected): Estimación de la f0 basada en (Gómez & Bonada) donde se han corregido manualmente algunos errores de la f0. También contiene valores cuantificados de note.f0 a escala referenciados a la escala temperada. Cada una de las 4 columnas del archivo corresponden a:
      $time\_seconds$, $energy$, $estimated f0\_Hz$, $estimatednote f0\_Hz$
    - Transcripción segmentación Notes (nombre del archivo.notes.Corrected): Contiene una línea por cada nota, incluyendo la siguiente información:
      $onset\_seconds$, $duration\_seconds$, $pitch\_MIDI$, $energy$

5 Experimentos

Tenemos por fin el algoritmo completamente desarrollado y la base de datos objeto de estudio definida. En este capítulo se realizarán una serie de pruebas para comprobar la efectividad del algoritmo.

5.1 Protocolo de pruebas

Antes de trabajar con el corpus de tonás vamos a trabajar con una serie de ejemplos de prueba para comprobar la veracidad de los resultados obtenidos.

5.1.1 Ejemplo serncillo

Vamos a construir un caso muy básico con dos melodías cortas sin sentido musical y contrastar si los datos calculados por nuestro algoritmo son correctos y coinciden con los datos calculados a mano.

Nuestras melodías van a ser:

R	[0, 3.2, 40], [3.2, 6, 30]
Q	[0, 1, 50], [1, 3, 20], [3, 5, 60]

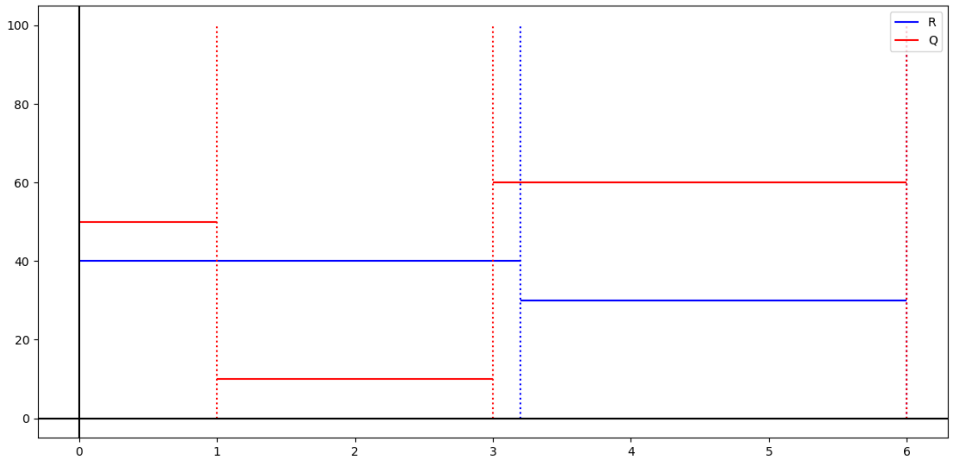
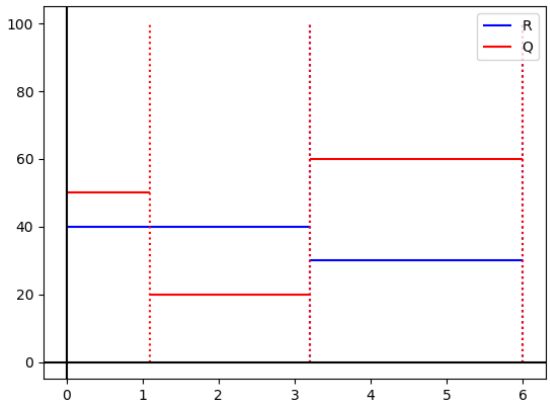


Figura 5.1 Primera prueba del algoritmo con un ejemplo sencillo.

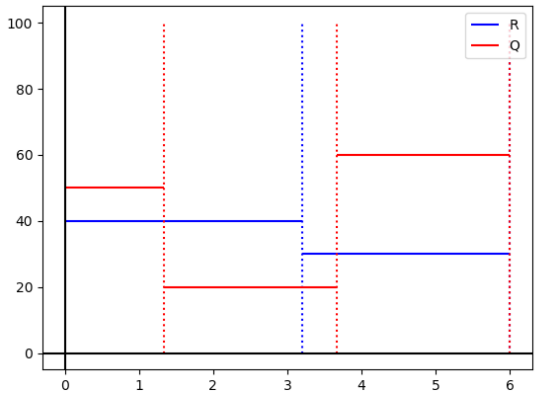
En este ejemplo el área inicial y el máximo valor de épsilon valen:

$areainicial$	158
ε_{max}	0.33333

Los dos únicos eventos que tiene este ejemplo tan básico se produce para la segunda nota de Q al encontrar a la primera nota de R, con valor de $\varepsilon = 0.1$ y un evento tipo final que coincide con maxeps:



(a) Instante en el que se produce el único evento entre las 2 melodías.



(b) Instante en el que Q está escalada ε_{max}.

Figura 5.2 Eventos del ejemplo sencillo.

En el primer evento, el área vale 137 y en el segundo 125.3335 con lo que el valor de épsilon que minimiza el área es ε =0.33333 resultando un área mínima comprendida entre los dos contornos melódicos de 125.3335.

5.1.2 Función "generate_melodies"

Dentro del módulo *auxiliar_functions* encontramos la función *generate_melodies* (código 5.1) la cual nos será de utilidad para poner a prueba el algoritmo. Se trata de una función que nos genera melodías aleatorias ya preparadas para poder probar directamente el algoritmo. Si se desea, en ella se pueden elegir el número de notas de las melodías, los intervalos de duración y la altura de sus notas.

