UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Facultat de Belles Arts de Sant Carles

Doctorado en Arte: Producción e Investigación

El Fuzzy Clustering y la Similitud Musical: Aplicación a la Composición Asistida por Ordenador

Tesis Doctoral

Autor: D. Brian Santiago Martínez Rodríguez

Directores: Dr. D. Vicente Liern Carrión
Dra. Dña. Teresa Cháfer Bixquert

El Fuzzy Clustering y la Similitud Musical: Aplicación a la Composición Asistida por Ordenador

Composición Musical Asistida por Ordenador Mediante Métodos de Agrupamiento Difuso

Resumen

La composición musical asistida por ordenador es un área de conocimiento que tiene sus orígenes en la segunda mitad del siglo pasado. Durante sus más de sesenta años de existencia han aparecido numerosas propuestas para abordar el problema de la creatividad artificial aplicada al ámbito de la variación musical, la emulación de estilos, la escritura automatizada de contrapunto o la composición estocástica, entre muchos otros. En la presente memoria propondremos un nuevo método para la generación computacional de variaciones y transiciones a partir de material musical proporcionado por el compositor, ya sea de carácter melódico, rítmico, armónico o tímbrico. La originalidad de nuestro método radica en la construcción de nuevos algoritmos basados en las técnicas de agrupamiento difuso, capaces incorporar el orden de los elementos de los conjuntos de datos durante el proceso de partición. Para implementar computacionalmente estas técnicas hemos diseñado el software MERCURY mediante el que realizaremos distintos experimentos cuyos resultados, en forma de transiciones musicales, ilustrarán la utilidad de nuestra propuesta. Completaremos la presente investigación con la composición de la obra *Transiciones difusas*, para cuarteto de cuerdas, adjunta como apéndice. La metodología propuesta implica formular una nueva medida de la disimilitud musical, aplicable de forma general a la comparación de dos secuencias numéricas cualesquiera, con las que se pueda representar cualquier tupla de atributos musicales. Es posible, por tanto, aplicar esta disimilitud sobre ámbitos más teóricos como los sistemas de afinación. Finalmente propondremos diversos métodos para estimar la compatibilidad entre un conjunto de notas y un sistema de afinación generando, en última instancia, transiciones entre diferentes sistemas.

Resum

La composició musical assistida per ordinador és una àrea de coneixement que té els seus orígens a meitat del segle passat. Durant els seus més de seixanta anys d'existència han aparegut nombroses propostes per a abordar el problema de la creativitat artificial aplicada a l'àmbit de la generació de variacions, emulació d'estils, escriptura automatitzada de contrapunt i composició de música estocàstica, entre molts altres. En aquesta memòria proposarem un nou mètode per a crear variacions i transicions entre material musical preexistent, ja siga de caràcter melòdic, rítmic, harmònic o tímbric. L'originalitat del nostre mètode radica en la construcció d'algoritmes basats en la tècnica de fuzzy clustering, capaços de realitzar agrupaments en què es té en compte l'ordre dels elements dels conjunts de dades. Per a implementar aquestes tècniques, hem dissenyat el programari Mercury mitjançant el qual es realitzaran experiments amb transicions entre melodies, ritmes i seqüències harmòniques que il·lustraran la utilitat de la nostra proposta, i que culminaran amb la composició de l'obra *Transicions difuses*, adjunta com a apèndix. La metodologia proposada no només té conseqüències pràctiques, sinó que implica formular una nova mesura de la dissimilitud musical, aplicable de forma general a la comparació de qualsevol parell de seqüències numèriques, que puguen representar melodies, ritmes, harmonies o timbres. Un cop establert com valorar la dissimilitud, aquesta també pot aplicar-se a àmbits molt més teòrics, com són els sistemes d'afinació. Proposarem diversos mètodes per a estimar la compatibilitat entre un conjunt de notes i un sistema d'afinació i generar, en última instància, transicions entre dos sistemes d'afinació. Aquesta tasca pot facilitar la interpretació d'obres en un sistema d'afinació diferent d'aquell per al qual van ser concebudes, sempre que s'exigisca que el nivell de compatibilitat entre els dos sistemes siga acceptable.

Summary

Computer-assisted composition is an area of knowledge that has its origins in the middle of the last century. During its more than sixty years of existence, numerous proposals have appeared to address the problem of artificial creativity applied to the field of generation of variations, emulation of styles, automated counterpoint writing, stochastic music composition, among many others. In this report we will propose a new method to create variations and transitions between pre-existing musical material, be it melodic, rhythmic, harmonic or timbre-related. The originality of our method lies in the construction of algorithms based on the technique of fuzzy clustering, capable of performing groupings in which the order of the elements of the data sets is taken into account. To implement these techniques, we designed the software MERCURY through which experiments will be performed with transitions between melodies, rhythms and harmonic sequences that will illustrate the usefulness of our proposal, and that will culminate with the composition of the work *Fuzzy Transitions*, attached as an appendix. The proposed methodology not only has practical consequences, but also implies formulating a new measure of musical dissimilarity, applicable in a general way to the comparison of any pair of numerical sequences, which may represent melodies, rhythms, harmonies or timbres. Once established how to assess the dissimilarity, this can also be applied to much more theoretical areas, such as tuning systems. We will propose various methods to estimate the compatibility between a set of notes and an tuning system and, in the last instance, generate transitions between two tuning systems. This work can facilitate the interpretation of works in a tuning system different from that for which they were conceived, whenever it is required that the level of compatibility between both systems is acceptable.

Agradecimientos

En esta tesis doctoral he tenido la fortuna de aglutinar las que –con permiso de la poesía– quizá sean mis tres grandes pasiones: la composición musical, las matemáticas y la programación informática. Sin embargo, la presente memoria no resume únicamente este proceso de investigación, que me ha ocupado por varios años: representa, además, la culminación de una larga fase vital de aprendizaje y superación que comenzó en mi infancia –cuando aprendí a programar mis propios videojuegos sobre BASIC en el ordenador *Spectrum ZX* que mis abuelos me habían regalado– y que concluye al redactar estas palabras para dar paso a un nuevo ciclo. Sin embargo, este camino no me hubiera sido posible sin el amor, la ayuda y la amistad de numerosas personas, a las cuales debo tanto como aprecio y respeto. La siguiente reseña no es más que una pequeña muestra de la gratitud que siento por que sean, o hayan sido, una parte importante de mi vida.

Me gustaría agradecer en primer lugar a mis directores de tesis Vicente Liern y Teresa Cháfer, con los que he podido desarrollar una estimulante labor investigadora. Gracias Vicente por todo lo que me has enseñado durante estos años, compartiendo conmigo una pequeña parte de tu sabiduría. Gracias Teresa por toda la ayuda y los consejos que me has prestado, que tanto me han orientado a lo largo de este camino. Gracias a ambos por el trato tan cercano que siempre me habéis ofrecido: trabajar con vosotros ha sido sin duda lo mejor de esta tesis.

Gracias a mis padres, Santiago y Chelo, y a mi tía María José, así como a mis abuelos Josefa y Alejandro. Sin su enorme esfuerzo –colmando su vida de renuncias y sacrificios– y sin su infinita generosidad, no sólo esta tesis o mis estudios

anteriores no hubieran sido posibles; nada de cuanto he podido conseguir me hubiera sido acaso ofrecido.

Quiero continuar agradeciendo a algunos de mis profesores: gracias a Claudia, por confiar tanto en mí y demostrarme que es frente a la adversidad donde los sueños pueden ser alcanzados; a Jose Manuel que, aunque oficialmente no ha sido mi profesor, me ha dado grandes lecciones de un exquisito maridaje entre matemática y gastronomía; a Ximo, por sus seminarios de *mastering* y producción; a Tomás, por sus maravillosas clases magistrales de análisis y armonía; a Enrique, por todas esas conversaciones interminables en las que hablábamos de todo, a veces incluso de música; a Roberto, por ofrecerme su casa y la sabiduría de toda una vida; Andrés, por demostrarme el valor de la intuición en la música; a César, por descubrirme los secretos que rigen el laberinto del serialismo; a Ramón, por enseñarme a construir la música; a Tamarit, por grabar en mi subconsciente las normas del contrapunto; a Maria José, por sus inolvidables clases de piano; a Fernando, por compartir conmigo su amistad, su erudición y su gusto por lo perfecto, y en especial a Gregorio, por iniciarme en el mundo de la composición electroacústica y la informática musical.

Gracias también –con unos años de retraso– a Ibáñez, por enseñarme Cálculo Diferencial; a Bernabéu, por sus penetrantes clases de Física General; a Azcárraga, por el rigor, la palabra y la Mecánica Cuántica; a Caselles, por mostrarme como se tiene que dar una clase perfecta, y muy especialmente a Jeff, por sus clases de inglés, filosofía, literatura, poesía, pintura, música y de vida –todo ello aderezado con un sentido del humor muy *british*– a lo largo de muchos, muchos años.

Es tiempo ahora de agradecer a mis grandes amigos Pepe y Alberto, dos mentes privilegiadas, por la ayuda que me han proporcionado en las fases iniciales de esta tesis –cuando era aún un concepto difuso pendiente de germinar– y por tantos y tantos buenos momentos en los que hemos celebrado la suerte de conocernos. También a los no menos geniales *camarada* Jose Javier y *croscópico* David, dos almanseños virtuosos y notables jugadores de mus.

Gracias a Nina, Casañs y Jose Luís, por soportar mis meses anacoretas de encierro y clausura: su paciencia y cariño se han mostrado más resistentes que la exigencia rigurosa de mi quehacer cotidiano. Gracias a mi hermano Alberto, con el que comparto el gusto por el buen *rock*, por descubrirme lo que es el mágico fenómeno de la *aurora borealis*.

Gracias Jaume por tu alegría y apoyo incondicional: la frescura joven de tus palabras es mi fuente inspiradora de ilusión, un bálsamo contra el desgaste de las horas.

Dedicado a la memoria de mi abuelo Alejandro, carpintero y ebanista, y de mi abuela Josefa, sirvienta y ama de casa.

Índice general

Índice de figuras

Índice de tablas

Glosario

- ***Fuzzy / soft***: Relativo a la lógica difusa.
- ***Hard / crisp***: Relativo a la lógica binaria.
- ***Cluster***: Grupo, racimo, conjunto.
- ***Clustering***: Agrupamiento.
- ***Centroide***: Punto central del *cluster*.
- **CAO**: **C**omposición **A**sistida por **O**rdenador.
- **AI**: ***A****rtificial* ***I****ntelligence*.
- **FCM**: Algoritmo ***F****uzzy* ***c****-****M****eans*.
- **FOCM**: Algoritmo ***F****uzzy* ***O****rdered* ***c****-****M****eans*.
- **FCT**: Algoritmo ***F****uzzy* ***C****omplete* ***T****ransitions*.
- **MIR**: ***M****usic* ***I****nformation* ***R****etrieval*.
- **FFT**: ***F****ast* ***F****ourier* ***T****ransform*.
- **HPS**: ***H****armonic* ***P****roduct* ***S****pectrum*.

Capítulo 1

Introducción

«La música nace libre; conseguir la libertad es su destino.»

«Music was born free; and to win freedom is its destiny.»

(Busoni, 1911, pág. 5)

1.1 Justificación

La idea de inventar una máquina capaz de crear resulta fascinante en sí misma: no sólo emplaza nuestro intelecto a una función de *metacreador*, es decir, un creador de creadores; también nos obliga a teorizar y sistematizar los procesos creativos humanos –quizá aquellos más íntimos y definitorios del ser– mediante los cuales, a lo largo de milenios, hemos representado, bajo los términos de la mímesis, la naturaleza de los mundos exterior e interior; redimiendo, al mimo tiempo, pasiones y temores bajo el influjo de la catarsis.

La automática y la algoritmia no son disciplinas novedosas en la edad contemporánea. Desde el *Mecanismo de Anticitera* (Seaman y Rössler, 2011) hasta nuestros sofisticados *smartphones* han existido en la historia numerosos ejemplos de máquinas, autómatas o robots, capaces de reemplazar la labor humana en múltiples ámbitos. Tampoco es novedosa su aplicación a la creación artística, en este caso de índole musical: sirvan como ejemplo la maravillosa *Arca Musarithmica* inventada por el jesuita Atanasio Kircher a durante el siglo XVII (Bumgardner, 2009), o el celebérrimo *Musikalisches Würfelspiel* creado por W. A. Mozart (Levy, 2005).

La aparición de las computadoras digitales y su vertiginosa expansión en términos de capacidad de cálculo y almacenamiento es el hecho diferencial que, a mediados del siglo XX, posibilita el desarrollo de una nueva disciplina del conocimiento: la creatividad artificial. Todas las propuestas, ideas o fabulaciones realizadas hasta entonces se verán superadas en expectativas por las posibilidades que, al menos desde el punto de vista teórico, ofrece la computación moderna; dando lugar al mismo tiempo a una verdadera revolución en el terreno cultural, un seísmo tecnológico que afecta desde las propuestas realizadas por los creadores de vanguardia, hasta la vigencia de los postulados éticos y estéticos del arte mismo.

La música no permanece ajena a las posibilidades de esta nueva realidad. Muy pronto aparecen los primeros hitos en la incipiente historia de la *computer composition*: el trabajo experimental de Lejaren Hiller y Leonard Isaacson, plasmado en la *Suite Illiac* (Hiller e Isaacson, 1959); el primer lenguaje de programación musical desarrollado por Max Mathews, denominado *MUSIC I* (Roads y Mathews, 1980); o la utilización de las computadoras para la generación de música estocástica en la serie de obras *ST* de Xenakis (Xenakis, 1971), son sólo algunos ejemplos de la incorporación de la máquina digital en la creación musical. De esta manera, la composición asistida por ordenador se muestra como un problema abordable en términos de la inteligencia artificial. Tal y como propone Roads en su artículo *Research in music and artificial intelligence* (1985), el objetivo de un asistente inteligente para la composición debe ser «apoyar al compositor en fases de composición altamente creativas»[1], facilitando además la creación del esquema formal de la composición y codificando adecuadamente el material musical en términos computacionales.

Más allá de conseguir que el ordenador sea una especie de *calculadora* musical para facilitar la labor del compositor, el propio Hiller se pregunta: «¿Por qué programar una computadora digital para generar música?»; resulta necesario entonces argumentar una respuesta sólida, en términos de la filosofía de la ciencia, que anule la falacia de considerar como únicamente válido aquel conocimiento cuya utilidad sea eminentemente práctica. Dobrian (1993) plantea algunas de las posibles respuestas: si la tecnología nos atrae *per se* y nos deslumbra, requiere entonces un uso ético del que no se pueda desprender ningún comportamiento potencialmente lesivo; si los conocimientos derivados de la composición con computadores son propios de la ciencia base, contribuyen por tanto en la ampliación del conocimiento humano general; si al desarrollar métodos algorítmicos para la composición asistida por ordenador es necesario estudiar y modelar los procesos

[1] «The overall goal of an intelligent composer's assistant is to support the composer in highly creative phases of composition. This includes the creation of the plan and architecture of a composition, and the encoding of musical material into the working score» (Roads, 1985, pág. 170).

cognitivos que subyacen en el acto creativo humano, entonces comprenderemos mejor el funcionamiento de nuestro propio intelecto.

A lo largo de las últimas seis décadas han aparecido innumerables acercamientos al problema de la composición musical asistida por ordenador[2], utilizando cada vez construcciones teóricas más complejas fundamentadas a su vez por la disponibilidad de máquinas cada vez más potentes. Sin embargo a fecha actual parece no existir ningún método único o definitivo que sea capaz de emular con resultados realmente brillantes la creatividad musical humana. Escrutados de forma exhaustiva los acercamientos tradicionales, parece extenderse una cierta tendencia a la generación de técnicas híbridas, así como una relativa necesidad de encontrar nuevos paradigmas exploratorios que propongan caminos hasta ahora no transitados.

El ámbito de investigación de esta tesis, lejos de estar cerrado, continúa suscitando gran interés en las comunidades científica y artística de nuestra actualidad, tal y como pone de manifiesto la existencia de numerosas revistas especializadas, congresos, simposios y grupos de investigación de carácter público[3] y privado[4], todos ellos relativos a la creación musical computarizada.

Entre las múltiples cuestiones que plantea la marea computacional de nuestro tiempo, existe una que resulta para el melómano particularmente inquietante: el *test de Turing* del compositor musical. Toda la tradición de la denominada *música culta* occidental se basa en la escucha, estudio y admiración de un reducido grupo de creadores cuyas contribuciones al repertorio musical han logrado sobrevivir al transcurso de los años y a los cambios en las modas, estéticas y hábitos musicales. Su corpus musical es el legado –en algunas ocasiones reducido y selecto, en otras vasto y pródigo– que trasciende la figura del artista y lo vincula estrechamente con la sociedad en la que estuvo inmerso. Así pues, ¿adorarán los humanos de un no muy lejano futuro a esos compositores digitales que todavía hoy hemos de crear? ¿Se hallará entonces –si es que aún ha de existir–, en una híbrida mezcla de intuición y algoritmo, aquella eterna relación del artista con su tiempo?

[2]En el artículo *AI methods in algorithmic composition: A comprehensive survey* de Fernández y Vico (2013) podemos encontrar una exhaustiva relación de diferentes técnicas, experimentos y software desarrollados hasta 2013 en el ámbito de la composición asistida por ordenador.

[3]Véase por ejemplo el proyecto *Melomics* y el sistema *IAMUS*, desarrollados en la Universidad de Málaga, (Sánchez y col., 2013; Ball, 2012; Vico, Sanchez y Albarracín, 2011).

[4]Véase a modo de anécdota la siguiente noticia *Huawei used AI technology to complete Schubert's unfinished symphony* (Rowat, 2019).

1.2 Problema de investigación

En general, los compositores conocen que la verdadera dificultad de la composición musical no radica tanto en la creación de un buen motivo, tema o melodía, sino en el uso que se haga de éste. En el equilibrio entre el arte de la variación, ampliamente estudiado en numerosos tratados[5], y la repetición musical, tan necesaria como peligrosa, es donde se cimientan las bases del discurso musical occidental al menos hasta la aparición de las vanguardias[6] en la segunda mitad del siglo XX. Como veremos en el próximo capítulo, la investigación en creatividad musical artificial no ha sido ajena a este hecho y son numerosos los acercamientos que se han planteado durante las últimas décadas para tratar de resolver el problema de la generación automatizada de variaciones, mutaciones o transiciones musicales, normalmente utilizando material melódico proporcionado por el usuario y codificado convenientemente para su procesamiento computacional. Los paradigmas *clásicos* más frecuentemente empleados para realizar este tipo de variaciones desde un tema A hacia un tema B se basan principalmente en la utilización de algoritmos genéticos (Horner y Goldberg, 1991); gramáticas formales (Keller y Morrison, 2007); operadores de cambio aplicados de forma recursiva (Mongeau y Sankoff, 1990), o el uso de autómatas celulares (Miranda, 1990), entre otros. El problema de la generación computacional de variaciones y transiciones musicales continúa abierto y constituirá en la presente tesis nuestro principal problema de investigación, así como nuestro fundamental objeto de estudio.

Subsidiariamente, la definición y cálculo computacional del concepto de similitud melódica, es decir, la diferencia o semejanza entre dos melodías cualesquiera, es una cuestión de vital importancia en los ámbitos de la composición y del análisis musical asistidos por ordenador[7]. De hecho, la estimación de una similitud entre dos melodías puede utilizarse directamente sobre el reconocimiento automático de patrones melódicos, así como en determinados procesos computacionales para la generación de variaciones y transformaciones melódicas. Existe bastante literatura que aborda los métodos más importantes para la medida y definición de la similitud melódica (Maidín, 1998; Aloupis y col., 2006), implementando compu-

[5] Véase por ejemplo *Fundamentos de la composición Musical* de Schöenberg, (2001).

[6] Algunos compositores de la segunda mitad del siglo XX desarrollarán planteamientos compositivos en los cuales la narrativa en la música será deliberadamente evitada, tratando de liberar la música de esa rígida concepción formal impuesta por la tradición de la música culta occidental. Sirvan como ejemplo la *Momentform* propuesta por Karlheinz Stockhausen en su obra *Kontakte*(1958-60) (Momentform: Neue Zusammenhange zwischen AuffUhrungsdauer, Werkdauer und Moment), o la utilización del *I Ching* para la composición de la obra *Music of Changes* (1951) de John Cage (Cage, 1961b).

[7] Véanse por ejemplo el proyecto *DARMS* (Erickson, 1975) o los trabajos en el análisis musical asistido por ordenador desarrollados por Gross en *A set of computer programs to aid in music analysis* (1975) o por Wenker en A computer-aided analysis of Anglo-Canadian folktunes (1979).

tacionalmente alguno de ellos y probando su eficiencia sobre el corpus de melodías procedente de los corales de J.S. Bach, con interesantes resultados [8]. Artículos como *MelodyShape at MIREX 2014 Symbolic Melodic Similarity* de Urbano (2014) o *Geometric algorithms for melodic similarity* de Laitinen y Lemström (2010) ponen de manifiesto la actualidad de la investigación en similitud melódica, problema procedente del *Music information retrieval* (MIR), así como se relación con la composición asistida por ordenador y la generación automatizada de variaciones melódicas.

Por tanto, los principales problemas de investigación que abordaremos en esta tesis son los siguientes:

1. *Creación de un método computacional para realizar transiciones melódicas completas entre dos melodías cualesquiera, que permita generar progresivas variaciones y transformaciones del material melódico inicial hasta convertirlo por completo en el material melódico final. Este proceso proporcionará al compositor novedoso material melódico potencialmente utilizable en la creación musical, subyugado a revisión por sus criterios estéticos.*

2. *Diseño de un método para el cálculo de la disimilitud melódica, implementable computacionalmente, que tenga en consideración el orden de las notas musicales y que pueda ser aplicado a la comparación de melodías con distinta longitud o número de elementos.*

3. *Construcción de algoritmos que obtengan transiciones para cualquier otro parámetro musical como ejemplo el ritmo, la armonía, el timbre o el sistema de afinación; creando de esta manera transiciones entre distintos ritmos, transiciones entre distintas secuencias de acordes, transiciones entre distintos timbres o incluso transiciones entre distintos sistemas de afinación.*

[8] Véase nuestro anterior trabajo *Técnicas computacionales para el cálculo de la similitud melódica*, Martínez (2018).

1.3 Objetivos

Para responder a los problemas de investigación que hemos formulado anteriormente, nos planteamos dos tipos de objetivos: generales, que serán de aplicación en todo el ámbito del presente proceso de investigación, y específicos, que perseguirán resultados concretos

1.3.1 Objetivos generales

- Encontrar nuevos procedimientos algorítmicos que nos permitan realizar una transición musical completa desde una determinada melodía inicial a otra final.
- Generalizar los procedimientos propuestos a otras características de la música tales como el ritmo, la armonía, el timbre, los sistemas de afinación, o cualquier otra propiedad musical.
- Demostrar la aplicabilidad de estas nuevas técnicas para su uso en la composición musical, utilizándolas en la composición de una nueva obra.

1.3.2 Objetivos específicos

Para profundizar en un nivel de concreción más detallado, nos hemos planteado una serie de objetivos específicos cuya consecución será necesaria para alcanzar los objetivos generales:

- Realizar una revisión bibliográfica para determinar las principales líneas de investigación en el ámbito de la composición asistida por ordenador desde sus inicios hasta la actualidad y enmarcar nuestra investigación en este ámbito.
- Encontrar una representación matemática adecuada para la representación simbólica de la melodía, que contemple su organización y secuenciación temporal.
- Conocer los métodos existentes de clasificación (*clustering*) denominados *k-means* y *c-means*. Manejar el concepto de partición de tipo *hard* (*crisp*) y de tipo *soft* (*fuzzy*) de un conjunto de datos en un espacio métrico de dimensión finita.
- Conocer los distintos tipos de medidas de disimilitud existentes basados en los procesos de *clustering*, así como las métricas o funciones distancia más comúnmente utilizadas en los algoritmos de *clustering*.

- Definir una medida de la disimilitud melódica que nos permita establecer un criterio objetivo bajo el cual determinar el grado de proximidad existente entre dos melodías cualesquiera.
- Desarrollar un nuevo método de *clustering*, basado en el *fuzzy c-means*, en el que se incorpore el orden de las secuencias de datos para la realización de la partición de forma congruente con el orden de los centroides. Diseñar su implementación algorítmica.
- Definir una medición de la disimilitud basada en el *clustering*, capaz de proporcionar un valor numérico que cuantifique la disimilitud entre dos secuencias ordenadas cualesquiera.
- Diseñar un procedimiento capaz de realizar transiciones completas entre dos secuencias ordenadas cualesquiera.
- Aplicar las nuevas definiciones de disimilitud y métodos algorítmicos propuestos a la melodía.
- Extender los resultados obtenidos al resto de características musicales de tipo simbólico como el ritmo, la armonía o el timbre.
- Definir matemáticamente el concepto de sistema de afinación, revisando la literatura científica existente al respecto y proponer nuevos métodos para delimitar la compatibilidad de un conjunto de notas con un determinado sistema de afinación, así como calcular la similitud existente entre diversos sistemas de afinación.
- Implementar computacionalmente los nuevos métodos propuestos en un software que sea capaz de calcular la disimilitud y realizar las transiciones musicales, permitiendo al usuario tanto introducir los *input* musicales como exportar los resultados obtenidos (*output*) en un formato simbólico estándar (*MusicXML*) fácilmente legible por cualquier software comercial de notación musical.

1.4 Metodología empleada

1.4.1 Naturaleza de la investigación

En la presente tesis se presenta una hibridación entre la investigación de naturaleza teórica y la investigación aplicada, de características más prácticas y centrada en resolver problemas muy concretos. Por ejemplo, encontraremos una aproximación exploratoria de tipo básica en los capítulos dedicados a la búsqueda de nuevas definiciones para la disimilitud entre secuencias de elementos, así como en lo referente a nuevos acercamientos formales para la definición de los sistemas de afinación y su compatibilidad con un conjunto de notas, presentes en los últimos capítulos. La investigación se volverá más aplicada en las secciones dedicadas a la búsqueda de nuevos algoritmos con la finalidad de realizar transiciones completas entre secuencias melódicas, rítmicas, armónicas o tímbricas, así como en la implementación informática realizada de dichas técnicas.

1.4.2 Revisión bibliográfica sistematizada

La revisión bibliográfica que se ha realizado en la presente tesis, materializada en forma de estado de la cuestión en el capítulo 2, es una revisión bibliográfica de tipo sistematizada. Las características de dicha revisión consisten en que se ha utilizado una metodología estandarizada (Codina, 2017) para evitar posibles sesgos en la selección de artículos o inclusión de aquellos que no procedan de fuentes aceptadas por la comunidad científica. De esta manera nuestra revisión bibliográfica intenta evitar la arbitrariedad, sesgo o subjetividad. Además, se han consultado las principales fuentes y sistemas de recuperación de información asociadas al tema de investigación, citando las fuentes utilizadas, las palabras clave y los criterios de selección de artículos. Por último, hemos pretendido que sea reproducible, ya que cualquier otro investigador podría llegar a los mismos resultados que nosotros, comprobando la validez de nuestro trabajo.
Para garantizar el correcto desarrollo de la revisión se ha utilizado el marco de trabajo SALSA (*Search, AppraisaL, Synthesis and Analysis)* descrito en *A typology of reviews: an analysis of 14 review types and associated methodologies* (Grant y Booth, 2009). Esta metodología considera indispensables la existencia de los cuatro elementos para una revisión bibliográfica sistematizada: búsqueda de los artículos, valoración de los mismos, método de síntesis y fase análisis de resultados o redacción del estado de la cuestión. La búsqueda se realizará con criterios de inclusión o exclusión bien definidos, expresados en nuestro caso mediante palabras clave, con la finalidad de obtener aproximadamente un banco de un centenar de artículos relevantes para la redacción de nuestra revisión bibliográfica.

Las palabras clave utilizadas han sido: *music, melody, rhythm, chord, harmonic, composition, computer, computer music, computer-aided composition, algorithmic composition, computer-assisted composition, automatic, program, process, software, markov, grammar, stochastic, generation, cellular, genetic, network, clustering.* Los criterios de inclusión para la inclusión de los artículos en la colección final han sido el cumplimiento de alguna de estas palabras clave y el número de citas ese artículo. Los artículos con cero número de citas han sido desechados. Las fuentes donde se han buscado los artículos han sido *JSTOR*, *Google Scholar* y las siguientes revistas:

- Sound and Music Computing conference (SMC).
  `http://www.smcnetwork.org/`
- Mathematics and Computation in Music (MCM).
  `http://www.smcm-net.info/`
- The International Computer Music Association (ICMC).
  `http://www.computermusic.org/`
- Computer Music Journal.
  `http://www.computermusicjournal.org/`
- The Computer Journal.
  `https://academic.oup.com/comjnl/`
- Journal of the American Musicological Society.
  `http://www.ams-net.org/pubs/jams.php/`
- Computers in the Humanities.
  `https://www.jstor.org/journal/comphuma/`
- Journal of New Music Research.
  `https://www.tandfonline.com/loi/nnmr20/`
- The Journal of the Acoustical Society of America.
  `https://asa.scitation.org/journal/jas/`
- The Journal of Music, Technology and Education.
  `https://www.ingentaconnect.com/content/intellect/jmte/`
- The Music Information Retrieval Evaluation eXchange .
  `https://www.music-ir.org/mirex/wiki/MIREX_HOME/`

1.4.3 Acotación del área de estudio

En lo referente a las transiciones entre distintos materiales musicales, el área de estudio ha sido acotada, deliberadamente y por motivos de simplicidad, a la música representada de manera simbólica mediante notación convencional occidental, circunscrita al sistema temperado igual de doce tonos; excluyendo por tanto otros tipos de música como por ejemplo la de carácter folclórico, la música microtonal, la música en otros sistemas de afinación, la música electroacústica, o las nuevas grafías propias de determinadas estéticas de vanguardia. Dichas realidades musicales, más complejas, podrán ser estudiadas en futuras líneas de investigación. En lo referente a los sistemas de afinación, hemos continuado la propuesta de definición *fuzzy* realizada por Liern (2005), mostrando su adaptación únicamente a los principales sistemas de afinación históricos, ya que resultaría inabarcable su ejemplificación en todos los sistemas de afinación inventados de los que tenemos constancia y constituiría un trabajo de investigación completo en sí mismo.

1.4.4 Fases de la investigación

El orden lógico en la planificación de las fases de este trabajo de investigación ha estado delimitado por el cumplimiento de los diferentes objetivos específicos y generales.

1. Realización de una revisión bibliográfica organizada en forma de estado de la cuestión, en la que se han resumido las principales líneas de investigación y aportaciones realizadas en el ámbito de la composición musical asistida por ordenador desde sus inicios hasta el momento actual.

2. Se ha decidido utilizar una representación de tipo vectorial para la melodía, describiéndola como una sucesión ordenada de n notas, donde cada nota es un vector en un espacio métrico de dimensión d y cada coordenada del vector representa una característica musical. Se ha decidido utilizar el MIDI *pitch* para caracterizar la altura de la nota, el MIDI *velocity* para la intensidad y un coeficiente δ para caracterizar la duración de la nota. Los silencios, *staccati* y acentos se caracterizarán como dimensiones extra con valor 1 ó 0.

3. Se ha propuesto una medida de la disimilitud entre dos melodías distintas basada en la agregación de la distancia existente entre cada una de las notas de ambas melodías. Sin embargo, cuando se comparan melodías de diferente número de notas es necesario establecer un criterio para seleccionar qué notas de la melodía más larga serán comparadas con qué notas de la melodía más corta. Se propone solucionar dicho problema mediante la utilización de los métodos de *clustering*.

4. Revisión de las bases teóricas y el funcionamiento algorítmico de los diferentes tipos de métodos de *clustering* utilizados en el ámbito de la minería de datos. Se ha profundizado en la variante *fuzzy* del algoritmo *k-means* estándar, denominado c-means y se ha estudiado su utilización para la comparación de melodías.

5. Se han investigado distintos tipos de medidas de disimilitud existentes basados en los procesos de *clustering*, así como las funciones distancia más comúnmente utilizadas.

6. Se ha propuesto el algoritmo FCM, consistente en una modificación del algoritmo *fuzzy c-means* en el que se han incorporado una función de vecindad gaussiana.

7. Se implementa computacionalmente el nuevo algoritmo FCM. Se comprueba que nuestro método es capaz de realizar el proceso de agrupamiento respetando el orden existente en las secuencias introducidas como datos a *particionar* y centroides iniciales.

8. Se propone una nueva medida de la disimilitud melódica basada en nuestro algoritmo FCM, capaz de tener en cuenta el orden de las secuencias de notas. Se implementa computacionalmente y se comprueban sus resultados mediante numerosos ejemplos.

9. Se trabaja en el software MERCURY, desarrollando los módulos de entrada y salida de datos mediante el formato *MusicXML* e implementando la libraría gráfica de notación musical.

10. Se desarrolla el proceso de *cuantificación*, a través del cual podemos aproximar a notación musical convencional los estadios intermedios arrojados por el algoritmo FCM. Se implementa en MERCURY.

11. Se generaliza el algoritmo FCM para el caso de la armonía, el ritmo y el timbre (espectros energéticos), adaptando para estos cuatro casos el concepto de disimilitud previamente definido para el caso de la melodía. Se implementa en MERCURY. El programa es capaz de calcular la disimilitud y realizar transiciones incompletas, basadas en el FCM para los cuatro casos básicos (melodía, ritmo, armonía y timbre).

12. Se propone el algoritmo FCT. Dicho algoritmo aplica reiteradamente el algoritmo FCM y cada vez que finaliza añade un punto nuevo a la secuencia de centroides. Obtenemos que el algoritmo FCT es capaz de realizar una

transformación completa, proporcionando numerosos estadios intermedios, entre dos secuencias numéricas cualesquiera.

13. Se implementa el algoritmo FCT en MERCURY, obteniendo las transiciones completas para el caso de la melodía, el ritmo, la armonía y el timbre. Dicha implementación nos proporciona innumerable material musical a modo de transiciones o variaciones entre material musical inicial establecido por el usuario.

14. Ampliamos nuestro estudio al caso de los sistemas de afinación. Revisamos la literatura científica existente al respecto. Proponemos un nuevo método, hibridando las definiciones de Liern (2005) y nuestro algoritmo FCM, para calcular la compatibilidad de un conjunto de notas con un determinado sistema de afinación. Se implementa en MERCURY, utilizando para ello el algoritmo *Cooley-Turkey* para la Transformada Rápida de Fourier, y el método del *Harmonic Product Spectrum* para la detección de frecuencia fundamental. Se realizan experimentos para comprobar la compatibilidad entre una serie de notas medidas experimentalmente y un sistema de afinación.

15. Se define la disimilitud entre sistemas de afinación mediante el método FCM, así como las transiciones entre sistemas de afinación mediante el método FCT. Se implementa en MERCURY. Se realizan experimentos en los que se obtienen transiciones completas entre distintos sistemas de afinación, así como una medida de disimilitud entre ellos.

16. Se utiliza MERCURY para la realización de la composición *Transiciones difusas*, ejemplificando el uso y la utilidad que nuestra novedosa propuesta al ámbito de la composición musical asistida por ordenador puede tener en el terreno de la creación artística.

17. Difusión de resultados en congresos y publicaciones especializadas.

1.5 Estructura de la tesis

La presente memoria se estructura en distintos capítulos que vienen a reflejar las diferentes etapas por las que hemos pasado en nuestro proceso investigador. Tras una introducción, el capítulo 2 presenta el estado de la cuestión, que ilustra las aportaciones más relevantes encontradas en el ámbito de la composición musical asistida por ordenador. El planteamiento seleccionado para ello ha sido de tipo cronológico, resumiendo por orden de publicación un gran número de artículos, publicaciones de revistas, comunicaciones de congresos, libros y tesis doctorales relacionadas con alguna de las líneas de investigación tradicionales, comenzando por los pioneros en la creación musical computarizada de los años cincuenta y llegando hasta las más recientes propuestas de nuestros días.

El capítulo 3 recoge el marco teórico fundamental de nuestra presente investigación. En él hemos plasmado los principios matemáticos y algorítmicos básicos sobre los que se ha desarrollado esta tesis, resumiendo los planteamientos formales de los algoritmos de *hard* y *soft clustering* utilizados frecuentemente en el ámbito de la minería de datos, junto con las funciones distancia más comúnmente utilizadas y explicando las aportaciones que hemos realizado. Nuestra contribución principal, pieza clave en el engranaje del presente trabajo, consiste en la definición de las funciones de vecindad y su incorporación en el algoritmo *Fuzzy c-Means* (FCM) propuesto por Bezdek (1981). Dicha modificación da lugar al algoritmo *Fuzzy Ordered c-Means* (FOCM), con el que podremos definir una medida de similitud para cualquier pareja de secuencias de elementos, y posteriormente al algoritmo *Fuzzy Complete Transitions* (FCT), gracias al cual podremos generar transiciones desde cualquier secuencia de datos inicial a otra final.

En el capítulo 4 acotaremos la aplicación de los algoritmos y cálculos de disimilitud propuestos en el capítulo anterior a cada una de las tres características básicas de la música: la melodía, el ritmo y la armonía. Estableceremos una manera de parametrizar las distintas características musicales relativas a cada uno de estos tres casos (alturas, intensidad, duración, silencio, acentos, articulaciones, etc.) y realizaremos una implementación computacional a través de nuestro software Mercury que nos permitirá la ejecución de numerosos ejemplos tanto de cálculo de disimilitudes como de realización de distintas transiciones musicales, poniendo de manifiesto la potencia del método presentado para la generación de nuevo material de tipo melódico, rítmico o armónico. Por último, estudiaremos en los mismos términos el concepto de timbre musical, parametrizando matemáticamente el espectro energético de frecuencias de varios sonidos, sometiéndolos tanto a cálculos de disimilitud como a transiciones espectrales.

En el capítulo 5 estudiaremos los sistemas de afinación, generalizando su definición a través de un conjunto de intervalos y funciones generadoras. Ilustraremos dicha definición con los principales sistemas de afinación de la tradición musical occidental. Comprobaremos que el software MERCURY es capaz de calcular de manera sencilla la compatibilidad entre sistemas introducida en Liern (2005). Además, mostraremos que la compatibilidad se puede generalizar mediante la obtención de transiciones entre sistemas de afinación. De hecho, se realizarán comparaciones entre distintos sistemas de afinación gracias a la adaptación de las definiciones de disimilitud y los métodos de para las transiciones propuestas en el capítulo 3 a las características de los éstos.

En el capítulo 6 expondremos las conclusiones a las que nos ha conducido nuestra labor investigadora, resumiendo las dificultades encontradas y evaluando el grado de consecución de los objetivos alcanzado. Además presentaremos futuras líneas de investigación y posibles mejoras tanto de los métodos algorítmicos presentados como de su implementación informática, además de discutir sobre la aplicabilidad en la composición musical de los mismos.

Por último, en el Apéndice I se presenta parte del código relativo al módulo de cálculo de MERCURY, escrito en lenguaje *C#*. En el Apéndice II se presentará la composición musical *Transiciones difusas* creada utilizando los métodos propuestos en el presente trabajo de investigación, junto con las tablas de material melódico, rítmico y armónico generado con MERCURY y utilizado para la elaboración de la pieza, poniendo de manifiesto algunas de las múltiples posibilidades creativas que ofrece al compositor el método para la composición asistida por ordenador aquí propuesto.

Capítulo 2

Estado de la cuestión

«Al oír por primera vez la idea de música por computador, cualquier persona podría preguntarse: "¿Por qué programar una computadora digital para generar música?" La respuesta a esta cuestión no es sencilla [...] y nos lleva además a la pregunta de hasta dónde es posible expresar los principios de la estética y la forma musicales de manera conveniente para su procesado computacional.»

«Upon first hearing of the idea of computer music, a person might ask: "Why program a digital computer to generate music?" The answer to this question is not simple [...] it also raises the question of how far it is possible to express musical and aesthetic principles in forms suitable for computer processing.»

(Hiller e Isaacson, 1959, pág. 1)

2.1 Introducción

En el presente capítulo se presenta la redacción del estado de la cuestión relativo a nuestras investigación. Continuando con la metodología expuesta en el capítulo introductorio, se ha realizado una revisión bibliográfica sistematizada, resultando en una colección final de aproximadamente cien publicaciones relevantes que han sido resumidos tratando de mostrar las características más relevantes de cada artículo. Conviene recordar algunas de las referencias bibliográficas destacables relativas a la composición musical asistida por ordenador que se han sido utilizadas:

- Manning (2013). *Electronic and computer music.*
- Holmes (2012). *Electronic and experimental music: technology, music, and culture.*
- Rumsey y McCormick (2012). *Sonido y grabación. Introducción a las técnicas sonoras.*
- Nierhaus (2009). *Algorithmic composition: paradigms of automated music generation.*
- Supper (2004). *Música electrónica y música con ordenador: historia, estética, métodos, sistemas.*
- Holmes (2002). *Electronic and experimental music. Pioneers in technology and composition.*
- Núñez (1993). *Informática y electrónica musical.*

A continuación expondremos de forma cronológica los resultados de nuestro trabajo de revisión bibliográfica sobre la composición musical asistida por ordenador, comenzando por los años cincuenta y llegando hasta la actualidad, mostrando las aportaciones más relevantes que han sido presentadas a la comunidad científica en cada década.