Código 5.1 Función para generar las melodías de referencia y de consulta.

```
# El primer dato de cada melodía tendrá por defecto tiempo de comienzo cero,
# los demas se generaran aleatoriamente
def generate_melodies():
    """Generador de melodías"""
    R = [[0, round(random.uniform(1, 4), 2), random.randint(100, 1000)]]
    Q = [[0, round(random.uniform(1, 4), 2), random.randint(100, 1000)]]

    # creando las notas de R:
    time = R[0][1]
    for i in range(1, 15): # R tendra 5 notas(de momento, por tener algo)
        newnote = []
        newnote.append(R[i-1][1])
        newtime = random.uniform(1, 4) # cada nota durara entre 1 y 4 secs
        time += newtime
        newnote.append(round(time, 2))
        # los valores posibles en frecuencia
        newpitch = random.randint(100, 1000)
        newnote.append(newpitch)
        R.append(newnote)

    # creando las notas de Q:
    t = Q[0][1]
    for j in range(1, 10): # Q tendra 3 notas
        newnote = []
```

```
        newnote.append(Q[j-1][1])
        newtime = random.uniform(1, 4)
        t += newtime
        newnote.append(round(t, 2))
        newpitch = random.randint(100, 1000)
        newnote.append(newpitch)
        Q.append(newnote)

    # vamos a machacar el valor del final de la última nota para que siempre
    # coincida con el de R(extender el valor de la ultima nota de Q hasta el
        final de los tiempos)

    # maxeps=(R[-1][1]-Q[-1][1])/len(Q)
    # Q[-1][1]=R[-1][1]
    return(R, Q)
```

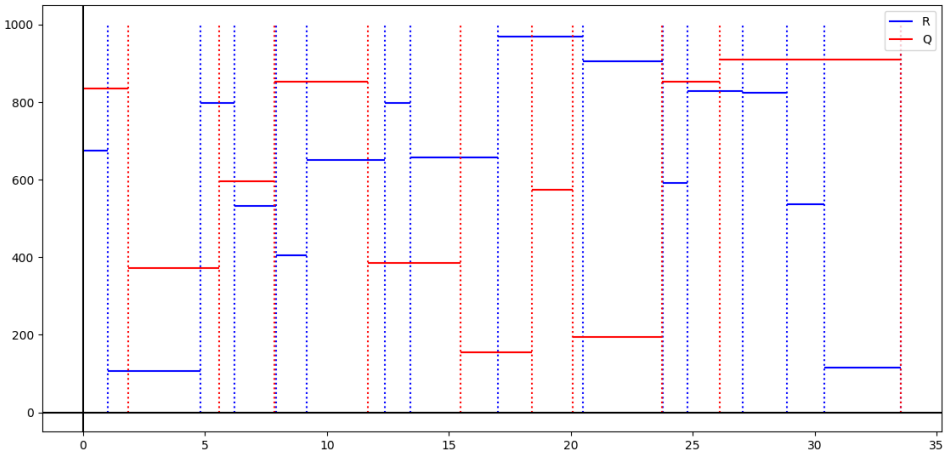


Figura 5.3 Ejemplo con generate_melodies.

En el ejemplo de la figura 6, Q tiene un total de 10 notas y R de 15, con una duración aleatoria de cada nota de entre 1 y 4 segundos y una altura o tono aleatorio de entre 100 y 1000dB.

Al contemplar los resultados vemos que se han producido un total de 15 eventos, siendo $maxeps = 0.493$ y $areainicial = 0.049999999997$. En *result* hemos guardado el área mínima y el ε para el que se produce: $result = [13088.899999999998, 0.13000000000000034]$

$areainicial$	0.049999999997
ε_{max}	0.33333

5.1.3 Función "comprueba_area(R,Q,q)", calculo alternativo de comprobación

En el siguiente punto de esta sección vamos a empezar a trabajar con la base de datos real. Antes de entrar en ella vamos a aplicarle a las melodías de ejemplo anteriores un algoritmo alternativo para comparar con el método del presente trabajo.

Este método alternativo pretende sacar las mismas conclusiones que nuestro algoritmo, con la salvedad de que no lo hace de manera eficiente optimizando el tiempo de cómputo y operaciones. Esta nueva versión es mucho más simple conceptualmente, pues calcula el área de una forma tradicional, calculando el área cada vez que se produce un evento como la suma de todos los rectángulos que producen cada pareja de notas de R y Q. Se trata de un algoritmo exhaustivo no eficiente, en el que aseguramos que el cálculo de los parámetros de salida son los deseados aunque para ello haya empleado más recursos de los necesarios, con la finalidad de comprobar la veracidad de los datos proporcionados por el algoritmo eficiente.

De esta forma vamos a trabajar con el ejemplo anterior, y se le van a aplicar a este par de melodías los dos algoritmos. Por un lado vamos a guardar en la lista *ayuda* las áreas de cada evento calculadas con el algoritmo eficiente. Por otro lado, en el fichero *auxiliar_functions* tenemos la función *comprueba_area* (código 5.2) la cual necesita como parámetros de entrada (R, Q, q) con son respectivamente las melodías R y Q y q es la lista en la que cada elemento es un objeto Event que contienen todos los eventos de cada nota de Q. Con esta información podemos calcular el área para cada ε aplicando la función de *area_inicial*, pues esta al fin y al cabo calcula el área para una posición estática de un par de melodías.

Código 5.2 Función para comprobación del cálculo del área.

```
def comprueba_area(R, Q, q):
    """Función para comprobar el área para un epsilon concreto"""
    ordered_events = []
    for query_note in q:
        for event_in_query_note in query_note.eps:
            ordered_events.append(event_in_query_note)
    ordered_events = sorted(ordered_events)

    areas = []
    for one_event in ordered_events:
        Qaux = []
        for j in range(len(Q)):
            notaux = []
            notaux.append(Q[j][0]+j*one_event[0])
            notaux.append(Q[j][1]+(j+1)*one_event[0])
            notaux.append(Q[j][2])
            Qaux.append(notaux)
        # print(Qaux)
        Qaux[-1][1] = R[-1][1]
        pak = area_inicial.initial_area(R, Qaux)[0]
        areas.append(round(pak, 6))
    return(areas)
```

Tenemos por tanto dos listas, *ayuda* y *areas*, que contienen la misma información. Si restamos los valores elemento a elemento podremos ver la precisión de la diferencia del cálculo del área según los dos métodos.

Como puede observarse en los resultados, el peor de los casos comete errores del orden de 10^{-7}, lo cual es insignificante comparado con el orden de magnitud de las áreas que se están calculando que son del orden de 10^5, de forma que no tienen efecto a la hora de la toma de decisiones sobré qué épsilon produce el menor área.

5.2 Probando el algoritmo con la base de datos

En esta sección vamos a explicar cómo se ha manipulado la base de datos para que sea compatible para aplicarle nuestro algoritmo. Después vamos a probarla realizando una serie de experimentos comparando las deblas y los martinetes del corpus de tonás.

Como apoyo a esta parte se ha creado el archivo *dataframe* (código 5.3) y la función *prepare_melody* del módulo *auxiliar_functions*. El primer paso es importar las librearías *pandas* y *glob*, las cuales permiten trabajar con objetos tipo *dataframe* e importar archivos de Excel de una carpeta a Python de forma automática.

Código 5.3 Archivo con las funciones necesarias para preparar los datos a partir de los archivos .csv.

```
# Script que carga los archivos automaticamnte
```

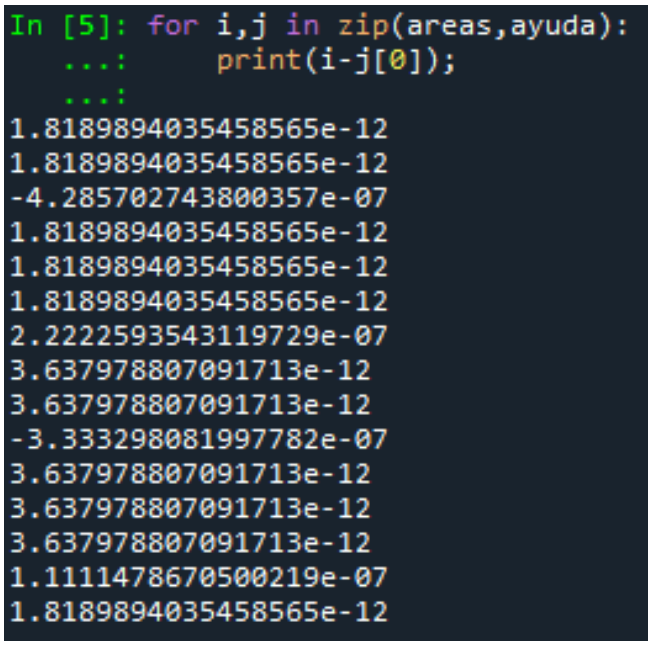


Figura 5.4 Diferencia de áreas calculadas por los dos métodos.

```
import pandas as pd
import glob


def dframe(exclude):

    if exclude == 'exclude_pabon&mairena':
        references_array = exclude_pabon_and_mairena()

    if exclude == 'none_deblas':
        references_array = all_deblas()

    if exclude == 'none_martinetes':
        references_array = all_martinetes()

    # el siguiente bucle es para 2 cosas, tratar los datos como floats(por algú
        n motivo a veces lo trata como str) y para que las melodías empiecen
        desde el segundo cero
    for reference in references_array: # recorremos cada cancion
        start_time = float(reference.iloc[0, 0])
        for i in range(len(reference)): # recorremos cada nota de la canción.
            Con este bucle forzamos el comienzo de la primera nota en el
            instante cero y además conseguimos asegurar que siempre trate los
            datos como float(porque si no a veces los trata como str)
            if type(reference.iloc[i, 0]) == str:
                reference.iloc[i, 0] = float(reference.iloc[i, 0])
            if type(reference.iloc[i, 1]) == str:
                reference.iloc[i, 1] = float(reference.iloc[i, 1])
            if type(reference.iloc[i, 2]) == str:
                reference.iloc[i, 2] = float(reference.iloc[i, 2])
            reference.iloc[i, 0] = reference.iloc[i, 0]-start_time
    return references_array
```

```
def exclude_pabon_and_mairena():
    references_array = []
    # el metodo .drop para quitar la primera columna/fila?
    for path_name in glob.glob("./DB_files/deblas/*.csv"):
        if ('TPabon' not in path_name) and ('AMairena' not in path_name):
            references_array.append(pd.read_csv(
                path_name, names=["inicio", "duracion", "tono", path_name]).drop
                    ([0], axis=0))
    return references_array


def all_deblas():
    references_array = []

    for path_name in glob.glob("./DB_files/deblas/*.csv"):
        references_array.append(pd.read_csv(
            path_name, names=["inicio", "duracion", "tono", path_name]).drop
                ([0], axis=0))
    return references_array


def all_martinetes():
    references_array = []
    # el metodo .drop para quitar la primera columna/fila?
    for path_name in glob.glob("./DB_files/martinetes/*.csv"):
        references_array.append(pd.read_csv(
            path_name, names=["inicio", "duracion", "tono", path_name]).drop
                ([0], axis=0))
    return references_array
```

En este script encontramos la función $dframe$, la cual crea la lista tr cuyos elementos son objetos tipo $dataframe$ y cada uno de estos objetos se corresponde a una melodía.

Estos objetos se han creado a partir de un archivo .csv del corpus tonás, y por cómo están estructurados los datos no están preparados para poder aplicarle el algoritmo directamente. Va a ser necesario convertir la tipología de cada uno de los datos, ya que estos están insertados como si fueran cadenas de caracteres en vez de floats.

Además será necesario desplazar toda la melodía, nota a nota, para que el comienzo del cante de las deblas y martinetes empiecen en el instante cero ya que muchas de las transcripciones tienen un silencio previo al comienzo del cante.

Por último será necesario añadir notas nuevas a la melodía. Esto se debe a que el corpus objeto de estudio no contempla los silencios como notas, por lo tanto será necesario añadir notas con un pitch de 0dB con la longitud del espacio que exista entre dos notas consecutivas en la que una no empiece en el mismo instante de tiempo en el que acabó la anterior.

A continuación vamos a aplicarle el algoritmo usando como consulta la debla de Tomás Pavón y contrastarla con el dataset de deblas.

5.2.1 Experimento 1: Clasificación de las deblas según su escuela

Tomás Pavón y Antonio Mairena fueron dos grandes cantaores flamencos del siglo XX con una influencia en el género indiscutible. En la siguente tabla (5.1) puede observarse una clasificación de las deblas registradas

en la base de datos según se parezca más a la debla cantada por Tomás Pavón o por Antonio Mairena para valorar la influencia de estos dos artistas.

Tabla 5.1 Clasificación de las deblas según su similitud al estilo de Tomás Pavón o Antonio Mairena.