2.2 Investigaciones pioneras

El trabajo de Hiller e Isaacson culmina en la creación de la *Suite Illiac* para cuarteto de cuerdas, tradicionalmente considerada como la primera obra que utiliza algoritmos computacionales en su composición. Sin embargo, el propio Hiller reconoce que existen algunos experimentos de composición asistida por ordenador previos a la publicación de su obra. La canción *Push Button Bertha*, resultado de la programación de Douglas Bolitho y Martin L. Klein, es el más documentado de estos experimentos (Klein, 1957; Pierce, 1961; Hiller, 1968; Ames, 1987). El 15 de Julio de 1956 esta canción fue retransmitida en el programa de televisión *Adventure Tomorrow*, menos de un mes antes de la primera presentación pública de una aún inconclusa Suite Illiac [1], que sería finalizada por Hiller en noviembre de 1956 (Ariza, 2011).

[1] El 9 de agosto de 1956 Hiller realizó una interpretación en público de su obra en la Universidad de Illinois (Hiller e Isaacson, 1959).

Con anterioridad a los trabajos de Hiller e Isaacson y de Klein y Bolitho, en 1955 los autores Caplin y Prinz realizan los primeros experimentos de los que existe constancia relativos a la composición musical asistida por ordenador (Martínez, 2019). Prinz tenía conocimientos de programación en la computadora *Mark I* (Prinz, 1952), desarrollada en 1949 por la compañía inglesa *Ferranti Ltd* [2] e implementa en Mark I, gracias a los conocimientos musicales de Caplin, el *Musikalisches Würfespiel* de W. A. Mozart. El sistema compositivo de Caplin y Prinz ignora las armonías propuestas por Mozart para la mano izquierda, utilizando por tanto únicamente las líneas melódicas de la mano derecha. El resultado podía escucharse gracias al rudimentario sistema de síntesis sonora del ordenador. También podía imprimirse o visualizarse como una cadena de símbolos alfanuméricos (Ariza, 2011).

Caplin y Prinz continúan al menos hasta 1960 realizando experimentos con el ordenador Mark I y a partir de 1959 utilizando el computador *Ferranti Mercury*, más potente que su predecesor, culminando en la composición *The Foggy, Foggy Dew*. Para la determinación aleatoria de las notas de esta obra se utilizaron tablas de probabilidad de transición, de forma similar a una cadena de *Markov* de primer orden (Hiller, 1968; Martínez, 2019).

En 1964 la investigadora Harriet Padberg publica su tesis *Computer-composed canon and free-fugue* (1964) en la que resume su aportación a la composición asistida por ordenador. Padberg programa dos sistemas, el *Program 1* y el *Program 2*, ambos en *Fortran*. El primero, escrito para la computadora *IBM 1620*, permite la creación de cánones de hasta cuatro voces basados en una serie de notas. El segundo sistema, escrito para la computadora *IBM 7072*, amplía las limitaciones del anterior sistema e incorpora la posibilidad de crear una fuga en estilo libre.

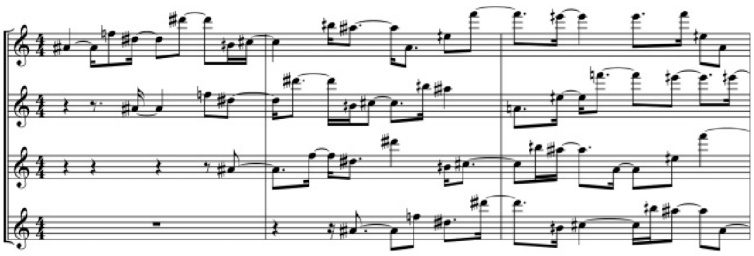


Figura 2.1: Fragmento de *Composition 4* de H. Padberg, transcrito en formato MusicXML por Christopher Ariza (Ariza, 2011, pág. 52).

[2] Una de las más antiguos sonidos generados por un ordenador y la grabación más antigua existente de música producida por ordenador es la interpretación en 1951 de las obras *God Save the Queen*, *Baa Black Sheep* y *In the Mood* mediante la computadora *Ferranti Mark I* de la Universidad de Manchester (Ariza, 2011, pág. 43).

Las series de notas están limitadas a escalas con un número de notas comprendido entre 5 y 24 notas por octava. El sistema permite las operaciones de retrogradación e inversión de las series. El resultado compositivo de los programas se obtiene en unas tablas que hacen la función de partitura. Padberg incluye cinco composiciones de ejemplo en su tesis, que han sido transcritas en formato *MusicXML* por Ariza (Ariza, 2011).

Brooks Jr y col. (1957) proponen un método de composición de melodías basado en las cadenas de Markov de orden m. Se analizan 37 melodías para obtener las probabilidades de los distintos elementos, variando el orden de la cadena desde los bigramas ($m = 2$) y trigramas ($m = 3$) hasta las probabilidades de octavo orden, utilizando para estos conteos un computador. La altura de las notas se codifica numéricamente, y los valores de duración se representan dividiendo la melodía en células mínimas representadas por el valor de la semifusa. De estas probabilidades se derivan unas tablas de que se utilizarán para la composición de nuevas melodías mediante un proceso aleatorio. Dicha síntesis melódica se realiza incorporando diferentes restricciones métricas para configurar ritmos melódicos deseados. En total se sintetizan 600 melodías y se puede observar un resultado muy interesante: las melodías generadas mediante órdenes muy bajos (la síntesis con $m = 1$ obtiene la distribución del monograma, es decir, aleatoria) presentan un alto grado de aleatoriedad, sin embargo aquellas generadas con probabilidades de órdenes muy altos producen melodías que son el resultado de unir fragmentos de las melodías originales (Martínez, 2019).

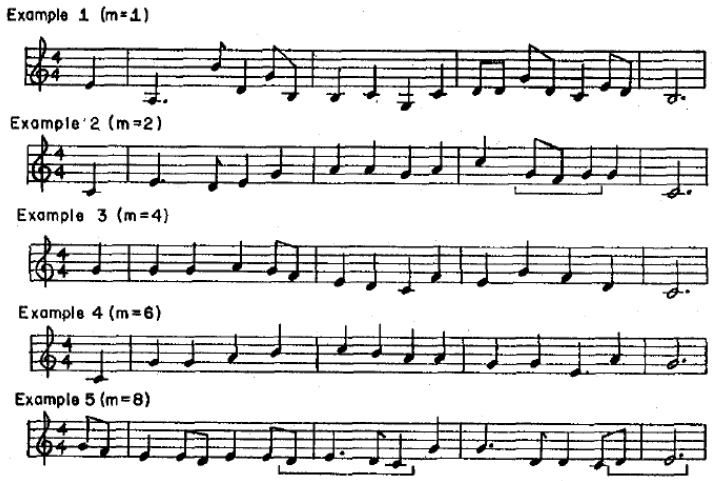


Figura 2.2: Algunas melodías obtenidas para distintos órdenes de Markov m (Brooks Jr y col., 1957, pág. 181).

Los resultados obtenidos utilizando ordenes intermedios son los que presentan mayor originalidad; sin embargo producen en ocasiones melodías *degeneradas* (e.g. largas secuencias melódicas ascendentes o descendentes) con lo que es necesario un filtrado humano para discriminar las piezas que tienen interés musical de aquellas que no lo tienen. En este punto, el método utilizado basado en cadenas de Markov no es todavía lo suficientemente complejo para cumplir nuestros requerimientos estéticos (Martínez, 2019).

Entre los primeros experimentos en el ámbito de la composición asistida por ordenador realizados a finales de la década de los cincuenta destaca el trabajo del compositor Lejaren Hiller desarrollado en la Universidad de Illinois con su colaborador Leonard Isaacson, matemático de la *Standard Oil Company*. Trabajando en la universidad, Hiller tuvo acceso al ordenador *ILLIAC I (Illinois Automatic Computer*, un computador construido en el año 1952, consistente en una máquina de cinco toneladas de peso y cerca de 2800 válvulas de vacío que se programaba mediante cinta de papel perforada (Holmes, 2012; Martínez, 2019). Para solucionar el gran número de obstáculos que el carácter exploratorio de su investigación les va a acarrear, Hiller e Isaacson dividen su trabajo en distintos estadios de desarrollo (Holmes, 2012):

1. Elegir un estilo de composición polifónico simple, que sea adaptable a la programación informática. Se selecciona por tanto un estilo estricto de contrapunto.

2. Determinar cómo se puede codificar la información musical y demostrar que la programación informática puede soportar las técnicas musicales estándar, produciendo un resultado legible e interpretable por cualquier músico.

3. Demostrar que un ordenador puede producir estructuras musicales nuevas, codificando elementos como dinámicas y ritmo.

4. Mostrar cómo los ordenadores pueden ser útiles para los compositores de música contemporánea en su intento de desarrollar nuevos tipos de música con elementos musicales no convencionales (Hiller e Isaacson, 1959).

El resultado de sus investigaciones se ve plasmado en lo que se considera la primera gran composición musical creada con la ayuda de un ordenador: la *Suite Illiac*, para cuarteto de cuerdas, finalizada en noviembre de 1956. Los cuatro movimientos se construyen en paralelo con los cuatro puntos de la investigación descritos anteriormente: el primer movimiento se diseña para obtener sencillas melodías diatónicas sobre las que progresivamente se van añadiendo voces hasta llegar a sonar las cuatro simultáneamente; en el segundo movimiento se exploran con ma-

yor profundidad las reglas del contrapunto mediante términos matemáticos para demostrar que la música tradicional puede ser codificada en términos computacionales, alcanzando al final de este movimiento el nivel más alto de complejidad con la realización de 1900 operaciones aritméticas; el tercer movimiento lleva a los investigadores a pretender objetivos musicales, intentando demostrar que mediante la programación informática se pueden codificar tanto el ritmo como la dinámica musical. Se utiliza, además, una escala cromática cuyos disonantes resultados obligan a los investigadores a incorporar una serie de reglas tradicionales para mejorar el proceso de selección, finalizando en una composición muy similar a la de algunas obras de compositores contemporáneos; el cuarto y último experimento las reglas compositivas utilizadas no derivan de la tradición musical sino de disciplinas extramusicales con la intención de demostrar que una computadora convenientemente programada puede generar música interesante incluso cuando esté utilizando reglas que nada tienen que ver con el arte del sonido. Los investigadores exploran la utilización de métodos estocásticos probabilistas basados en cadenas de Markov (Hiller e Isaacson, 1959; Hiller, 1959; Martínez, 2019).

La obra fue interpretada en público en la Universidad de Illinois el 9 de agosto de 1959 en su versión incompleta de tres movimientos y presentada a la comunidad científica el 28 de agosto de 1956 en la *11th National Meeting of the Association for Computing Machinery* para ser finaliza en noviembre del mismo año y publicada en la editorial *Board of the New Music Edition* de nueva York.

Hiller continuará sus investigaciones sobre música y computación en la Universidad de Illinois, desarrollando en 1962 junto a Robert A. Baker en *Fortran* y mediante un computador *IBM7094* el programa *MUSICOMP (MUsic SImulator-Interpreter for COMpositional Procedures)*, capaz de automatizar subrutinas compositivas y construir de piezas musicales a partir de la definición previa de reglas. La obra *Computer Cantata* publicada en 1963 presenta los resultados de las pruebas de eficiencia y uso realizadas con MUSICOMP (Hiller y Baker, 1964).

En 1957 Max Mathews desarrolla en los Laboratorios Bell de Nueva Jersey el lenguaje *MUSIC I*. Se trata de un lenguaje de programación orientado a la composición musical asistida por ordenador desarrollado en lenguaje ensamblador para ejecutarse en un computador IBM704. Es monofónico, capaz por tanto de generar una sola onda triangular. Sólo permite establecer los parámetros de amplitud, frecuencia y duración de cada sonido. La salida se almacena en una cinta magnética que posteriormente se somete al proceso de conversión digital / analógica gracias a que los Laboratorios Bell, en esos años, eran los únicos en los Estados Unidos que tenían un conversor DA (conversor de tecnología de válvulas de 12 bits). La primera composición musical realizada por ordenador es *The Silver Scale*, fechada

el 17 de mayo de 1957 por su autor Newman Guttman. Consiste en una sencilla melodía de 17 segundos a través de la computadora y Music I.

Mientras tanto Mathews continua trabajando en el *MUSIC II*, que será terminado en 1958. No ofrece muchos cambios con respecto al Music I. Se trata de un software sea más versátil y funcional que su predecesor, que aumenta el número de voces hasta una polifonía de 4 voces e incrementa el número de algoritmos disponibles para la síntesis sonora. Introduce la síntesis por tabla de ondas, haciendo posible sintetizar sonidos mucho más complejos.

La introducción del IBM 7090, con mejor rendimiento gracias a los transistores, lleva a Mathews a desarrollar el *MUSIC III*. La tercera versión del MUSIC introduce tantas innovaciones que podemos hablar de un software prácticamente nuevo, más que una versión superior. Se trata de una herramienta flexible y bien configurada, adecuada para el uso profesional por parte de los compositores. Introduce el concepto de *Unit Generators*[3] (UG) erigiéndose como todo un punto de inflexión para el desarrollo posterior de la composición asistida por ordenador. Introduce el concepto de SCORE[4] y ORCHESTRAS[5].

MUSIC IV es desarrollado en 1963 resultado de la colaboración entre Max Mathews y Joan Miller. Introduce mejoras de manejo en términos de programación, más que en términos musicales. Será el último de la familia, *MUSIC V* completado en 1968 el que finalmente transcienda los límites de los Laboratorios Bell y se convierta en un paradigma de referencia para futuros softwares de composición musical asistida por ordenador como Music360 (desarrollado por Barry Vercoe en el MIT), Music 10 (desarrollado por John Chowning y James Moorer en la Stanford University) o lenguajes más modernos como CSound. Se desarrolla para una computadora IBM360 y bajo el lenguaje de programación *Fortran* y mejora muchas imperfecciones de las versiones anteriores facilitando al mismo tiempo la introducción de datos. Este software se declara de libre difusión no comercializable y supone el fin de la investigación de Max Mathews para el software de síntesis sonora (Holmes, 2012).

[3]Las Unit Generators, también llamados Opcode, son macros que realizan funciones específicas y que pueden ser invocadas por comandos simples de manera muy rápida. Permiten obtener datos de salida y también aceptar entradas y al vincular varias UG, es posible construir fácilmente herramientas más elaboradas, capaces de realizar funciones complejas como por ejemplo la construcción de instrumentos, que se realiza mediante la conexión de varias UG. Lenguajes de programación contemporáneos como Max / Msp o CSound por ejemplo, utilizan todavía el concepto UG introducido por Mathews a principios de los años sesenta.

[4]SCORE hace referencia al conjunto de parámetros definibles por el usuario para estructurar y organizar el contenido de la música, llamando a determinados instrumentos de la orquesta.

[5]ORQUESTRA hace referencia al conjunto de instrumentos que se pueden utilizar en el SCORE mediante parámetros definibles por el usuario.

Ya a finales de los años sesenta, Mathews desarrolla *GRAPHIC 1*, un sistema informático interactivo que puede traducir imágenes dibujadas con un lápiz óptico sobre un terminal de pantalla en un sonido sintetizado. El usuario puede insertar imágenes y gráficos directamente en la memoria de una computadora por el simple hecho de dibujar estos objetos. Permite modificar, borrar, duplicar así como almacenar los dibujos. Mathews introduce de esta manera el concepto de composición interactiva en tiempo real mediante una pantalla de computadora (Mathews y Rosler, 1968).

En 1970 Mathews presenta *GROOVE: Generated Real time Output Operations on Voltage controlled Equipment*, consistente en una interfaz de pantalla de visualización para simplificar la gestión de la síntesis digital en tiempo real (Mathews y Moore, 1970).

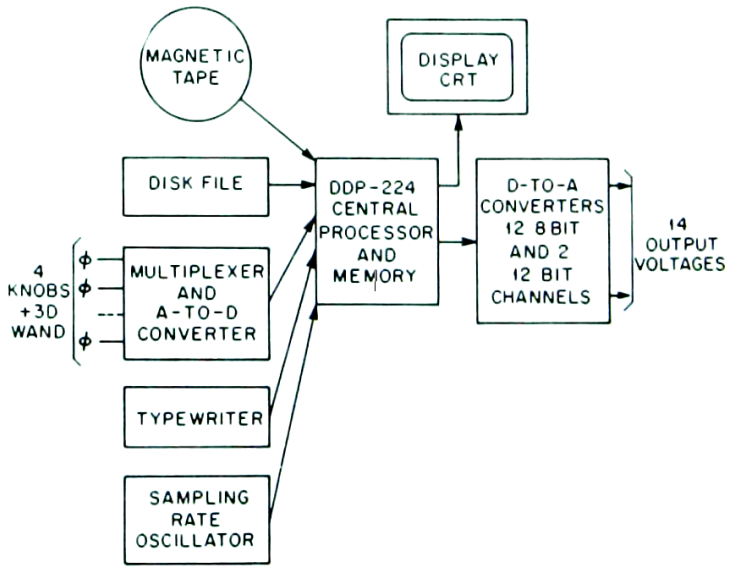


Figura 2.3: Diagrama de bloques del sistema GROOVE (Mathews y Moore, 1970, pág. 716).

En 1961 aparece el *Stochastic Music Program (SMP)*, inventado por Y. Xenakis y ampliamente utilizado por el compositor en sus obras ST/10-1, 080262; ST/48-1, 240162; *Atrées* y *Morsima-Amorsima* [6]. Los procesos compositivos implementados son esencialmente los mismos métodos estocásticos utilizados manualmente para la composición de *Achorrhipsis* y están descritos en los capítulos I y V del

[6] Para más información véase *Música de la Arquitectura* de Xenakis (2009).

libro *Formalized music: thought and mathematics in composition* (1971). En la siguiente figura podemos ver el flujo general del programa (Myhill, 1978):

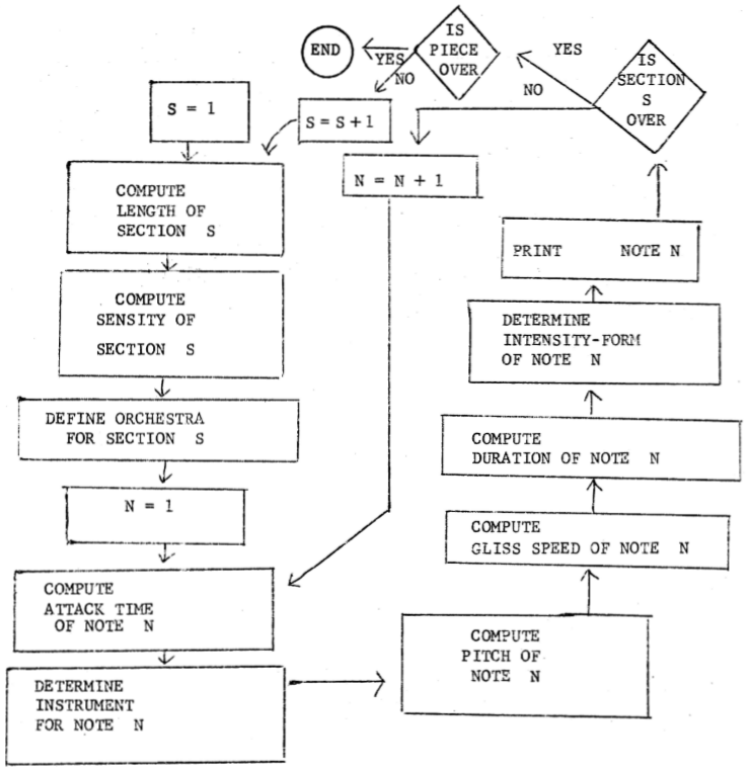


Figura 2.4: Flujo de funcionamiento general del programa SMP escrito por Xenakis (Myhill, 1978, pág. 272).

Dicho programa será mejorado por Myhill en 1978, tratando de hacerlo compatible con *MUSIC V* y buscando un camino hacia un lenguaje musical semi-estocástico (Myhill, 1979; Harley, 2004).

Desde 1954 hasta 1964 Michael Koenig trabaja como compositor y asistente en el estudio WDR. Comienza a estar interesado en la composición asistida por ordenador durante la década de los 60, y resultado de dicho interés es su programa *PROJECT ONE*. Dicho software se desarrolla en el Instituto de Sonología de la Universidad de Utrecht en 1964 y está orientado a la composición serial asistida por ordenador. Se basa en la contraposición de dos conceptos opuestos: regularidad frente a irregularidad. El programa es capaz de componer secciones, cada una de

las cuales ha sido generada mediante distintos tipos de procesos. El resultado es una tabla con determinados parámetros que una vez impresa ha de ser transcrita a partitura (Koenig, 1970a). A principios de los setenta, Koenig presenta la segunda versión de su software para la composición asistida por ordenador *PROJECT TWO* que permite al usuario una mayor influencia en el proceso compositivo y que deja por tanto en el olvido su anterior creación *PROJECT ONE*. En este último, la forma general se determina aleatoriamente siempre dentro de determinados límites. El usuario ha de introducir mucha información para definir los rangos de valores de la música así como las decisiones que el programa tomará para generar la forma final de la pieza (Koenig, 1970b). Durante toda esta década Koenig desarrollará *Sound Synthesis Program (SSP)*, creado poco después que el Instituto de Sonología adquiriese un nuevo ordenador. Representa el sonido como una serie de amplitudes en el tiempo y para utilizarlo, el usuario ha de establecer dos listas, una para la amplitud y otra para los valores temporales (Holmes, 2012).

2.3 Años 70

Truax (1973) incorpora en sus sistema *POD* estructuras similares a las propuestas por Koenig en las cuales se ofrecen posibles variaciones aleatorias limitadas dentro de un intervalo. El sistema POD está formado por una serie de programas para la síntesis sonora en tiempo real y la composición interactiva desarrollados por el autor en el Instituto de Sonología de Utrecht en colaboración con Simon Fraser. Los métodos de síntesis, monofónicos, incorporan el método de síntesis FM descubierto por John Chowning. El compositor puede trabajar con POD en diferentes niveles: eligiendo los objetos sonoros, especificando el algoritmo a utilizar en las reglas de selección (por ejemplo la distribución de Poisson) y sus variables implicadas (densidad, intervalos de frecuencia, amplitudes) (Truax, 1977; Truax, 1984).

James Anderson Moorer (1972) presenta un experimento de composición computacional, continuando con sus investigaciones anteriores sobre la producción de música tonal enmarcada en la tradición occidental, distanciándose de los métodos utilizados por Hiller e Isaacson. De esta forma Moorer critica la impredecibilidad de las melodías, así como el uso libre de la disonancia, más propio de una estética vanguardista que de la tradición clásica o romántica, que se puede encontrar en la partitura de la *Suite Illiac*. Al mismo tiempo destaca el interés de los resultados obtenidos por Brooks Jr y col. (1957). El experimento propuesto consiste en un método de composición secuencial; primero se selecciona la macroforma de la pieza, después se seleccionan los acordes y por último se conforma la melodía. La macroestructura se determina mediante la selección de dos valores, restringidos a potencias de dos, que determinan el número de grupos mayores y el número de grupos menores. A continuación se determina el número de tiempos que se incluyen en cada grupo menor y se acepta o se rechaza la forma obtenida en función de la restricción impuesta sobre la longitud total de la pieza. En la selección de acordes se utilizan probabilidades de primer orden que expresan la posibilidad que tiene cada acorde de aparecer. Sobre estas probabilidades se aplican reglas de selección, como por ejemplo que las secuencias de acordes generadas que nunca alcancen la tónica son automáticamente rechazadas, o secuencias armónicas de tipo ABBA en las que el acorde A no sea tónica son rechazadas también. De esta forma, como la selección del acorde es previa a la creación de la melodía resuelve el problema de qué acorde seleccionar para acompañar a una melodía dada, ya que en esta propuesta la melodía se deduce de la sucesión armónica. Con respecto a los ritmos de cada grupo menor independiente cabe mencionar que éstos son seleccionados mediante procesos aleatorios, incluyendo la única restricción de la definición de una longitud máxima y mínima para cada nota. En la figura 2.5 podemos ver algunos de los resultados de este experimento, que pueden ser im-

presos o reproducidos mediante un conversor digital-analógico, presentando líneas melódicas que no han sido editadas o filtradas por el criterio humano, compuestas bajo parámetros que maximizan la similitud de las melodías creadas con los postulados gramaticales de la tradición tonal occidental.

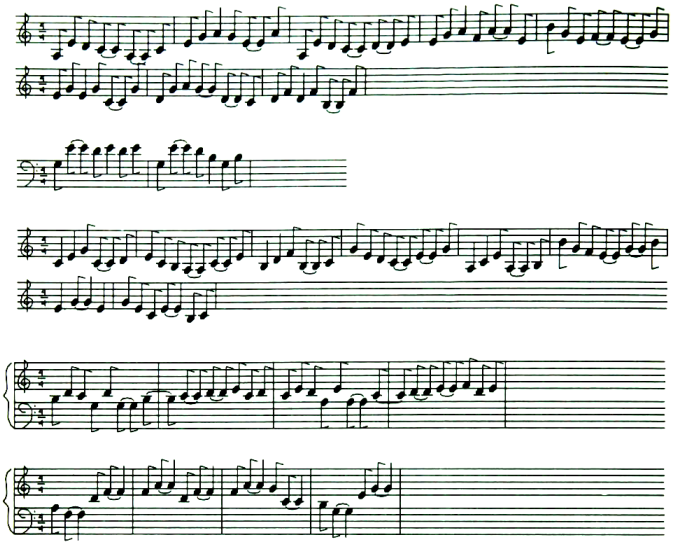


Figura 2.5: Ejemplo de cinco melodías generadas de forma consecutiva, sin su correspondiente acompañamiento armónico m (Moorer, 1972, pág. 112).

Lidov y Gabura (1973) proponen un algoritmo para la generación de melodías basado en un modelo de lenguaje formal, aplicando técnicas analíticas procedentes del estructuralismo, encajando los modelos lingüísticos como paradigma explicativo del fenómeno musical. Se deduce la estructura de la frase musical desde simples patrones rítmicos mediante una gramática generativa que coordina métrica y tonalidad, representados como criterios formales independientes. Los intervalos musicales son notados mediante tres símbolos (n, i, X); n representa la nota, i representa el intervalo, y X es una letra que representa la categoría de efecto de contorno (P significa frase, M miembro, G grupo, d derecha, g izquierda, m compás, etc.). Sobre este conjunto de símbolos se definen las reglas gramaticales que finalmente darán lugar a la composición. Los resultados de esta propuesta son melodías que no siguen patrones armónicos convencionales como consecuencia de las gramáticas definidas mediante funciones interválicas en vez de funciones armónicas.

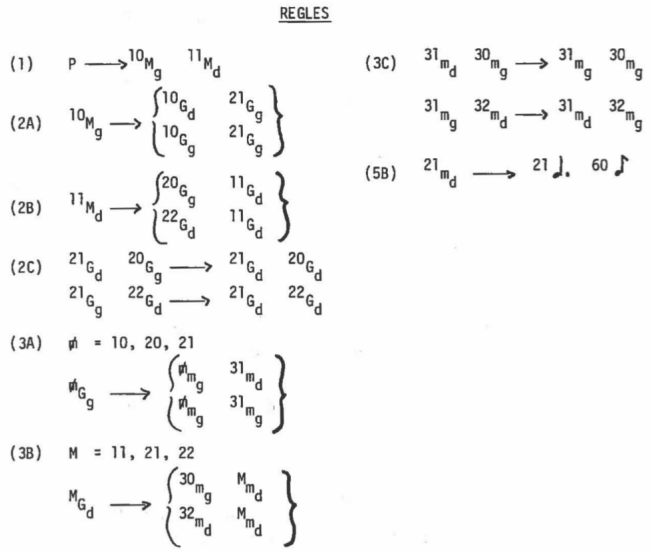


Figura 2.6: Extracto de la gramática rítmica (Lidov y Gabura, 1973, pág. 24).

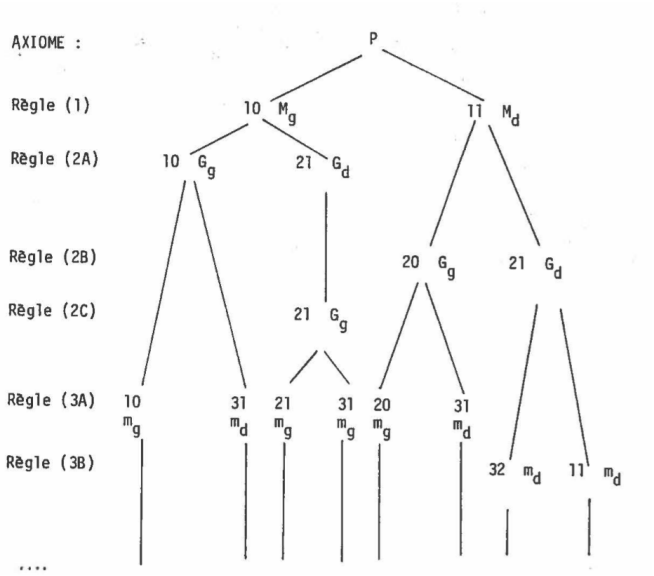


Figura 2.7: Árbol de derivación parcial de una base rítmica arbitraria (Lidov y Gabura, 1973, pág. 25).

En 1975 Tipei presenta *MP1*, un algoritmo capaz de generar música mediante un ordenador. El programa se concibe como un asistente capaz de suplantar algunas de las rutinas que existen en el trabajo cotidiano del compositor. MP1 es capaz de desarrollar tan sólo unas pocas tareas, las más sencillas, pero constituye un buen punto de partida para posteriores investigaciones sobre la composición asistida por ordenador. Ofrece tres ámbitos de aplicación de sus funcionalidades: música tradicional de carácter tonal y polifónica, música serial y música probabilística (Tipei, 1975; Martínez, 2019). El programa trabaja bajo los siguientes supuestos:

- Los sonidos pueden ser descritos en un espacio vectorial, de tal forma que cualquier composición puede ser expresada en forma tensorial.
- Cualquier sucesión continua de sonidos puede ser descrita por una cadena de Markov.
- Componer significa tomar decisiones. Estas decisiones son representadas mediante reglas de operación.

Además, se compone de una serie de subrutinas que pueden ser utilizadas entre sí de forma muy abstracta (PRECON, LIST, TRIM, FUTURE, PASPRE, PATTY, LOOKIN, WHICH, FUN, TYPLOO, AUXIL, entre otras). La información que arroja el ordenador como *output* son cadenas de símbolos que pueden fácilmente ser transcritos en notación musical.

Roads (1978) resume en su libro *Composing grammars* las principales aplicaciones de las gramáticas formales formuladas por Chomsky[7] a la composición de estructuras musicales. Expone los conceptos necesarios para entenderlas, su clasificación y las herramientas relativas a la notación que permiten describir sus propiedades estructurales: estructuras de árboles o metalenguajes simbólicos aplicados a gramáticas con finalidades compositivas. Además, presenta y compara algunos programas informáticos que han implementado gramáticas con finalidades compositivas, como el programa *SCORE* [8], el programa *LEXEM*, el programa *NGRAM* o el programa *DUR*, algunos de ellos aún en fase desarrollo en la fecha de publicación del documento (Roads, 1978).

En el mismo año Bales (1978) presentan *AMUS: Automatic Music System*, diseñado y desarrollado como un instrumento musical económico capaz de realizar variedad de propósitos, además de *Monodram*, la primera composición creada con este sistema. Según los autores, el programa proporciona al compositor una serie de herramientas compositivas que le liberan de una gran cantidad de trabajo

[7] Véase *Formal properties of grammars* (Chomsky, 1963).
[8] Véase *SCORE: A Musician's Approach to Computer Music* (Smith, 1972).

tedioso y le permiten dedicar más tiempo a los aspectos creativos del arte. El sistema incorpora un generador de sonido analógico controlado por técnicas digitales a través de un procesador Motorola M6800 que es capaz de controlar los parámetros de duración, frecuencia, amplitud y envolventes de cada tono producido. Además, existen distintas configuraciones preestablecidas de contenido armónico (forma de onda, tremolo y vibrato). El procesador admite partituras escritas en notación AMUS y las transcribe en información entendible por el módulo de generación de sonido, admitiendo hasta 8 voces independientes. La composición de *Monodram* se realiza mediante un programa escrito en BASIC se pueden generar las partituras en la notación adecuada para AMUS y permite seleccionar entre cuatro técnicas compositivas distintas:

- Composición estocástica dodecafónica totalmente libre.
- Composición estocástica microtonal, en la que la altura, la duración y el número de voces se eligen aleatoriamente.
- Utilización de música creada por el usuario en la cual se incorporan nuevos fragmentos generados por el programa.
- Utilización de citas o fragmentos de composiciones preexistentes relativas a variedad de estilos y períodos, seleccionados por el usuario.

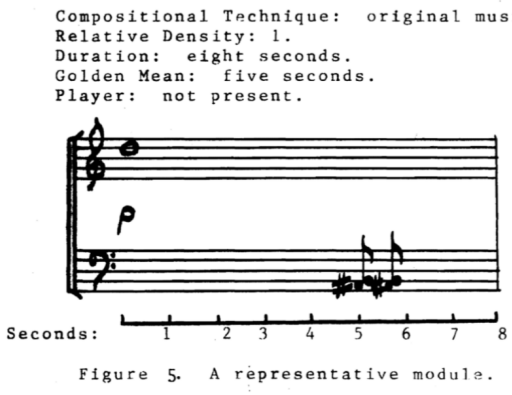


Figura 2.8: Ejemplo de utilización de un módulo de AMUS (Bales, 1978, pág. 467).

Justus Matthews (1978) presenta el programa *MUSIC 3150*, un software diseñado en *Fortran* para la composición musical para instrumentos acústicos convencionales. El programa pregunta al usuario por el número inicial para la generación

de número aleatorios (semilla) y a partir de una serie de diez contornos probabilísticos genera progresivamente las notas. El usuario puede establecer la duración total de la composición así como el número de voces. El programa ofrece la opción de orquestar la música generada de forma automática. También ha de especificar los valores máximos y mínimos de los parámetros musicales necesarios para la generación de notas aleatorias. También ha de establecer el parámetro de densidad sonora máxima y mínima que existirá entre las voces.

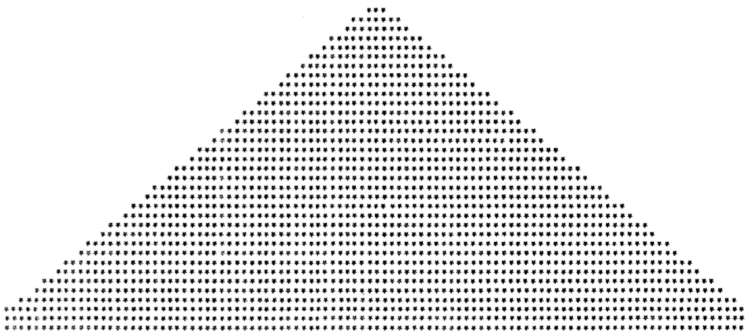


Figura 2.9: Ejemplo del contorno probabilístico número 4 del programa MUSIC 3150 (Matthews, 1978, pág. 261).

2.4 Años 80

Green (1980) presenta su software *PROD*, un programa para la composición musical asistida por ordenador basado en las gramáticas formales. Como parámetro de entrada PROD utiliza una gramática proporcionada por el usuario. El resultado de salida es una partitura musical generada automáticamente siguiendo la gramática establecida. Además se pueden obtener múltiples partituras a partir de una gramática ya que el programa posee un comportamiento no determinista, controlable por el usuario, en la que se utilizan procesos estocásticos para determinar el resultado. Sobre estos procesos estocásticos, el usuario tiene la capacidad de controlar cuándo y en qué grado ocurrirán los procesos aleatorios. Como viene siendo común en los formalismos gramaticales, la gramática se expresa mediante símbolos terminales y no terminales, así como a través de reglas de producción[9], y sólo se introducirán procesos aleatorios en aquellas producciones que posean más de una alternativa, especificando la probabilidad de ocurrencia de cada caso posible.

```
cx5 : cbar bar bar bar bar gbar ;

bar : cbar | gbar ;

cbar : major(c4) major(c4) major(c4) major(c4) ;

gbar : seven(g4) seven(g4) seven(g4) seven(g4) ;

major : note(trumpet,$1,8,100,,0)
        note(,third($1))
        note(,fifth($1),,,,8) ;

seven : note(trumpet,$1,8,100,,0)
        note(,third($1))
        note(,fifth($1))
        note(,,seventh($1),,,,8) ;
```

Figura 2.10: Ejemplo de gramática como parámetro de entrada (Green, 1980, pág. 107).

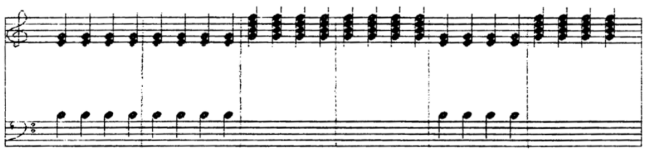


Figura 2.11: Partitura resultante (Green, 1980, pág. 108).

[9]Para más información sobre teoría y notación relativa a gramáticas formales véanse los capítulos 2, 4 y 12 del libro *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition* de Martin y Jurafsky (2009).

Una interesante aportación a la generación de melodías utilizando una solución híbrida entre cadenas de Markov y gramáticas formales reside en el programa *Melody-one*, presentado por Abel (1981) en su artículo *Computer Composition of Melodic Deep Structures*. El programa, escrito sobre *LISP*, es capaz de generar melodías agradables y coherentes con los postulados de la música tonal occidental a través del desarrollo de patrones melódicos procedentes de motivos musicales. La técnica básica que se utiliza para la generación de las notas es una cadena de Markov lineal, controlada por una gramática que genera un conjunto de posibles notas permitidas en función de la última nota generada. La gramática de transición utilizada puede verse en el siguiente diagrama

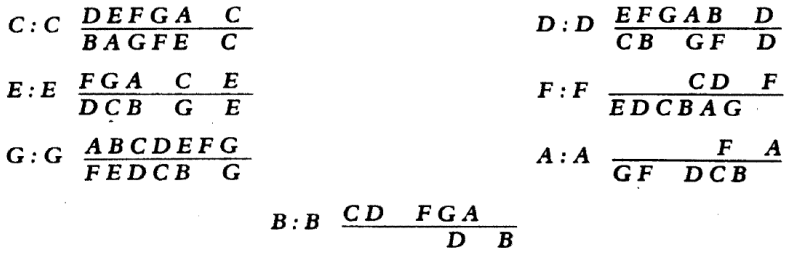


Figura 2.12: Gramática de transición utilizada en Melody-one (Abel, 1981, pág. 159).

El programa puede ser enriquecido mediante la utilización de gramáticas más sofisticadas que contemplen por ejemplo la utilización de notas alteradas, obteniendo material musical más interesante. Sin embargo Melody-one no contempla la generación del ritmo o del acompañamiento armónico y por tanto queda fuera del alcance del programa; éste retornara únicamente una sucesión de notas que deberán ser expresadas en ritmos y armonizadas de forma manual por el usuario (Martínez y Liern, 2019).

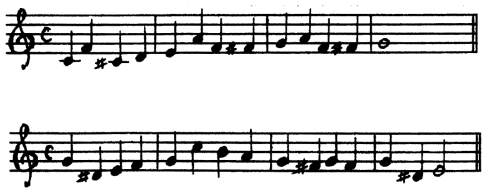


Figura 2.13: Melodías de cuatro compases generadas mediante una gramática de notas alteradas (Abel, 1981, pág. 168).

El investigador y compositor Ames (1982) presenta *Protocol*, una obra para piano de 17 minutos duración aproximada y compuesta mediante el uso de una computadora. Si bien el autor no sistematiza ningún método ni desarrolla ningún software específico para la composición musical, su acercamiento al computador radica en la utilización del mismo como un asistente para la realización automatizada de tareas algorítmicamente complejas en las que introduce algunas ideas novedosas que merece la pena destacar; Ames utiliza sofisticadas técnicas para la generación del material musical como por ejemplo la selección de notas aleatorias más elaborada que el simple método de prueba y error, la adquisición de estrategias jerárquicas para imponer restricciones estilísticas, la generación de campos armónicos disonantes de forma automatizada, o la utilización de curvas logarítmicas para determinar la distribución de las voces de un acorde en función de su tesitura.

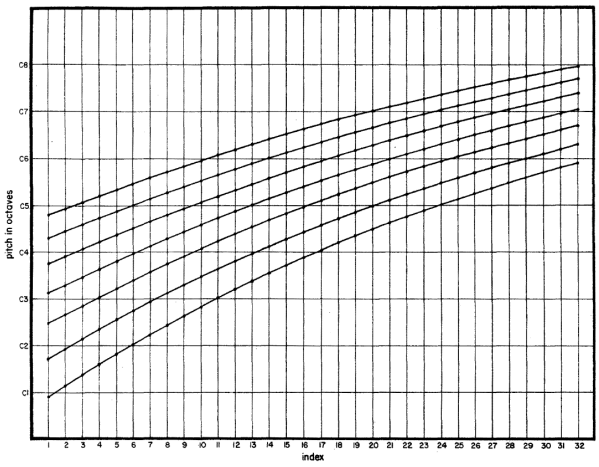


Figura 2.14: Curvas logarítmicas para la distribución de las voces de los acordes (Ames, 1982, pág. 138).

En el artículo *Compositional applications of stochastic processes* (Jones, 1981) podemos encontrar un resumen de la utilización en la composición musical de distintas técnicas estocásticas; además de las conocidas aportaciones a la música estocástica de Hiller (1959), Xenakis (1971), Koening (1970a), Truax (1973) y Roads (1978), existen otras interesantes propuestas. Lorrain (Lorrain, 1980) define distintas distribuciones de probabilidad para controlar procesos estocásticos entre las que se encuentra la función de decaimiento aleatorio, utilizada en 1979 para la composición de su movimiento orquestal *Firelake*.

$$N(t) = N_0 e^{-kt}$$

Por ejemplo se puede realizar la selección de alturas con la función anterior, estableciendo un centro tonal asimilable a N_0 y calculando las alturas sobre el total cromático sobre distancias t. También puede aplicarse al cálculo de la longitud de determinados eventos, pausas o patrones rítmicos. Jones (1980) presenta sus obras *Laitrapartial* y *Macricisum* en las que se utiliza las cadenas de Markov simples para la generación de timbres brillantes y cristalinos con alto contenido en parciales, así como cadenas de órdenes más complejos para la generación de ritmos.

Jones muestra además como se pueden utilizar estructuras motívicas, en lugar de notas individuales, como estados de un modelo de Markov para la generación de material musical (Nierhaus, 2009).

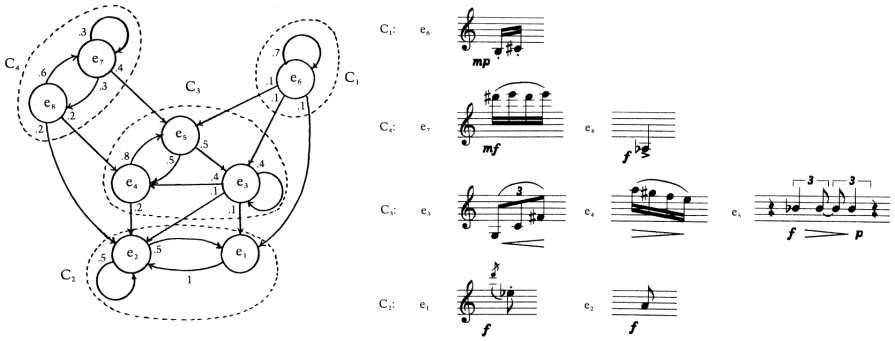


Figura 2.15: El modelo de Markov y motivos como elementos terminales (Jones, 1980, pág. 49).

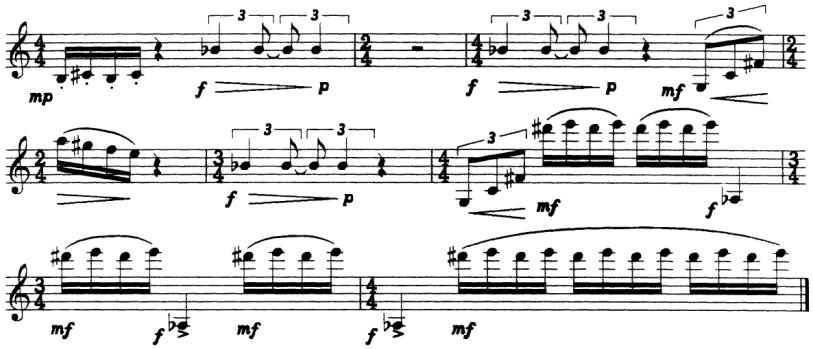


Figura 2.16: Ejemplo de producción musical generado por el sistema (Jones, 1980, pág. 50).

Michael Matthews (1981) presenta un software para realizar de forma automática operaciones relativas a la *Pitch Class Set Theory*[10] introducida por Milton Babbitt y formulada por Allan Forte. El programa posee determinadas subrutinas capaces de hacer lo siguiente: organizar cualquier conjunto de *pitch class* en su mejor orden normal; comparar la forma primaria de un conjunto dado con la tabla de formas primarias para determinar el nombre del conjunto y su vector interválico; calcular todas las transposiciones de las formas original e invertida; modificar los miembros individuales de cualquier vector interválico, transformando de esta manera el conjunto dado en otros conjuntos relacionados.

McNabb (1981) presenta *Dreamsong*, una composición electroacústica basada en la fusión de sonidos sintetizados digitalmente con otros grabados, de tal forma que se fundan en un continuo de material sonoro. El programa principal que se utiliza es *MUS10*, la versión de Leland Smith de *MUSCMP* y descendiente directo de *MUSIC IV* de Max Mathews.

Meinecke (1981), perteneciente a la Universidad de *North Texas State*, propone una serie de algoritmos compositivos escritos en Fortran e integrados en el sistema *M.U.S.I.C* (*McGrill University System for Interactive Computing*) que incluyen algoritmos de generación de alturas y de ritmos además de utilidades para el análisis y la correlación de melodías. El programa utiliza algoritmos estocásticos de segundo orden; es decir, la selección de cada intervalo se basa en función de los dos intervalos anteriores, basando esta selección en una cadena de Markov tridimensional. La selección de la dirección del intervalo se realiza en función de cuatro contornos melódicos predefinidos. El estilo de las melodías generadas depende de las melodías utilizadas como *corpus de entrenamiento* para la generación de la cadena de Markov (Martínez, 2019).

Hiller (1982) presenta la subrutina *STOCH*, escrita en Fortran IV, utilizada para la composición del primer movimiento de su obra *Algorithms III*. Esta subrutina expresa correctamente la producción de parámetros musicales de acuerdo con distribuciones de probabilidad dependientes del tiempo. STOCH selecciona siete parámetros para cada nota, de manera que cada parámetro es seleccionado de independientemente: la altura, la octava, el nivel dinámico, el timbre, el ataque, el ritmo y un multiplicador para el ritmo. El programa es capaz de manejar sucesos aleatorios desde orden cero a orden cuatro.

Entre los primeros experimentos sobre la generación de música autosemejante cabe destacar la aportación de Bolognesi (1983) en el artículo *Automatic composition: Experiments with self-similar music*. Tomando como punto de partida el

[10]Para más información sobre la *pitch class set theory* véase *The structure of atonal music* de Forte (1973).

sorprendente resultado del experimento de Voss y Clarke (1978) en el que encuentran que numerosos estilos musicales cumplen hasta cierto punto el denominado *comportamiento* $1/f$. Esta conclusión observacional sugiere que es más conveniente el uso de procesos estocásticos con un espectro de tipo $1/f$ (ruido rosa) que procesos con espectro constante (ruido blanco) para controlar fluctuaciones temporales de parámetros como la frecuencia o la intensidad en experimentos de composición automática. Las propiedades de la música generada con este algoritmo muestran la relación entre el espectro $1/f$ y las estructuras jerárquicas, en concreto las melodías estocásticas generadas con este algoritmo pueden ser consideradas como autosemejantes, siendo esta autosemejanza la forma más primitiva de estructura jerárquica.

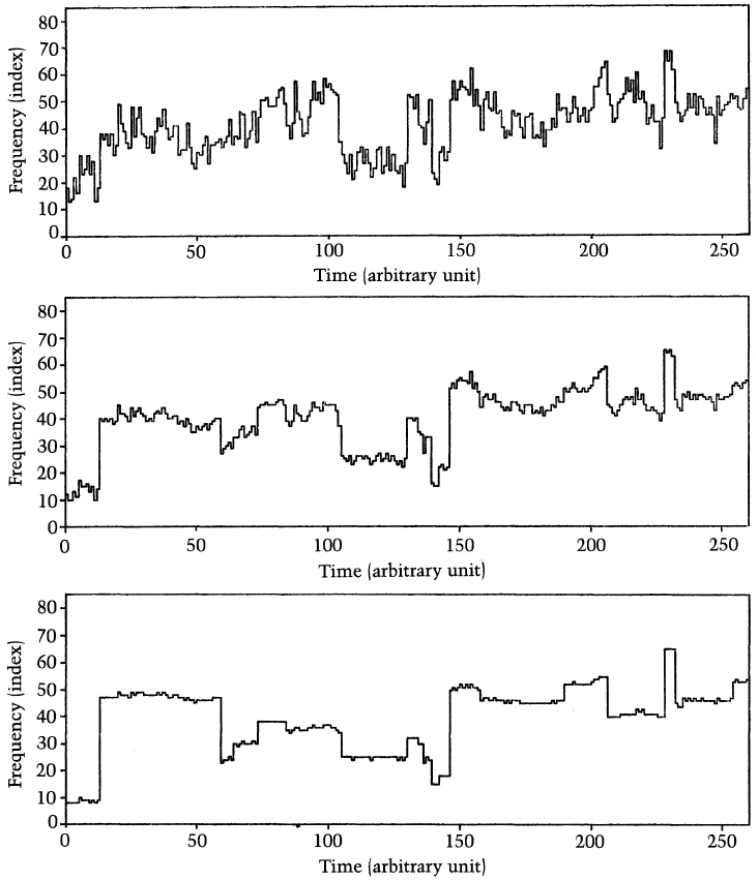


Figura 2.17: Tres melodías generadas mediante ruido $1/f$ (Bolognesi, 1983, pág. 30).

Steedman (1984) propone la utilización de una gramática generativa para la producción de secuencias de acordes en estilo Jazz. El carácter recursivo de las mismas hace, según el autor, que las gramáticas de este tipo sean un formalismo adecuado para describir las reglas que rigen estas secuencias. En el ejemplo proporcionado en este artículo Steedman es capaz de generar todos los miembros de una gran clase de armonías muy complejas denominadas *Blues de doce compases*[11] a partir de un número muy pequeño de reglas de producción. Estas reglas pueden ser fácilmente ampliadas, aumentando de esta forma su ámbito de aplicación. Por ejemplo se pueden configurar para que realicen variaciones de los acordes de la canción *I Got Rhythm* de Gershwin sustituyendo la primera de las reglas por un armazón de tónicas, dominantes y subdominantes (Steedman, 1984).

En 1985 Vercoe escribe el lenguaje de programación para sonido *CSound*, escrito en *C* y válido para cualquier sistema operativo. Finalizado en el *MIT*, CSound está basado en un lenguaje anterior del propio autor denominado *Music 11* que continua el modelo iniciado por Max Mathews en su serie de lenguajes *MUSIC-N* desarrollados en los Laboratorios Bell, asimilando el modelo de Unit Generators y la división entre ORCHESTRA y SCORE. El desarrollo de CSound continuará durante los noventa y la primera década del siglo XXI por el científico John Ffich en la Universidad de Bath. Actualmente posee alrededor de 1700 UG (*Unit Generators*) y es completamente modular y extensible por el usuario (Boulanger, 2000).

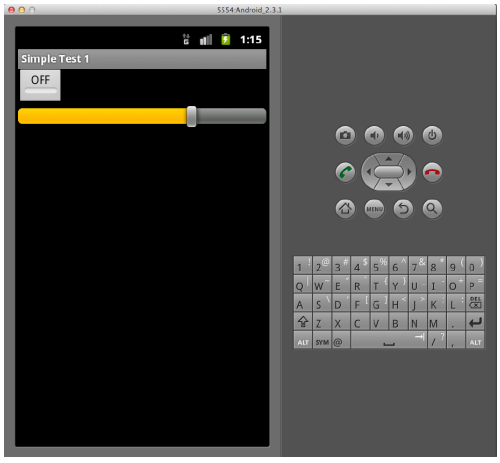


Figura 2.18: Ejemplo de integración de CSound en Android (Yi y Lazzarini, 2012, pág. 5).

[11] El Blues de doce compases o *Twelve Bar Blues* es una progresión de acordes típica del *blues*. Para más información véase (Benward, 2014).

Dodge (1988) presenta una nueva composición asistida por ordenador fundamentada en la geometría fractal: *Profile*, compuesta en el *Center for Computer Music* en el *Brooklyn College* en el año 1984, y será la primera de una lista de composiciones realizadas por el mismo autor que utilizan algoritmos similares [12]. El algoritmo utilizado en la composición de *Profile* utiliza un ruido de tipo $1/f$ para generar las alturas, las duraciones y la amplitud de cada una de las tres líneas melódicas que componen la estructura de la obra. La estructura global de la obra implementa una analogía musical con las *Curvas de Peano*[13] La forma en la que se generan las notas en cada linea melódica es la siguiente: el compositor especifica el número de *pitch classes* que serán generadas; la primera línea comprende al menos tantas notas como límite del pitch class especificado, aunque usualmente el número es mayor a causa de la naturaleza del ruido $1/f$. Posteriormente el programa continúa con la segunda línea, creando una secuencia de notas para cada una de las notas de la primera línea hasta que se alcanza una cierta diversidad de pitch classes. En la tercera línea se genera una secuencia de notas para cada una de las notas de la segunda línea. En este punto el programa almacena los pitch classes generados, pero sin valores temporales. En el siguiente paso del proceso se le asignarán duraciones mediante un algoritmo recursivo muy similar.

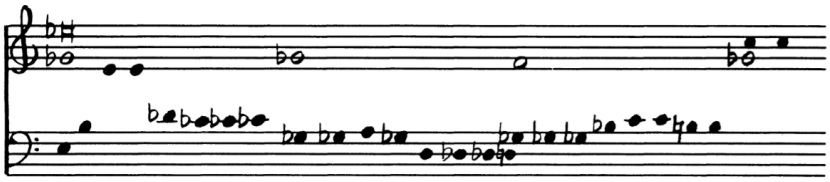


Figura 2.19: Ejemplo de las tres líneas melódicas generadas por el proceso (Dodge, 1988, pág. 13).

Beyls (1989) ha sido uno de los pioneros en la utilización de autómatas celulares para la generación de estructuras musicales, expandiendo los acercamientos tradicionales al tema e inventando métodos adicionales para la definición de vecindad que permiten considerar un mayor número de estados para las celdas que serán consideradas en el cálculo del siguiente estado. Realiza una asignación de alturas en función de la posición de las celdas mientras que los ritmos resultan de la regla

[12] Otras de las composiciones mediante algoritmos fractales de Dodge son *Roundelay* (1985), *A Postcard from the Volcano* (1986), *Song Without Words* (1986), *Viola Elegy* (1987).

[13] Las *curvas de Peano* toman su nombre en honor al matemático italiano *Giuseppe Peano*. Son curvas continua que recubren todo el plano y se obtienen mediante una sucesión de curvas continuas, sin ningún tipo de intersección entre sí, que en el caso límite convergen a una curva que posee propiedades fractales, ya que su dimensión topológica es 1 pero su dimensión fractal *Hausdorff-Besicovitch* es 2 (Lipschutz, 1970).

en la que dos alturas idénticas sucesivas se fusionan en una nota de valor doble. Beyls (2003) presenta su programa *CA Explorer* en el que se genera la estructura musical con un autómata celular monodimensional procesado por un algoritmo genético.

Uno de los primero autores en considerar las redes neuronales para la composición asistida por ordenador ha sido Todd (1989). En su artículo *A connectionist approach to algorithmic composition* explica como se puede representar mediante el acercamiento neuronal un factor determinante en la música como es el tiempo, proporcionando dos alternativas capaces de resolver tal cuestión y aplicándolas a la producción de melodías. Lewis (1989) continúa con esta línea de investigación, proponiendo variantes al paradigma de la *Creación por Refinamiento (CBR)*[14] previamente intratable, mostrando sencillos ejemplos computacionales de secuencias armónicas generadas mediante un nuevo método denominado *attentional CBR*.

(0.200000 I III VII IV III V VII VI V VI VI I)
(0.200000 I II II VII V VII II VII VI III II I)
(0.500000 I IV I II V IV III IV VI II V I)
(0.500000 I V IV V IV V II VII VII III VII I)
(0.800000 I V IV I V V I VI IV V I I)
(0.800000 I I II V VI IV VI VII I I V I)

Figura 2.20: Secuencias de acordes generadas por el algoritmo *attentional CBR* entrenado sobre un corpus de progresiones armónicas sencillas (Lewis, 1989, pág. 183).

En 1990 aparece el artículo *Algorithmic Composition: Quantum Mechanics and the Musical Domain* en el que Bain (1990) propone aplicar las ecuaciones que rigen la mecánica cuántica (Ecuación de Schöringer) sobre el ámbito de la composición musical, realizando una analogía entre las frecuencias discretas de un sistema cuántico, como por ejemplo sucede en un átomo, y las frecuencias discretas producidas por los instrumentos tradicionales, ya que éstas últimas se encuentran relacionadas mediante integrales múltiples de algunas frecuencias fundamentales por lo que pueden ser expresadas como un conjunto discreto y cuantizado de ondas estacionarias. Los resultados compositivos que obtiene Bain de esta original aproximación se verán reflejados en su obra *Retreat from Quiescence*.

[14]La Creación por Refinamiento (CBR) es un paradigma de redes neuronales desarrollado específicamente para problemas relacionado con la creatividad artificial, como por ejemplo la composición mediante ordenador. El método CBR consiste en una fase de aprendizaje supervisado seguida de una fase de creación en la cual se realiza una creación aleatoria va siendo sometida a procesos de mejora hasta que cumple con los criterios deseados. Lewis expone que las aplicaciones del método CBR en la composición musical son muy limitadas a causa del elevado gasto computacional que requiere el algoritmo de *retropropagación* (Lewis, 1991).

2.5 Años 90

En 1990 Francis Courtot presenta en su artículo *A constraint-based logic program for generating polyphonies* una versión aún en desarrollo del lo que poco después sería uno de los primeros entornos gráficos de composición algorítmica, denominado *CARLA*. Este software se muestra como un intento de utilizar una interfaz de programación visual sobre un sistema de programación basado en la lógica y programado sobre el lenguaje *Prolog II*, de tal manera que permite a cada compositor definir su propio conjunto de reglas sintácticas generadoras de material musical. Uno de los primeros compositores en utilizar CARLA será el francés Philippe Hurel en su obra *Six miniatures en trompe l'ail* como consecuencia de un encargo del *Ensemble Intercontemporain*. Otros compositores que utilizarán este sistema son M. Stroppa en su obra *Elet..., fogytiglan*; M. Jarrell, quien utiliza CARLA para generar polifonías basadas en determinados contenidos interválicos, o el propio F. Courtot.

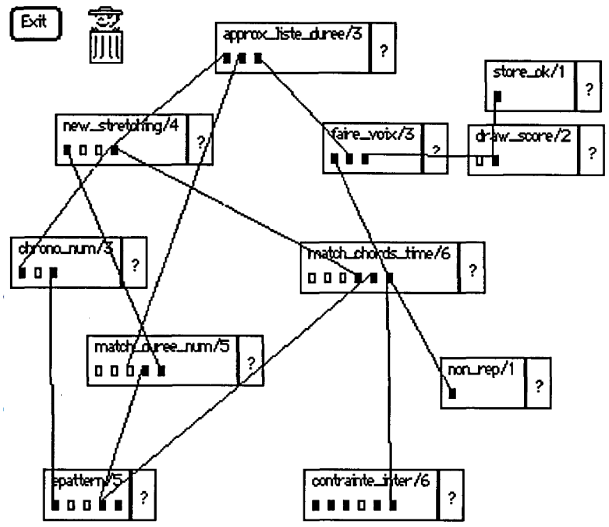


Figura 2.21: Red definida en CARLA por el compositor Philippe Hurel para su obra su obra *Six miniatures en trompe l'ail* (Courtot, 1992, pág. 203).

También en el IRCAM y de manera simultánea, tras varios prototipos como *Preform*, *Esquisse* o *Berriere 90*, comienza el proyecto *PatchWork*, dirigido por Assayag y col. (1999). PatchWork es una herramienta gráfica para la composición asistida por ordenador, escrita en *Common Lisp* para Macintosh. Cualquier ins-

trucción puede ser transcrita en una operación gráfica a través de un conjunto de cajas relacionadas entre sí. El núcleo del programa contiene un amplio conjunto de cajas predefinidas, cada una de ellas capaz de realizar una función determinadas. La interfaz gráfica de PatchWork utiliza el paradigma de orientación al evento y permite posicionar, mover y editar las cajas a través del ratón y del teclado. Está basado en la noción de *Patch*: un diseño de los elementos visuales dentro de una ventana.

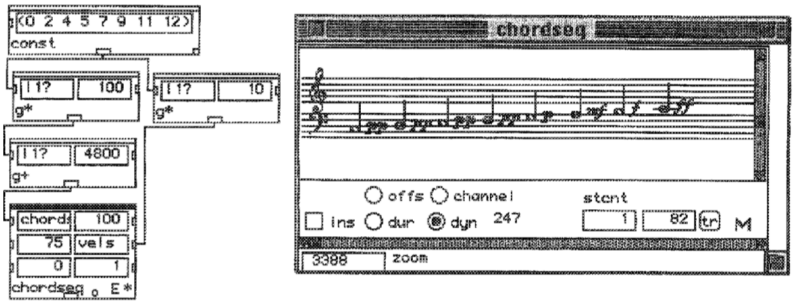


Figura 2.22: Ejemplo de patch diseñado en PatchWork (Assayag y col., 1999, pág. 203).

PatchWork permite ser ampliado de manera sencilla mediante una arquitectura de librerías; de esta manera cualquier usuario puede empaquetar sus *patchs* en archivos externos que pueden ser invocados en cualquier momento desde PatchWork. Muchas de las librerías que algunos compositores han desarrollado se han ido incorporando en la distribución inicial del programa como por ejemplo el módulo *Esquisse*, desarrollado por Tristan Murail para operaciones relacionadas con el espectralismo; *Chaos/Alea*, desarrollado por Mikhail Malt con implementación de procesos estocásticos y modelos dinámicos; *Kant*, escrito por Gerard Assayag, Carlos Agon, Joshua Fineberg y Camilo Rueda, para la transcripción inteligente de ritmos; *PWConstrains*, desarrollada por Mikael Laurson para la programación basada en reglas; *Situation*, escrita por Camilo Rueda y Antoine Bonnet para la generación de estructuras armónicas y rítmicas basadas en restricciones; y *Repmus*, desarrollada por Gerar Assayag y Claudy Malherbe para la generación y procesado de estructuras rítmicas y armónicas (Assayag y col., 1999).

Basándose en la experiencia obtenida por el desarrollo de su predecesor PatchWork, *Open Music* implementa un conjunto de nuevas funcionalidades que resultan en una poderosa y flexible herramienta compositiva. El programa está desarrollado sobre el lenguaje de programación *CLOS (Common Lisp Object System* y proporciona una potente interfaz gráfica orientada a objetos con la que la programación se puede estructurar de forma visual. Existen numerosas *clases* con

comportamientos y funcionalidades preestablecidos que pueden interconectarse entre sí. La notación musical se realiza en *Common Music Notation (CMN)*, una librería gráfica multiplataforma desarrollada por Bill Schottstaedt en el *Center for Computer Research in Music and Acoustics*, de la Universidad de Stanford.

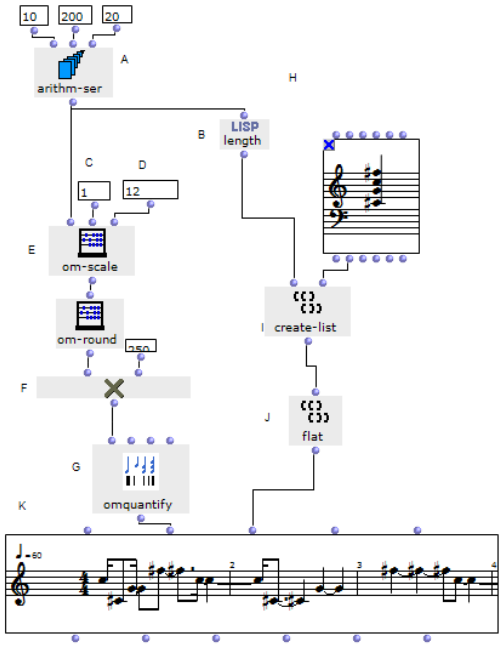


Figura 2.23: Ejemplo de patch procedente del Tutorial nº30 incluido con la versión 6.12 (Assayag y col., 1999, pág. 4).

De la misma manera que en su predecesor, Open Music trabaja con el concepto de *patch* donde el usuario ubica los objetos, funciones, clases, *subpatches* y demás elementos para posteriormente conectarlos entre sí y especificar, de forma gráfica, los algoritmos deseados. Las funciones en Open Music poseen tantas salidas (*outlets*) como necesiten, a diferencia de PatchWork que sólo disponía de un outlet por función. El mecanismo de abstracción que permite la arquitectura de patch es muy clara: cuando un patch se introduce dentro de otro éste aparece dibujado con sus entradas (*inlets*) y salidas (*outlets*), admitiendo además la recursividad, por lo tanto un patch puede llamarse a sí mismo. OpenMusic se ha mostrado con el tiempo como una herramienta muy potente para el diseño de sofisticados métodos con los que generar estructuras musicales cuyo uso se ha generalizado hasta nuestros días (Assayag y col., 1999).

Millen (1990) presenta su software *Cellular Automata Music* en el que se pueden emplear autómatas celulares de una, dos y hasta tres dimensiones para generar las notas musicales gracias a una matriz de valores MIDI y duraciones. En el artículo Generation of formal patterns for music composition by means of cellular automata, Millen explica el funcionamiento de un autómata celular unidimensional con dos estados posibles por celda y un conjunto de vecindad consistente en cinco celdas, en las cuales se pueden obtener patrones musicales que, en función del conjunto de reglas establecido, pueden ser interpretados en términos de la repetición con variación.

En el artículo *Genetic algorithms and computer-assisted music composition* (Horner y Goldberg, 1991) podemos encontrar una de las primeras aplicaciones de los algoritmos genéticos a la composición musical asistida por ordenador. Los autores describen la técnica evolutiva para la generación de material melódico a modo de transiciones iterativas entre dos pequeñas melodías (*Thematic bridging*). La técnica modifica sucesivamente un patron melódico inicial y compara el resultado con un patrón melódico final, siendo seleccionados los resultados con mejor función *fitness*. Para la generación de la composición, los autores utilizan seis ciclos como los anteriores para producir material melódico que posteriormente será estructurado a modo de canon imitativo a cinco voces.

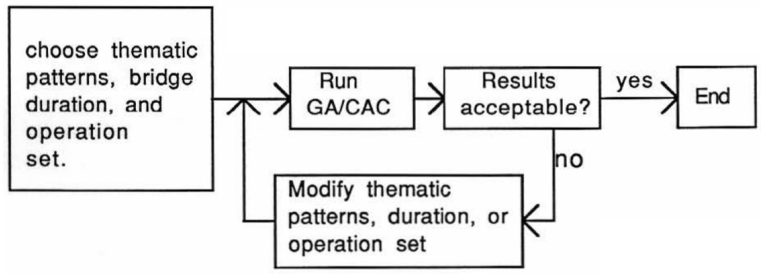


Figura 2.24: Esquema de la modificación iterativa utilizada por Horner y Goldberg (Horner y Goldberg, 1991, pág. 2).

Winsor (1991) presenta *Pat-Proc*, un programa interactivo, escrito inicialmente en *BASIC* y posteriormente transcrito en *C*, con la finalidad de generar partituras de hasta 20 voces utilizando algunas de las técnicas existentes en la música minimalista, como por ejemplo la utilización de procesos cíclicos (denominados *loops* en inglés) establecidos por el usuario o procesos algorítmicos para la generación melódica, como por ejemplo rotaciones y retrogradaciones. El propio Winsor utiliza Pat-Prop para la composición de distintas obras como por ejemplo *Dulcirner dream*, *Dervish 2*, *Les Chemins des Nuages*, y *Suite for Disklavier*.

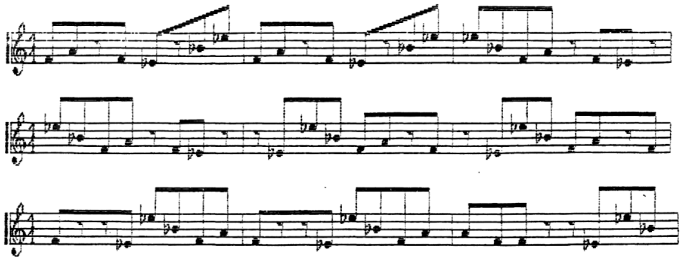


Figura 2.25: Ejemplo de rotaciones y retrogradaciones especificadas por el usuario en PatProc sobre una melodía inicial (Winsor, 1991, pág. 117).

Miranda (1993) presenta su software *CAMUS (Cellular Automata MUSic*, junto con *CAMPLAY* y*PROCAMUS*[15]. El autor propone una representación para las triadas sobre un espacio bidimensional denominado *von Neuman Musical Space* (Miranda, 1993) que contiene el rango completo de pares de intervalos, expresados por su número de semitonos, que puede contener cualquier triada. Sobre este espacio el autor propone dos tipos de autómatas celulares: el primero es el *Juego de la vida* de Conway (1970), el segundo es el *autómata celular cíclico*[16]. Ambos autómatas trabajan en paralelo para la determinación del material musical y de la orquestación, respectivamente.

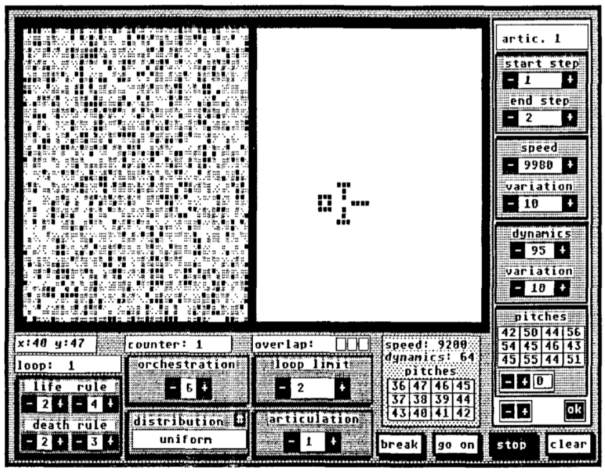


Figura 2.26: Pantalla principal de CAMUS (Miranda, 1990, pág. 15).

[15]Para más información sobre PROCAMUS véase *Cellular Automata Music Investigation* (Miranda, 1990).

[16]Véase Cyclic cellular automata in two dimensions (Fisch, Gravner y Griffeath, 1991).

Una interesante aportación al ámbito de la composición musical utilizando algoritmos genéticos es la aplicación *GenJam*, presentada por Biles (1994). Consiste en un modelo que genera improvisaciones en estilo Jazz, manteniendo relaciones jerárquicas entre distintas ideas melódicas sugeridas por la progresión de acordes preestablecida que se está interpretando. GenJam es capaz de interpretar sus propios solos sobre el acompañamiento de una sección rítmica estándar y al mismo tiempo recibir información en tiempo real de un humano que se utiliza para mejorar los valores de *fitness* óptimos para cada uno de los compases y fraseos. Para improvisar sobre una tonalidad establecida, el programa lee la progresión armónica desde un archivo que además le proporciona el tempo, el estilo rítmico y el número de compases de *coros* para la duración del solo. El programa es compatible con secuencias MIDI para piano, bajo y percusión generadas previamente con el programa *Band-in-a-Box*.

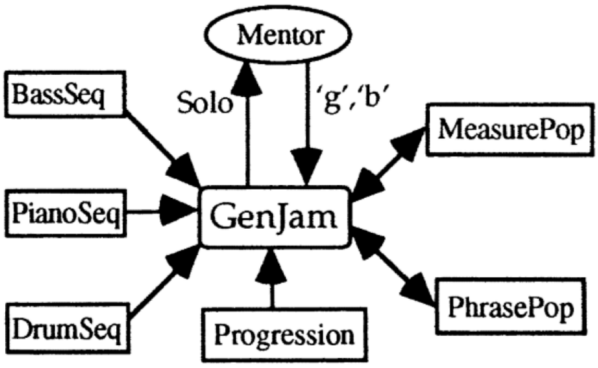


Figura 2.27: Arquitectura del sistema GenJam (Biles, 1994, pág. 132).

Otro ejemplo de aplicación de algoritmos evolutivos a la composición musical lo encontramos en el artículo *Harmonization of Musical Progressions with Genetic Algorithms* en el que los autores Horner y Ayers (1995) implementan un algoritmo de estas características para la realización de progresiones armónicas complejas, que puede llegar a incluir acordes de séptima, dominantes secundarias, acordes cromáticos e incluso modulaciones. El programa también es capaz de armonizar una melodía, generando las voces internas, si se proporciona la secuencia armónica a seguir. El algoritmo genético en este caso contiene una función de *fitness* obtenida a partir de las numerosas restricciones armónicas presentes en la armonía tonal.

Jacob (1995) presenta *variations*, un sistema para la composición algorítmica basada en los algoritmos genéticos. El programa es un intento de reducir los procesos compositivos a unas pocas reglas sencillas que puedan ser fácilmente transformadas en instrucciones computacionales. El procedimiento básico del funcionamiento de variations es el siguiente:

- Se define un conjunto de motivos primarios que serán utilizados.
- Se componen frases mediante la superposición y la secuenciación de varios motivos.
- Se crean frases, uniendo motivos seleccionados y motivos modificados mediante la selección evolutiva.
- Se juntan las frases, creando fragmentos musicales más amplios.

El componente humano se encuentra en la definición del conjunto primario de motivos y en la posterior escucha mediante la herramienta *Ear*. Si el usuario escucha el fraseo generado y lo desaprueba, el programa desechará el resultado obtenido de tal manera que al final el usuario va obteniendo un conjunto de frases utilizables que pueden ser nuevamente recombinadas para obtener más material musical.

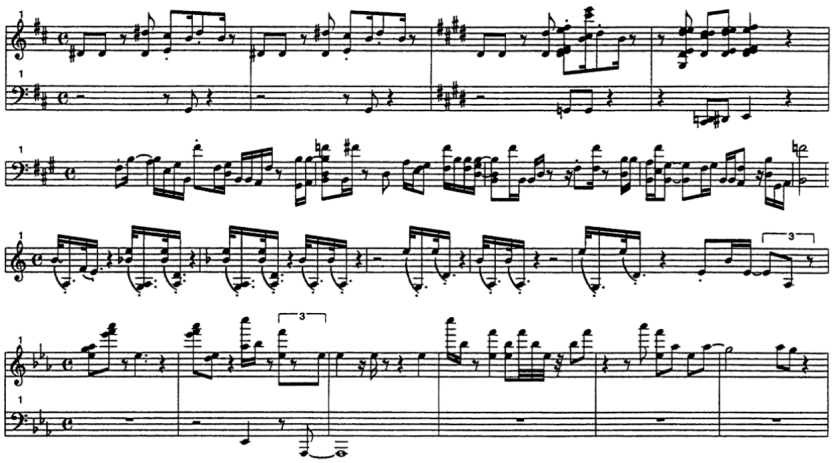


Figura 2.28: Ejemplos de variaciones obtenidas como resultado (Jacob, 1995, pág. 455).

Otros autores experimentan con la utilización de algoritmos genéticos en distinto ámbitos de la composición musical son Hartmann (1990) o McIntyre (1994).

En el ámbito de la utilización de redes neuronales con propósitos compositivos cabe mencionar el sistema *HARMONET*, presentado por Hild, Feulner y Menzel (1992), capaz de armonizar melodías en el estilo de los corales de J. S. Bach mediante la utilización de redes neuronales supervisadas por un conjunto jerárquico de reglas, combinando las ventajas de ambos métodos. El sistema ha de ser entrenado con un corpus de doce corales de Bach utilizando la retropropagación. El sistema obtiene el esqueleto armónico reduciendo cada coral a sus armonías fundamentales, considerando las corcheas y semicorcheas como notas ornamentales ajenas a la armonía (Martínez, 2019).

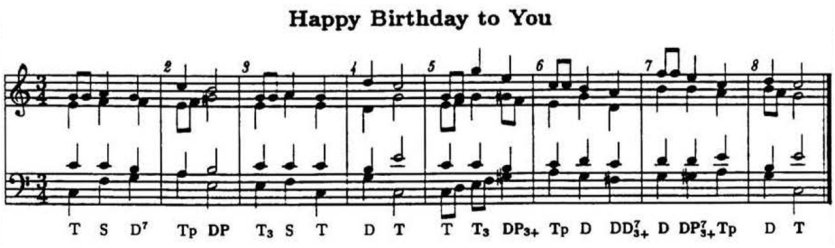


Figura 2.29: *Cumpleaños feliz* armonizado por HARMONET (Hild, Feulner y Menzel, 1992, pág. 273).

En una línea de investigación muy parecida, Mozer (1994) propone la red neuronal *CONCERT*. Este sistema se entrena con un corpus de melodías de las que extrae regularidades existentes en progresiones de notas y progresiones armónicas que serán interpretadas como restricciones melódicas y estilísticas. CONCERT incorpora restricciones de tipo psicoacústico extraídas de estudios psicológicos sobre la percepción humana. Los resultados de este sistema parecen ser superiores, bajo una escucha final, a los resultados obtenidos por cadenas de Markov de tercer orden (Martínez, 2019).

Laine (1997) propone la generación de patrones musicales mediante la utilización de neuronas artificiales mutuamente inhibidas, conocidas como *sistemas MINN*. Estos sistemas se caracterizan por un descenso del nivel de activación de las neuronas conectadas y están relacionadas con el funcionamiento del cerebro en fenómenos biológicos en los que existe una acción motriz mantenida por largo tiempo como por ejemplo en el latido cardíaco, al andar o al nadar. El sistema implementado con finalidades compositivas es capaz de generar una amplia variedad de figuras rítmicas y al mismo tiempo procesos cíclicos de carácter repetitivo.

Puckette (1986) presenta Patcher, un entorno gráfico para el control y configuración de los objetos en el entorno *Max*, inventado por él mismo durante la década de los ochenta (Puckette, 1996). Su interfaz gráfica continua con la estructura de patchs, permitiendo al usuario conectar módulos a través de cuerdas por los que circulan mensajes de control o señales. Estos módulos generalmente representan unidades de procesamiento, entradas o salidas, *buffers* o conexiones MIDI.

Patcher es escrito por Miller Puckette para la composición de la obra *Plutón* del compositor Philippe Manoury. Patcher inicialmente se ejecutaba sobre Macintosh y se encargaba de realizar únicamente control MIDI. Posteriormente Patcher fue rediseñado por David Zicarelli, conociéndose como *Max/Opcode* e introduciendo muchas mejoras en la interfaz de usuario así como novedosos objetos externos (Puckette, 2001).

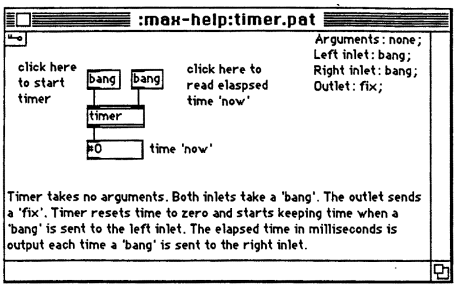


Figura 2.30: Ventana de ayuda en la primera versión de Max (Puckette, 1986, pág. 429).

El proyecto *IRCAM Musical Workstation*, iniciado en 1989, introduce una nueva versión de Max que incorpora el procesamiento en tiempo real de señales de audio. Este software se distribuye bajo el nombre de *Max/ISPW* (*IRCAM Signal Processing Workstation*) y se organiza en dos componentes; el primero es la interfaz gráfica de usuario conocida como *NeXTSTEP*, el segundo es un pequeño motor de ejecución en tiempo real llamado *FTS (Faster Than Sound* basado en el procesador i860 de Intel (Déchelle y col., 1999).

El equipo de IRCAM *Real time systems*, creado en 1995 por François Déchelle, comienza nuevos desarrollos basados en los componentes de ISPW. Sobre la evidencia de que el desarrollo de nuevo hardware ya no valía la pena al ser muy costoso, se toma la decisión de abandonarlo para desarrollar en su lugar programas informáticos con alto nivel de *portabilidad*. Para ayudar en este desarrollo multiplataforma se decide separar la interfaz gráfica de usuario del motor de ejecución en tiempo real, haciendo así independientes la evolución de las partes gráficas y de procesado de audio. Esta versión se denomina *Max/FTS* (Déchelle y col., 1999).

Al mismo tiempo, Miller Puckette comienza el desarrollo de *Pure Data* con el objetivo de remediar algunas debilidades de Max en el campo de la gestión dinámica de las estructuras de datos (Kreidler, 2009). Pure Data utiliza una arquitectura de dos componentes similar a Max/FTS y consigue la portabilidad en el lado gráfico a través de la adopción del kit de herramientas *TCL/TK* (Puckette, 1996).

Con la aparición del lenguaje Java, se amplía la posibilidad de realizar interfaces gráficas multiplataforma. La nueva implementación en Java de la interfaz gráfica de usuario de Max/FTS comienza a finales de 1996 y se le proporciona el nombre de *jMax* (Déchelle y col., 1999).

Al reutilizar la parte de audio de Pure Data, David Zicarelli lanza a fines de 1997 el paquete *MSP (Max Signal Processing)* para Max/Opcode con el que consigue incorporar la síntesis en tiempo real y el procesamiento de señal a Max / Opcode, para las plataformas Macintosh (Puckette, 2007). A causa de la elegancia de la interfaz de usuario, el número de objetos disponibles y la fácil conexión con *plug-ins VST* han hecho de este sistema, denominado *Max/MSP*, un éxito inmediato y lo han coronado con un estándar internacional en los campos de la composición asistida por ordenador y la síntesis sonora[17]. Actualmente el programa está bajo desarrollo y mantenimiento de la empresa se software *Cycling '74*, la cual ha desarrollado sus propias librerías para el manejo de video, *Jitter*, basadas en gráficos *OpenGL* y proporcionando capacidad de cálculo con matrices.