Más similares a Tomás Pavón	Más similares a Antonio Mairena
Naranjito	TalegondeCordoba
JHeredia	Turronero
Chocolate	RafaelRomeroElGallina
JAlmaden	
ChanoLobato	
PdeLucia	
JMerce	
MSimon	
MVargas	
DiegoClavel	

Como puede observarse en la tabla 5.1, la influencia de Tomás Pavón fue mayor que la de Antonio Mairena, no por ello quitar mérito al legado de Mairena.

En la tabla (5.2) se listan el conjunto de deblas de la base de datos ordenadas por similitud a cada uno de los dos artistas (de más parecidas a menos parecidas) junto con el valor del área mínima entre las dos deblas.

Tabla 5.2 Lista de áreas mínimas de deblas ordenadas de más a menos parecidas para los maestros Tomás Pabón y Antonio Mairena.

Órden de similitud a Tomás Pavón		Órden de similitud a Antonio Mairena	
Nombre	Área mínima	Nombre	Área mínima
ChanoLobato	175.996	RafaelRomeroElGallina	316.651
DiegoClavel	233.203	ChanoLobato	333.954
MSimon	234.062	DiegoClavel	373.511
JAlmaden	244.288	Chocolate	385.568
JMerce	244.417	PdeLucia	385.859
PdeLucia	260.604	JMerce	390.645
Chocolate	283.834	JHeredia	403.098
Naranjito	302.860	Naranjito	418.298
JHeredia	361.878	Turronero	424.494
RafaelRomeroElGallina	410.94	JAlmaden	436.779
Turronero	425.660	MSimon	462.328
MVargas	507.174	TalegondeCordoba	509.092
TalegondeCordoba	569.778	MVargas	511.022

En la figura 5.5 se muestra la representación gráfica de los contornos melódicos de la deblas cantadas por Tomás Pavón y por Diego Clavel, antes de aplicarle el algoritmo de escalado para minimizar el área entre los contornos melódicos. Como se observa la deblas de Diego Clavel es más larga que la de Tomás Pavón, por lo que escalaremos linealmente la debla de Pavón.

En la figura 5.6 vemos en más detalle los 7 primeros segundos de las deblas de Pavón y Diego Clavel una vez escalada la debla de menor duración al épsilon que minimiza el área entre los dos contornos melódicos. En este tramo puede apreciarse bastante bien la similitud entre los contornos melódicos de estas dos deblas.

5.2.2 Experimento 2: Comparación de deblas

Para este experimento vamos coger el conjunto de deblas y vamos a aplicarle el algoritmo a cada una de los elementos que la componen, contrastándolas con el resto de deblas de la base de datos. De esta manera generamos una matriz de distancias en las que ambos ejes de la matriz corresponden al listado de deblas y cada celda de la matriz corresponde al área mínima entre las dos deblas correspondiente a sus índices.

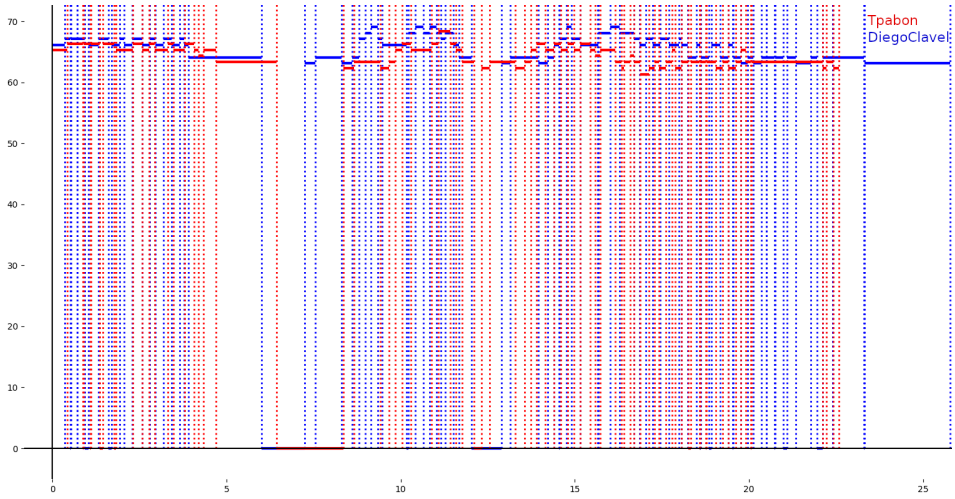


Figura 5.5 Representación de los contornos melódicos de las deblas de Tomás Pavón y Diego Clavel.

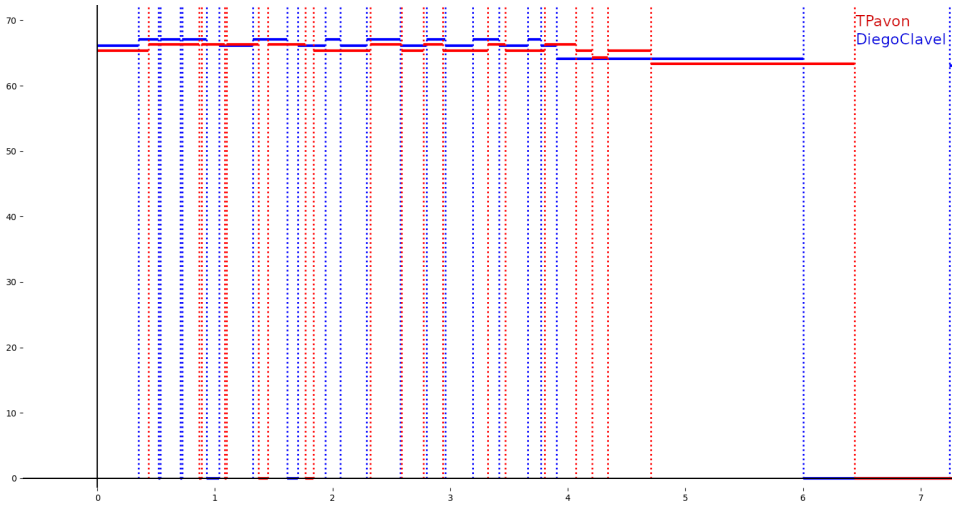


Figura 5.6 Detalle de la primera frase de las deblas de Tomás Pavón y Diego Clavel tras escalar la debla de Tomás Pavón al épsilon que minimiza el área.