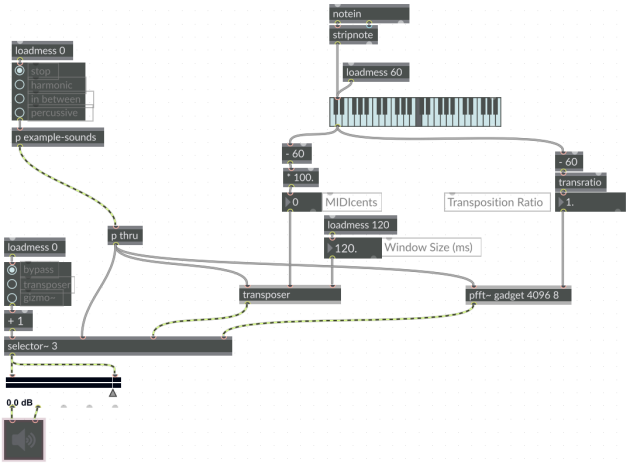


Figura 2.31: Ejemplo de patch en Max/MSP en la versión 7.3.5.

[17]Para más información sobre el lenguaje Max/MSP véase el libro *Electronic music and sound design* de los autores Cipriani y Giri (2010).

McCartney (1996) presentan *SuperCollider*, un entorno para programación orientada a la composición musical asistida por ordenador y la síntesis de sonido en tiempo real. Contiene un lenguaje de programación, un sistema de clases orientado a objetos, funciones predefinidas, una gran librería de objetos para el procesado de la señal de audio, incluso una pequeña interfaz de usuario para crear programas interactivos con el usuario o interaccionar con instrumentos MIDI. El usuario puede escribir algoritmos compositivos o destinados a la síntesis sonora en un lenguaje de programación de alto nivel. De esta manera se permite la creación de instrumentos y procesos compositivos muy complejos con un nivel de flexibilidad superior al permitido por otros lenguajes de síntesis coetáneos.

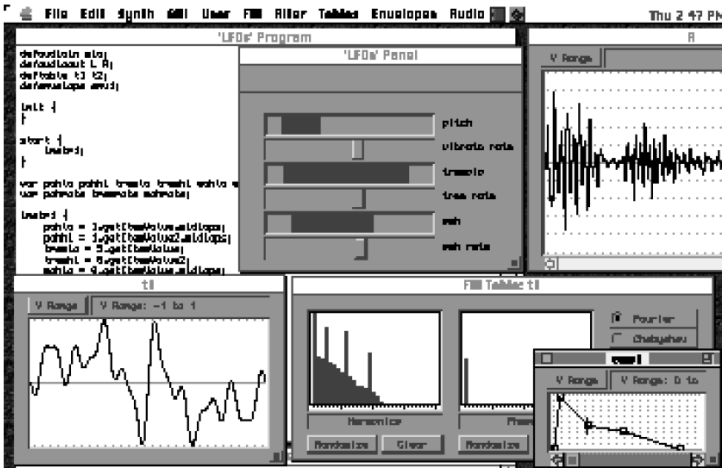


Figura 2.32: Entorno de desarrollo de la versión inicial de SuperCollider (McCartney, 1996, pág. 257).

El origen de SuperCollider se encuentra en programas diseñados a principios de los noventa por McCartney que todavía no tenían la funcionalidad en tiempo real, como por ejemplo *Synth-O-Matic* o *Pyrite*. A partir de la versión 3, el entorno de SuperCollider ha sido separado en dos componentes: el servidor (*scsynth*) y el cliente (*sclang*). Estos componentes se comunican a través del protocolo *Open Sound Control (OSC)* (McCartney, 2002).

Desde su aparición en 1996, el entorno ha disfrutado de una amplia acogida, siendo utilizado por innumerables compositores que han ido desarrollando una amplia base de datos de librerías y patchs en ámbitos de la investigación en acústica, la composición algorítmica, la síntesis musical o incluso en el reciente campo del *live coding* con aplicaciones como *ixi lang* (Magnusson, 2011).

2.6 Años 2000

Biletskyy (2000) presenta el software *Doctor Webern*, un entorno visual para la composición asistida por ordenador basada en la linealidad temática. El programa permite al usuario introducir temas musicales representados como objetos abstractos, almacenarlos y editarlos por aumentación, disminución o retrogradaciones. Los temas se pueden distribuir de manera gráfica en cada una de las voces existentes en la composición. De esta manera Doctor Webern no incorpora herramientas derivadas de la inteligencia artificial ya que es el usuario quien establece las reglas y realiza en última instancia las decisiones sobre qué combinaciones de temas son aceptables.

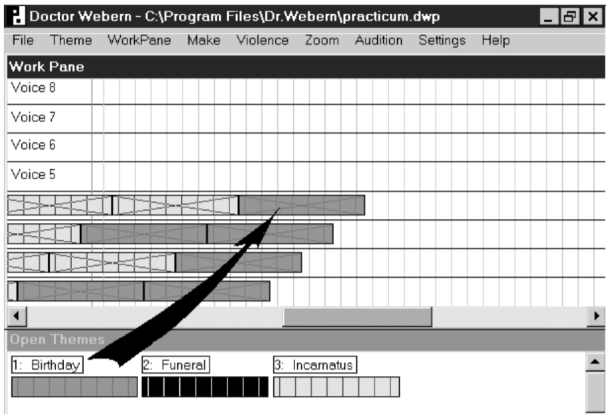


Figura 2.33: Ventana principal del software Doctor Webern (Biletskyy, 2000, pág. 36).

Jones (2000) presenta en su artículo *A Computational Composer's Assistant for Atonal Counterpoint* la aplicación informática *Counterpoint Assistant* (CPA), un programa destinado a la composición asistida por ordenador escrito en *Macintosh Common LISP* (MCL). Proporciona una herramienta matemática para encontrar transiciones y desplazamientos temporales sobre líneas melódicas creadas por el usuario que pueden ser superpuestas para encontrar soluciones a las restricciones armónicas o contrapuntísticas especificadas por el usuario. El programa, a diferencia de otras aplicaciones que son capaces de componer contrapunto imitando estilos preexistentes, se limita a proporcionar al compositor un conjunto de herramientas con las que poder construir un conjunto coherente de elementos armónicos y melódicos. El proceso que sigue es el siguiente: primero el usuario compone manualmente un conjunto de melodías, para a continuación definir las restricciones armónicas, melódicas o contrapuntísticas. El programa calcula y evalúa todas las

combinaciones posibles de estas melodías eliminando progresivamente las combinaciones que no encajan con las restricciones establecidas, quedando como resultado final aquellas combinaciones que sí las cumplen, ordenándolas siguiendo determinados criterios.

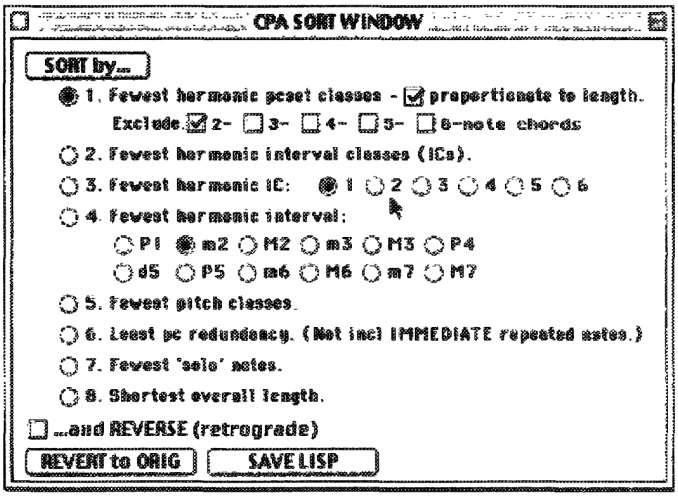


Figura 2.34: Ventana de ordenación en CPA en la ventana de *sort* (Jones, 2000, pág. 37).

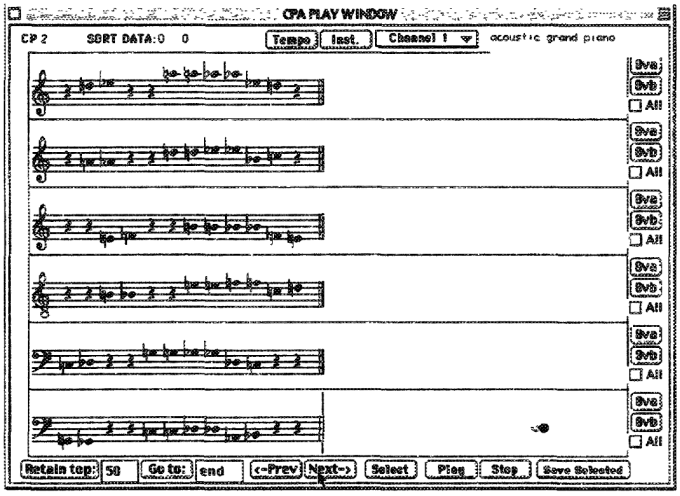


Figura 2.35: Ejemplo de contrapunto mostrado en la ventana de *play* (Jones, 2000, pág. 37).

Tokui e Iba (2000) desarrollan un sistema interactivo denominado *CONGA* (*Composition in genetic approach*) que permite obtener patrones rítmicos mediante técnicas de computación evolutiva. Desarrollado en *Borland C++* para la plataforma *Windows*, el programa está basado en el estándar MIDI para la codificación de los patrones rítmicos mediante su valor de *velocity*, comprendido en un valor numérico entre 0 y 127.

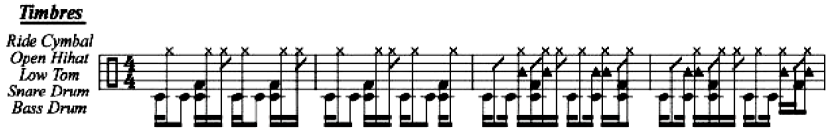


Figura 2.36: Ejemplo de frase rítmica generada por *CONGA* (Tokui e Iba, 2000, pág. 10).

En el artículo *Using multiattribute prediction suffix graphs to predict and generate music*, los autores Triviño-Rodriguez y Morales-Bueno (2001) proponen un sistema de composición asistida por ordenador basándose en modelos de Markov avanzados denominados *Probabilistic Suffix Automata* (PSA), consistentes en cadenas probabilísticas de longitud L variable. Los resultados experimentales descritos por los autores, consistentes en la generación de melodías a partir del sistema entrenado previamente sobre un corpus compuesto por cien corales de J. S. Bach, demuestran una alta semejanza en términos de utilización de alturas y de duraciones existentes entre las melodías generadas y las compuestas por Bach (Martínez, 2019).

Farbood y Schöner (2001) presentan un sistema capaz de generar de contrapunto mediante ordenador en el estilo de Palestrina, tomando como *input* un *cantus firmus* proporcionado por el usuario. Los autores proponen un modelo basado en la utilización de cadenas de Markov de segundo orden. Cada regla compositiva del contrapunto se implementa como una tabla de probabilidad donde las transiciones interválicas prohibidas poseen una probabilidad de cero. De esta manera, la *tabla armónica* determina cuándo el intervalo entre la nota y su respectivo cantus firmus es permisible; la *tabla melódica* describe la probabilidad de un intervalo melódico entre dos notas consecutivas del contrapunto; la *tabla de cadencias* asegura que haya una cadencia apropiada al final de cada ejemplo; la *tabla cromática* proporciona pesos muy bajos a las notas alteradas cromáticamente, suficientemente bajos como para que sean utilizadas únicamente cuando no existe otra solución; la *tabla de buenos movimientos paralelos* previene de la existencia de demasiadas terceras, sextas y décimas paralelas; la *tabla de movimientos cercanos* prohíbe determinados movimientos directos, como por ejemplo quintas y octavas escondidas; la *tabla de movimiento de salida* prohíbe salir de un unísono por

movimiento directo; la *tabla de movimiento general* asigna altas probabilidades a los movimientos contrarios y oblicuos, y baja a los movimientos directos; y por último la *tabla de clímax* proporciona la probabilidad de cero si el cantus firmus y el contrapunto se mueven en la misma dirección (Martínez, 2019).

	m2u	M2u	m3u	M3u	P4u	P5u	m6U	P8u	m2d	M2d	m3d	M3d	P4d	P5d	P8d	P1
m2u	0	.45	.2	.2	0	0	0	0	.035	.035	.025	.025	.01	.01	.009	.001
M2u	.45	.45	.03	.03	0	0	0	0	.01	.01	.005	.005	.004	.004	.002	.001
m3u	.45	.45	.03	.03	0	0	0	0	.01	.01	.005	.005	.004	.004	.002	.001
M3u	.35	.35	.05	.05	0	0	0	0	.05	.05	.025	.025	.025	.025	.024	.001
P4u	.065	.065	0	0	0	0	0	0	.4	.4	.02	.02	.01	.01	.009	.001
P5u	0	0	0	0	0	0	0	0	.52	.52	.01	.01	.01	.01	.009	.001
m6u	0	0	0	0	0	0	0	0	.51	.51	.025	.025	.01	.01	.009	.001
P8u	0	0	0	0	0	0	0	0	.51	.51	.025	.025	.01	.01	.009	.001
m2d	.05	.05	.005	.005	.002	.002	.002	.002	0	.467	.2	.2	.015	0	0	.001
M2d	.05	.05	.005	.005	.002	.002	.002	.002	.367	.367	.1	.1	.015	0	0	.001
m3d	.366	.366	.005	.005	.002	.002	.002	.002	.05	.05	.1	.1	0	0	0	.001
M3d	.366	.366	.005	.005	.002	.002	.002	.002	.05	.05	.1	.1	0	0	0	.001
P4d	.53	.53	.02	.02	.02	.02	.039	.02	0	0	0	0	0	0	0	.001
P5d	.454	.454	.03	.03	.01	.005	.005	.011	0	0	0	0	0	0	0	.001
P8d	.454	.454	.03	.03	.01	.005	.005	.011	0	0	0	0	0	0	0	.001
P1	.1	.1	.08	.08	.05	.05	.04	.03	.1	.1	.08	.08	.05	.04	.02	0

Figura 2.37: Tabla para las reglas de transición melódica (Farbood y Schöner, 2001, pág. 3).

Los autores Puente, Alfonso y Moreno (2002) proponen la herramienta *GEMUSIC* mediante la cual se pueden generar computacionalmente líneas melódicas que suenan similares a las supuestamente compuestas por humanos. El acercamiento de esta propuesta consiste en la utilización de gramáticas evolutivas.

En su artículo *A musical learning algorithm*, el autor Cope (2004) presenta *Gradus*, un programa informático que a partir del análisis de un conjunto de contrapuntos de primera especie a dos voces, es capaz de generar contrapuntos nuevos, similares a los modelos, sobre un *cantus firmus* proporcionado. Gradus utiliza seis categorías de objetivos, que vienen a resumir las reglas existentes en las especies del contrapunto de Fux.

Millen (2004) presenta un software para la composición asistida por ordenador basado en los autómatas celulares. El programa, escrito en los lenguajes *Cocoa* y *Objective-C*, permite controlar al compositor las celdas que estarán disponibles para representar las alturas y las duraciones de las notas. El programa permite la modificación en tiempo real de otros parámetros como el tempo o la transposición del resultado musical generado. Las distintas configuraciones obtenidas pueden ser guardadas en un fichero para su posterior reutilización. El autómata celular implementado es de tipo *K3R1* [18] tiene una vecindad de tres celdas y un rango de una celda a cada lado de la celda objetivo. El estado cero se representa con el color negro, el estado uno por el colo rojo y el estado dos por el color amarillo. La regla del autómata se basa en la media de la suma de los estados de las celdas vecinas. Existen siete posibilidades, desde cero hasta seis. Mediante la selección

[18] Para más información sobre este tipo de autómatas véase *A new kind of science* (Wolfram, 2002).

de la regla 1599, el autor muestra *Hanging Gardens*, una composición generada con este autómata.

Buteau (2006) propone un método para el análisis motívico basado en el *clustering*. Para los autores, un motivo musical es una sucesión finita y no nula de tonos musicales. Un conjunto de motivos puede ser ordenado mediante técnicas de clustering. Se realiza una implementación del modelo y se prueba con la obra de Robert Schumann *Träumerei*, comparando los resultados con los del teórico musical Repp. El modelo había sido previamente implementado por los autores Mazzola, Zahorka y Stange-Elbe en 1996, como un módulo del software *RUBATO*, y es en 2002 cuando se vuelve a implementar en Java proporcionándole nuevas funcionalidades y mayor eficiencia computacional.

Weinberg, Godfrey y Rae (2007) describen un sistema musical interactivo que, mediante la utilización de algoritmos genéticos, realiza una colaboración entre músicos humanos y un robot que improvisa sobre un xilófono. El robot se diseña para responder a las señales de entrada humanas de forma acústica, evolucionando en tiempo real una frase musical generada por un humano mediante un algoritmo genético basado en funciones de fitness. Las conexiones entre los intérpretes humanos y el sistema robótico se realizan mediante el protocolo MIDI para el piano digital y desde MAX/MSP para el resto de instrumentos, utilizando el objeto *pitch* capaz de detectar la frecuencia de una señal generada por un instrumento acústico.

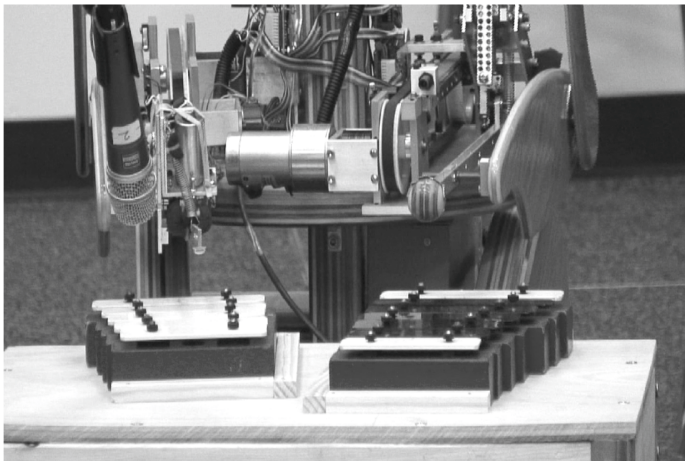


Figura 2.38: Brazos robóticos interpretando una melodía (Weinberg, Godfrey y Rae, 2007, pág. 193).

Tzimeas y Mangina (2007) presentan el software *Jazz Sebastian Bach*, una herramienta para composición asistida por ordenador basada en una nueva idea relacionada con algoritmos genéticos y sus mecanismos para modificar un tema compuesto por Bach hasta que parezca compuesto en estilo Jazz. La propuesta discute las dificultades algorítmicas más importantes de los *Automatic Fitness Assessment* (AFA) y los *Interactive Genetic Algorithm* (IGA) implementados sobre sistemas compositivos basados en algoritmos genéticos. Los autores analizan y proponen una nueva función de fitness denominada *Critical Damped Oscillator* capaz de resolver determinados problemas musicales.

Keller y Morrison (2007) presentan *Impro-Visor*, un software implementado en *Java* capaz de crear solos de jazz, en tiempo real, sobre progresiones de acordes establecidas. El programa se basa en la generación automática de melodías utilizando gramáticas probabilísticas.

Los autores Gimenes y Miranda (2008) introducen *iMe*, *Interactive Musical Environments*, un sistema musical interactivo basado en los agentes inteligentes que es capaz de aprender a generar música de forma autónoma en tiempo real, bajo la perspectiva de la cognición y percepción humana.

De León y col. (2008) proponen la caracterización de la melodía como un conjunto de reglas derivadas de un sistema genético de tipo *fuzzy*. El objetivo de la investigación es encontrar una técnica que pueda distinguir entre archivos *MIDI* que contengan líneas melódicas de aquellos otros que no. La metodología utilizada es la siguiente: en primer lugar las pistas MIDI son descritas como un conjunto de características estadísticas. A continuación se someten a un proceso de creación de reglas de tipo *crisp* que sean capaces de caracterizar dichas pistas melódicas. A continuación los conjuntos reglas son convertidos en conjuntos de tipo *fuzzy*, proporcionando un método para convertir los atributos melódicos de tipo *crisp* en atributos *fuzzy* sustituyendo la figura del experto humano por la de un algoritmo genético que, proporcionados las definiciones lingüísticas de cada atributo, aprende automáticamente el conjunto fuzzy de parámetros asociados al mismo; esta combinación se conoce como sistema genético fuzzy.

Laurson, Kuuskankare y Norilo (2009) presentan *PWGL (PatchWork Graphical Language*, un entorno de programación visual para la música diseñado sobre *Lisp* y *OpenGL* desde el 2002 hasta 2008. Puede utilizarse para la composición asistida por ordenador, para el análisis musical o para la síntesis sonora. Al igual que *Open Music*, PWGL es un sucesor directo de PatchWork, sin embargo PWGL proporciona un entorno unificado, dedicado exclusivamente a la síntesis sonora que permite definir de forma visual instrumentos y partituras, así como manejar la información de control sobre éstos.

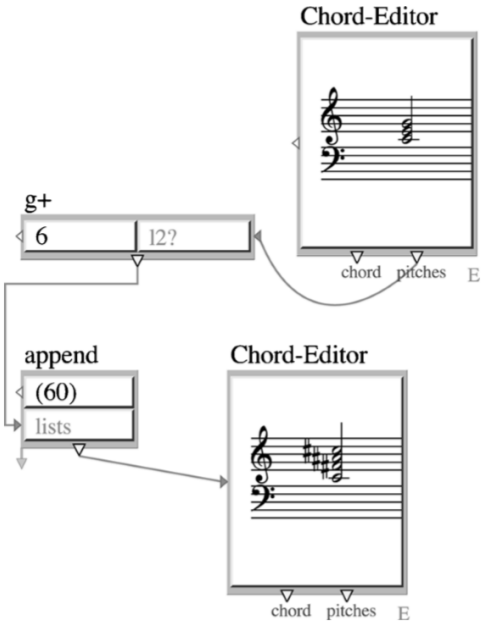


Figura 2.39: Ejemplo de patch en PWGL (Laurson, Kuuskankare y Norilo, 2009, pág. 23).

Psenicka (2009) presenta *FOMUS (FOrmat MUSic)*, un programa de notación musical que automatiza muchas de la labores de creación de partituras a partir de datos de eventos *en bruto* generados desde cualquier otro sistema. El propósito es separar las convenciones de notación y elementos fundamentales de la música tales como altura, duración, dinámica y articulaciones. FOMUS difiere de programas previos como OpenMusic o PWGL en que éste produce partituras totalmente acabadas a través de archivos de salida totalmente compatibles con otras aplicaciones profesionales de edición en formato de *LilyPond* o *MusicXML*.

Los autores Falkenstein y Tlalim (2009) desarrollan *Alter Ego*, un sistema *standalone* programado sobre *SuperCollider*. Posee dos modos, el de entrenamiento y el de *playback*. Su funcionamiento se basa en la implementación de algoritmos genéticos y cadenas de Markov, codificando la amplitud, los parámetros de tiempo de ataque y release, las curvas de desvanecimiento, o el panorámico entre otros (Martínez, 2019).

2.7 Años 2010 y situación actual

Cuthbert y Ariza (2010) presenta *music21* un conjunto de herramientas programadas sobre Python para la musicología asistida por ordenador y la codificación simbólica de la música. El sistema, orientado a facilitar la tarea investigadora tanto a aquellos músicos con poca experiencia en el ámbito de la programación como a programadores con pocos conocimientos musicales, es capaz de proporcionar una gran cantidad de información sobre la partitura previamente codificada en formato MusicXML, mostrando una gran potencia y flexibilidad para el análisis asistido por ordenador. También puede utilizarse para la composición asistida por ordenador, gracias a su sistema de codificación de la notación musical de forma simbólica y su integración completa con el lenguaje de programación Python. En el artículo Feature Extraction and Machine Learning on Symbolic Music using the music21 Toolkit, los autores Cuthbert, Ariza y Friedland (2011) demuestran como el *framework music21* puede integrarse dentro de las técnicas procedentes del *machine learning* y ser aplicadas al análisis musical asistido por ordenador. Esta herramienta proporciona sofisticados métodos para la extracción de características, en este caso musicales, con las que caracterizar distintas partituras (Martínez, 2019).

```
from music21.musicxml import testFiles as xml
from music21.humdrum import testFiles as kern
# display 3D graphs of count, pitch, and duration
mozartStream = music21.parse(
    xml.mozartTrioK581Excerpt)
notes = mozartStream.flat.stripTies()
g = graph.Plot3DBarsPitchSpaceQuarterLength(
    notes, colors=['r'])
g.process()
```

```
# perform the same process on a different work
chopinStream = music21.parse(kern.mazurka6)
notes = chopinStream.flat.stripTies()
g = graph.Plot3DBarsPitchSpaceQuarterLength(
    notes, colors=['b'])
g.process()
```

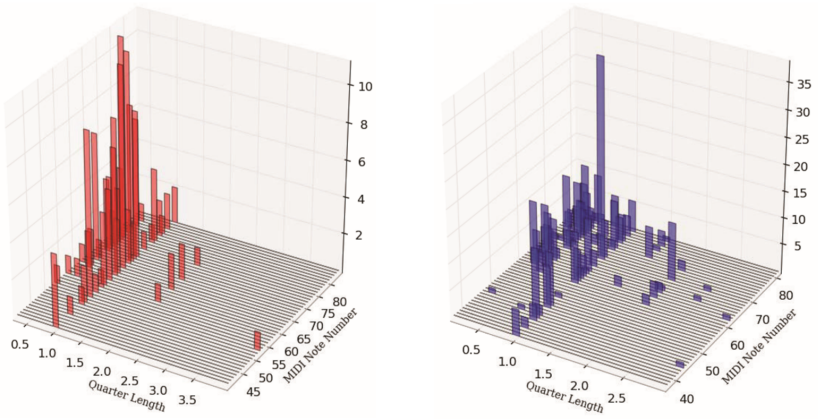


Figura 2.40: Ejemplo de código en music21 y su resultado (Cuthbert y Ariza, 2010, pág. 640).

Carpentier y Bresson (2010) presenta *Orchidée*, un sistema de orquestación asistida por ordenador programada sobre *OpenMusic* y *Matlab*. El acercamiento teórico a la tarea de la orquestación es como un problema de optimización multicriterio en el cual se define un punto ideal correspondiente a una configuración que optimiza todos los criterios. Aunque muchas veces dicho punto no existe, se pueden encontrar diferentes soluciones de compromiso denominadas soluciones de *Pareto*. En función de los objetivos del compositor, algunas regiones del conjunto de soluciones de Pareto son más interesantes que otras de acuerdo a las preferencias personales y específicas del compositor. El software ha sido probado por el compositor Jonathan Harvey en su obra *Speakings* con notable éxito, incorporando directamente en la partitura orquestal algunas secciones orquestadas por Orchidée, prácticamente sin revisión humana.

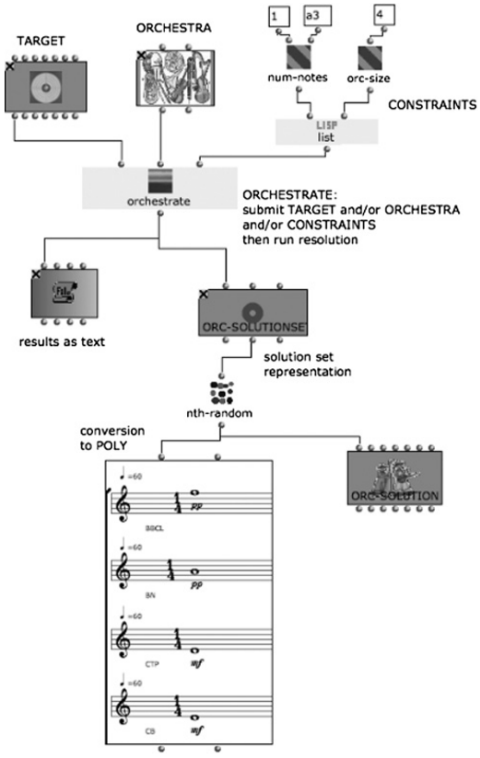


Figura 2.41: Ejemplo ejecución de Orchidée desde OpenMusic (Carpentier y Bresson, 2010, pág. 22).

Magnusson (2010) presenta *ixi lang*, un lenguaje de programación diseñado como una extensión de *SuperCollider* para el *Live coding*[19]. Ixi lang resuelve muchos problemas relacionados con la práctica del Live coding, haciendo el proceso más rápido, más comprensible y menos laborioso, acercando esta nueva disciplina a artistas que no necesariamente han de ser expertos programadores.

```
scale minor
tuning wcHarm

jarret    ->    piano[7   1   5   3   ]
jar2      ->    piano[  2   3   5     ]-12!16

jimi      ->   string[11   4 3233     ]!32
jimi >> distort

grid 4    |   |   |   |   |   |   |   |   |   |
ali       ->        |o   x   o   x   |
hat       ->        |  i i i i i i   |

ambient   ->    wind{0   1   3  8    0}!24
ambient >> reverb >> lowpass

future 4:4 >> swap jimi
>shift jarret 2
group drummer -> ali hat
doze drummer
future 12:1 >> perk drummer
```

Figura 2.42: Captura de pantalla de una sesión de ejemplo en ixi lang (Magnusson, 2010, pág. 2).

Van Der Merwe y Schulze (2011) en su artículo *Music generation with Markov models* propone el sistema *SuperWillow*, una modificación del sistema anterior para la composición musical asistida por ordenador *Willow*. El sistema utiliza el formato de representación simbólica MusicXML y se basa en las cadenas y modelos ocultos de Markov. El sistema primero analiza información relevante de archivos en XML como por ejemplo el tempo, las escalas, el compás, progresiones de acordes, progresiones rítmicas para posteriormente elaborar cadenas de Markov de diferente orden. Tras esta fase de análisis, el sistema comienza la fase de generación de música en la que se producirá una imitación de los datos introducidos en la fase de entrenamiento, junto con un número de restricciones impuestas, como por ejemplo que el compás ha de permanecer contante en toda la composición, que las notas simultáneas suenan como acorde no como arpeggio, que los instrumentos deben sonar sin silencios, que las composiciones deben finalizar con

[19] El Live coding consiste en la generación de música en tiempo real, mediante la introducción de instrucciones en directo mientras ésta suena, a la vez que se proyecta sobre una pantalla el contenido del computador del artista para que el público pueda observarlo. Para más información véase Nilson (2007).

cadencias, que las notas no deben exceder de un compás a otro o que no existan grupos rítmicos irregulares. Los resultados compositivos generados fueron sometidos a una encuesta en la que fueron comparados con composiciones generadas por un humano. En dicha encuesta participaron un total de 263 voluntarios y si bien los resultados mostraron que las composiciones humanas continuaban siendo preferidas, un 72 % de los encuestados no se consideraba capaz de distinguir entre las composiciones humanas y las generadas por SuperWillow.

Sandred (2010) presenta *PatchWork Musical Constraint system (PWMC)* un sistema para la composición musical asistida por ordenador en la que la generación de la partitura se entiende como un problema de resolución de restricciones musicales. El sistema se encuentra implementado en OpenMusic como una extensión del lenguaje *PWGL* (*PatchWork OpenGL*) y tiene el principal objetivo de solucionar el problema de la satisfacción de restricciones aplicadas a estructuras musicales polifónicas donde las alturas y los ritmos están indeterminados. Sandred utiliza PWMC para la composición de su obra *Labyrinths in the Wind*.

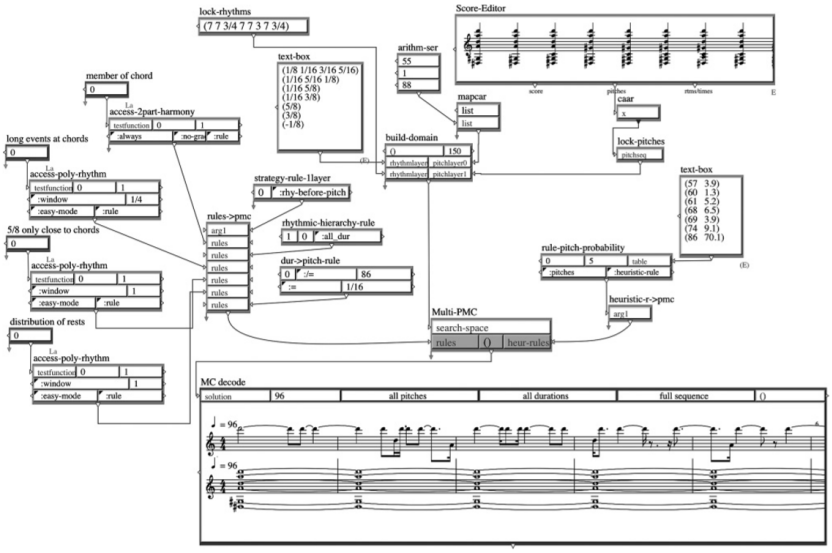


Figura 2.43: Patch para el comienzo de la composición *Labyrinths in the Wind* (Sandred, 2010, pág. 21).

En su artículo *Generating and evaluating musical harmonizations that emulate style* los autores Chuan y Chew (2011) presentan una solución híbrida al problema de la emulación de estilos aplicada a los procesos armónicos en la cual

se combinarán métodos de aprendizaje estadístico, como por ejemplo cadenas de Markov, con un armazón de reglas teóricas y musicales. El sistema es capaz de analizar la información de entrada y extraer variables capaces de caracterizar diversos estilos musicales. El sistema requiere como datos de entrenamiento líneas melódicas en formato MIDI con los acordes correspondientes a cada compás etiquetados en formato de texto y, mediante árboles de decisión, el sistema aprenderá las relaciones melódico-armónicas existentes en ese estilo. Previamente a esta fase de análisis se debe preprocesar los datos, sometiéndolos a una normalización de *pitch class* y a la determinación de la tonalidad de cada línea melódica (Martínez, 2019).

Los autores Honingh y Bod (2011) presentan una novedosa propuesta para la clasificación automática de géneros musicales a través de procesos de *clustering*. En su artículo proponen un experimento en el que se van a clasificar distintas obras pertenecientes a compositores de diferentes períodos en función del número de ocurrencias de cada una de las seis categorías interválicas posibles. Se realizará un análisis y conteo de pitch class y de esta manera cada obra podrá ser representada en un espacio de seis dimensiones. Mediante técnicas típicas de *clustering* y tras una fase de entrenamiento, el sistema será capaz de distinguir entre determinados estilos musicales.

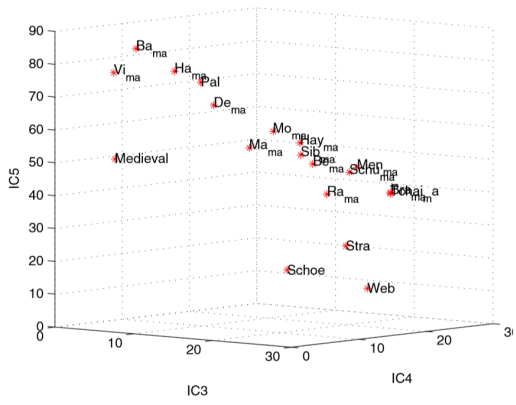


Figura 2.44: Ejemplo de agrupación de diferentes autores para las categorías interválicas 3, 4 y 5 (Honingh y Bod, 2011, pág. 348).

Agostini (2012) presentan *Bach: An environment for computer-aided composition in max*, una librería para Max que le proporciona funcionalidades para la notación musical simbólica y para el procesado de música en tiempo real. La librería gráfica

de *Bach* permite editar notas con el ratón, con el teclado y mediante mensajes de Max; soporta alteraciones microtonales; permite asociar *metadatos* a cada una de las notas y por último posee la funcionalidad de reproducción de la música a velocidad variable.

Collins (2012) presenta *Autocousmatic*, un sistema algorítmico implementado en SuperCollider capaz de generar música electroacústica a través de procesos de *machine-learning* y un corpus de *training*, consistente en una amplia base de datos de obras. El resultado musical generado por el sistema ha sido sometido a la crítica de tres compositores experimentados en el ámbito de la composición electroacústica, siendo muy críticos con el mismo y poniendo de manifiesto sus deficiencias compositivas.

En su artículo *ManuScore: Music Notation-Based Computer Assisted Composition*, los autores Maxwell, Eigenfeldt y Pasquier (2012) presentan *ManuScore*, una aplicación para la composición musical interactiva basada en la notación musical e inspirada en sistemas de aprendizaje y generación. Las capacidades generativas del sistema están focalizadas en la continuación de un fragmento musical proporcionado por el usuario, que sea capaz de escribir nueva música de una manera similar a como lo estaba haciendo el humano.

Los autores Cherla, Purwins y Marchini (2013) proponen un enfoque similar mediante un software capaz de generar variaciones de una melodía monofónica proporcionada, mediante la utilización de técnicas de aprendizaje no supervisado, utilizando cadenas y modelos ocultos de Markov. Los resultados obtenidos han sido sometidos a la revisión por parte de un panel de expertos, a los que se les ha sometido a una cuestionario de evaluación, obteniéndose un resultado favorable ya que, en general, los expertos encontraron musicalmente interesantes las variaciones generadas que poseían un ritmo regular y variaciones de altura con respecto a las notas de las melodías originales (Martínez, 2019).

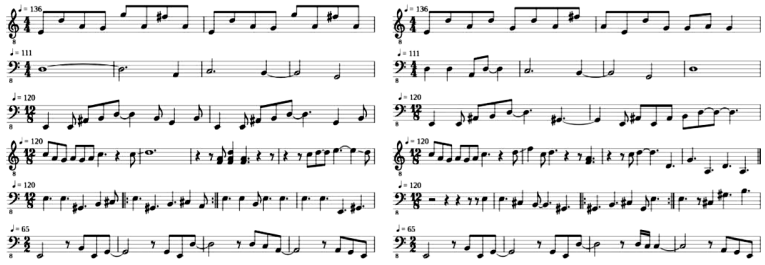


Figura 2.45: Comparación entre melodías iniciales (derecha) y sus variaciones (izquierda) (Cherla, Purwins y Marchini, 2013, pág. 76).

Deal y Sanchez (2013) propone su obra *Goldstream Variations*, compuesta gracias a la librería para Max denominada *ML Toolkit* (Smith y Garnett, 2012) que incorpora técnicas de *machine learning* en la composición asistida por ordenador con las que el compositor pretender expandir al máximo la capacidad de ML para generar variedad sonora.

Neuman (2013) en si artículo *Generative Grammars for Interactive Composition Based on Schaeffer TARTYP* propone un sistema capaz de generar complejas mediante la utilización de gramáticas de estructura de frase, propuestas por Chomsky (1957), y derivadas de la *Tableau Récapitulatif de la Typologie* propuesta por Schaeffer (1966). El algoritmo ha sido implementado como cuatro clases de Java que han sido embebidas como extensiones de las clases de Max/MSP.

Los autores Quick y Hudak (2013) proponen una nueva categoría de gramática para la generación de estructuras musicales armónicas y métricas: las gramáticas generativas temporales con grafos (*TGGG*). Este tipo de gramáticas poseen dos características distintas, en primer lugar su dimensión temporal en la que las reglas de producción están parametrizadas por la duración de las frases; en segundo lugar por el hecho de que sus árboles de análisis están representados en forma de grafos que pueden compartir nodos, incorporando de esta forma la repetición musical en la propia gramática. La implementación de estas gramáticas se ha realizado en el lenguaje *Haskell*. Los resultados se muestran prometedores considerando la relativa sencillez de la gramática utilizada como experimento (Martínez, 2019).

Los autores Long, Wong y Sze (2013) presentan en su artículo *T-Music: A melody composer based on frequent pattern mining* un software capaz de componer melodías a partir de un texto proporcionado por el usuario, encontrando las correlaciones existentes entre texto y música. Estas correlaciones se representan en los denominados *frecuent patterns (fp)*. El sistema se compone de dos fases: en la fase inicial se buscan estas correlaciones gracias una base de datos generada a partir de numerosas canciones. A continuación y basándose en las correlaciones encontradas, el sistema compone una melodía para la letra proporcionada basándose en un autómata probabilístico.

Es destacable la aportación a la creatividad musical artificial realizada por los autores Sánchez y col. (2013) en el proyecto *Melomics*, en la cual un ordenador es capaz de generar obras completas, utilizando para ello sofisticados algoritmos genéticos. Algunas de las composiciones más destacables generadas fueron interpretadas en público y sometidas a una evaluación por parte de melómanos experimentados, con óptimos resultados [20].

[20] Se pueden escuchar fragmentos de estas composiciones en la siguiente página web `http://www.melomicsrecords.com/` .

Carretero presenta su tesis *El proceso de composición musical a través las técnicas bio-inspiradas de inteligencia artificial: investigación desde la creación musical* (2013) en la que se investiga sobre la aplicación de los autómatas celulares y los P-sistemas en la composición musical asistida por ordenador, proporcionando determinadas herramientas computacionales que pueden facilitar la labor creativa a otros compositores.

McVicar, Fukayama y Goto (2014) desarrollan *AutoRhythmGuitar*, un software capaz de generar ritmos de guitarra en un espacio de tablaturas. El programa toma como parámetros de entrada una secuencia de acordes y genera una partitura tradicional en formato MusicXML. Funciona mediante el modelo de los *n-gramas*, requiriendo una fase de entrenamiento con tablaturas procedentes de cinco guitarristas famosos en formato *GuitarPro*. La evaluación final del sistema muestra como los modelos generan partituras realistas y factibles (Martínez, 2019).

Los autores Klügel (2014) presentan *FugueGenerator*, un software orientado a la composición musical asistida por ordenador de forma colaborativa, en la que los usuarios manipulan una superficie de control bidimensional con el que se establece el contorno de determinados eventos musicales que controla el funcionamiento de determinados procesos generativos estocásticos. Con esta superficie de control y basándose en el la propuesta anterior de Wallis, Ingalls y Campana (2008) los autores categorizan determinadas emociones en la música generada en función de su valencia y su agitación.