Obtenemos así la matriz de distancias, matriz simétrica que contiene números reales no negativos como elementos, y a partir de la cual podemos generar el mapa de calor de la figura 5.7

A la luz del mapa de calor de la figura 5.7, donde se representa el grado de similitud según intensidad de color, parece que las dos deblas más parecidas son la de ChanoLobato y la de JAlmaden. Vemos también cómo la debla más parecida a la de Tomás Pavón es la debla de Chano Lobato, tal y como veíamos en el experimento 1. Se aprecia también la similitud entre los cantes de Pavón y de Mairena. Las deblas que más distan, es decir, las menos similares parecen ser las de Diego Clavel y la de Talegón de Cordoba.

5.2.3 Experimento 3: Comparación de martinetes

Este experimento es totalmente análogo al anterior, con la salvedad de que esta vez usaremos los martinetes del corpus de tonás en vez de las deblas. De este modo, generamos de nuevo una matriz de distancias al comparar todos los martinertes entre sí y generamos un mapa de calor (figura 5.8) a partir de la matriz de distancias. Gracias a que la muestra de martinetes es bastante mayor que la de deblas, nos permite observar en el mapa de calor de la figura 5.8 muchos más pares de canciones mucho más similares entre sí.Por destacar algunos vemos celdas muy oscuras (es decir, martinetes muy similares) como son la de El Torta con la de Tía Anica la Piriniaca o la de Miguel Vargas con la de Juan Talega. Vemos también cómo las líneas horizontal y vertical correspondientes al martinete de Manuel Agujetas son de un color muy claro, es decir, es bastante distinta de cualquier otro martinete, por tener un estilo muy propio.

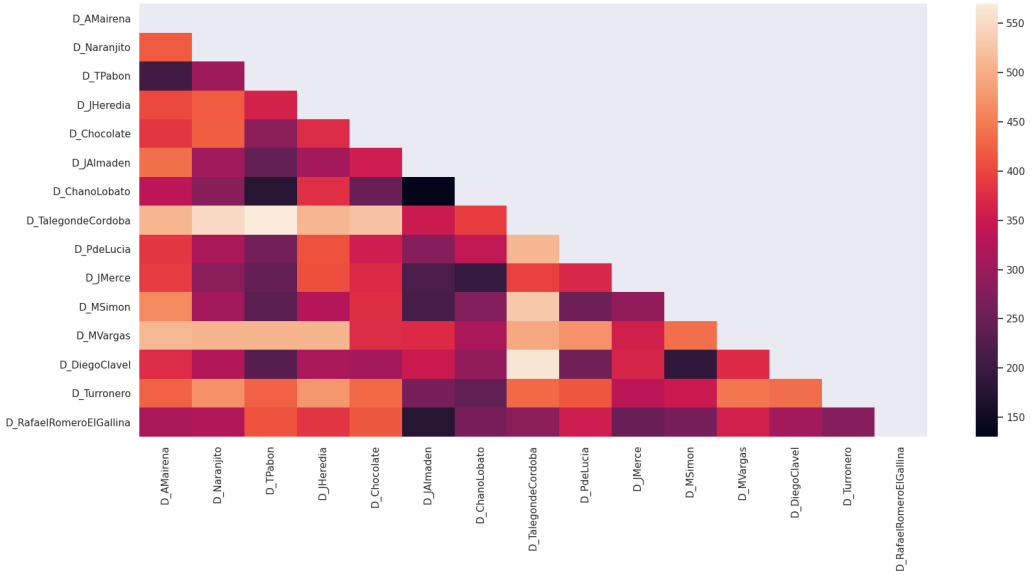


Figura 5.7 Mapa de calor de la matriz de distancias de deblas.

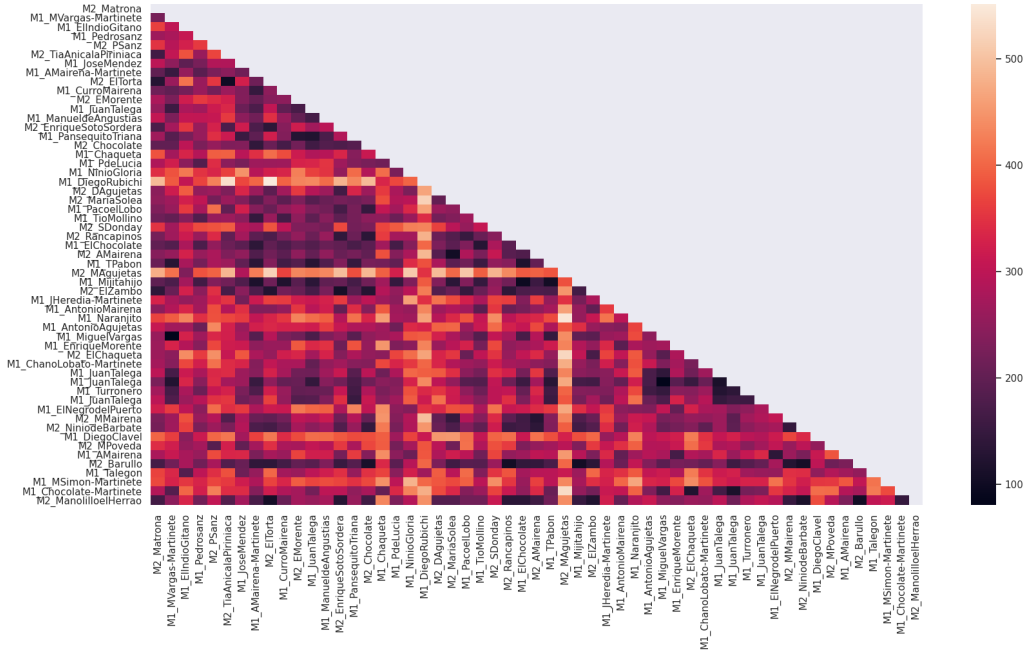


Figura 5.8 Mapa de calor de la matriz de distancias de los martinetes.

5.2.4 Experimento 4: Deblas y martinetes, comparativa de toda la base de datos

En esta ocasión vamos a tomar los subconjuntos de deblas y martinetes y vamos a comparar todas las melodías entre sí, generando una matriz de distancias y el mapa de calor asociado con todas las canciones que componen la base de datos del corpus de tonás (figura 5.9) .