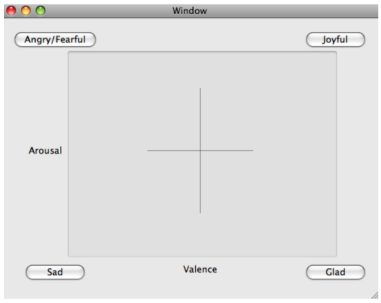


Figura 2.46: Interfaz de usuario para producir música respecto la valencia y agitación (Wallis, Ingalls y Campana, 2008, pág. 2).

Los autores Nuanáin, Herrera y Jorda (2015) presentan en su artículo *Target-based rhythmic pattern generation and variation with genetic algorithms* un sistema basado en algoritmos genéticos que, dado un patrón rítmico, automáticamente genera patrones relacionados a modo de variaciones. Se utilizan dos medias de

la distancia, la distancia *Hamming* y la distancia *directed-swap* para medir la similitud rítmica utilizable en las funciones de *fitness* del algoritmo genético. Los autores implementan este modelo en Max/MSP y Pure Data bajo el nombre de *SimpleGA* para a continuación proporcionarle la interfaz *GenDrum* capaz de crear patrones rítmicos polifónicos con sonidos sintetizados.

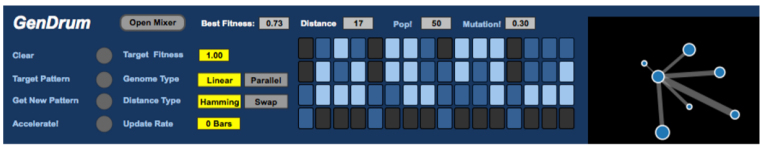


Figura 2.47: Interfaz de GenDrum (Nuanáin, Herrera y Jorda, 2015, pág. 61).

Los autores Loughran, McDermott y Neil (2015) proponen la composición de melodías polifónicas para piano a través de algoritmos evolutivos gramaticales implementados con funciones de fitness basadas en la Ley de *Zipf* [21]. Los autores realizan experimentos con una población de 200 durante 50 generaciones, con un coeficiente de mutación de 0,01, un coeficiente de cruce de 0,7 y un elitismo de tamaño uno.

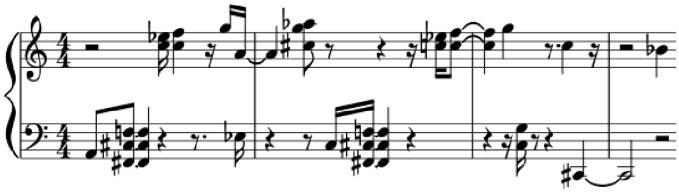


Figura 2.48: Ejemplo de melodía generada (Loughran, McDermott y Neil, 2015, pág. 278).

Los autores Agustín-Aquino, Junod y Mazzola en el libro *Computational Counterpoint Worlds. Mathematical Theory, Software, and Experiments* (2015) ofrecen un amplio catálogo de técnicas en las que se implementan nuevos sistemas de contrapunto, definidos mediante aparatos algebraicos, capaces de establecer sistemas de composición totalmente nuevos. Los autores muestran diversos resultados teóricos obtenidos, extendiendo las reglas del contrapunto a la música microtonal así como una implementación informática de estos modelos.

[21] La Ley de Zipf ha sido formulada de forma empírica en los años cuarenta por el lingüista George Kingsley Zipf. Según esta ley la mayoría de las lenguas cumple que la frecuencia de aparición de las distintas palabras sigue una distribución que puede aproximarse por $P_n \propto 1/n^a$ (Montemurro, 2001).

Manaris, Stevens y Brown (2016) presentan *JythonMusic* un entorno de código libre desarrollado en *Python* orientado a la creación de música y a la programación creativa. Soporta la composición algorítmica, la programación dinámica y el *live coding*, además de ofrecer funcionalidades de conexión con muchos dispositivos, pianos digitales, *smartphones* y *tablets* mediante el protocolo *OSC* y el protocolo *MIDI*.

Sly, Agarwala e Inoue (2017) proponen la utilización de redes neuronales recurrentes (*RNN*) para la composición asistida por ordenador de música con forma bien definida y que incorpore convenciones estilísticas. Sin ninguna base teórica predefinida, los modelos generan música sintácticamente correcta capaz de engañar el oído humano.

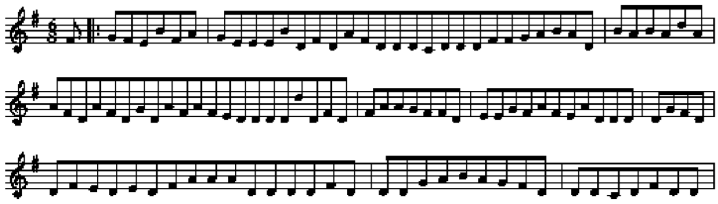


Figura 2.49: Ejemplo de música generada por el modelo basado en RNN (Sly, Agarwala e Inoue, 2017, pág. 7).

Antoine y Miranda (2017) presentan en su artículo *Computer Generated Orchestration: Towards Using Musical Timbre in Composition* un sistema para la orquestación generada por ordenador basado en técnicas de *machine learning* tales como el algoritmo *Support vector Machine (SVM)*. El corpus de entrenamiento se construye con 250 ejemplos etiquetados para cada atributo del clasificador que funcionará como método automático de clasificación del timbre.

Los autores Yu y Wong (2017) basándose en el software *T-Music*, proponen en su artículo *A Melody Composer for Both Tonal and Non-Tonal Languages* dos nuevos métodos para la extracción de estos patrones desde composiciones instrumentales preexistentes y los aplican tanto a lenguajes tonales como a los no tonales.

Louzeiro (2017) presenta un sistema en red de tipo cliente-servidor para la composición musical en tiempo real mediante el cual un director media la interacción entre un solista y un ensemble de instrumentistas que están leyendo a primera vista una partitura generada automáticamente.

Navarro (2017) presenta en su tesis *Sociedades Humano-Agente: Un Caso de Estudio en Creatividad Musical* un framework basado en el modelo de sociedad

humano-agente, mediante el cual se permite el intercambio de información entre un grupo de humanos y una sociedad de agentes virtuales con el objetivo de generar música a partir de una entrada proporcionada por el usuario, que puede consistir en una secuencia armónica tonal o a partir de los colores de una imagen.

En el artículo *Combinatorics, Probability and Choice in Music Composition: Towards an Aesthetics of Composing Systems for Non-Musicians*, Albini (2018) presenta la obra *Ricercari Diatonici, for people, composing system and harpsichordist*, en la que mediante la descarga de una aplicación móvil el usuario es capaz de generar aleatoriamente un grandísimo número de obras. Además, éste puede escuchar la composición generada y si cumple con sus criterios estéticos personales, seleccionarla y enviarla de vuelta al compositor. De esta manera el usuario actúa como un filtro estético para la música generada aleatoriamente. Las obras seleccionadas por un gran número de usuarios fueron posteriormente interpretadas en público.

Krzyzaniak (2018) presenta una técnica de aprendizaje interactiva para un sistema de percusión robótico, basado en técnicas de *machine learning* capaz de analizar y producir patrones estadísticos a través de la observación en tiempo real de intérpretes humanos.

En el artículo *Generating new musical works in the style of Milton Babbitt*, los autores Bemman y Meredith (2018) proponen un sistema para automatizar los procedimientos compositivos del compositor estadounidense Milton Babbit y generar obras completamente nuevas coherentes con el estilo compositivo del autor. Los procesos y técnicas compositivas estudiados son: *all-partition array*, *time-point system* y *equal-note-value strings* (Bemman y Meredith, 2016).

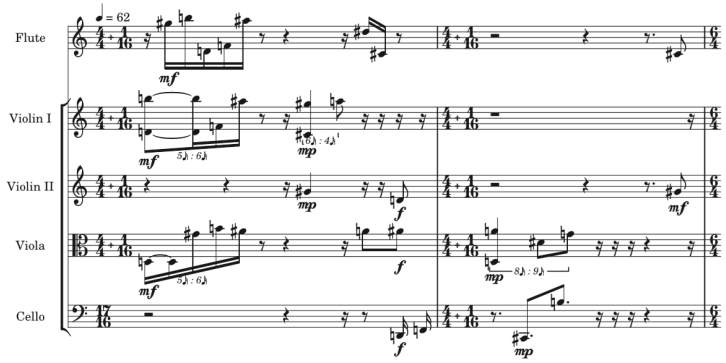


Figura 2.50: Primeros compases de la obra generada emulando el estilo de Milton Babbit (Bemman y Meredith, 2018, pág. 73).

2.8 Resumen

La evolución de la composición musical asistida por ordenador se remonta a los inicios de la computación moderna y avanza hasta nuestros días en paralelo con el desarrollo de los ordenadores y su vertiginoso aumento en la capacidad de cálculo. Desde la primera presentación pública de la *Suite Illiac*, tradicionalmente considerada como la primera obra compuesta utilizando la ayuda de un computador, han aparecido numerosas propuestas que han tratado de resolver problemas compositivos muy específicos; otras sin embargo, han propuesto aplicaciones informáticas o métodos algorítmicos de carácter más general, aplicables a la vertiente creativa de la *computer music*. En todo caso, podemos afirmar que los acercamientos tradicionales al problema de la composición musical asistida por ordenador en su vertiente simbólica, es decir, creación mediante ordenador de música expresada en partituras mediante notación musical convencional para su posterior interpretación, se pueden resumir en las siguientes grandes líneas de investigación:

- *Implementación de reglas y restricciones compositivas.*
- *Procesos estocásticos y aleatoriedad controlada.*
- *Modelos generativos basados en cadenas de Markov.*
- *Sistemas caóticos, autosemejantes o fractales.*
- *Utilización de autómatas celulares.*
- *Algoritmos genéticos y bioinspirados.*
- *Utilización de sistemas basados en agentes.*
- *Utilización de gramáticas formales.*
- *Técnicas procedentes del* machine learning.

Lejos de ser una cuestión cerrada, el panorama actual de la composición musical asistida por ordenador se presenta como un amplio campo de investigación en el que existe una fuerte tendencia a la hibridación de muchos de los paradigmas explicados anteriormente, fusionando elementos propios de la ciencia y del arte e incorporando, al mismo tiempo, las posibilidades creativas existentes en la interacción entre humano y máquina.

Capítulo 3

Marco teórico: *clustering*, disimilitud y transiciones difusas

«Si la palabra "música" es sagrada y está reservada para los instrumentos del siglo dieciocho y diecinueve, podemos sustituirla por un término más significativo: organización del sonido.»

«If this word "music" is sacred and reserved for eighteenth- and nineteenth-century instruments, we can substitute a more meaningful term: organization of sound.»

(Cage, 1961a, pág. 5)

3.1 Introducción

Este capítulo constituye el marco conceptual básico de la presente tesis. Describiremos en primer lugar los fundamentos teóricos y algorítmicos del método de agrupamiento *k-means* y continuaremos con la versión difusa del mismo denominada *Fuzzy c-Means* (FCM). Llegados a este punto presentaremos dos nuevos algoritmos: el primero es el denominado *Fuzzy Ordered c-Means* (FOCM), con el que conseguiremos realizar una partición de tipo *fuzzy* de un conjunto de datos ordenados tomando en consideración el orden de sus elementos. El segundo algoritmo propuesto es el *Fuzzy Complete Transitions* (FCT), gracias al cual po-

dremos generar transiciones completas entre dos secuencias con distinto número de elementos constituyentes.

3.2 Los métodos de agrupamiento

Los métodos de agrupamiento (*clustering*), también denominados análisis de *cluster*, análisis de segmentación o análisis taxonómico, son métodos que consisten en crear grupos de elementos (*clusters* o racimos) dentro de un conjunto de datos inicial de manera que los elementos comprendidos en cada grupo puedan ser considerados como similares entre sí, y distintos con respecto a los elementos de otro grupo. A diferencia de los métodos de clasificación, en los cuales a los elementos se les asigna con una clase preexistente, en los métodos de clustering las diferentes clases o subgrupos en los que se va a dividir el conjunto de datos han de ser definidas con anterioridad a la ejecución de la fase de análisis. El procedimiento de clustering consistirá en encontrar una partición de un conjunto de datos $\mathbf{X}$ que satisfaga unos determinados criterios de agrupamiento.

Estas técnicas juegan un papel fundamental en los distintos paradigmas sobre los que se basa la inteligencia artificial, la minería de datos o el reconocimiento automatizado de patrones. En Gan, Ma y Wu (2007) podemos encontrar algunos interesantes ejemplos de la utilización de los procesos de clustering en diversos ámbitos científicos: en el campo de la biotecnología, la utilización del análisis de clustering permite determinar la expresión de determinados genotipos (Yeung, Medvedovic y Bumgarner, 2003); en el ámbito de la medicina, el análisis de cluster se utiliza frecuentemente para el mantenimiento y mejora del sistema sanitario de salud, así como para la prevención y detección de enfermedades (Clatworthy y col., 2005); en el sector de la economía, las técnicas de *clustering* se pueden utilizar para determinar segmentos de mercado e identificar posibles objetivos comerciales (Saunders, 1980); en el reconocimiento de patrones y en el procesamiento de automático de imágenes, los algoritmos de clustering funcionan de forma muy eficiente para encontrar los bordes de los objetos o para comparar los histogramas de colores de distintas imágenes (Comaniciu y Meer, 1999).

Según Buhmann (1995), la resolución de un problema mediante un algoritmo de clustering tiene que seguir las siguientes cuatro fases: representación de datos, modelado, optimización y validación. La fase inicial de representación de datos determina las estructuras que serán descubiertas en los agrupamientos de los datos. La fase de modelado define el concepto de clusters y el criterio mediante el cual se diferenciarán los diferentes grupos. La fase de optimización mejora las

prestaciones de eficiencia y eficacia del algoritmo, validando sus resultados en una última fase mediante conjuntos de datos de test.

Generalmente, los algoritmos de clustering pueden ser divididos en las dos grandes categorías: *hard* clustering (también llamado *crisp clustering*) y *soft* clustering (también llamado *fuzzy clustering*). La diferencia fundamental es que, en el *hard* clustering, un elemento del conjunto de datos pertenece únicamente a un cluster, mientras que en el *fuzzy* clustering un elemento puede pertenecer simultáneamente a varios clusters, quedando establecidos en el proceso de partición los coeficientes de pertenencia de cada elemento a cada uno de los diferentes clusters.

Los algoritmos convencionales de *hard* clustering pueden ser clasificados a su vez en dos grandes categorías: los algoritmos jerárquicos y los algoritmos *particionales*. Existen a su vez dos tipos de algoritmos jerárquicos: los algoritmos jerárquicos divisivos y los algoritmos jerárquicos aglomerativos. En los primeros, el algoritmo procede de arriba a abajo, empezando por el cluster más grande (un cluster que contiene todos los elementos) y efectuando progresivas divisiones sobre éste. En los de tipo aglomerativo, el algoritmo funciona al revés, definiendo primero los clusters más pequeños posibles (de un único elemento) para ir fusionándolos progresivamente en clusters más grandes. Para conjuntos de datos muy grandes, los métodos jerárquicos son poco prácticos ya que su comportamiento es $O(n^2)$ en cuanto a necesidades de espacio de memoria y $O(n^3)$ en cuanto a tiempo de computación (donde n es el número de puntos del conjunto de datos). Al contrario que los algoritmos jerárquicos, los algoritmos partitivos crean una superposición de distintas particiones de elementos (Gan, Ma y Wu, 2007).

A continuación será necesario definir algunos conceptos preliminares. Seguiremos el criterio explicado por Jain y Dubes (1988); dado un conjunto de datos, denominaremos *elemento* (también *tupla*, *punto* o *individuo*) a cada unidad mínima de información perteneciente a este conjunto de datos. Cada elemento llevará asociado un total de q magnitudes escalares denominadas *características* (también *variables*, *observables*, o *atributos*). El término *cluster* (o *grupo*, o *clase*) se utilizará para designar a cada una de las c agrupaciones realizadas del conjunto de datos. Se entiende que los elementos que pertenecen a un determinado cluster comparten propiedades o características entre sí y son diferenciables de los elementos pertenecientes a otro cluster. En los procesos de *fuzzy* clustering esta distinción ya no está tan clara. El termino *centroide* denomina el punto central de cada uno de los clusters. El conjunto de n datos se denotará como

$$\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^q \tag{3.1}$$

donde cada $\mathbf{x}_i = \{x_{i1}, x_{i2}, \ldots, x_{iq}\}$, con $\mathbf{x}_i \in \mathbb{R}^q$, será un punto de q características perteneciente a un espacio métrico q-dimensional $\mathbb{R}^q$. El índice i designará el i-ésimo elemento $\mathbf{x}_i$; el escalar x_{ik} designará el valor de la k-ésima característica de $\mathbf{x}_i$. El número de atributos q se conoce como la *dimensionalidad* del conjunto de datos X, y tendrá que ser un número finito y entero mayor que cero. En dicho espacio existe una función distancia que ha de cumplir con las características expuestas en la sección 3.4.

3.3 Particiones *hard* y particiones *soft*

En el proceso de *hard* clustering los distintos algoritmos asignan a cada elemento del conjunto de datos uno de los grupos o clases preexistentes, asumiendo que un elemento puede pertenecer únicamente a cluster, de tal manera que el resultado de este tipo de procesos puede ser expresado mediante una matriz $\mathbf{U}$ de $n \times c$ elementos, donde cada elemento toma los valores cero o uno, cumpliéndose unas determinadas condiciones que garantizan que cada elemento puede pertenecer solo a un cluster y que no pueden existir clusters vacíos. En el *fuzzy* clustering se produce un cambio radial; se asume que un elemento puede pertenecer a varios clusters al mismo tiempo, estableciéndose unos coeficientes de pertenencia que expresan el grado relación de cada elemento a cada uno de los clusters. De esta forma, los elementos de la matriz pueden tomar cualquier valor real entre cero y uno siempre que sigan cumpliendo con las condiciones requeridas (al contrario que en el caso hard en el que los coeficientes admiten solo dos valores). Los resultados nuevamente pueden ser expresados en términos de una matriz $\mathbf{U}$, aunque esta matriz ahora ya no será tan dispersa, ya que existirán muchos menos elementos iguales a cero. La condición que ha de regir este particionado es la siguiente: la suma de las probabilidades de pertenencia de un elemento a todos los grupos ha de ser igual a uno, es decir, la suma de todos los elementos de cada fila de la matriz ha de ser igual a la unidad (Gan, Ma y Wu, 2007).

En Bezdek (1981) y Bezdek y col. (1999) encontramos el fundamento teórico necesario para definir los distintos tipos de particionado de un conjunto de datos. Supongamos que $\mathbf{X}$ es un conjunto finito de n elementos tal que $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$ y queremos repartir los elementos del conjunto $\mathbf{X}$ en un número c de subconjuntos $\mathbf{C} = \{\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_c\}$ con $2 \leq c \leq n$. Esta familia de subconjuntos $\{\mathbf{C}_j : 1 \leq j \leq c\} \subset \mathbf{X}$ será una partición de tipo *hard*[1] si se cumplen las siguientes condiciones:

[1]La terminología *hard* y *fuzzy* se utilizará respectivamente para distinguir a procesos en los que no interviene la lógica fuzzy (*hard*), de los que sí que requieren de lógica *fuzzy* (*soft*).

$$\bigcup_{j=1}^{c} \mathbf{C}_j = \mathbf{X} \tag{3.2}$$

$$\mathbf{C}_j \cap \mathbf{C}_k = \varnothing, \quad 1 \leq j \neq k \leq c \tag{3.3}$$

es decir, si la unión de todos los subconjuntos es igual a conjunto inicial, pero la intersección de cada par de subconjuntos distintos es nula, por lo que cada elemento $\mathbf{x}_i$ puede pertenecer únicamente a un subconjunto $\mathbf{C}_j$.

Podemos definir la matriz $\mathbf{U} = [u_{ij}]$ que representará los coeficientes de pertenencia de cada elemento $\mathbf{x}_i$ a cada subconjunto $\mathbf{C}_j$. Es fácil demostrar que la matriz U representará una partición de tipo *hard* de $\mathbf{X}$ si y sólo si se cumplen las siguientes tres condiciones:

$$u_{ij} \in \{0, 1\}, \quad 1 \leq j \leq c, \quad 1 \leq i \leq n \tag{3.4}$$

$$\sum_{j=1}^{c} u_{ij} = 1, \quad 1 \leq i \leq n \tag{3.5}$$

$$0 < \sum_{i=1}^{n} u_{ij} < n, \quad 1 \leq j \leq c \tag{3.6}$$

la primera condición significa exige que los coeficientes u_{ij} sólo podrán tomar los valores 1 ó 0. La segunda implica que cada elemento $\mathbf{x}_i$ pertenece a uno y sólo uno de los c subconjuntos. La tercera condición garantiza que no existe ningún subconjunto vacío y que no hay ningún subconjunto igual a todo el conjunto $\mathbf{X}$ (por tanto el número de subconjuntos es mayor o igual a dos). Resulta equivalente denominar *c-partición hard* de $\mathbf{X}$ tanto a $\{\mathbf{V}_j\}$, como a u_j o a $\mathbf{U}$.

Definición 3.3.1 c-Partición hard*, (Bezdek, 1981) . Sea* $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ *un conjunto finito de n elementos donde cada* $\boldsymbol{x}_i \in \mathbb{R}^q$ *es un observable definido por q características; sea* $\boldsymbol{C}_{cn}$ *el conjunto de matrices reales de dimensión* $c \times n$*; sea c un entero,* $2 \leq c < n$*. El espacio de particiones* hard *para* $\boldsymbol{X}$ *es el conjunto*

$$\mathbf{M}_c = \left\{ \mathbf{U} \in \mathbf{C}_{cn} \middle| u_{ij} \in \{0, 1\} \forall i, j; \sum_{j=1}^{c} u_{ij} = 1 \forall i; 0 < \sum_{i=1}^{n} u_{ij} < n \right\} \tag{3.7}$$

El número de posibles particiones *hard* es finito pero crece muy rápidamente; de hecho Duda, Hart y Stork (1973) proponen la siguiente fórmula para calcular las distintas particiones *hard* posibles de un conjunto $\mathbf{X}$ en c subconjuntos no vacíos,

dados unos valores c y n cualesquiera que no sean los triviales:

$$|\mathbf{M}_c| = \frac{1}{c!}\left[\sum_{j=1}^{c}\binom{c}{j}(-1)^{c-j}j^n\right] \tag{3.8}$$

Por ejemplo, si $c = 10$ y $n = 25$, hay 10^{18} posibles 10-particiones *hard* distintas de los 25 puntos del conjunto de datos, resultando muy complicado encontrar la óptima mediante búsqueda exhaustiva.

Ejemplo 3.3.1 *Supongamos que tenemos el siguiente conjunto de frutas*

$$\boldsymbol{X} = \{x_1 = \text{piña}, x_2 = \text{naranja}, x_3 = \text{plátano}\} \tag{3.9}$$

sobre el que queremos realizar una partición hard *para clasificarlas en dulces o ácidas.*
El número total de posibles 2-particiones *hard* (*dulces* o *ácidas*) de los tres datos ($n = 3$) sería

$$|\mathbf{M}_2| = \frac{1}{2!}\left[\sum_{j=1}^{2}\binom{2}{j}(-1)^{2-j}j^3\right] = \frac{1}{2}(-2+8) = 3 \tag{3.10}$$

Y por tanto las posibles particiones de $\mathbf{X}$ serían sólo estas tres matrices:

$$\mathbf{U}_1 = \begin{bmatrix}0 & 1\\ 0 & 1\\ 1 & 0\end{bmatrix} \quad \mathbf{U}_2 = \begin{bmatrix}0 & 1\\ 1 & 0\\ 1 & 0\end{bmatrix} \quad \mathbf{U}_3 = \begin{bmatrix}1 & 0\\ 0 & 1\\ 1 & 0\end{bmatrix} \tag{3.11}$$

En las que hemos eliminado todas aquellas que no cumplen las condiciones 3.4, 3.5 y 3.5, además de aquellas que son equivalentes gracias a una simple reordenación de columnas. Sin embargo, la partición óptima, y por tanto adecuada a las propiedades observables de nuestros datos (las tres frutas en este caso) sería la siguiente:

$$\mathbf{U} = \begin{bmatrix}0 & 1\\ 0 & 1\\ 1 & 0\end{bmatrix} \tag{3.12}$$

Donde las filas representarían a cada una de las frutas (piña, naranja, plátano) y las columnas indicarían si es dulce o ácida. Observamos como cada fruta ha sido asignada con un coeficiente 0 ó 1 para establecer la partición a la que pertenece. Continuemos con este ejemplo para ilustrar como nuestra realidad física supera el modelo matemático que acabamos de definir: supongamos que queremos clasificar otras frutas como por ejemplo la manzana, el melocotón, o la ciruela, que pueden

ser al mismo tiempo tanto *dulces* como *ácidas*. ¿Cómo afrontaría este reto nuestro método de particionado *hard*? Clasificará cada una de estas ambiguas frutas con la propiedad que estuviese presente en mayor grado; de esta forma si un melocotón es más *dulce* que *ácido*, será clasificado como *dulce* obviando toda la información relativa a su grado de acidez. Si queremos modelar matemáticamente la complejidad presente en esta situación necesitamos un tipo de particionado que sea capaz de incluir toda una gama de graduaciones entre las clases de particiones. Ruspini (1970) proporciona una definición que soluciona este problema: los coeficientes de pertenencia u_{ij} ahora podrán ser cualquier valor real comprendido en el intervalo $[0, 1]$. Hablaremos en este caso de una partición de tipo *soft*, también denominada partición *fuzzy*.

Definición 3.3.2 c-Partición fuzzy*, (Bezdek, 1981). Sea* $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ *un conjunto finito de n elementos donde cada* $\boldsymbol{x}_i \in \mathbb{R}^q$ *es un observable definido por q características; sea* $\boldsymbol{C}_{cn}$ *el conjunto de matrices reales de dimensión* $c \times n$*; sea c un entero,* $2 \leq c < n$*. El espacio de particiones* fuzzy *para* $\boldsymbol{X}$ *es el conjunto*

$$\mathbf{M}_{fc} = \left\{ \mathbf{U} \in \mathbf{C}_{cn} \middle| u_{ij} \in [0,1] \forall i, j; \sum_{j=1}^{c} u_{ij} = 1 \forall i; 0 < \sum_{i=1}^{n} u_{ij} < n \right\} \tag{3.13}$$

Observamos que la suma de cada u_{ij} para cada fila de $\mathbf{U}$ sigue valiendo 1, pero como los valores permitidos para los coeficientes de pertenencia son ahora $0 \leq u_{ij} \leq 1, \forall i, j$ es posible que para $\mathbf{x}_i$ se produzca una distribución arbitraria de los valores de sus coeficientes de pertenencia a lo largo de los c subconjuntos *fuzzy* que forman la partición de $\mathbf{X}$. Podría darse el caso en el que en una fila, todos los coeficientes valgan 0 excepto uno que valga 1 ya que la partición *hard* M_c es un subconjunto de la partición *fuzzy* $\mathbf{M}_{fc}$, por lo que podemos considerar la partición de tipo *fuzzy* como una generalización del proceso de particionado *hard*.

Ejemplo 3.3.2 *Supongamos que tenemos el siguiente conjunto de frutas*

$$\mathbf{X} = \{x_1 = \text{piña}, x_2 = \text{naranja}, x_3 = \text{plátano}, x_4 = \text{manzana}, x_5 = \text{melocotón}\} \tag{3.14}$$

sobre el que queremos realizar una partición *fuzzy* para clasificarlas en *dulces* o *ácidas*. Un resultado posible de esta clasificación sería el siguiente:

$$\mathbf{U} = \begin{bmatrix} 0{,}07 & 0{,}93 \\ 0{,}14 & 0{,}86 \\ 0{,}97 & 0{,}03 \\ 0{,}35 & 0{,}65 \\ 0{,}68 & 0{,}32 \end{bmatrix} \tag{3.15}$$

Podemos observar intuitivamente como esta forma de realizar particiones *fuzzy* resulta más adecuada para trabajar con conjuntos de datos reales y modelar situaciones físicas en las que se describan interacciones reales entre los componentes de los datos. Existen infinitas posibles particiones *fuzzy* de un conjunto de datos $\mathbf{X}$. En los próximos epígrafes de este capítulo se describirán distintos algoritmos que son capaces de generar este tipo de particiones bajo determinados criterios de búsqueda.

3.4 Funciones distancia

En Mazón (2011) encontramos una definición del concepto de *distancia* y su relación con el espacio métrico:

Definición 3.4.1 Distancia. *Sea X un conjunto, una* distancia *en X es una función de $X \times X$ en $\mathbb{R}^+$ que a cada par de elementos $\boldsymbol{x}, \boldsymbol{y} \in X$ le asocia el número $d(\boldsymbol{x}, \boldsymbol{y})$, satisfaciendo las siguientes propiedades:*

(1) $d(\mathbf{x}, \mathbf{y}) = 0 \iff \mathbf{x} = \mathbf{y}$,

(2) $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$,

(3) $d(\mathbf{x}, \mathbf{y}) \leq d(\mathbf{x}, \mathbf{z}) + d(\mathbf{y}, \mathbf{z})$ (desigualdad triangular)

Al par (X, d) se le denomina *espacio métrico*. Si $\|\cdot\|$ es una norma en un espacio vectorial E, entonces la función:

$$d(\mathbf{x}, \mathbf{y}) \coloneqq \|\mathbf{x} - \mathbf{y}\| \tag{3.16}$$

es una distancia en E. Consecuentemente, cada espacio normado es un espacio métrico. En particular, el espacio euclídeo q-dimensional $\mathbb{R}^q$ es un espacio métrico con distancia dada por:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{q} (x_i - y_i)^2} \tag{3.17}$$

Esta distancia euclidiana es probablemente la distancia más utilizada y conocida; sin embargo existen muchas otras funciones distancia que siguen cumpliendo con las condiciones anteriormente descritas y que pueden ser utilizadas en las distintas técnicas de clustering. A continuación mostraremos algunas funciones distancia más utilizadas (Gan, Ma y Wu, 2007).

3.4.1 Distancia euclidiana

Como hemos visto anteriormente, para dos puntos $\mathbf{x}$ e $\mathbf{y}$ pertenecientes a un espacio métrico q-dimensional, la distancia euclidiana se define como

$$d_{\text{euc}}(\mathbf{x}, \mathbf{y}) = \left[\sum_{k=1}^{q} (x_k - y_k)^2 \right]^{\frac{1}{2}} \tag{3.18}$$

3.4.2 Distancia Manhattan

También denominada distancia *taxi driver*, ya que la distancia se define como la suma del valor absoluto de la diferencia de cada par de componentes, de forma análoga a como un taxista mediría la distancia en una ciudad formada por manzanas perfectamente cuadradas. Para dos puntos $\mathbf{x}$ e $\mathbf{y}$ pertenecientes a un espacio q-dimensional, la distancia Manhattan se define como

$$d_{\text{man}}(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{q} |x_k - y_k| \tag{3.19}$$

3.4.3 Distancia máxima o Chebyshev

Se define como el máximo valor de la distancia de los atributos, por tanto para dos puntos $\mathbf{x}$ e $\mathbf{y}$ pertenecientes a un espacio q-dimensional, la distancia máxima es

$$d_{\max}(\mathbf{x}, \mathbf{y}) = \max_{k=1}^{q} |x_k - y_k| \tag{3.20}$$

3.4.4 Distancia Minkowski

Para dos puntos $\mathbf{x}$ e $\mathbf{y}$ pertenecientes a un espacio q-dimensional, la distancia Minkowski es

$$d_{\min}(\mathbf{x}, \mathbf{y}) = \left[\sum_{k=1}^{q} |x_k - y_k|^r \right]^{\frac{1}{r}}, \quad r \geq 1 \tag{3.21}$$

El coeficiente r se denomina el *orden* de la distancia. La distancia Minkowski supone una generalización de las distancias euclidianas, Manhattan y máxima, ya que si tomamos $r = 2$ obtenemos la distancia euclidiana, si tomamos $r = 1$ obtenemos la distancia Manhattan, y en el límite $r \to \infty$, se puede demostrar que

$$\lim_{r \to \infty} \left[\sum_{k=1}^{q} |x_k - y_k|^r \right]^{\frac{1}{r}} = \max_{k=1}^{q} |x_k - y_k| \tag{3.22}$$

3.4.5 Distancia media euclidiana

Es una modificación de la distancia euclidiana en la cual dos puntos con ningún atributo en común no tendrán nunca una distancia más pequeña que otro par de puntos cualquiera que contengan los mismos valores de atributos. Para dos puntos $\mathbf{x}$ e $\mathbf{y}$ pertenecientes a un espacio q-dimensional, la distancia media se define como

$$d_{\text{euc}}(\mathbf{x}, \mathbf{y}) = \left[\frac{1}{q}\sum_{k=1}^{q}(x_k - y_k)^2\right]^{\frac{1}{2}} \tag{3.23}$$

3.4.6 Distancia Chord

La distancia Chord, propuesta por Kenkel y Orlóci (1986) es una modificación de la distancia euclidiana. Está definida por la distancia de la cuerda que une dos puntos normalizados unidos por una hiperesfera de radio uno. Se puede calcular directamente de datos no normalizados de la siguiente manera

$$d_{\text{chord}}(\mathbf{x}, \mathbf{y}) = \left[2 - 2\frac{\sum_{k=1}^{q} x_k y_k}{\|\mathbf{x}\|_2 \|\mathbf{y}\|_2}\right]^{\frac{1}{2}} \tag{3.24}$$

donde $\|\cdot\|_2$ es la norma L_2

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{k=1}^{q} x_k^2} \tag{3.25}$$

$$\|\mathbf{y}\|_2 = \sqrt{\sum_{k=1}^{q} y_k^2} \tag{3.26}$$

3.4.7 Distancia geodésica

Propuesta por Legendre y Legendre (1983). La distancia geodésica es una transformación de la distancia de Chord y se define como el arco de longitud mínima que conecta los dos puntos normalizados sobre la superficie de una hiperesfera de radio uno

$$d_{\text{geo}}(\mathbf{x}, \mathbf{y}) = \arccos\left(1 - \frac{d_{chord}(\mathbf{x}, \mathbf{y})}{2}\right) \tag{3.27}$$

3.4.8 Distancia Mahalanobis

La distancia de Mahalanobis se define como

$$d_{\text{mah}}(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{x} - \mathbf{y})\Sigma^{-1}(\mathbf{x} - \mathbf{y})^T} \tag{3.28}$$

donde Σ es la matriz de covarianzas $q \times q$ en la que el elemento (r, s) contiene la covarianza entre la variable x_r y x_s, calculada de esta manera

$$\Sigma = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1q} \\ c_{21} & c_{22} & \cdots & c_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ c_{q1} & c_{q2} & \cdots & c_{qq} \end{pmatrix} \tag{3.29}$$

$$c_{rs} = \frac{1}{n} \sum_{i=1}^{n} (x_{ir} - \bar{x}_r)(x_{is} - \bar{x}_s) \tag{3.30}$$

$$\bar{x}_k = \frac{1}{n} \sum_{i=1}^{n} x_{ik}, \quad k = 1, 2, \ldots, q \tag{3.31}$$

Requiere un elevado coste computacional, ya que para calcular la matriz de covarianza se ha de utilizar los n elementos que forman el conjunto de datos.

3.4.9 Distancia media de resta de atributos

Propuesta en Czekanowski (1962). Para dos puntos $\mathbf{x}$ e $\mathbf{y}$ pertenecientes a un espacio q-dimensional, se define como

$$d_{\text{mra}}(\mathbf{x}, \mathbf{y}) = \frac{1}{q} \sum_{k=1}^{q} |x_k - y_k| \tag{3.32}$$

3.4.10 Índice de asociación

Propuesta en Whittaker (1952). Para dos puntos $\mathbf{x}$ e $\mathbf{y}$ pertenecientes a un espacio q-dimensional, se define como

$$d_{\text{ind}}(\mathbf{x}, \mathbf{y}) = \frac{1}{2} \sum_{k=1}^{q} \left| \frac{x_k}{\sum_{l=1}^{q} x_l} - \frac{y_k}{\sum_{l=1}^{q} y_l} \right| \tag{3.33}$$

3.4.11 Métrica de Canberra

Propuesta en Legendre y Legendre (1983). Para dos puntos $\mathbf{x}$ e $\mathbf{y}$ pertenecientes a un espacio q-dimensional, se define como

$$d_{\text{can}}(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{q} \frac{|x_k - y_k|}{|x_k| + |y_k|} \tag{3.34}$$

3.4.12 Coeficiente de Czekanowski

Propuesta en Johnson y Wichern (2014). Para dos puntos $\mathbf{x}$ e $\mathbf{y}$ pertenecientes a un espacio q-dimensional, se define como

$$d_{\text{cze}}(\mathbf{x}, \mathbf{y}) = 1 - \frac{2 \sum_{k=1}^{q} \text{mín}(x_k, y_k)}{\sum_{k=1}^{q} (x_k + y_k)} \tag{3.35}$$

3.4.13 Coeficiente de divergencia

Propuesta en Legendre y Legendre (1983). Para dos puntos $\mathbf{x}$ e $\mathbf{y}$ pertenecientes a un espacio q-dimensional, se define como

$$d_{\text{div}}(\mathbf{x}, \mathbf{y}) = \left[\frac{1}{q} \sum_{k=1}^{q} \left(\frac{x_k - y_k}{x_k + y_k} \right)^2 \right]^{\frac{1}{2}} \tag{3.36}$$

3.4.14 Métrica discreta

Propuesta en Kaufman y Rousseeuw (2009). Para dos puntos $\mathbf{x}$ e $\mathbf{y}$ pertenecientes a un espacio q-dimensional, se define como

$$d_{\text{dis}}(\mathbf{x}, \mathbf{y}) = \begin{cases} 1, & \text{si } \mathbf{x} = \mathbf{y} \\ 0, & \text{si } \mathbf{x} \neq \mathbf{y} \end{cases} \tag{3.37}$$

3.5 El algoritmo *k-Means*

El algoritmo *k-means*, descrito inicialmente por MacQueen (1967), es uno de los métodos de clustering más ampliamente utilizados. Se puede clasificar como un método de clustering de tipo particional, no jerárquico, en el cual se divide el conjunto de datos en un número k de grupos, cada uno de ellos con un *centroide* denominado *media* (means). Este algoritmo requiere establecer el número de clusters k de antemano[2], así como realizar una inicialización previa de los grupos. Los resultados de agrupación obtenidos dependerán de forma determinista tanto del número de clusters como de la inicialización realizada, por lo que para confiar en los resultados será conveniente repetir el procedimiento con distintas inicializaciones para un determinado número k.

El algoritmo asignará cada elemento al cluster cuyo centroide le sea más cercano, pudiendo pertenecer cada elemento a un único cluster. En cada iteración se actualizarán las posiciones de los centroides, hasta que se cumpla un determinado criterio de parada, bajo el cual se considerará que el algoritmo ha finalizado. Como hemos visto, el funcionamiento del algoritmo tiene dos fases principales: la fase de inicialización y la fase de iteración. En la primera fase se asignará de forma aleatoria cada uno de los n elementos a uno de los k clusters. En la fase de iteración, el algoritmo calculará la distancia entre cada elemento y cada centroide, reasignando el elemento al cluster cuyo centroide tiene más cercano. Podemos formular el algoritmo k-means como un problema de optimización de una función objetivo que será minimizada bajo unas determinadas condiciones de convergencia (Gan, Ma y Wu, 2007).

Definición 3.5.1 Función objetivo k-means*, (Bezdek, 1981). Sea* $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\} \subset \mathbb{R}^q$ *un conjunto de datos de n elementos. La función objetivo* k-means $J_w \colon \boldsymbol{M}_c \times \mathbb{R}^{cq} \to \mathbb{R}^+$ *se define como*

$$J_W(U, \mathbf{v}) = \sum_{i=1}^{n} \sum_{j=1}^{c} u_{ij} (d_{ij})^2 \tag{3.38}$$

donde $d_{ij} = d(\mathbf{x}_i, \mathbf{v}_j)$ es una función distancia[3] calculada entre el elemento i y el centroide j; $\mathbf{v} = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_c) \in \mathbb{R}^{cq}, \mathbf{v}_j \in \mathbb{R}^q \forall j$ es el conjunto de centroides de los clusters; $\mathbf{v}_j$ es el centroide del cluster $u_j \in U, 1 \leq j \leq c$; y la matriz

[2] tradicionalmente en este algoritmo se denomina con la letra k el número de clusters, en lugar de la letra c que hemos venido utilizando hasta este momento.

[3] Nótese que, si bien usualmente se utiliza como función distancia la métrica euclidiana, cualquiera de las funciones descritas en el apartado 3.4 podría ser empleada.

$\mathbf{U} = [u_{ij}] \in \mathbf{M}_{cp}$ es la matriz de pertenencia relativa a una partición de tipo *hard*, que cumple las siguientes condiciones:

1. $u_{ij} \in 0, 1 \, \forall i = 1, 2, \ldots, n; j = 1, 2, \ldots, c$
2. $\sum_{j=1}^{c} u_{ij} = 1 \, \forall i = 1, 2, \ldots, n$

La optimización se realizará si y solo si los coeficientes u_{ij} se calculan de la siguiente manera

$$u_{ij} = \begin{cases} 1, & \text{si } d(\mathbf{x}_i, \mathbf{v}_j) = \min\limits_{1 \leq t \leq c} d(\mathbf{x}_i, \mathbf{v}_t) \\ 0, & \text{en otro caso.} \end{cases} \tag{3.39}$$

y los centroides de cada cluster u_j se calculan mediante

$$\mathbf{v}_j = \frac{\sum_{i=1}^{n} u_{ij} \mathbf{x}_i}{\sum_{i=1}^{n} u_{ij}} \tag{3.40}$$

Podemos escribir el pseudocódigo del algoritmo de la siguiente manera

Algoritmo 3.5.1 [K-MEANS] (Duda, Hart y Stork, 1973), (Bezdek, 1981)

(1) Establecer el número de clusters c, $2 \leq c < n$, inicializar $U^{(0)} \in M_c$. Para cada paso $l, l = 0, 1, 2, \ldots :$
(2) Calcular los c centroides $v_j^{(l)}$ con la expresión 3.40 y con $U^{(l)}$.
(3) Actualizar la matriz $U^{(l)}$ con la expresión 3.39 $\forall i, j$.
(4) Comparar $U^{(l)}$ y $U^{(l+1)}$ con cualquier norma matricial. Si $\|U^{(l+1)} - U^{(l)}\| \leq \epsilon_L$ entonces parar de iterar: de otra forma, establece $l = l + 1$ y vuelve al paso 2.

Nótese que la complejidad del algoritmo es $O(ncq)$, donde n es el número de elementos del conjunto de datos, c es el número de clusters y q es la dimensionalidad del espacio métrico (Phillips, 2002). A causa de esta dependencia es un algoritmo eficiente para analizar grandes conjuntos de datos, ya que su coste computacional es lineal con respecto al tamaño de los conjuntos; sin embargo es interesante señalar que el algoritmo no trabaja tan eficazmente cuando el número de dimensiones q es muy elevado. En este tipo de algoritmo, la convergencia hacia un mínimo absoluto no está garantizada ya que puede suceder que el algoritmo encuentre un mínimo local, no óptimo, en el que se detenga. Esto se puede evitar iterando el algoritmo con diferentes inicializaciones; algunas de éstas proporcionarán mejores resultados de agrupamiento que otras no tan favorables por lo que podremos

evaluar la calidad del resultado en función de la inicialización utilizada. Existen numerosas variaciones de este algoritmo estándar, como el algoritmo *k-armonic*, el algoritmo *EM Gaussiano* o el *algoritmo Fuzzy c-means*, del cual hablaremos a más adelante (Gan, Ma y Wu, 2007).

Ejemplo 3.5.1 *Podemos definir un sencillo clasificador de hojas de árboles teniendo en cuenta únicamente las dimensiones de sus hojas, es decir, midiendo el ancho y el largo de cada hoja. La siguiente figura muestra un pequeño conjunto de estas duplas de datos recogidas experimentalmente y pertenecientes a tres clases de árboles distintas (pino, álamo y haya). Se puede observar como los datos se agrupan claramente en tres clusters. Se mostrará el resultado de aplicar el algoritmo de k-means para una inicialización aleatoria de tres clusters ($k = 3$), y utilizando una métrica euclidiana.*

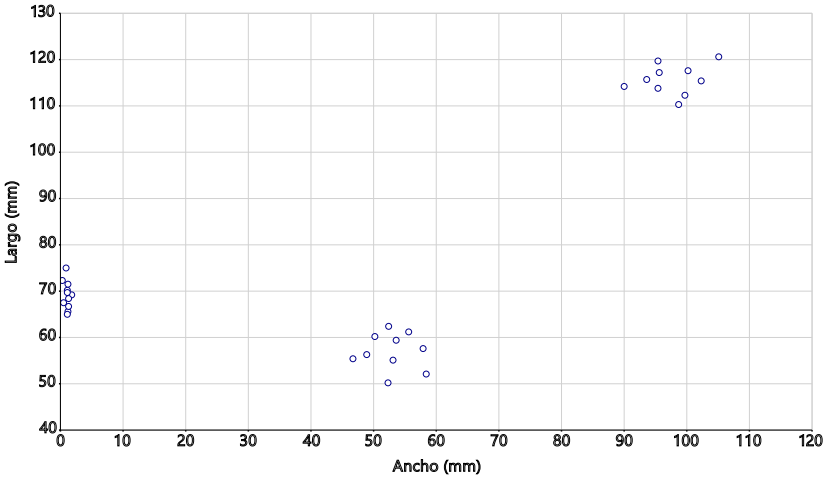


Figura 3.1: Conjunto de datos con las distintas medidas de las dimensiones de las hojas.

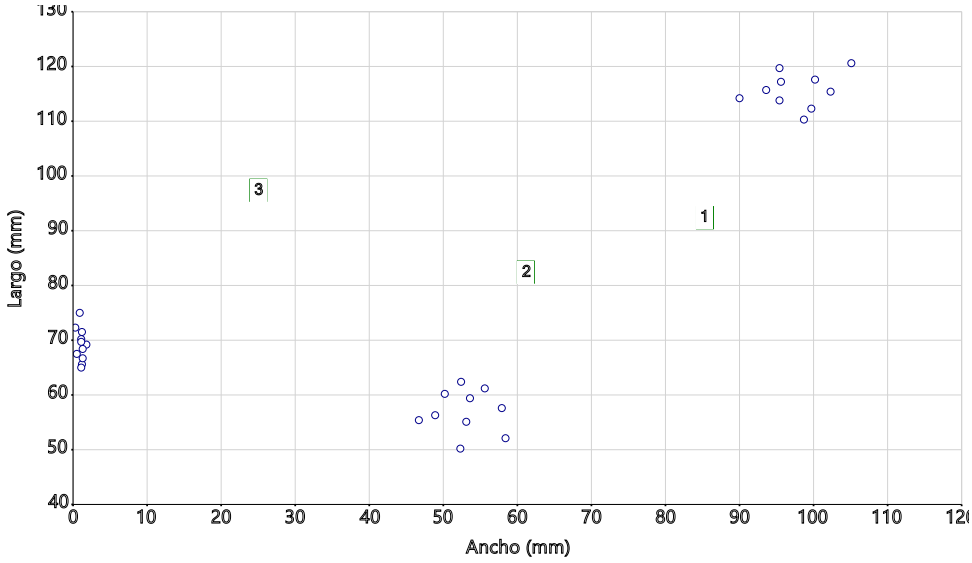


Figura 3.2: Inicialización de los tres centroides.

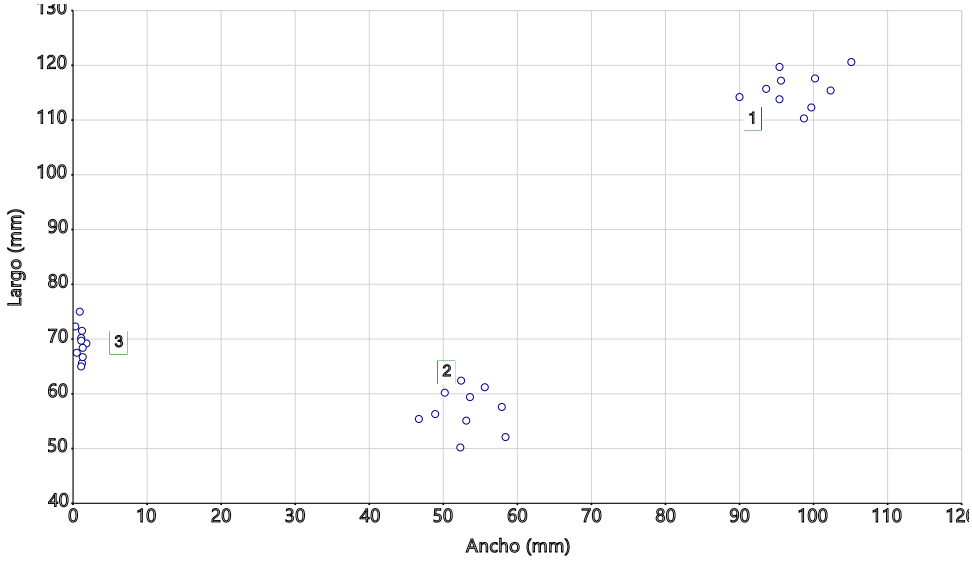


Figura 3.3: Proceso de convergencia: paso 1.

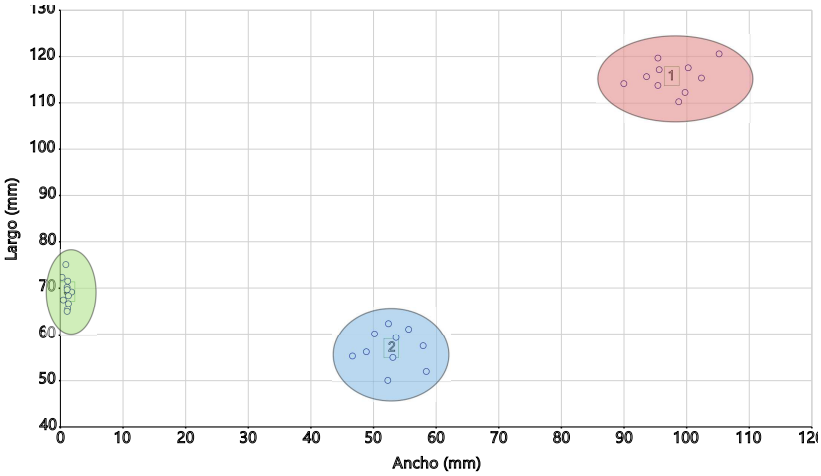


Figura 3.4: Proceso de convergencia: paso 2.

> *Nótese como los centroides, en su proceso iterativo hacia la convergencia final, atraviesan distintos estadios intermedios. Dicha secuencia de posiciones constituye una transición desde la posición inicial hasta la posición final.*

Es interesante representar los coeficientes de pertenencia 3.39 mediante un código de color asignado para cada cluster. Podemos comprobar como los coeficientes u_{ij} tienen únicamente los valores 0 ó 1 y por lo tanto un elemento pertenece sólo a un cluster.

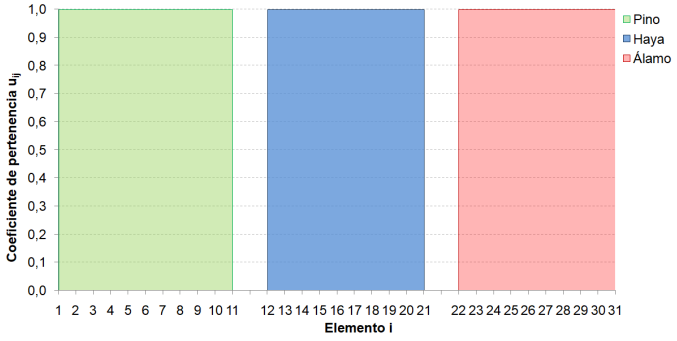


Figura 3.5: Coeficientes de pertenencia de cada elemento a cada cluster.

3.6 El algoritmo *Fuzzy c-Means* (FCM)

El algoritmo *fuzzy c-means* supone una generalización de las funciones descritas en 3.38, transformándolas en una familia infinita de funciones. La primera de estas generalizaciones la realizó Dunn (1973), siendo posteriormente formulado por Bezdek (1981) como una extensión del algoritmo k-means.

Definición 3.6.1 Funciones objetivo fuzzy c-means*, (Bezdek, 1981). Sea* $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^q$ *un conjunto de datos de n elementos. La función objetivo fuzzy* c-means $J_w \colon \boldsymbol{M}_{fc} \times \mathbb{R}^{cq} \to \mathbb{R}^+$ *se define como*

$$J_\lambda(U, \mathbf{V}) = \sum_{i=1}^{n} \sum_{j=1}^{c} u_{ij}^{\lambda} (d_{ij})^2 \tag{3.41}$$

donde

$$U \in \mathbf{M}_{fc} \tag{3.42}$$

es una partición *fuzzy* de $\mathbf{X}$,

$$\mathbf{V} = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_c) \in \mathbb{R}^{cq}, \quad \mathbf{v}_j \in \mathbb{R}^q \tag{3.43}$$

es el conjunto de centroides asociados a los clusters $u_j, 1 \leq j \leq c$, y

$$d_{ij} = d(\mathbf{x}_i, \mathbf{v}_j) \tag{3.44}$$

es cualquier función distancia en $\mathbb{R}^q$; u_{ij} es el coeficiente de pertenencia del elemento $\mathbf{x}_i$ al cluster j; y por último $\lambda \in [1, \infty)$ es el exponente de peso, o grado de *fuzziness* del proceso.
La función originariamente propuesta por Dunn (1973) se obtiene estableciendo $\lambda = 2$ y seleccionando la distancia euclidiana $d(ij) = d_{euc}(ij)$. Posteriormente fue generalizada de por Bezdek (1973) en la siguiente familia de funciones $\{J_\lambda | 1 \leq \lambda < \infty\}$. Podemos observar ahora las funciones objetivo tienen la distancia ponderada por los coeficientes de pertenencia u_{ij}. Como $\mathbf{M}_{fc}$ es una partición fuzzy, dichos coeficientes pueden tener cualquier valor entre $[0, 1]$.
El proceso de fuzzy clustering se conseguirá mediante una optimización iterativa de la función objetivo J_λ, actualizando en cada iteración tanto los coeficientes de pertenencia u_{ij} como los centroides $\mathbf{v}_j$ a través de las siguientes expresiones (Bezdek, 1981)

$$u_{ij} = \frac{1}{\sum_{k=1}^{c} \left[\frac{d(\mathbf{x}_i, \mathbf{v}_j)}{d(\mathbf{x}_i, \mathbf{v}_k)} \right]^{\frac{2}{\lambda - 1}}} \tag{3.45}$$

$$\mathbf{v}_j = \frac{\sum_{i=1}^{n} u_{ij}^{\lambda} \cdot \mathbf{x}_i}{\sum_{i=1}^{n} u_{ij}^{\lambda}} \tag{3.46}$$

La matriz $\mathbf{U}: n \times c$ es ahora una partición fuzzy de $\mathbf{X}$, construida por los coeficientes de pertenencia u_{ij}

$$U_{ij} = \begin{pmatrix} u_{11} & \cdots & u_{1c} \\ \vdots & \ddots & \vdots \\ u_{n1} & \cdots & u_{nc} \end{pmatrix} \tag{3.47}$$

Como condición sobre la partición fuzzy seguimos teniendo que la suma de los coeficientes de pertenencia de un elemento a todos los centroides tiene que ser igual a la unidad.

$$\sum_{j=1}^{c} u_{ij} = 1, \quad \forall 1 \leq i \leq n. \tag{3.48}$$

Escrito en pseudocódigo, el algoritmo fuzzy c-mean propuesto por Bezdek queda de la siguiente manera:

Algoritmo 3.6.1 [FUZZY C-MEANS (FCM)] (Bezdek, 1973)

(1) Establecer el número de clusters c, $2 \leq c < n$. Escoger cualquier función distancia en $\mathbb{R}^q$; establecer λ, $1 \leq \lambda < \infty$. Inicializar $U^{(0)}$. Para cada iteración $l, l = 0, 1, 2, \ldots, :$
(2) Calcular los centroides $\{v_j^{(l)}\}$ mediante $U^{(l)}$ y la expresión (3.46).
(3) Actualizar $U^{(l)}$ mediante $\{v_j^{(l)}\}$ y la expresión (3.45).
(4) Comparar $U^{(l)}$ con $U^{(l+1)}$ utilizando cualquier norma matricial, siendo $\epsilon \in (0, 1)$ un criterio de parada arbitrario. Si $\| U^{(l+1)} - U^{(l)} \| \leq \epsilon$ entonces parar, de lo contrario establecer $l = l + 1$ y volver al paso 2.

Ejemplo 3.6.1 *Supongamos que ampliamos el número de medidas del conjunto de datos presentado en el ejemplo 3.5.1 tal y como vemos en la figura que se muestra a continuación. Aplíquese el algoritmo fuzzy c-means con valor de* $\lambda = 2$, $c = 3$ *y métrica euclidiana. Se utilizará la misma inicialización que en mencionado ejemplo para realizar una comparativa de los resultados obtenidos.*

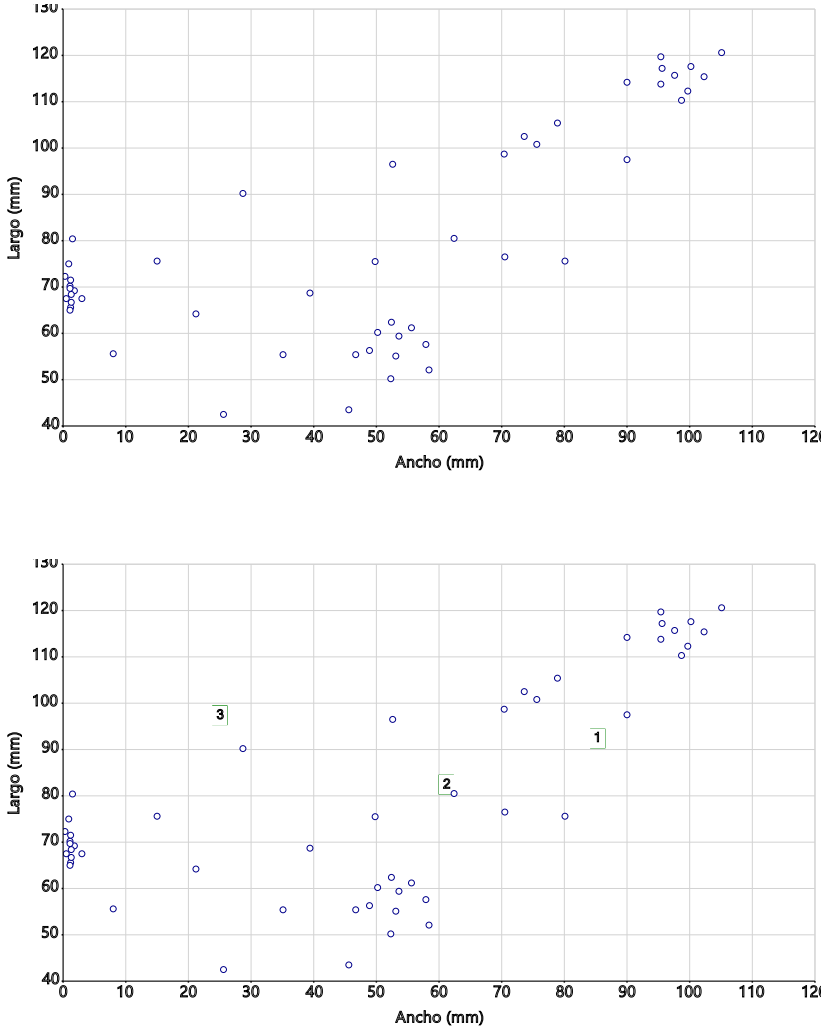


Figura 3.6: Inicialización de los centroides.

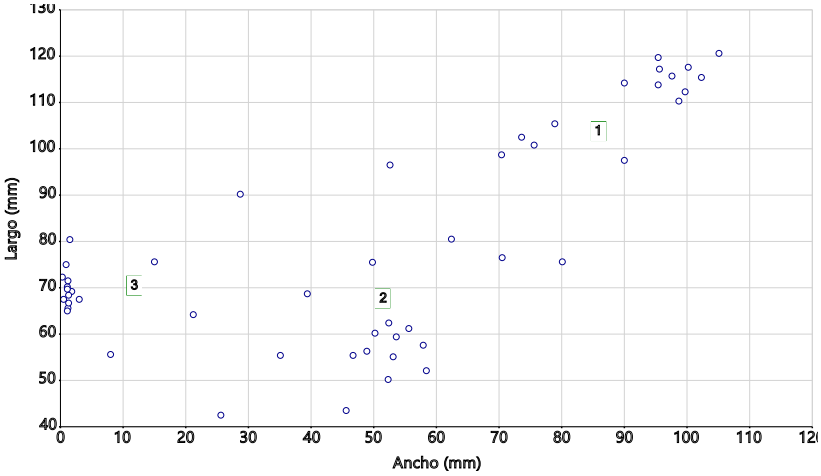


Figura 3.7: Aplicación del algoritmo FCM: paso 1.

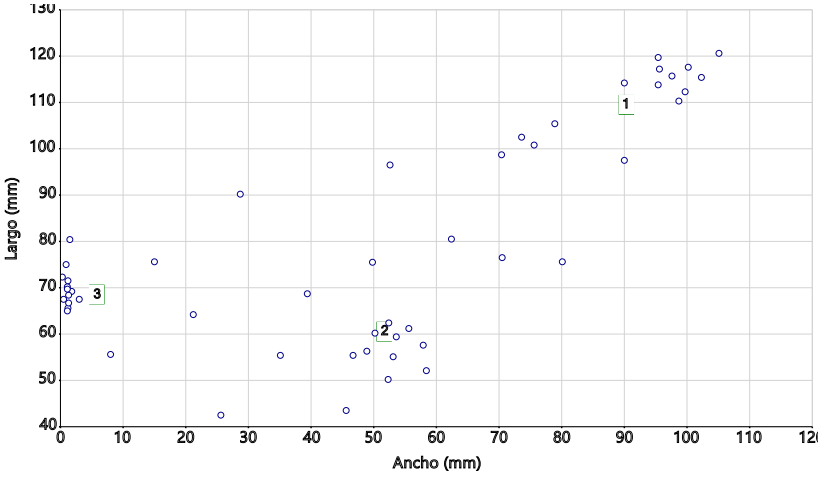


Figura 3.8: Aplicación del algoritmo FCM: paso 2.

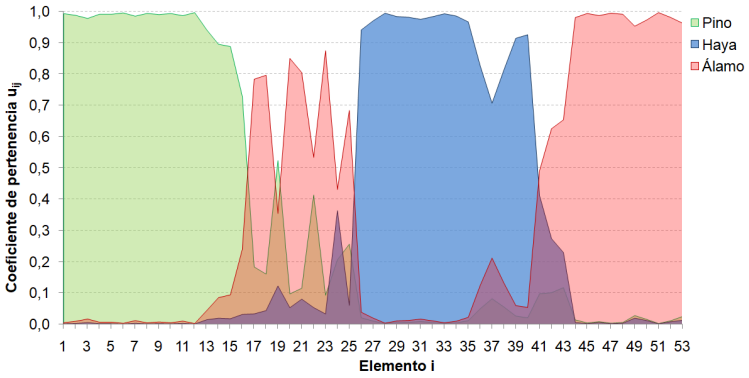


Figura 3.9: Coeficientes de pertenencia u_{ij}.

Podemos comparar los resultados obtenidos para los coeficientes de pertenencia en este caso con los obtenidos en el ejemplo 3.5.1. Observamos como los valores de u_{ij} pueden ahora tener cualquier valor comprendido entre el 0 y el 1, y por tanto un mismo elemento pertenecer en distinto grado a varios clusters simultáneamente.

3.7 El algoritmo *Fuzzy Ordered c-Means* (FOCM)

Propuesto por Martínez y Liern (2017), consiste en una variación del algoritmo fuzzy c-means en el cual se tiene en cuenta el orden de los elementos de $\mathbf{X}$ para realizar la partición fuzzy. La propuesta surge de la necesidad de comparar secuencias de datos, es decir, conjuntos de datos en los que el orden juega un papel determinante. Supongamos que tenemos una secuencia $\mathbf{X} = \{\mathbf{x}_i\}_{i=1}^n$ de n elementos, donde cada $\mathbf{x}_i \in \mathbb{R}^q$ y queremos realizar una partición de tipo fuzzy en un número c de grupos, pero en la cual el orden de los elementos de $\mathbf{X}$ esté correlacionado con el orden de los centroides $\mathbf{v}_j$; es decir, que sobre aquellos elementos que tengan posiciones iniciales exista una predisposición mayor a ser incluidos en los grupos iniciales, y que sobre aquellos elementos que tengan unas posiciones más cercanas al final exista una mayor predisposición hacia su pertenencia a grupos finales.

Esta dependencia de los índices i (que representan a los n elementos de la secuencia $\mathbf{X}$), con los índices j (que representan los distintos c centroides de la partición fuzzy) será expresada en términos matemáticos mediante *funciones de vecindad*, que proporcionaran un coeficiente de peso comprendido entre $[0, 1]$ para cada una de las distintas duplas (i, j).

De esta manera la dependencia con el orden podrá ser introducida en el algoritmo de clustering ya que utilizaremos los pesos proporcionados por las funciones de vecindad para modificar los coeficientes de pertenencia correspondientes a la matriz $\mathbf{U}$. Dicha dependencia con el orden únicamente podrá ser definida mediante particiones de tipo fuzzy en las cuales la matriz U puede ser continua, es decir, ya no está formada por unos o ceros sino que admite que sus elementos tomen cualquier valor real entre 0 y 1, con la condición de que la suma de los valores de cada fila sea igual a la unidad.

3.7.1 Funciones de vecindad

Definición 3.7.1 *Una función continua $f : \mathbb{R}^2 \to \mathbb{R}$ es una función de vecindad entre dos secuencias $\{\mathbf{x}_i\}_{i=1}^n$ y $\{\mathbf{v}_j\}_{j=1}^c$ con $c < n$ si*

$$\int_0^{n-1} f(i,j)di < \infty, \quad \forall j \in \{1, 2, \ldots, c\}, \quad \forall i \in \{1, 2, \ldots, n\} \tag{3.49}$$

donde n, c son los números de elementos de la primera y segunda secuencia, respectivamente.

Definición 3.7.2 Función de vecindad gaussiana

$$f(i,j) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\left[\frac{1}{2\sigma^2}\left(i-\frac{(n-1)}{(c-1)}j\right)^2\right]} \tag{3.50}$$

donde n, c son los números de elementos de la primera y segunda secuencia, respectivamente, y σ la anchura de la gaussiana. Podemos definir tres sub tipos de funciones de vecindad gaussianas (ancha, normal, estrecha) estableciendo esto valores de la desviación estándar $\sigma_{ancha} = \frac{n}{3}$, $\sigma_{normal} = \frac{n}{6}$ y $\sigma_{estrecha} = \frac{n}{12}$ respectivamente

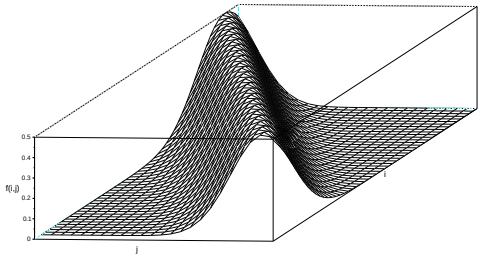


Figura 3.10: Función de vecindad gaussiana.

Definición 3.7.3 Función de vecindad exponencial

$$f(i,j) = e^{-\frac{1}{\tau}\left|i-\frac{(n-1)}{(c-1)}j\right|} \tag{3.51}$$

donde n, c son los números de elementos de la primera y segunda secuencia, respectivamente, y τ la anchura de la función. Podemos definir tres sub tipos de funciones de vecindad exponenciales (ancha, normal, estrecha) estableciendo esto valores de $\tau_{ancha} = \frac{n}{2}$, $\tau_{normal} = \frac{n}{4}$ y $\tau_{estrecha} = \frac{n}{8}$ respectivamente.

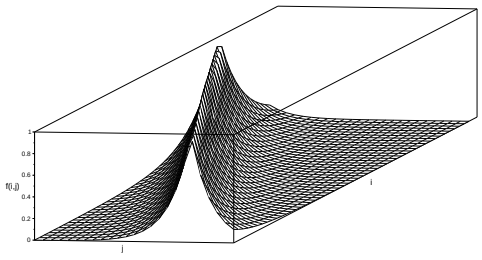


Figura 3.11: Función de vecindad exponencial.

Definición 3.7.4 Función de vecindad triangular

$$f(i,j) = \begin{cases} \frac{i}{\mu}, & \text{si } j \neq 0 \text{ y } 0 \leq i \leq \mu \\ 1 - \frac{(i-\mu)}{(n-\mu-1)}, & \text{si } j = 0 \text{ ó } \mu < i \leq n-1 \end{cases} \tag{3.52}$$

donde

$$\mu = \frac{(n-1)}{(c-1)} j, \tag{3.53}$$

n, c son los números de elementos de la primera y segunda secuencia, respectivamente

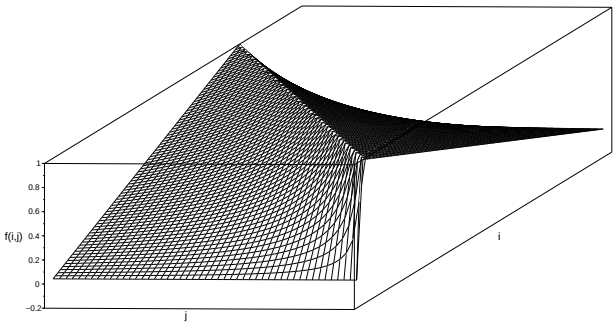


Figura 3.12: Función de vecindad triangular.

Definición 3.7.5 Función de vecindad rectangular

$$f(i,j) = \begin{cases} 0, & \text{si } 0 \leq i < \mu - {}^{L}/_{2} \\ 1, & \text{si } \mu - {}^{L}/_{2} \leq i \leq \mu + {}^{L}/_{2} \\ 0, & \text{si } \mu + {}^{L}/_{2} < i \leq n-1 \end{cases} \tag{3.54}$$

donde

$$\mu = \frac{(n-1)}{(c-1)} j \tag{3.55}$$

n, c son los números de elementos de la primera y segunda secuencia, respectivamente, y

$$L = \frac{n-1}{3} \tag{3.56}$$

es la anchura del rectángulo.

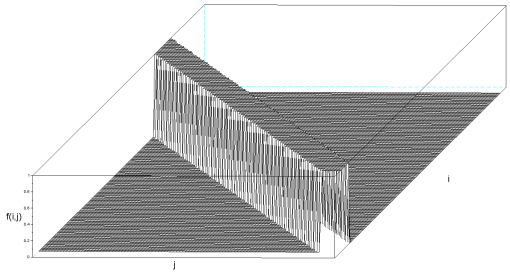


Figura 3.13: Función de vecindad rectangular.

Definición 3.7.6 Función de vecindad trapezoidal

$$f(i,j) = \begin{cases} \frac{i}{\mu - L/2}, & \text{si } 0 \leq i < \mu - L/2 \\ 1, & \text{si } \mu - L/2 \leq i \leq \mu + L/2 \\ \frac{L/2+\mu-i}{n-\mu-L/2-1} + 1, & \text{si } \mu + L/2 < i \leq n-1 \end{cases} \tag{3.57}$$

donde

$$\mu = \frac{(n-1)}{(c-1)} j, \tag{3.58}$$

n, c son los números de elementos de la primera y segunda secuencia, respectivamente, y

$$L = \frac{n-1}{3} \tag{3.59}$$

es la anchura del trapecio.

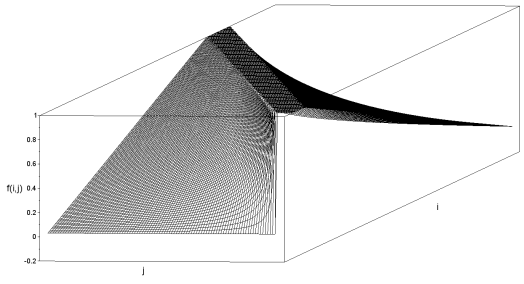


Figura 3.14: Función de vecindad trapezoidal.

Definición 3.7.7 Función de vecindad sigmoidal

$$f(i,j) = \frac{1}{1 + e^{[a \cdot b(i-c)]}} \tag{3.60}$$

donde

$$\begin{aligned} a &= -2j/(c-1) + 1 \\ b &= \tau/(n-1) \\ c &= (n-1)/2 \end{aligned}$$

siendo n, c los números de elementos de la primera y segunda secuencia, respectivamente, y τ es la anchura del escalón de la función sigmoidal. Podemos definir tres sub tipos de funciones de vecindad sigmoidales (ancha, normal, estrecha) estableciendo estos valores del parámetro τ, que representa la pendiente del escalón de la sigmoidal: $\tau_{plana} = 5$, $\tau_{normal} = 15$ y $\tau_{vertical} = 150$.

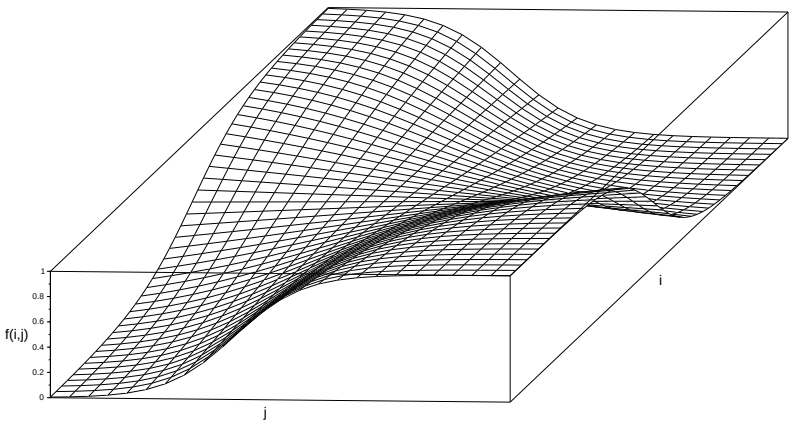


Figura 3.15: Función de vecindad sigmoidal.

Definición 3.7.8 Función de vecindad discreta

$$f(i,j) = \begin{cases} 1, & \text{si } i = j \\ 0, & \text{si } i \neq j \end{cases} \tag{3.61}$$

3.7.2 Descripción del algoritmo FOCM

Nuestra propuesta será modificar el algoritmo FCM expuesto en 3.6 de tal manera que el orden de las secuencias influya en el resultado del procedimiento de *clustering* (Martínez y Liern, 2017). Introduciremos un cambio sustancial: para cada paso del algoritmo, una vez que la partición $\mathbf{U}$ haya sido calculada asignaremos un peso a cada elemento u_{ij} calculado gracias a una función de vecindad $f(i,j)$ específica, previamente seleccionada. Para poder cumplir con los criterios de convergencia del algoritmo FCM expuestos en 3.48, normalizaremos la nueva matriz $\hat{\mathbf{U}}$ de la siguiente manera

$$u_{ij} = \frac{1}{\sum_{k=1}^{c} \left[\frac{d(\mathbf{x}_i, \mathbf{v}_j)}{d(\mathbf{x}_i, \mathbf{v}_k)}\right]^{\frac{2}{\lambda-1}}} \tag{3.62}$$

$$\hat{u}_{ij} = \frac{u_{ij} \cdot f(i,j)}{\sum_{k=1}^{c} u_{ik} \cdot f(i,k)}. \tag{3.63}$$

$$\mathbf{v}_j = \frac{\sum_{i=1}^{n} \hat{u}_{ij}^{\lambda} \cdot \mathbf{x}_i}{\sum_{i=1}^{n} \hat{u}_{ij}^{\lambda}} \tag{3.64}$$

Algoritmo 3.7.1 [Fuzzy ordered c-means (FOCM)] (Martínez y Liern, 2017)

(1) Escoger cualquier función de vecindad conveniente.
(2) Escoger cualquier función distancia d en $\mathbb{R}^q$; establecer λ, $1 \leq \lambda < \infty$. Inicializar $\widehat{U}^{(0)}$. Para cada iteración $l, l = 0, 1, 2, \ldots, :$
(3) Calcular los centroides $\{v_j^{(l)}\}$ con $\widehat{U}^{(l)}$ y la expresión 3.64.
(4) Actualizar $\widehat{U}^{(l)}$ utilizando las ecuaciones 3.62, 3.63 y $\{v_j^{(l)}\}$.
(5) Comparar $\widehat{U}^{(l)}$ con $\widetilde{U}^{(l+1)}$ utilizando cualquier norma matricial, siendo $\epsilon \in (0,1)$ un criterio de parada arbitrario. Si $\| \widehat{U}^{(l+1)} - \widehat{U}^{(l)} \| \leq \epsilon$ entonces parar, de lo contrario establecer $l = l + 1$ y volver al paso 3.

Ejemplo 3.7.1 *Repitamos el experimento del ejemplo 3.5.1 pero esta vez utilizando el algoritmo Fuzzy Ordered c-means, con una función de vecindad de tipo gaussiana, métrica euclidiana,* $\lambda = 2$*,* $c = 3$*, y la misma inicialización.*

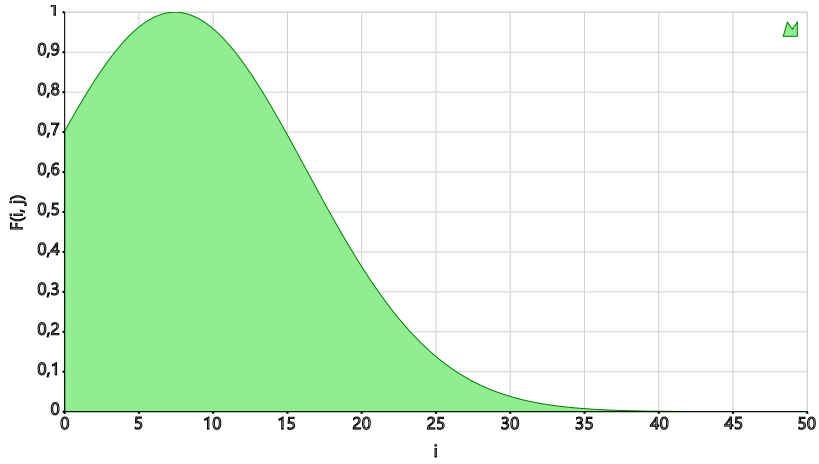


Figura 3.16: Ejemplo de función de vecindad gaussiana para los valores de $i = 53$ y $j = 1$.

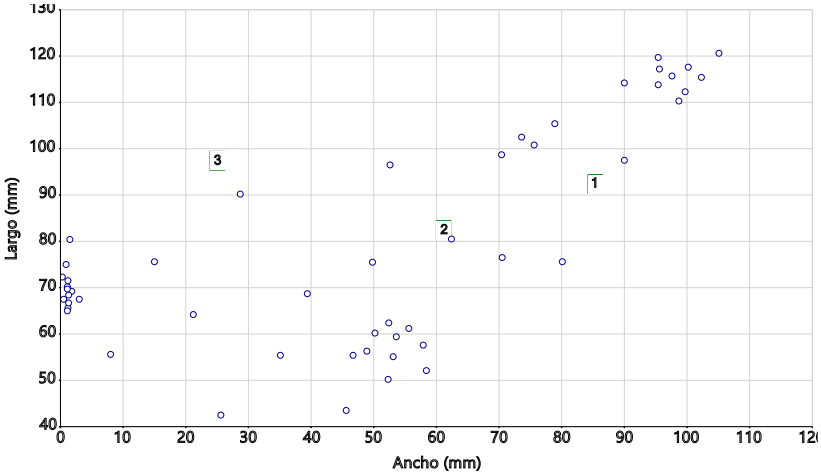


Figura 3.17: Inicialización de los centroides.

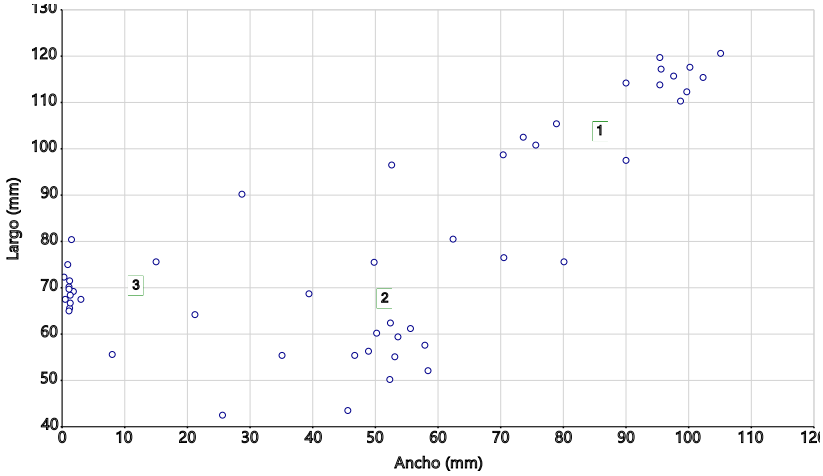


Figura 3.18: Aplicación del algoritmo FOCM: paso 1.

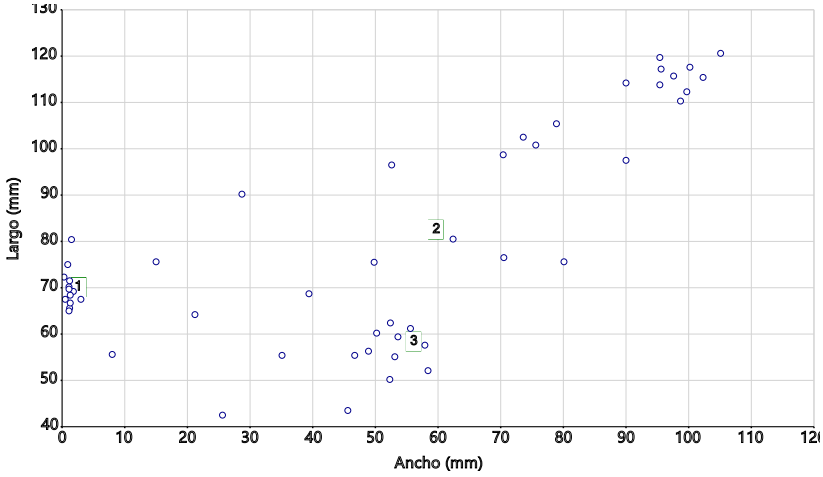


Figura 3.19: Aplicación del algoritmo FOCM: paso 2.