En las figuras 5.9 y 5.10 pueden observarse la existencia de 3 regiones ligeramente diferenciadas por la intensidad de colores del mapa de calor. El sector de arriba a la izquierda de la figura 5.10 corresponde con el subconjunto de deblas, que al ser más parecidas entre sí se ven de un color más oscuro. A su vez, el sector de abajo a la derecha de la figura 5.10 también se muestra de un color más oscuro al tratarse de la región correspondiente al subconjunto de martienetes, también más parecidos entre sí. Sin embargo, en el rectángulo de abajo a la izquierda tiene una intensidad de color más clara, correspondiendo a la región en la que se están comparando deblas con martientes, menos parecidos entre sí, reflejando así que parece existir una separación entre estos dos estilos, lo que muestra que, además de las tareas de búsqueda por escalado,

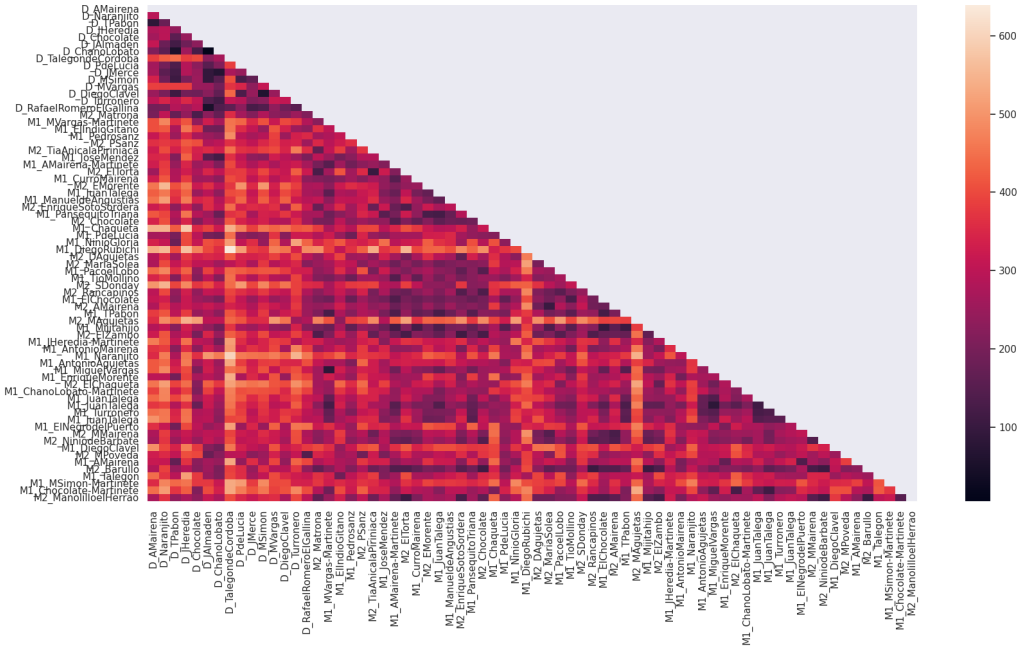


Figura 5.9 Mapa de calor de la matriz de distancias de toda la base de datos(deblas y martinetes).

nuestro método también podría usarse para clasificación de estilos, aunque para esto habría que hacer un estudio más elaborado que caería fuera del objetivo de este trabajo.

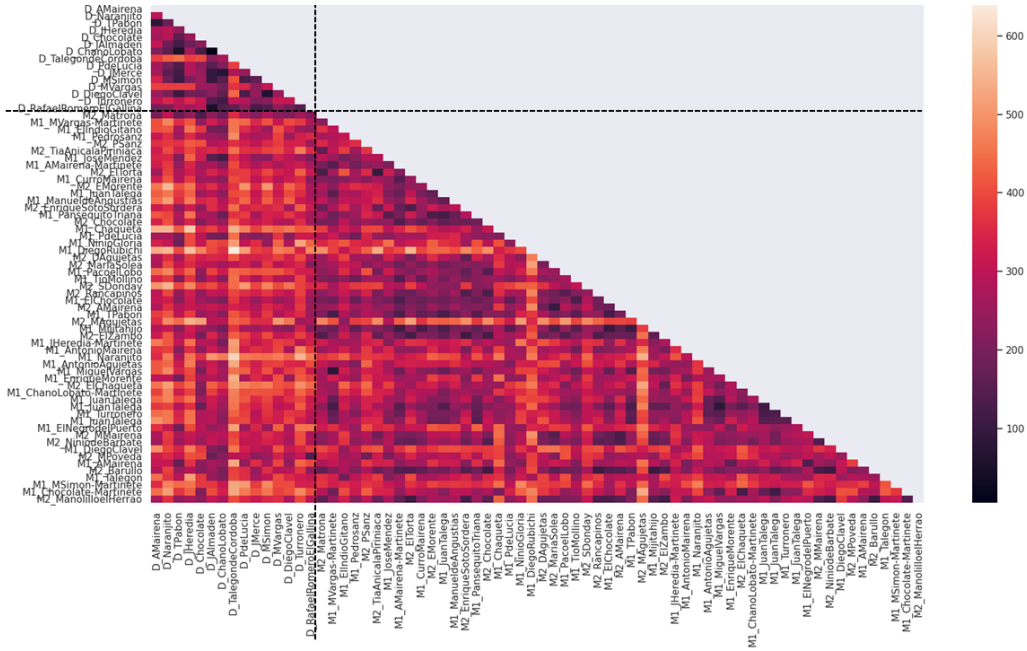


Figura 5.10 Separación de regiones de deblas y martinetes en el apa de calor de la matriz de distancias de toda la base de datos.

5.3 Comparativa de coste computacional

La aplicación de técnicas geométricas en sistemas de recuperación de información musical ha sido ampliamente utilizado como vimos en el capítulo introductorio. Sin embargo uno de los problemas de las técnicas geométricas es su coste compuacional, por ello el algoritmo propuesto para el problema de escalado supone una buena mejoría del tiempo de cómputo y por tanto de la eficiencia del mismo con respecto a otros

algoritmos propuesto anteriormente.

En esta sección vamos a comparar el tiempo de computo que supone realizar los experimentos del capítulo anterior aplicando el algoritmo implementado con respecto a un algoritmo exhaustivo que recalcula las áreas en cada evento. Este segundo algoritmo es el descrito en la sección 5.1.3 (ver código 5.2) que se utilizó para comprobar los resultados calculados con el algoritmo eficiente.

Más concretamente vamos a realizar esta comparativa para los experimentos de: aplicación del algoritmo de escalado lineal a dos melodías (tabla 5.3), clasificación de deblas según similitud a Tomás Pavón y Antonio Mairena (tabla 5.4), cálculo de la matriz de distancias del subconjunto de deblas (tabla 5.5), cálculo de la matriz de distancias del subconjunto de martinetes (tabla 5.6) y cálculo de la matriz de distancias del corpus de tonás (tabla 5.7).

Tabla 5.3 Comparativa de tiempos para el experimento de comparación de dos melodías.

Algoritmo	Tiempo
Eficiente	0.004072109858194987
No eficiente	0.07136336962381998667

Tabla 5.4 Comparativa de tiempos para el experimento de clasificación de deblas según su escuela.

Algoritmo	Tiempo
Eficiente	0.43394931157430012
No eficiente	5.058680772781372

Tabla 5.5 Comparativa de tiempos para el experimento de comparación de todas las deblas entre sí.

Algoritmo	Tiempo
Eficiente	2.2145053545633951333
No eficiente	20.599667231241862

Tabla 5.6 Comparativa de tiempos para el experimento de comparación de todas los martinetes entre sí.

Algoritmo	Tiempo
Eficiente	12.608126401901245
No eficiente	103.64595588048299333

En cada una de estas tablas pueden observarse los tiempos que tardó en ejecutarse cada uno de los dos algoritmos, el desarrollado e implementado en este trabajo y un algoritmo exchaustivo que recalcula el área en cada evento. Para el experimento de aplicación del algoritmo de escalado lineal a dos melodía se ha sacado una media tras comparar dos a dos la debla de Tomás Pavón con el resto de deblas del corpus de tonás. Las conclusiones que podemos extraer de estas tablas se discuten en la tabla final de resultados (tabla 6.1) del capítulo 6.

Tabla 5.7 Comparativa de tiempos para el experimento de comparación de todo el corpus de tonás.

Algoritmo	Tiempo
Eficiente	23.251962661743164
No eficiente	269.84739510218302667

6 Conclusiones y trabajo futuro

La música flamenca es un arte milenario que puede verse profundamente beneficiado de ser estudiado con rigor desde las matemáticas y la computación. El presente documento supone un pequeño ejemplo del trabajo que puede realizarse en pos de preservar y popularizar el flamenco haciendo uso de la ciencia y la tecnología. A las vista de los resultados obtenidos en los experimentos realizados en el capítulo 5 se pueden extraer como resumen tres ideas principales:

- El algoritmo propuesto supone una notable mejoría de la eficiencia de la complejidad computacional respecto a un algoritmo de exhaustivo de escalado.
- El problema de similitud geométrica en el escalado lineal puede usarse para distintas aplicaciones: operaciones de consulta comparando dos melodías, clasificación por escuelas o clasificación por estilos, siendo quizás esta última en la que menos rendimiento se observa.
- El escalado lineal disminuye considerablemente la medida de similitud entre dos cantes, lo que indica que es una técnica indicada para detectar versiones de una misma canción.

A continuación desarrollamos un poco estos aspectos.

6.1 Eficiencia computacional

En los 4 exeperimentos en los que se ha evaluado el tiempo de cómputo, la aplicación del algoritmo propuesto frente a un algoritmo de fuerza bruta ha supuesto una mejoría considerable, reduciendo el tiempo de cómputo notablemente tal y como refleja la tabla 6.1

Tabla 6.1 Tiempo medio de cómputo de cada algoritmo para diferentes experimentos.

Algoritmo / Experimento	Eficiente	No eficiente
Comparación de dos melodías	0.004072109858194987	0.07136336962381998667
Clasificación en las escuelas de Pavón o Mairena	0.43394931157430012	5.058680772781372
Comparación de todas las deblas entre sí	2.2145053545633951333	20.599667231241862
Comparación de todas los martinetes entre sí	12.608126401901245	103.64595588048299333
Comparación de todas la base de datos entre sí	23.251962661743164	269.84739510218302667

Esto supone una reducción del tiempo de cómputo del 94.29 % al comparar dos melodías sueltas, un 91.42 % en la clasificación de las deblas en las escuelas de Tomás Pavón o Antonio Mairena, un 89.25 % en la comparación de todas las deblas entre sí, un 87.83 % al comparar todos los martinetes entre sí y un 91.38 % al comparar toda corpus de deblas y martinetes entre sí, lo que supone un autentico éxito de cara a optimizar recursos de tiempo y memoria al aplicar este algoritmo para resolver el problema de escalado lineal a dos melodías.

6.2 Clasificación de melodías entre deblas y martinetes

El flameco es un arte vivo y en constante cambio, con distintas interpretaciones de un mismo cante y con numerosas ornamentaciones e improvisaciones propias de cada cantaor. Esto supone una dificultad a la hora de tratar el problema de clasificación musical desde una perspectiva puramente geométrica y se requiere incluir en el método algorítmico aspectos músico-culturales propios de los cantes en cuestión. En el mapa de calor de la figura 5.9 se muestra una visualización de la similitud entre melodías a partir de la matriz de distancias generada tras comparar al completo el corpus de tonás. Aunque parece existir regiones ligeramente diferenciadas, no parece ser una diferenciación lo suficientemente conclusoria como para clasificar tonás en los subestilos de deblas y martinetes.

Concluimos por tanto que abordar geométricamente la clasificación de tonás entre deblas y martinetes puede suponer un complemento para dicha clasificación, pero no un resultado concluyente.

6.3 Aplicación de consulta entre melodías

En las imágenes 5.5 y 5.6 de la sección 5 veíamos la similitud de los contornos melódicos entre dos deblas muy parecidas (la de Tomás Pavón y la de Diego Clavel).