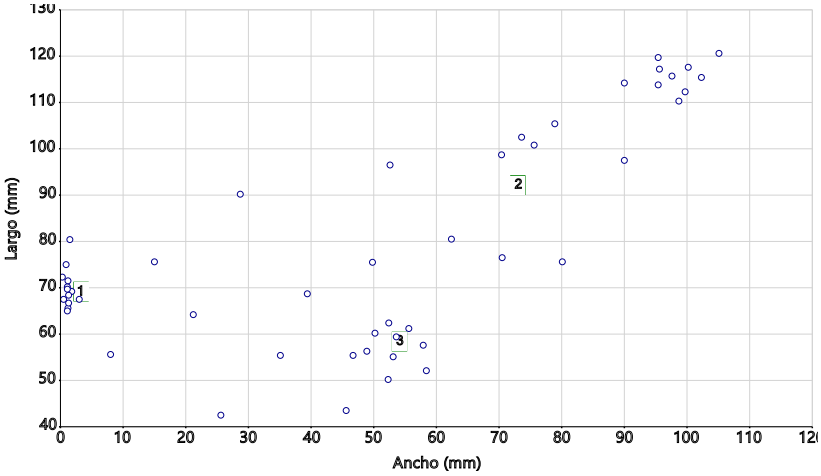


Figura 3.20: Aplicación del algoritmo FOCM: paso 3.

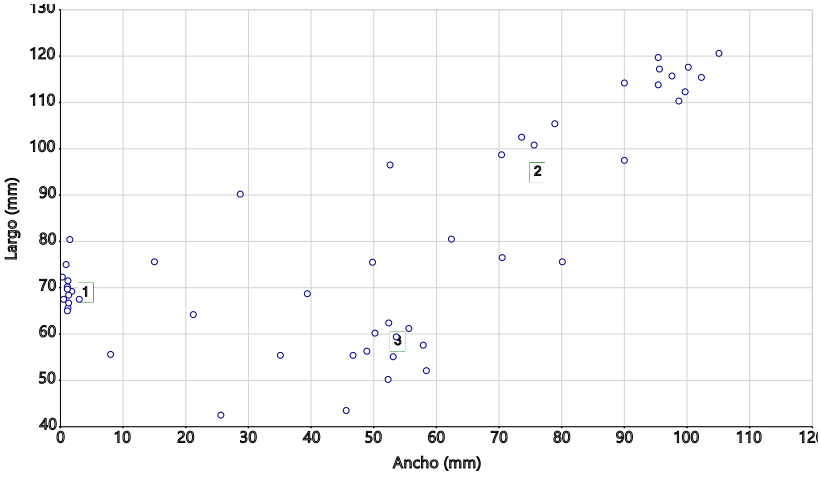


Figura 3.21: Aplicación del algoritmo FOCM: paso 4.

Podemos comparar los resultados obtenidos para los coeficientes de pertenencia en este caso con los obtenidos en el ejemplo anterior. Observamos como los valores de u_{ij} se han ordenado con respecto al orden inicial de los clusters (pino, álamo y haya).

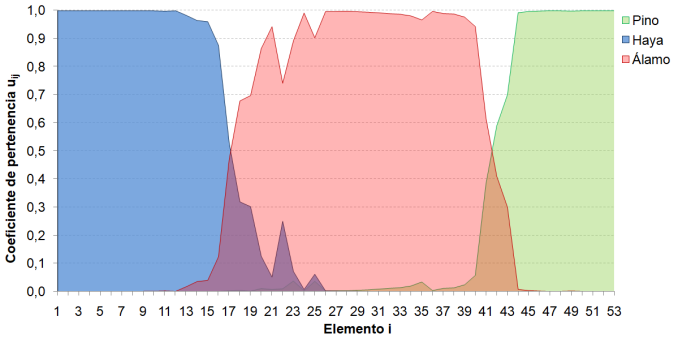


Figura 3.22: Coeficientes de pertenencia u_{ij} calculados con FOCM.

3.7.3 Transiciones difusas simples

Como hemos visto, durante la ejecución de los algoritmos de clustering FCM y FOCM los centroides sufren en cada iteración una modificación de sus coordenadas desde el estado inicial hasta el estado final en el que se ha conseguido minimizar una determinada función objetivo bajo un criterio de parada arbitrario. Se puede entender que la secuencia de posiciones intermedias por las que cada centroide pasa constituye una transición desde su posición inicial hasta su posición final en la que se ha minimizado paulatinamente dicha función objetivo, y por tanto se ha reducido progresivamente la distancia entre este centroide y los miembros del cluster asociados a él. El carácter determinista de estos algoritmos garantiza que dada una inicialización siempre se obtendrán los mismos estados parciales y final.

Definición 3.7.9 Transición difusa simple. *Sea* $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\} \subset \mathbb{R}^q$ *un conjunto de datos en un espacio métrico* $\mathbb{R}^q$. *Sea* $\boldsymbol{V}^l = \{\boldsymbol{v}_1^l, \boldsymbol{v}_2^l, \ldots, \boldsymbol{v}_c^l\} \in \mathbb{R}^{cq}$, $v_j \in \mathbb{R}^q$ *con* $1 \leq j \leq c$ *y* $2 \leq c \leq n$ *el conjunto de* c *centroides calculados mediante la aplicación del algoritmo FCM o FOCM sobre el conjunto de datos* $\boldsymbol{X}$ *en la iteración número* l, *para un total de* k *iteraciones. Sea* $\boldsymbol{V}^0 = \{\boldsymbol{v}_1^0, \boldsymbol{v}_2^0, \ldots, \boldsymbol{v}_c^0\}$ *la inicialización previa de los centroides requerida a la ejecución del algoritmo. Una transición difusa desde los centroides iniciales* $\boldsymbol{V}^0$ *hasta los centroides finales* $\boldsymbol{V}^k$ *viene definida por la siguiente secuencia*

$$\mathscr{T} = \{\mathbf{V}^0, \mathbf{V}^1, \ldots, \mathbf{V}^k\} = \{\mathbf{V}^l\}_{l=0}^k, \mathbf{V}^l \in \mathbb{R}^{cq} \tag{3.65}$$

con $0 \leq k < \infty$; $0 \leq l \leq k$; y donde cada $\mathbf{V}^l = \{\mathbf{v}_1^l, \mathbf{v}_2^l, \ldots, \mathbf{v}_c^l\} \in \mathbb{R}^{cq}$ representa el conjunto de centroides del proceso en cada iteración l, y $\mathbf{V}^0 = \{\mathbf{v}_1^0, \mathbf{v}_2^0, \ldots, \mathbf{v}_c^0\}$ representa la inicialización de los centroides previa a la ejecución del algoritmo, y c es el número de centroides.

3.8 El algoritmo *Fuzzy Complete Transitions* (FCT)

3.8.1 Descripción del algoritmo FCT

En esta sección proponemos un nuevo algoritmo de clustering, basado en el FOCM, capaz de realizar transiciones completas entre dos secuencias cualesquiera, de distinto número de elementos, respetando el orden de los mismos durante el proceso. Dadas dos secuencias de n y c elementos en un espacio métrico $\mathbb{R}^q$, el algoritmo inicializa el conjunto de centroides con la secuencia que deseamos modificar, usualmente la que posee un menor número de elementos c, siendo $c < n$. Esta secuencia se denominará en lo sucesivo como *secuencia B*. Nuestro objetivo será llegar mediante progresivas modificaciones de B a la secuencia final de n elementos, nuestra meta, denominada *secuencia A*. Asignaremos con los elementos de A el conjunto de datos inicial sobre el que se calcularán las sucesivas particiones fuzzy. Aplicaremos FOCM y una vez que éste haya acabado, añadiremos un nuevo centroide al conjunto final de centroides, ejecutando nuevamente el algoritmo y repitiendo el proceso hasta que el número de centroides de la secuencia B haya alcanzado al número de datos de la secuencia A. En este punto el algoritmo converge y la secuencia B se ha convertido exactamente en la secuencia A, proporcionando un gran número de estados intermedios por el camino.

Algoritmo 3.8.1 [Algoritmo FCT (Fuzzy Complete Transitions)]

(1) Escoger cualquier función de vecindad conveniente.
(2) Escoger cualquier función distancia d en $\mathbb{R}^q$; establecer λ, $1 \leq \lambda < \infty$. Establecer $c_0 = c$.
(3) Inicializar los centroides: asignar los c_0 centroides con los valores desde los que se desea partir. Para cada iteración $l, l = 0, 1, 2, \ldots, :$
(4) Si $c_l > n$ parar; de lo contrario si $c_l \leq n$ hacer:
(5) Si $c_l = n$ establecer como función de vecindad la vecindad discreta (véase 3.7.8), de lo contrario continuar con la misma función de vecindad.
(6) Actualizar $\widehat{U}^{(l)}$ utilizando las ecuaciones 3.62, 3.63 y $\{v_j^{(l)}\}$.
(7) Comparar $\widehat{U}^{(l)}$ con $\widetilde{U}^{(l+1)}$ utilizando cualquier norma matricial, siendo $\epsilon \in (0, 1)$ un criterio de parada arbitrario. Si $\| \widehat{U}^{(l+1)} - \widehat{U}^{(l)} \| \leq \epsilon$ entonces saltar al paso 8. En caso contrario establecer $l = l+1$, calcular los centroides $\{v_j^{(l)}\}$ con $\widehat{U}^{(l)}$ y la expresión 3.64 y volver al paso 6.
(8) Elegir los dos puntos consecutivos h, $h+1$ con $1 \leq h \leq c_l - 1$ para los que la distancia entre ellos es máxima, es decir, para la iteración actual l seleccionar un h tal que $\text{máx}_{h=1}^{c_l-1}[d(\mathbf{v}_h^l, \mathbf{v}_{h+1}^l)]$.

(9) Establecer $l = l + 1$. Añadir a $\mathbf{V}^l$ un centroide nuevo en la posición $h + 1$ de la secuencia de centroides, con los atributos calculados como la media: $(\mathbf{v}_h^l + \mathbf{v}_{h+1}^l)/2$. Actualizar $c_l = c_l + 1$. Volver al paso 4.

El conjunto de todos los estados por los que atraviesa el conjunto de centroides constituye una transición difusa completa desde el estado inicial $\mathbf{V}^0$ hasta el estado final $\mathbf{V}^k$. En dicho estado final el número de centroides c_k es igual al número de puntos n y cada centroide i es igual al elemento i, cumpliéndose la condición 3.66.

3.8.2 Transiciones difusas completas

Resulta posible realizar una transición desde cualquier estadio inicial, representado por los centroides $\mathbf{V}^0 = \{\mathbf{v}_1^0, \mathbf{v}_2^0, \ldots, \mathbf{v}_{c_0}^0\}$, hasta el conjunto de datos inicial $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, de tal manera que tras un total de k iteraciones

$$\mathbf{v}_i^k = \mathbf{x}_i, \forall 1 \leq i \leq n \tag{3.66}$$

Definición 3.8.1 Transición difusa completa. *Sea* $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\} \subset \mathbb{R}^q$ *un conjunto de datos en un espacio métrico* $\mathbb{R}^q$. *Sea* $\boldsymbol{V}^l = \{\boldsymbol{v}_1^l, \boldsymbol{v}_2^l, \ldots, \boldsymbol{v}_{c_l}^l\} \in \mathbb{R}^{cq}, v_j \in \mathbb{R}^q$ *con* $1 \leq j \leq c_l$ *y* $2 \leq c_l \leq n$ *el conjunto centroides calculados mediante el algoritmo FCT (véase 3.8.1) sobre el conjunto de datos* $\boldsymbol{X}$ *en la iteración número* l, *para un total de* k *iteraciones. Sea* $\boldsymbol{V}^0 = \{\boldsymbol{v}_1^0, \boldsymbol{v}_2^0, \ldots, \boldsymbol{v}_c^0\}$ *la inicialización previa de los centroides requerida a la ejecución del algoritmo. Una transición difusa completa desde los centroides iniciales* $\boldsymbol{V}^0$ *hasta los centroides finales* $\boldsymbol{V}^k$ *viene definida por la siguiente secuencia*

$$\widehat{\mathscr{T}} = \{\mathbf{V}^0, \mathbf{V}^1, \mathbf{V}^2, \ldots, \mathbf{V}^k\} = \{\mathbf{V}^l\}_{l=0}^k, \mathbf{V}^l \in \mathbb{R}^{c_l q} \tag{3.67}$$

con $0 \leq k < \infty$; $0 \leq l \leq k$; y donde cada $\mathbf{V}^l = \{\mathbf{v}_1^l, \mathbf{v}_2^l, \ldots, \mathbf{v}_c^l\} \in \mathbb{R}^{cq}$ representa el conjunto de centroides del proceso en cada iteración l; $\mathbf{V}^0 = \{\mathbf{v}_1^0, \mathbf{v}_2^0, \ldots, \mathbf{v}_c^0\}$ representa la inicialización de los centroides previa a la ejecución del algoritmo, y c_l es el número de centroides existentes en cada iteración c_l.

Estas transiciones consistirán en un acercamiento de los centroides hacia el total de puntos de $\mathbf{X}$ en las que se añadirán progresivamente nuevos centroides, de tal manera que el número de éstos crezca hasta llegar a ser igual que el número de puntos del conjunto $\mathbf{X}$. Para cualquier iteración l se cumplirá

$$
\begin{aligned}
c_l &\in [c, n] \\
c_0 &= n \\
c_k &= n
\end{aligned}
$$

donde c es el número de centroides iniciales, determinados por la inicialización, n es el número de puntos del conjunto $\mathbf{X}$, c_l es el número de centroides para cada iteración l, y k es el número total de iteraciones necesarias para finalizar la transición completa.

3.9 Medida de la disimilitud

Una vez realizada la partición del conjunto de datos en los requeridos clusters, a menudo es útil definir determinadas magnitudes con las cuales podamos medir el grado de similitud entre cada uno de los grupos, su cohesión o la cercanía entre alguno de los elementos y unos determinados grupos. Estas propiedades, que en general no van a satisfacer todos los requisitos de una función distancia (véase 3.4, 3.5, 3.6), se denominarán *disimilitudes* y nos proporcionarán una valiosa información relativa a las características del agrupamiento realizado. A continuación resumiremos algunas de las *disimilitudes* tradicionalmente más utilizadas en el análisis de cluster, para a continuación proponer dos nuevas medidas de la disimilitud basadas en el *fuzzy clustering* y los algoritmos FCM y FOCM.

3.9.1 Disimilitud de vecino más cercano

Propuesta por Williams y Lambert (1966). Supongamos que hemos realizado una partición $\mathbf{U}$ (exclusivamente de tipo *hard*) de un conjunto de n datos $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ sobre un total de c, $2 \leq c < n$ clusters. Supongamos que $\mathbf{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_r\}$ y $\mathbf{Z} = \{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_s\}$ son dos clusters cualquiera, que contienen un número r y s de elementos de $\mathbf{X}$ respectivamente. La disimilitud[4] de vecino más cercano se define como

$$\mathscr{D}_{near}(\mathbf{Y}, \mathbf{Z}) = \min_{1 \leq i \leq r, 1 \leq j \leq s} d(\mathbf{y}_i, \mathbf{z}_j) \tag{3.68}$$

donde d es cualquier función distancia.

3.9.2 Disimilitud de vecino más lejano

Propuesta por Duran y Odell (1974). Sea d una función distancia. La disimilitud de vecino más lejano se define como

$$\mathscr{D}_{far}(\mathbf{Y}, \mathbf{Z}) = \max_{1 \leq i \leq r, 1 \leq j \leq s} d(\mathbf{y}_i, \mathbf{z}_j) \tag{3.69}$$

[4]Nótese que ahora no podemos hablar de función distancia ya que, en general, no se tiene porqué cumplir que $\mathscr{D}_{near}(\mathbf{Y}, \mathbf{Z}) \leq \mathscr{D}_{near}(\mathbf{Y}, \mathbf{W}) + \mathscr{D}_{near}(\mathbf{Z}, \mathbf{W}), \forall\ \mathbf{Z}, \mathbf{Y}, \mathbf{W} \subset \mathbf{X}$.

3.9.3 Disimilitud media entre vecinos

Propuesta por Duran y Odell (1974). Sea d una función distancia. La disimilitud media entre vecinos se define como

$$\mathscr{D}_{ave}(\mathbf{Y}, \mathbf{Z}) = \frac{1}{rs} \sum_{i=1}^{r} \sum_{j=1}^{s} d(\mathbf{y}_i, \mathbf{z}_j) \tag{3.70}$$

3.9.4 Disimilitud estadística

Propuesta por Duran y Odell (1974). Sea d una función distancia. La disimilitud estadística se define como

$$\mathscr{D}_{stat}(\mathbf{Y}, \mathbf{Z}) = \frac{rs}{r+s} (\bar{\mathbf{y}} - \bar{\mathbf{z}})(\bar{\mathbf{y}} - \bar{\mathbf{z}})^T \tag{3.71}$$

donde

$$\bar{\mathbf{y}} = \frac{1}{r} \sum_{i=1}^{r} \mathbf{y}_i \tag{3.72}$$

$$\bar{\mathbf{z}} = \frac{1}{s} \sum_{j=1}^{s} \mathbf{z}_j \tag{3.73}$$

3.9.5 Disimilitud entre centroides

Supongamos que hemos realizado una partición $\mathbf{U}$ (*hard* o *fuzzy*) de un conjunto de n datos $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ sobre un total de c, $2 \leq c < n$ clusters. Las posiciones de cada uno de los centroides $\mathbf{v}_j$, $1 \leq j \leq c$ vienen definidas por la expresión

$$\mathbf{v}_j = \frac{\sum_{i=1}^{n} u_{ij}^{\lambda} \mathbf{x}_i}{\sum_{i=1}^{n} u_{ij}^{\lambda}} \tag{3.74}$$

donde u_{ij} son los coeficientes de pertenencia de la matriz de partición $\mathbf{U}$, y λ es el grado de *fuzzyness* del proceso de *clustering*. Nótese que cuando el *clustering* sea de tipo *hard*, $\lambda = 1$.
La disimilitud[5] entre dos centroides cualquiera $\mathbf{v}_i, \mathbf{v}_j$ estará definida por la si-

[5] En este caso concreto es fácil demostrar que esta disimilitud sí que satisface las condiciones de una función distancia.

guiente expresión

$$\mathscr{D}_{mean}(\mathbf{v}_i, \mathbf{v}_j) = d(\mathbf{v}_i, \mathbf{v}_j) \tag{3.75}$$

donde d es cualquier función distancia.

3.9.6 Disimilitud media entre conjunto de datos y centroides

Definición 3.9.1 Disimilitud media. *Supongamos que hemos realizado una partición* $\boldsymbol{U}$ *(*hard *o* fuzzy*) de un conjunto de* n *datos* $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ *generando un total de* c, $2 \leq c < n$ *clusters* $\boldsymbol{V} = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_c\}$. *La disimilitud media entre el conjunto de datos y el conjunto de centroides se define como*

$$\mathscr{D}(\mathbf{X}, \mathbf{V}) = \frac{1}{n \cdot c} \sum_{i=1}^{n} \sum_{j=1}^{c} u_{ij} \cdot d(\mathbf{x}_i, \mathbf{v}_j) \tag{3.76}$$

donde u_{ij} *son los coeficientes de pertenencia de la matriz de partición* $\boldsymbol{U}$*, y* d *es cualquier función distancia.*

3.9.7 Disimilitud media ordenada entre conjunto de datos y centroides

Definición 3.9.2 Disimilitud media ordenada. *Supongamos que hemos realizado una partición* $\boldsymbol{U}$ *de tipo* fuzzy *de un conjunto de* n *datos* $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ *mediante el algoritmo FOCM (3.7), generando un total de* c, $2 \leq c < n$ *clusters* $\boldsymbol{V} = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_c\}$. *La disimilitud media ordenada entre el conjunto de datos y el conjunto de centroides se define como*

$$\widehat{\mathscr{D}}(\mathbf{X}, \mathbf{V}) = \frac{1}{n \cdot c} \sum_{i=1}^{n} \sum_{j=1}^{c} \hat{u}_{ij} \cdot d(\mathbf{x}_i, \mathbf{v}_j) \tag{3.77}$$

donde $\hat{u}_{ij}$ *son los coeficientes de pertenencia de la matriz de partición* $\boldsymbol{U}$ *calculados mediante el algoritmo FOCM, y* d *es cualquier función distancia.*

3.10 Resumen

En este capítulo hemos descrito las características y funcionamiento de los métodos de clustering, haciendo especial énfasis en los algoritmos *k-means* (MacQueen, 1967) y *fuzzy c-means* (Bezdek, 1981). Se han enumerado las principales funciones distancia utilizadas más comúnmente en este tipo de procedimientos (3.4) así como las principales medidas de disimilitud existente entre conjunto de datos y centroides (3.9).

A continuación se ha propuesto un nuevo algoritmo denominado Fuzzy Ordered C-Means (FOCM) (véase 3.7), basado en el algoritmo fuzzy c-means (véase 3.6), mediante el cual podremos realizar procesos de clustering sobre conjunto de datos ordenados. Nuestro algoritmo FOCM aplica las denominadas funciones de vecindad (3.7.1) en el cálculo de los coeficientes de pertenencia de la matriz de partición fuzzy de la secuencia inicial; de esta manera se introduce la concordancia ordinal entre secuencia de datos y secuencia de centroides en el proceso de realización del agrupamiento *fuzzy*, quedando los elementos iniciales de la secuencia de datos más relacionados con los elementos iniciales de la secuencia de centroides, y al mismo tiempo los elementos finales de la secuencia de datos más relacionados con los elementos finales de la secuencia de centroides. El grado de dependencia con el orden y el comportamiento del algoritmo se verán fuertemente determinados por el tipo y configuración de función de vecindad empleada.

Una vez realizada la partición y establecidos los coeficientes de pertenencia de cada elemento de la secuencia de datos a cada uno de los centroides[6] podemos definir una medida de la disimilitud entre secuencia de datos inicial y centroides iniciales, en la cual acumularemos las distintas distancias existentes entre cada elemento de la secuencia inicial y la secuencia inicial de centroides, ponderada por su coeficiente de pertenencia anteriormente calculado (3.9.1 y 3.9.2). Con estas definiciones podremos comparar dos secuencias cualesquiera de datos ordenados, aunque contengan distinto número de elementos: estableceremos como secuencia de datos a la secuencia que contenga un número mayor de elementos; al mismo tiempo inicializaremos la secuencia de centroides con los elementos de la secuencia que tenga un menor número de elementos; seleccionaremos una determinada función de vecindad para la aplicación del algoritmo FOCM, y por último calcularemos la disimilitud utilizando los coeficientes finales.

[6]Recuerde el lector que al tratarse de una partición de tipo *fuzzy*, un elemento de la secuencia inicial puede pertenecer al mismo tiempo a varios centroides, siempre que cumpla con la condición de convergencia que exige que la suma de los coeficientes de pertenencia de un elemento cualquiera a cada uno de los centroides ha de ser igual a uno (3.48).

En el transcurso de cada iteración tanto del algoritmo FCM como del FOCM, los centroides van actualizando sus posiciones hasta la convergencia final en la que una determinada función objetivo ha sido minimizada (3.41). En el caso del FOCM, la secuencia de centroides puede sufrir incluso un *giro* para acomodar su centroides iniciales y finales con los elementos iniciales y finales de la secuencia de datos, respectivamente. Dichas posiciones intermedias constituyen una *transición difusa simple*. Además, gracias al algoritmo FOCM hemos definido una medida de la disimilitud media ordenada entre el conjunto ordenado de centroides y el conjunto ordenado de datos (véase 3.76).

> *Hemos presentado un nuevo algoritmo, denominado Fuzzy Complete Transitions (FCT). Dicho algoritmo aplicará de forma recurrente el algoritmo FCM, pero añadiendo un nuevo centroide cada vez que se produzca la convergencia del FCT. Cuando el número de centroides sea igual al número de elementos, la función de vecindad se cambiará a la vecindad discreta (3.7.8) y las posiciones finales de los últimos centroides serán movidos exactamente a las posiciones de los elementos de la secuencia inicial, donde el algoritmo convergerá y finalizará. De esta manera, para dos secuencias de datos ordenados cualesquiera de distinto número de elementos, podemos generar una transición completa de la de menor número de elementos hacia la de mayor número de elementos sencillamente acumulando todos los estados intermedios arrojados por la aplicación de nuestro algoritmo FCT. Dichas posiciones intermedias constituyen una* transición difusa completa.

La aplicación de los algoritmos FOCM y FCT, así como el cálculo de la disimilitud media ordenada, puede llevarse a cabo sobre cualquier conjunto ordenado de centroides y conjunto ordenado de datos, siempre que ambos posean el mismo número de atributos o características que puedan ser parametrizados como números reales sobre un espacio métrico de dimensión finita.

Capítulo 4

Comparación y transiciones entre melodías, ritmos, armonías y timbres

«La función del creador es pasar por tamiz los elementos que recibe, porque es necesario que la actividad humana se imponga a sí misma sus límites. Cuanto más vigilado se halla el arte, más limitado y trabajado, más libre es.»

(Stravinski, 1983, pág. 86)

4.1 Introducción

Este capítulo se exponen las principales aportaciones al ámbito de la composición musical asistida por ordenador que se han llevado a término en la presente investigación. Utilizando las herramientas matemáticas y algorítmicas propuestas en el capítulo anterior, se describirán nuevos procesos para la generación de transiciones y variaciones musicales ya sean de carácter melódico, rítmico, armónico o tímbrico. La implementación computacional realizada de estas técnicas ha cristalizado en el software MERCURY, que se muestra a lo largo del capítulo como una versátil y novedosa herramienta informática capaz de enriquecer el extenso catálogo de software destinado a asistir al compositor musical durante su arduo

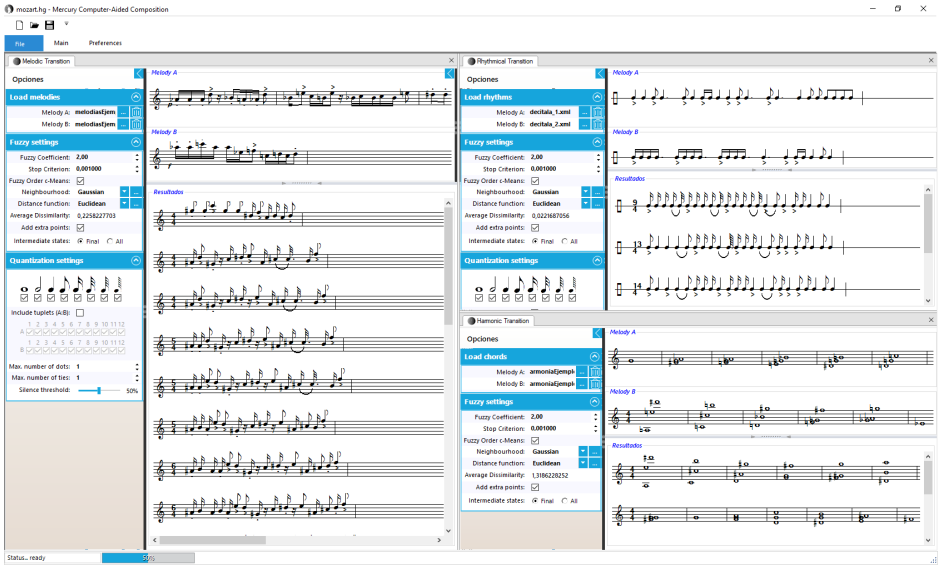
proceso creativo.
El procedimiento que seguiremos será el siguiente: en primer lugar nos centraremos en el ámbito melódico, formulando matemáticamente los atributos propios de la melodía como una secuencia ordenada de notas musicales de tal forma que sea posible su inclusión en los algoritmos FCM, FOCM y FCT (véase 3.6, 3.7, 3.8.1). A continuación, estudiaremos la disimilitud melódica gracias a la aplicación sobre la melodía de las definiciones de disimilitud media y disimilitud media ordenada, propuestas en el capítulo anterior (véase 3.9.1 y 3.9.2), ejemplificando su utilización y mostrando los diferentes resultados de disimilitud obtenidos en varios experimentos realizados con nuestro software MERCURY. Por último, nos centraremos en las transiciones melódicas, simples y completas, generadas con el algoritmo FOCM aplicado sobre dos melodías, así como en las transiciones melódicas completas obtenidas mediante la aplicación del algoritmo FCT. Ejemplificaremos el funcionamiento de estas transiciones mostrando los resultados computacionales procedentes de varios experimentos realizados en MERCURY con distintas líneas melódicas iniciales y finales, funciones de vecindad, funciones distancia y demás parámetros configurables.
A continuación extenderemos el proceso expuesto anteriormente sobre el resto de características musicales incluidas en nuestra investigación como son el ritmo, la armonía y finalmente el timbre. Definiremos en cada caso una disimilitud y mediante el algoritmo FCT generaremos transiciones rítmicas, armónicas y tímbricas, mostrándose los resultados computacionales de varios experimentos realizados mediante MERCURY.

Este capítulo pone de manifiesto las posibilidades de las técnicas de clustering definidas en el capítulo anterior aplicadas al ámbito de la composición musical asistida por ordenador. La implementación computacional de dichas técnicas, tanto para el cálculo de la disimilitud como para la generación de las transiciones musicales ha sido realizada mediante la programación completa del software MERCURY*, diseñado sobre los lenguajes de programación* C# *(para la parte de cálculo) y* VB.NET *(para la interfaz de usuario). Estas técnicas se muestran como una poderosa herramienta para la generación algorítmica de variaciones y transiciones de material musical preexistente, ya sean de carácter melódico, rítmico, armónico o tímbrico; proporcionando por tanto al compositor una fuente casi inagotable de ideas musicales con las que enriquecer su producción musical.*

4.2 Implementación computacional con Mercury®

4.2.1 Descripción del software Mercury

Presentamos en esta sección nuestro software MERCURY®, un nuevo programa orientado a la composición musical asistida por ordenador. MERCURY implementa los algoritmos descritos en los capítulos anteriores y es capaz de generar cuatro[1] tipos básicos de transiciones: melódicas, rítmicas, armónicas o tímbricas, desde una secuencia de material musical B hasta una secuencia A, de distinto número de elementos, proporcionadas por el usuario. Para este propósito, MERCURY implementa los algoritmos FCM, FOCM y FCT y es capaz de extraer las características musicales necesarias con las que representar los datos. Una vez que el usuario ha decidido con qué tipo de material inicial va a trabajar, ha configurado los parámetros y experimentado con los resultados obtenidos, el material musical generado por el programa puede ser fácilmente exportado a un archivo MusicXML que puede ser importado a cualquier programa de notación musical (e.g. *MuseScore*, *Finale*, *Sibelius*, etc.) donde el compositor puede continuar con su labor creativa (Martínez y Liern, 2019).



[1] Veremos en el capítulo siguiente que MERCURY también tiene otras funcionalidades relacionadas con la comparación de sistemas de afinación y transiciones entre los mismos, además de permitir el análisis FFT y detección de frecuencia fundamental desde archivos de audio mediante algoritmos como el *Harmonic Product Spectrum* o *Cepstrum*, tal y como se explicará en el siguiente capítulo.

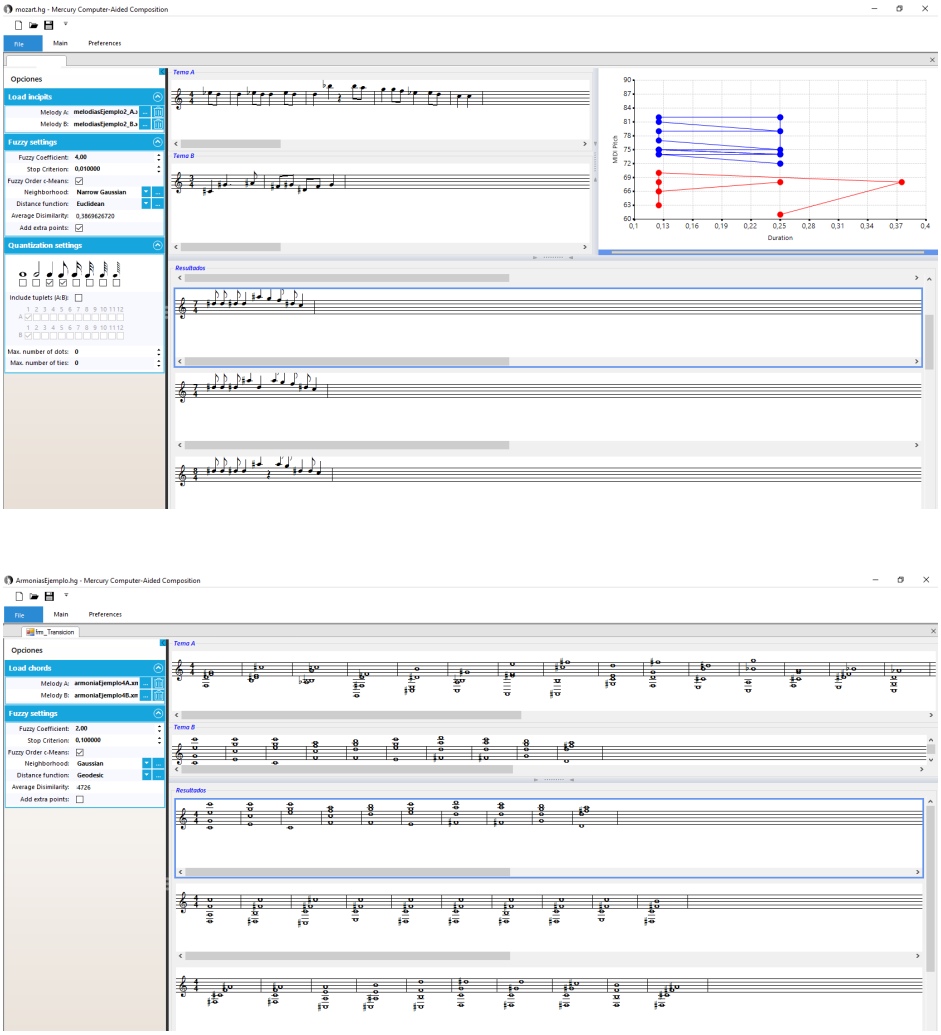


Figura 4.1: Capturas de pantalla del programa MERCURY.

4.2.2 Estructura del programa

Mercury se compone fundamentalmente de cuatro módulos independientes: el módulo de entrada y salida de datos, el módulo de cálculo, el módulo de la interfaz de usuario, y la librería de notación musical gráfica. Estos módulos se comunican entre sí mediante un conjunto de clases de dominio capaces de representar y compartir toda la información musical simbólica necesaria para la música (Martínez y Liern, 2019).

Módulo de I/O

Mercury recibe como parámetro de entrada partituras musicales en formato MusicXML, que pueden ser fácilmente generadas mediante cualquier programa de edición de partituras (*Musescore*©, *Sibelius*©, *Finale*©, *music21*©, etc.). El programa analiza la información incluida en la estructura de etiquetas del formato XML y lo traduce en objetos musicales de la capa de dominio. Una vez que la notación musical ha sido codificada, los algoritmos trabajan con información representada en vectores y matrices con las que se realizarán los cálculos matemáticos necesarios para obtener los valores de disimilitud y generar las transiciones entre distintos materiales musicales. En esta versión del software, la melodía está restringida a líneas monofónicas escritas únicamente en un pentagrama y una capa, sin ningún tipo de restricción en términos de duración, patrones rítmicos, articulaciones, dinámicas, silencios, etc. En el caso de las transiciones armónicas, la única restricción reside en el hecho de que las armonías inicial y final tienen que poseer el mismo número de voces, sin que exista ningún límite en el número de éstas o duración de la secuencia armónica. Por último, en el caso de las estructuras rítmicas, deben introducirse escritas en un único pentagrama y capa; el programa en este caso omitirá toda la información relativa a las alturas de las notas (Martínez y Liern, 2019).

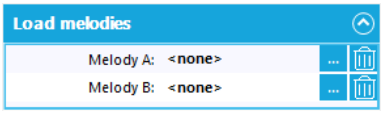


Figura 4.2: Menú para cargar dos melodías.

Todos los resultados generados por Mercury pueden ser fácilmente exportados a un fichero MusicXML, por lo que el usuario-compositor tiene la posibilidad de utilizarlos como material musical en su proceso compositivo. Además el programa permite escuchar directamente desde el programa, a través de la reproducción

MIDI, las melodías, acordes o ritmos generados para evaluar el material musical y filtrar los resultados obtenidos, seleccionando únicamente aquellos que encajen mejor con sus criterios compositivos. Los ejemplos musicales mostrados en el presente capítulo han sido generados en su totalidad con MERCURY, siguiendo este proceso descrito, y maquetados finalmente mediante un programa editor de partituras para su posterior exportación gráfica como formato vectorial *eps*.

Módulo de cálculo

En el módulo de cálculo, MERCURY implementa en el lenguaje de programación *C#* los algoritmos FCM, FOCM y FCT, así como numerosas funciones de vecindad y funciones distancia. En el Apéndice I encontramos el código relativo a estos cálculos. El proceso básico es el siguiente, una vez que las dos melodías, patrones rítmicos, secuencias armónicas o espectros tímbricos han sido representado en un espacio métrico $\mathbb{R}^q$, el algoritmo inicializará el conjunto de centroides con la secuencia que se desea modificar (*sequence B*). La secuencia objetivo (*sequence A*) se asigna a los elementos del conjunto de datos a *particionar*. Sobre este planteamiento, se aplica el algoritmo seleccionado, ya sea FCM, FOCM o FCT, configurado con los diversos parámetros de coeficiente *fuzzyness* λ, función de vecindad, función distancia y criterio de parada (Martínez y Liern, 2019).

El módulo de interfaz de usuario

La interfaz de usuario se ha desarrollado en VB.NET, utilizando la librería gratuita de controles gráficos *Syncfusion* sobre el sistema operativo *Windows* con la tecnología *.NET Framework 4.5*. El formulario principal permite al usuario crear, editar y guardar proyectos en un archivo de extensión *.hg*. Cada proyecto puede incluir un número ilimitado de formularios en forma de pestaña, que almacenan toda la configuración y transiciones obtenidas por el usuario. La información musical puede ser movida o compartida entre pestañas mediante *copy/paste* de tal forma que el usuario tiene una herramienta potente y flexible para realizar sus experimentos musicales. Además el usuario puede seleccionar la configuración para la reproducción MIDI, estableciendo los parámetros de *tempo* y programa, o seleccionando cualquier otro dispositivo MIDI conectado al ordenador para la reproducción del material musical, como por ejemplo un piano electrónico.

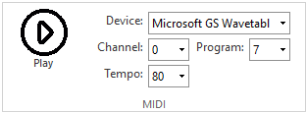


Figura 4.3: Menú para los ajustes MIDI.

El algoritmo FCT requiere que el usuario establezca diversos parámetros que se encuentran directamente relacionados con los resultados musicales obtenidos. Para ello Mercury cuenta con el menú de ajustes *fuzzy*, que permite controlar el coeficiente de *fuzzyness* λ, el criterio de parada del algoritmo. Los valores demasiado altos de dicho criterio de parada producirán que el algoritmo finalice sin la convergencia, mientras que valores demasiado bajos ralentizarán el proceso y requerirán de un mayor uso de memoria.

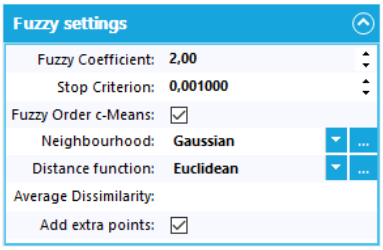


Figura 4.4: Menú de ajustes *fuzzy*.

El menú ofrece también al usuario la posibilidad de elegir entre varias funciones de vecindad (tres de la familia gaussiana, tres de la familia exponencial, cuatro triangulares, una rectangular, una trapezoidal y una sigmoidal), y también la posibilidad de elegir entre varias funciones distancia (Euclidiana, Euclidiana media, Manhattan, Manhattan media, Minkowski, Chebyshev, Acorde, Geodésica, Métrica de Canberra, Índice de divergencia y Métrica discreta). Cada una de estas funciones de vecindad o funciones distancia proporcionará estados intermedios muy diferentes. Las múltiples combinaciones de todos los valores del coeficiente de *fuzzyness*, todas las funciones de vecindad y todas las funciones de distancia brinda al usuario una gran cantidad de posibilidades para explorar y buscar el material musical deseado (Martínez y Liern, 2019).

La librería de notación musical gráfica

El módulo utilizado para mostrar de manera gráfica la música está basada la librería *PSALMControlLibrary* (*Polish System for Archivising Music Control Library*), un control gráfico de código abierto desarrollado para *.NET Framework* en 2010 por *Jacek Salamon*. Ha sido fuertemente modificado para cumplir con los requisitos de nuestro programa. En la siguiente figura podemos ver tres ejemplos de una melodía, una secuencia armónica y un patrón rítmico dibujados a través de la librería gráfica.