En la tabla 6.2 tomamos la debla de Tomás Pavón y la comparamos con el resto de deblas. El resultado mostrado es la diferencia entre el valor del área antes de que la melodía de consulta sea escalada con el área mínima tras aplicarle el algoritmo de escalado y escalarla un épsilon que minimiza el área. De este modo podemos tener una referencia de cuánto cambia la debla tras aplicarle el algoritmo:

Tabla 6.2 Diferencia entre el área inicial y el área mínima tras comparar la debla de Tomás Pavón con el resto de deblas.

Debla de TPavon contrastado con	areainicial - areamin
Naranjito	302.86059 - 302.8605 = **1.6672402e-08**
JHeredia	363.822780 - 361.8780 = **1.9446**
Chocolate	285.398988 - 283.8347 = **1.56424**
JAlmaden	368.34848 - 244.2887 = **124.0596**
ChanoLobato	335.309820 - 175.9965 = **159.3133**
TalegondeCordoba	569.79608 - 569.7780 = **0.01807201**
PdeLucia	344.199635 - 260.6047 = **83.5949**
JMerce	378.52967 - 244.4174 = **134.112**
MSimon	247.24796 - 234.0625 = **13.18545**
MVargas	528.1233 - 507.1743 = **20.94892**
DiegoClavel	245.501236 - 233.2030 = **12.29821**
Turronero	524.53528 - 425.6602 = **98.8749**
RafaelRomeroElGallina	426.775571 - 410.948 = **15.8275**

En la tabla 6.2 podemos apreciar que hay ocasiones en las que apenas es necesario escalar la melodía para encontrar el área de máxima similitud, pero existen muchos otros casos en los que hay que modificar mucho la melodía original para maximizar la similitud entre las melodía de consulta y de referencia (como son los casos de J. Almaden o J. Merce, por ejemplo). Gracias a aplicarle el algoritmo a este tipo de situaciones podemos identificar que dos melodías que a priori parecían muy distintas, en realidad sean muy parecidas pero que una esté cantada mucho más lenta que la otra por ejemplo.

Por tanto, cocluimos que aplicar el algoritmo parece dar muy buenos resultados para situaciones en las que hay dos tonás que son muy parecidas en cuanto a la forma de su perfil melódico pero la ejecución de los cantes están realizados a tempos distintos.

6.4 Trabajo futuro

Aún queda mucho por hacer en el campo de la teoría computacional de la música flamenca que ha comenzado a desarrollar el proyecto Cofla. Algunas sugerencias que continuen la senda de este proyecto pueden ser:

- Probar si se obtienen mejores resultados probando un escalado específico para cada nota en lugar de un escalado lineal para toda la melodía en su conjunto.
- Complementar la clasificación de deblas y martinetes con técnicas melódicas además de geométricas.
- Aplicación del algoritmo de escalado lineal geométrico para el estudio de otros palos e incluso de otros géneros musicales.
- Ampliación del corpus de deblas y martinetes para poner a prueba de nuevo el algoritmo propuesto.

Índice de Figuras

Índice de Tablas

Índice de Códigos

Bibliografía

[1] G. Aloupis, T. Fevens, S. Langerman, T. Matsui, A. Mesa, Y. Nunez, D. Rappaport, and G. Toussaint, *Algorithms for computing geometric measures of melodic similarity*, Computer Music Journal **30** (2006), no. 3, 67–76.

[2] B. Bartók and B Lord, *Texts and transcriptions of 75 folk songs from the milman parry collection, and a morphology of serbo croatian folk melodies*, Columbia University Press, 1951.

[3] M. A. Berlanga et al., *Música y religiosidad popular: Saetas y misereres en la semana santa andaluza*, (2001).

[4] L.E. Caraballo, J.M. Díaz-Báñez, F. Rodríguez, V. Sánchez-Canales, and I. Ventura, *Scaling and compressing melodies using geometric similarity measures*, Proc. European Workshop on Computational Geometry, EuroCG 2020.

[5] G. Castro, *Las mudanzas del cante en tiempos de silverio, análisis histórico*, vol. 4, Primento, 2014.

[6] C. Cronin, *Concepts of melodic similarity in music copyright infringement suits*, Computing in musicology: a directory of research, 1998.

[7] E. Gómez and C. López, *Criterios para la transcripción manual de la colección de tonás*, Barcelona, 2013.

[8] U. Hahn, N. Chater, and L. B. Richardson, *Similarity as transformation*, Cognition **87** (2003), no. 1, 1–32.

[9] I. Halmos, *Computational ethnomusicology in hungary in 1978*, University of Michigan Library, 1978.

[10] A. Holzapfel and Y. Stylianou, *Similarity methods for computational ethnomusicology*, Unpublished doctoral dissertation). University of Crete, Crete (2010).

[11] C. Kramer and L. J. Plenckers, *The structure of the saeta flamenca: An analytical study of its music. yearbook for traditional music*, 1998.

[12] P. Lefranc, *El cante jondo: del territorio a los repertorios: tonás, siguiriyas, soleares*, vol. 16, Universidad de Sevilla, 2000.

[13] F. Lerdahl and R. S. Jackendoff, *A generative theory of tonal music, reissue, with a new preface*, MIT press, 1996.

[14] Machado and A. Álvarez, *Colección de cantes flamencos recogidos y anotados por a. sevilla: El porvenir*, reedited Madrid: Cultura Hispinica, 1975, 1881.

[15] D. O. Maidín, *A geometrical algorithm for melodic difference*, Computing in musicology: a directory of research (1998), no. 11, 65–72.

[16] R. Molina and A. Mairena, *Mundo y formas del cante flamenco*, Revista de Occidente (1963).

[17] J. Mora, F. Gómez, E. Gómez, and J. M. Díaz-Báñez, *Melodic contour and mid-level global features applied to the analysis of flamenco cantes*, Journal of New Music Research **45** (2016), no. 2, 145–159.

[18] J. Mora, F. Gómez, E. Gómez, F. Escobar-Borrego, and J. M. Díaz-Báñez, *Characterization and melodic similarity of a cappella flamenco cantes*, Proceedings of ISMIR, 2010, pp. 9–13.

[19] S. Seeger, *Versions and variants of the tunes of "barbara allen"*, Selected Reports in Ethnomusicology, 1966.

[20] G. Tzanetakis, A. Kapur, W. A. Schloss, and M. Wright, *Computational ethnomusicology*, Journal of interdisciplinary music studies **1** (2007), no. 2, 1–24.