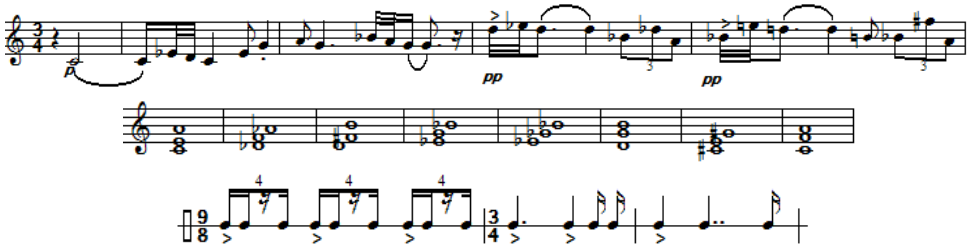


Figura 4.5: Una melodía, una secuencia armónica y un patrón rítmico mostrados en la librería de notación musical gráfica de Mercury (Martínez y Liern, 2019)

.

4.2.3 La Cuantización

Una vez que Mercury ha ejecutado cualquiera de sus algoritmos, ya sea FCM, FOCM o FCT, es necesario establecer un criterio para determinar la equivalencia entre los números reales generados por el algoritmo FCT y la notación musical simbólica más cercana que los representa. Este proceso, consistente en encontrar la notación musical que se corresponda con una serie de valores numéricos parametrizados, no es en absoluto trivial, ya que no existe una sencilla biyección que nos permita relacionar unívocamente cada q-tupla de valores de $\mathbb{R}^q$ en unas determinadas características musicales que puedan ser expresadas en términos de nuestra notación convencional. Nótese que realmente la notación musical tradicional de la música occidental nos proporciona una relación de tipo sobreyectiva, es decir, muchas figuraciones musicales distintas pueden generar la misma q-tupla de valores que en $\mathbb{R}^q$ la representan. Sucede entonces que para realizar el camino contrario, es decir, para asignar a cada q-tupla con una determinada figura musical, tendremos que realizar aproximaciones; es decir un proceso de *cuantización* (Martínez y Liern, 2019). El proceso de cuantificación será necesario para reasignar una característica de la notación musical tradicional a cada q-tupla de valores

contenida en cada una de las m notas de los l estados intermedios que arroja el algoritmo. En el caso del atributo de *MIDI pitch*, las notas se eligen de manera enarmónica mediante la selección del valor entero de MIDI pitch más cercano. De manera similar, la intensidad, representada musicalmente con los siguientes símbolos de *ppp*, *pp*, *p*, *mp*, *mf*, *f*, *ff*, y *fff*, se aproximarán al valor de velocidad MIDI más cercano de los símbolos anteriores (véase Selfridge-Field, 1997).

En el caso del coeficiente de duración, existe una amplia variedad de posibles notaciones rítmicas simbólicas cuyo coeficiente de duración es exactamente el mismo. Para poder aproximar el resultado numérico obtenido para el atributo de duración a su notación simbólica más cercana, el usuario debe especificar cuál de las posibles combinaciones de duraciones (redonda, blanca, negra, corchea, semicorchea, fusa, semifusa), número máximo de posibles notas ligadas, número máximo de posibles puntillos y tipos de grupos irregulares (3: 2, 5: 4, 7: 4, etc.). Para lograr esto, Mercury ofrece el *Menú de configuración de cuantización*. La siguiente figura muestra un ejemplo de cuantización definido por el usuario:

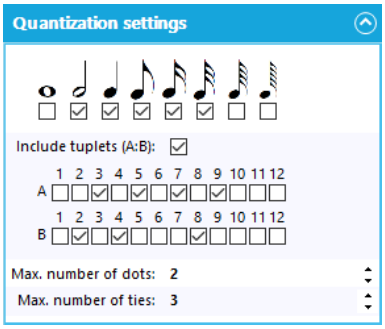


Figura 4.6: Ejemplo de cuantización establecida por el usuario.

Una vez establecida la configuración, Mercury calcula el coeficiente de duración para todas las combinaciones posibles permitidas, eliminando las repetidas, clasificando los resultados finales por un orden creciente y almacenándolos en memoria RAM para el caso en que la configuración de cuantificación no cambie durante los siguientes cálculos. Cuando el algoritmo ha finalizado y el programa necesita buscar la notación rítmica simbólica más adecuada, para cada nota elegirá de esta lista la notación rítmica con coeficiente de duración más cercano al valor numérico del coeficiente de duración calculado por el algoritmo (Martínez y Liern, 2019).

4.3 Parametrización de las características musicales

4.3.1 La altura

Como se propone en (León y Liern, 2012, pág. 454), la distancia en cents entre dos notas cuyas frecuencias son f_1 and f_2 se puede calcular mediante la siguiente expresión

$$d(f_1, f_2) = 1200 \times \left| \log_2 \left(\frac{f_1}{f_2} \right) \right| \text{ cents} \tag{4.1}$$

De acuerdo con el protocolo MIDI (Selfridge-Field, 1997), la altura de una nota se define mediante un número entero comprendido en el intervalo $[0, 127]$ (*MIDI-Pitch*), siendo el valor del Do_4 central[2] igual a 60 y el La_4 central igual a 69. Para el temperamento igual de 12 notas por octava, existe una diferencia exacta de 100 cents entre dos notas separadas por un número de *MIDI-pitch* (semitono). Si el diapasón correspondiente a la frecuencia f_{A_4} (usualmente establecido en $440Hz$) se corresponde al número de *midi-pitch* 69 entonces el número de *midi-pitch* de cualquier frecuencia f es

$$\nu = 69 + 12 \log_2 \Big(\frac{f}{f_{A_4}} \Big) \tag{4.2}$$

Para parametrizar la frecuencia de una nota utilizaremos en lo sucesivo el valor de su correspondiente número MIDI Pitch, con lo que la representación simbólica de la misma está garantizada a partir de una frecuencia de referencia f_{A_4}. Nada impide que este valor de MIDI Pitch tenga valores no enteros, permitiendo de esta forma la representación de música con un número de notas por octava diferente de doce, o la caracterización de músicas relativas a sistemas de afinación distintos al temperamento igual de doce notas por octava.

[2]Según el índice acústica científico. Otros índices acústicos establecen el do central como Do_5 (Sistema de Riemann), Do_3 (sistema francés), c' (nomenclatura Oxford), etc. (Latham, 2009, pág. 20).

4.3.2 La duración

Estableciendo la figura de la redonda como la unidad, es fácil definir un coeficiente de duración $\delta \in \mathbb{R}$. de una nota cualquiera, por complicada que sea su figuración rítmica. La figura de cuadrada tendrá coeficiente 2, la figura de la redonda 1, una blanca coeficiente 1/2, una negra 1/4, una corchea 1/8, semicorchea 1/16, fusa 1/32, semifusa 1/64, garrapatea 1/128. Como cada figura se encuentra en relación de dos con respecto a sus duraciones anteriores y posteriores, dicha relación puede ser generalizada como una sencilla potencia de dos

$$\alpha = \frac{1}{2^a}, \quad -1 \leq a \leq 7 \tag{4.3}$$

En la siguiente tabla podemos observar los diferentes valores que toma el parámetro a y el valor α para cada figura musical

Figura	a	α
cuadrada	-1	2
redonda	0	1
blanca	1	0.5
negra	2	0.25
corchea	3	0.125
semicorchea	4	0.0625
fusa	5	0.03125
semifusa	6	0,015625
garrapatea	7	0,0078125

Además, la presencia de cada puntillo multiplica este coeficiente de duración por el factor

$$\beta = \sum_{k=0}^{b} \frac{1}{2^k} = \frac{2^{b+1} - 1}{2^b}, \tag{4.4}$$

donde b es el número total de puntillos de la nota. Por otra parte, los grupillos o valores irregulares (tresillos, quintillos, etc.) pueden interpretarse como «léanse c notas donde caben d» y usualmente son denotados mediante la expresión $c : d$, modificando por tanto el coeficiente de duración de cada una de sus notas por el factor

$$\gamma = \frac{d}{c} \tag{4.5}$$

Teniendo en cuenta las expresiones (4.3), (4.4) y (4.5), para un número total τ de notas ligadas, el coeficiente total de duración tendrá la siguiente expresión

$$\delta = \sum_{i=1}^{\tau} (\alpha_i \cdot \beta_i \cdot \gamma_i) = \sum_{i=1}^{\tau} \left[\frac{1}{2^{a_i}} \cdot \frac{2^{b_i+1} - 1}{2^{b_i}} \cdot \frac{d_i}{c_i} \right] \tag{4.6}$$

4.3.3 La intensidad

En el protocolo MIDI (Selfridge-Field, 1997), la intensidad de una nota puede ser caracterizada mediante un valor numérico comprendido entre 1 y 127 denominado *key velocity*. Muchos teclados electrónicos miden la velocidad con la que una nota se presiona y realizan una transformación logarítmica para abarcar todo el posible rango dinámico, desde el *pianissimo* hasta el *fortissimo*. El número 0 se reserva para el mensaje de *Note Off*. El valor por defecto, para instrumentos sin capacidad de medir la intensidad de la nota, es el 64. En la figura 4.7 podemos ver como se distribuyen los valores de *key velocity* para las distintas intensidades musicales.

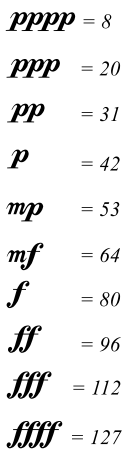


Figura 4.7: Diferentes valores de *key velocity* para dinámica musical.

En lo sucesivo, la intensidad musical será representada mediante el número de key velocity, comprendido entre 0 y 127, de acuerdo con la tabla de valores representada en la ilustración 4.7.

4.3.4 El silencio

El silencio es el elemento esencial de la música capaz de moldear las microestructuras rítmicas, temáticas o melódicas básicas que a su vez, mediante un proceso de condensación y acumulación, construyen el discurso musical de una obra. Resulta indispensable introducir una descripción matemática del silencio para incorporarlo en la parametrización de las características musicales que estamos realizando en este punto. Existen diferentes aproximaciones, no todas ellas igual de válidas, para realizar esto: en un primer lugar podemos considerar el silencio como una nota de duración δ pero con un valor de intensidad (*key velocity*) igual a cero; también existe la posibilidad de considerar un silencio como una nota con valor de frecuencia nula y por tanto MIDI pitch igual a cero; combinando ambas opciones, podríamos considerar un silencio como una nota con intensidad y frecuencia iguales a cero. Sin embargo todas estas opciones conllevan problemas en los algoritmos de clustering derivados del hecho de tener un punto con valor cero rodeado de otros puntos con valores distintos a cero; el algoritmo modificará hacia las frecuencias graves o hacia las notas de baja intensidad aquellas notas cercanas a las notas que sean silencio, ocasionando por tanto unos resultados indeseados desde el punto de vista musical, aunque perfectamente válidos desde el punto de vista del clustering. Necesitamos otra manera de representar los silencios que no afecte a los resultados, o que minimice su impacto en éstos.

*La propuesta que se realiza para la resolución de este problema proporciona unos resultados óptimos y será utilizada en lo sucesivo: se introducirá una nueva dimensión exclusivamente utilizada para caracterizar si una nota con coeficiente de duración δ es silencio o no. Esta dimensión será de tipo escalar y valdrá 0 cuando la nota no sea silencio, y 1 cuando la nota sea silencio. A continuación tendremos que establecer un valor ficticio para aquellos atributos musicales que están presentes en una nota pero que no existen en un silencio, como por ejemplo la frecuencia (*MIDI pitch*)o la intensidad (key velocity). La técnica propuesta aquí para la asignación de dicho valor ficticio consiste en realizar el valor medio del atributo calculado con los valores presentes en la nota inmediatamente anterior y la nota inmediatamente posterior.*

4.4 La melodía

Una nota musical puede determinarse mediante k características (e.g. altura, intensidad, duración, timbre, etc.) y por tanto puede ser expresada como un vector en $\mathbb{R}^q$, donde $q \leq d$. La manera mas sencilla de representar un nota musical consiste en establecer $q = 3$ y seleccionar los parámetros de altura, duración y silencio. Es posible trabajar con un número mayor de dimensiones de tal forma que podamos representar con mejor precisión la naturaleza de una nota musical, teniendo en cuenta parámetros musicales adicionales como la intensidad o el ataque. Podríamos introducir características adicionales para trabajar con géneros musicales alejados de la tradición cultural occidental, o que requieran más información como por ejemplo la música electroacústica, mediante la asimilación de dimensiones extra.

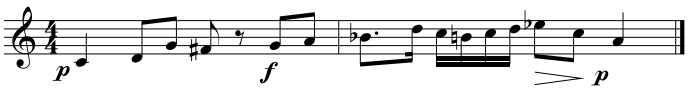
4.4.1 Definición de melodía

Definición 4.4.1 Nota musical, (Martínez y Liern, 2017). *Una nota musical es una q-tupla $x \in \mathbb{R}^q$ definida mediante q características musicales.*

La forma más conveniente de definir una nota musical es estableciendo $q = 4$ y asimilando a cada dimensión uno de los siguientes observables: MIDI pitch $[1, 127]$ (véase 4.2), coeficiente de duración $]0, \infty[$ (véase 4.6), key velocity $[1, 127]$, silencio $[0, 1]$. Una vez que hemos establecido las q características necesarias para definir el concepto de nota musical, podemos definir el concepto de melodía como una sucesión ordenada de n notas, siendo cada nota un vector en el espacio métrico q-dimensional $\mathbb{R}^q$.

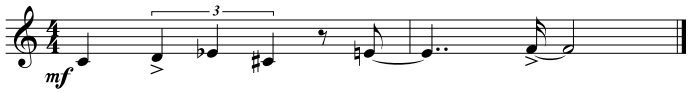
Definición 4.4.2 Melodía, (Martínez y Liern, 2017). *Una melodía $\mathscr{M}$ es una secuencia ordenada $\mathscr{M} = \{x_i\}_{i=1}^n$, donde cada $x_i \in \mathbb{R}^q$ es una nota musical definida mediante q características.*

Ejemplo 4.4.1 *Considérese la siguiente melodía y represéntese según la definición anterior en la que se describe cada nota como un conjunto de estos cuatro observables: coeficiente de duración (δ), MIDI pitch, silencio y key velocity.*



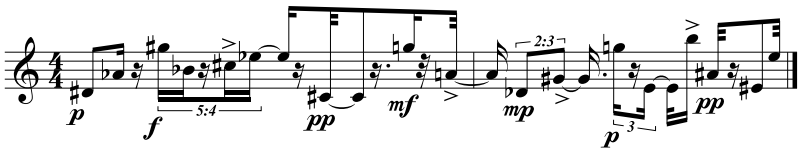
Nota	Coef. duración (δ)	MIDI Pitch	Silencio	Key velocity
1	0,2500	60	0	42
2	0,1250	62	0	42
3	0,1250	67	0	42
4	0,1250	66	0	42
5	0,1250	66,5	1	61
6	0,1250	67	0	80
7	0,1250	69	0	80
8	0,1875	70	0	80
9	0,0625	74	0	80
10	0,0625	72	0	80
11	0,0625	71	0	80
12	0,0625	72	0	80
13	0,0625	74	0	80
14	0,1250	75	0	80
15	0,1250	72	0	61
16	0,2500	69	0	42

Ejemplo 4.4.2 *Considérese la siguiente melodía y represéntese según la definición anterior en la que se describe cada nota como un conjunto de estos cuatro observables: coeficiente de duración (δ), MIDI pitch, silencio y key velocity.*



Nota	Coef. duración (δ)	MIDI Pitch	Silencio	Key velocity
1	0.2500	60	0	64
2	0,1667	62	0	79
3	0,1667	63	0	64
4	0,1667	61	0	64
5	0,1250	62,5	1	64
6	0,5625	64	0	64
7	0,5625	65	0	79

Ejemplo 4.4.3 *Considérese la siguiente melodía y represéntese según la definición anterior en la que se describe cada nota como un conjunto de estos cuatro observables: coeficiente de duración (δ), MIDI pitch, silencio y key velocity.*



Nota	Coef. duración (δ)	MIDI Pitch	Silencio	Key velocity
1	0,12500	63	0	42
2	0,06250	68	0	42
3	0,06250	74	1	61
4	0,05000	80	0	80
5	0,05000	70	0	80
6	0,05000	71,5	1	80
7	0,05000	73	0	100
8	0,11250	75	0	80
9	0,06250	68	1	56
10	0,15625	61	0	31
11	0,09375	70	1	48
12	0,06250	79	0	64
13	0,03125	74	1	72
14	0,09375	69	0	80
15	0,18750	61	0	53
16	0,28125	68	0	66
17	0,04167	79	0	42
18	0,04167	71,5	1	42
19	0,07292	64	0	42
20	0,06250	83	0	53
21	0,03125	70	0	31
22	0,06250	67,5	1	31
23	0,12500	65	0	31
24	0,03125	76	0	31

4.4.2 Melodías polifónicas

En la formulación matemática que estamos exponiendo, el caso de considerar melodías polifónicas no conduce directamente al problema de las *dimensiones vacías*. Una posible solución sería representar las diferentes alturas de las notas como un vector $\bar{\nu} \in \mathbb{R}^k$, donde k es el Mínimo Común Múltiplo del número de voces que aparecen en la melodía. En la figura 4.8 aparecen notas con 1, 2 y 3 voces, entonces $k = \text{m.c.m.}(1, 2, 3) = 6$. Consecuentemente las alturas de las voces podrían ser expresadas mediante un número de 6 dimensiones, repitiendo adecuadamente las alturas en cada voz.

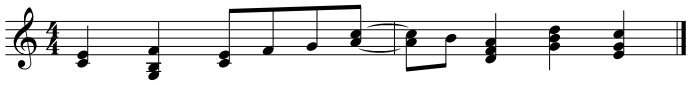


Figura 4.8: Ejemplo de melodía polifónica.

4.4.3 La disimilitud melódica

Comparación de dos melodías de igual número de notas

Si consideramos dos melodías $\mathscr{M}^A$ y $\mathscr{M}^B$, ambas pertenecientes a un espacio métrico q-dimensional, es posible medir una distancia entre ellas en el caso en el que tengan el mismo número de notas. Esto no significa necesariamente que las dos melodías tengan que tener la misma duración expresada en unidades de tiempo; significa que tienen que tener el mismo número de puntos. Para comparar estas dos melodías acumularemos la distancia parcial existente entre cada pareja de notas $\mathbf{x_i}$, $\mathbf{y_i}, i \geq 1$, respetando el orden establecido por la secuencia de notas de cada melodía.

Definición 4.4.3 Distancia media entre dos melodías, (Martínez y Liern, 2017). *Sean* $\mathscr{M}^A = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\}$ *y* $\mathscr{M}^B = \{\mathbf{y_1}, \ldots, \mathbf{y_n}\}$ *dos melodías, ambas con* n *notas, pertenecientes a un espacio métrico* q*-dimensional* $\mathbb{R}^q$*. Sea* $d : \mathbb{R}^q \times \mathbb{R}^q \to \mathbb{R}$ *una función distancia, la distancia media entre* $\mathscr{M}^A$ *and* $\mathscr{M}^B$ *se define como*

$$\bar{D}(\mathscr{M}^A, \mathscr{M}^B) = \frac{1}{n} \sum_{i=1}^{n} d(\mathbf{x_i}, \mathbf{y_i}). \tag{4.7}$$

La siguiente figura ejemplifica el cálculo de la distancia media entre dos melodías de cinco notas, representadas únicamente mediante las características de duración y MIDI pitch.

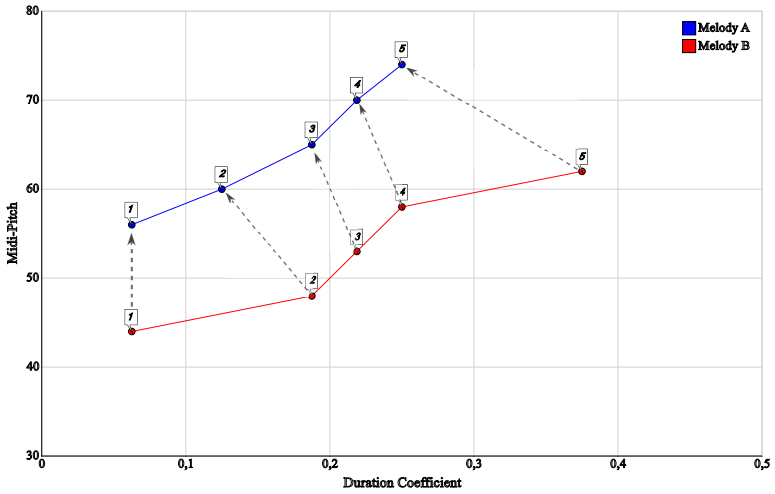


Figura 4.9: Ejemplo de cálculo de la distancia media entre dos melodías de cinco notas.

Si queremos comparar melodías de diferente número de notas, la ecuación 4.7 tiene que ser generalizada. Se propone la definición de una pseudodistancia basada en una partición de tipo *fuzzy*; realizaremos un *fuzzy clustering* entre las dos melodías $\mathscr{M}^A = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\}$ y $\mathscr{M}^B = \{\mathbf{y_1}, \ldots, \mathbf{y_m}\}$, siendo $n > m$, que nos permitirá estimar la distancia que separa a la melodía $\mathscr{M}^A$ de la melodía $\mathscr{M}^B$. A pesar de que esta medida no va a satisfacer los requisitos de una función distancia, nos podrá proporcionar información útil para establecer un grado de similitud o disimilitud entre ambas melodías. La manera de realizar este proceso de clustering será sencilla: la melodía con un mayor número de notas (caracterizada en los sucesivo por $\mathscr{M}^A$) se asimilará al conjunto de datos X sobre el que se va a realizar el particionado. La melodía con un menor número de notas (caracterizada en los sucesivo por $\mathscr{M}^B$) se asimilará a los centroides, haciendo $c = m$ e inicializando los centroides de los c clusters con las notas de $\mathscr{M}^B$. A continuación se realizará el proceso de *clustering* entre ambas.

Cuando se aplica un proceso de clustering de tipo hard (e.g. *k-means clustering*) sobre un conjunto de datos genérico X, el resultado es una partición de tipo *hard* (véase 3.3.1), el resultado es una partición discreta del conjunto de datos X en un total de c grupos, denominados *clusters*, de tal manera que cada elemento de X pertenece a uno y sólo uno de los clusters, tal y como podemos ver en la siguiente figura:

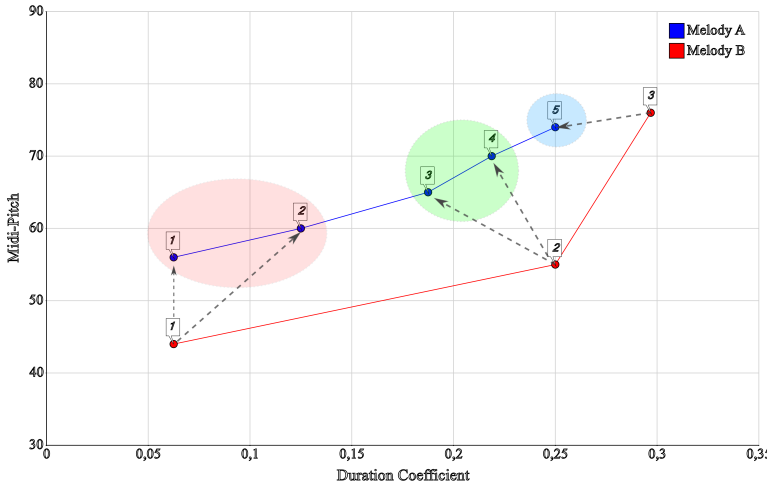


Figura 4.10: Ejemplo de partición de tipo hard sobre las cinco notas de la melodía A inicializando los tres centroides con las notas de la melodía B.

Sin embargo, para el proceso de comparación de melodías, utilizaremos particiones de tipo *fuzzy* en la cual el proceso de clustering proporcionará unos determinados coeficientes de pertenencia de cada elementos de X a cada cluster, de tal forma que se permite la pertenencia de un elemento de X a varios clusters simultáneamente. Esto nos conduce a aceptar las siguientes dos afirmaciones:

1. Asumimos que comparar cada nota de $\mathscr{M}^A$ solo con una nota de $\mathscr{M}^B$ no tiene sentido musical.

2. En el proceso de comparación melódica, el orden de la comparación es juega un papel clave. Las notas musicales se leen de forma secuenciada y por tanto su comparación, a pesar de realizarse mediante una aproximación vectorial, tiene que respetar esta secuencia.

Utilizaremos el fuzzy clustering para agrupar las n notas de la melodía $\mathscr{M}^A$ en m subconjuntos (tantos subconjuntos como notas tenga la melodía $\mathscr{M}^B$). Una vez este proceso ha finalizado, y tenemos la partición *fuzzy* de las notas de $\mathscr{M}^A$, seremos capaces de relacionar cada subconjunto de notas de $\mathscr{M}^A$ con una nota de $\mathscr{M}^B$ y finalmente, calcular una distancia media acumulando las distancias parciales de cada nota de $\mathscr{M}^A$ con cada nota de $\mathscr{M}^B$ ponderando dicha distancia parcial con su coeficiente de pertenencia.

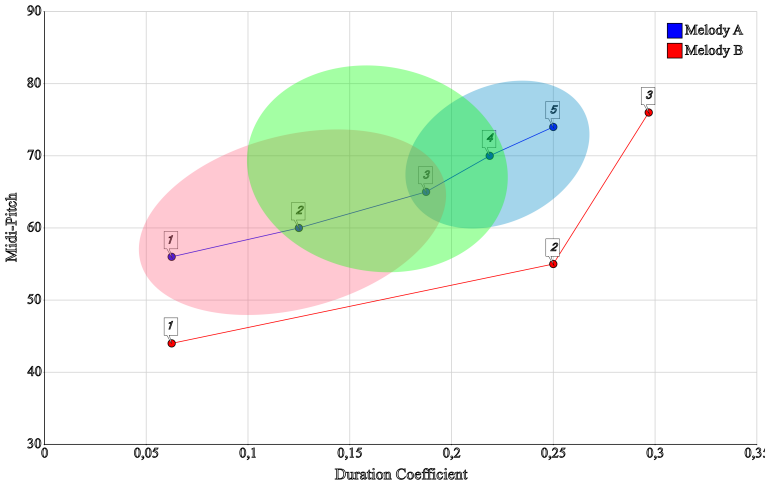


Figura 4.11: Ejemplo de partición fuzzy sobre las cinco notas de la melodía A inicializando los tres centroides con las notas de la melodía B.

Disimilitud melódica entre dos melodías de distinto número de notas

Consideremos a continuación dos melodías $\mathscr{M}^A$ y $\mathscr{M}^B$ con un número distinto de notas. Realicemos una partición *fuzzy* de las notas de $\mathscr{M}^A$ con los centroides iniciales proporcionados por las notas de $\mathscr{M}^B$, y apliquemos el algoritmo FCM (véase 3.6) l veces, hasta que el criterio de parada sea satisfecho. Una vez que el proceso de particionado ha sido completado, podemos utilizar la función de disimilitud presentada en 3.76 para definir una disimilitud entre $\mathscr{M}^A$ y $\mathscr{M}^B$ utilizando los coeficientes finales de pertenencia u_{ij} calculados en la matriz U y los centroides originales.

Definición 4.4.4 Disimilitud media entre dos melodías, (Martínez y Liern, 2017). *Sean* $\mathscr{M}^A = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\} \subset \mathbb{R}^q$ *y* $\mathscr{M}^B = \{\mathbf{y_1}, \ldots, \mathbf{y_m}\} \subset \mathbb{R}^q$ *dos melodías, donde* $n > m$. *Sea* $d : \mathbb{R}^q \times \mathbb{R}^q \to \mathbb{R}$ *una función distancia. Sean* u_{ij} *los coeficientes de pertenencia finales calculados con el algoritmo FCM. La disimilitud media* $\mathscr{D}$ *entre* $\mathscr{M}^A$ *y* $\mathscr{M}^B$ *se define como*

$$\mathscr{D}(\mathscr{M}^A, \mathscr{M}^B) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} u_{ij} \cdot d(\mathbf{x_i}, \mathbf{y_j}) \ . \tag{4.8}$$

Por construcción, $\mathscr{D}$ no considera el orden natural de la secuencia de notas existente en cada una de las melodías. De esta forma, la partición que el algoritmo FCM calcula no proporciona ningún peso especial a la distancia entre aquellas notas que se encuentran en posiciones más cercanas. Un ejemplo ilustrativo de este hecho se puede ver en la figura 4.12 en la que podemos ver tres melodías diferentes. Sin embargo, la melodía B es una retrogradación total de la melodía A; por tanto la disimilitud media entre $\mathscr{D}$ entre las melodías A y C será exactamente la misma que la disimilitud media entre las melodías B y C, resultado que no tiene sentido en términos musicales.

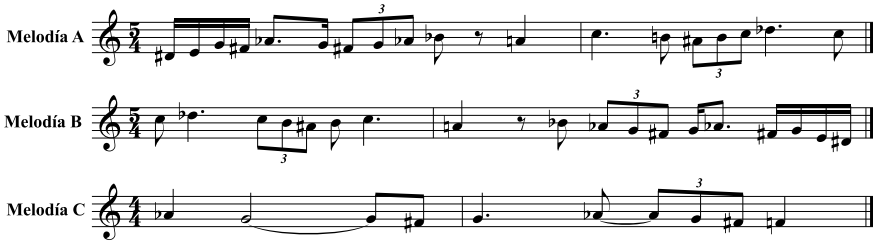


Figura 4.12: Tres melodías de ejemplo, donde la melodía B es una retrogradación de la melodía A.

$$\mathscr{D}(\mathscr{M}^A, \mathscr{M}^C) = 0{,}2677758444, \qquad \mathscr{D}(\mathscr{M}^B, \mathscr{M}^C) = 0{,}2677758444$$

Este ejemplo muestra una comparación de melodías sin tener en cuenta el orden de las notas, en el que se obtienen resultados alejados por completo de la lógica musical. Para acercarnos a la realidad de la música, es necesario introducir la dependencia con el orden dentro del método de estimación de la disimilitud. Tenemos que proporcionar mayores pesos a las notas que comparten posiciones cercanas dentro del orden de cada melodía, y reducir la contribución a la disimilitud total de aquellas parejas de notas cuyas posiciones son muy distintas, desde un punto de vista ordinal. Las funciones de vecindad introducidas en la sección 3.7.1 nos proporcionarán la información necesaria para introducir la dependencia con el orden y poder comparar secuencias de datos. Por tanto será necesario utilizar el algoritmo de *fuzzy clustering* FOCM (véase 3.7) para poder comparar dos melodías, teniendo en cuenta en esta comparación el orden de su secuencia de notas. Podemos utilizar la función de disimilitud presentada en 3.77 para definir una disimilitud melódica ordenada tal y como sigue a continuación.

Definición 4.4.5 Disimilitud media ordenada melódica, (Martínez y Liern, 2017). *Sean* $\mathscr{M}^A = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\} \in \mathbb{R}^q$ *y* $\mathscr{M}^B = \{\mathbf{y_1}, \ldots, \mathbf{y_m}\} \in \mathbb{R}^q$ *dos melodías de distinto número de notas. Sea* $d : \mathbb{R}^q \times \mathbb{R}^q \to \mathbb{R}$ *una función distancia. Sean* $\hat{u}_{ij}$ *los coeficientes de pertenencia finales calculados mediante el algoritmo FOCM de* $\mathscr{M}^B$ *sobre* $\mathscr{M}^A$*. La disimilitud media ordenada* $\widehat{\mathscr{D}}$ *de* $\mathscr{M}^A$ *sobre* $\mathscr{M}^B$ *se define como*

$$\widehat{\mathscr{D}}(\mathscr{M}^A, \mathscr{M}^B) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} \hat{u}_{ij} \cdot d(\mathbf{x_i}, \mathbf{y_j}). \tag{4.9}$$

Ejemplo 4.4.4 *Calcúlese la disimilitud media ordenada entre las siguientes dos melodías, utilizando diferentes funciones distancia y diferentes funciones de vecindad. Tómese el valor del grado de fuzzyness* $\lambda = 2$.

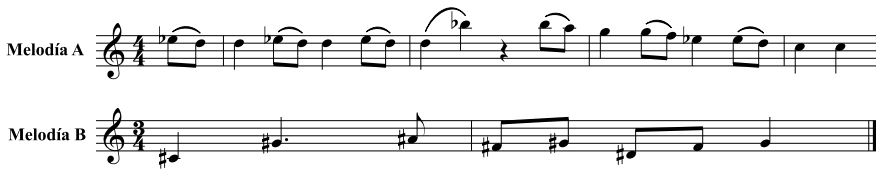


Figura 4.13: Melodías $\mathscr{M}^A$ y $\mathscr{M}^B$.

Tabla 4.1: Resultados del cálculo de la disimilitud media ordenada entre $\mathscr{M}^A$ y $\mathscr{M}^B$.

Func. Vecindad	Func. Distancia	$\widehat{\mathscr{D}}(\mathscr{M}^A, \mathscr{M}^B)$
Gaussiana	Euclidiana	1,2357124002
Gaussiana	Manhattan	1,2898533518
Gaussiana	Minkowski	1,2354807641
Exponencial	Euclidiana	1,2729243373
Exponencial	Manhattan	1,2895270583
Exponencial	Minkowski	1,2720456104
Triangular	Euclidiana	1,2514285494
Triangular	Manhattan	1,2700779102
Triangular	Minkowski	1,2509513433
Rectangular	Euclidiana	1,1997408204
Rectangular	Manhattan	1,2145868708
Rectangular	Minkowski	1,1985769688

4.4.4 Resultados computacionales: transiciones melódicas

Como hemos visto anteriormente, los algoritmos de clustering presentados además de permitirnos definir una medida de la disimilitud nos proporcionan una herramienta potente para generar nuevo material musical obtenido a partir de los estados intermedios que recorren los centroides. En esta sección veremos algunos ejemplos de su utilidad en la generación de transiciones melódicas.

Ejemplo 4.4.5 *Analicemos los estados intermedios por los que pasa* $\mathscr{M}^B$ *en cada una de las iteraciones del algoritmo FOCM, ejecutado en el ejemplo 4.4.4 con* $\lambda = 2$*, vecindad gaussiana y distancia euclidiana.*

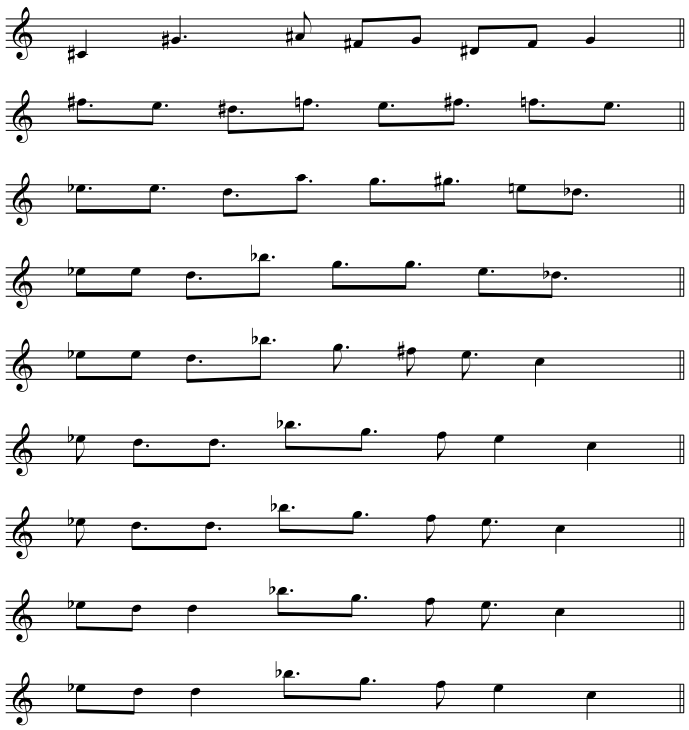


Figura 4.14: Estadios intermedios de $\mathscr{M}^B$ para cada iteración del algoritmo FOCM.

Podemos apreciar como la melodía $\mathscr{M}^B$ sufre un proceso de modulación acercándose a la melodía $\mathscr{M}^A$. Es especialmente notorio este hecho en el principio de la melodía mozartiana, con las dos notas *Re* repetidas después del *Mi*♭; así como en el salto de sexta menor tan característico producido entre las notas *Re* y *Si*♭.

Ejemplo 4.4.6 *Analicemos los estados intermedios finales, es decir, los estadios intermedios previos a la adición de un nuevo punto, por los que pasa $\mathscr{M}^B$ en una transición difusa completa hacia $\mathscr{M}^A$, ejecutado en el ejemplo 4.4.4 con $\lambda = 4$, vecindad gaussiana de tipo estrecha y métrica de tipo distancia media euclidiana.*

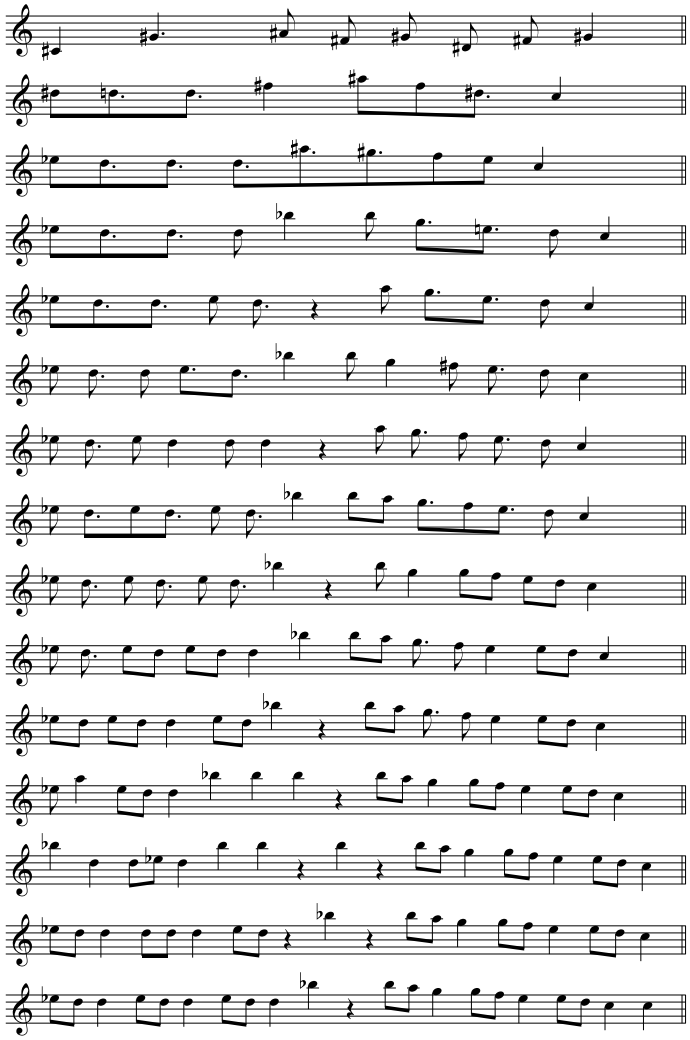


Figura 4.15: Estadios intermedios finales en la transición completa de $\mathscr{M}^B$ hacia $\mathscr{M}^A$.

Ejemplo 4.4.7 *Transición completa realizada mediante el algoritmo FCT entre la melodía A, procedente de 6 Kleine Klavierstücke Op.16, nº4 de Arnold Schoenberg, y la melodía B, procedente de Klavierstücke Op.33b del mismo autor, con los parámetros $\lambda = 2$, vecindad exponencial de tipo estrecha y métrica Manhattan.*

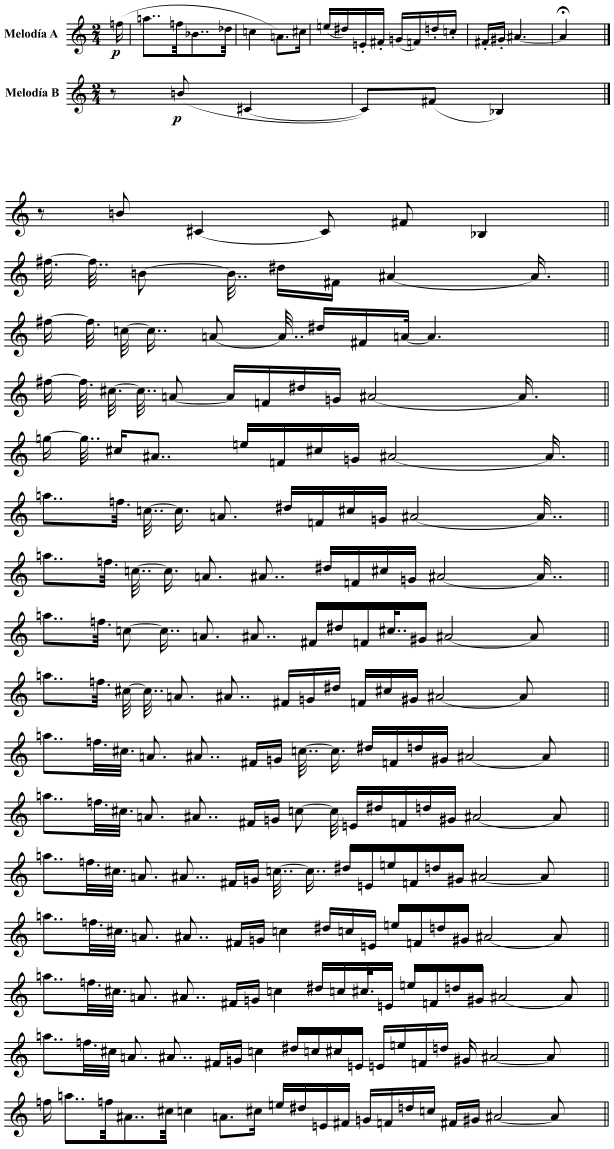
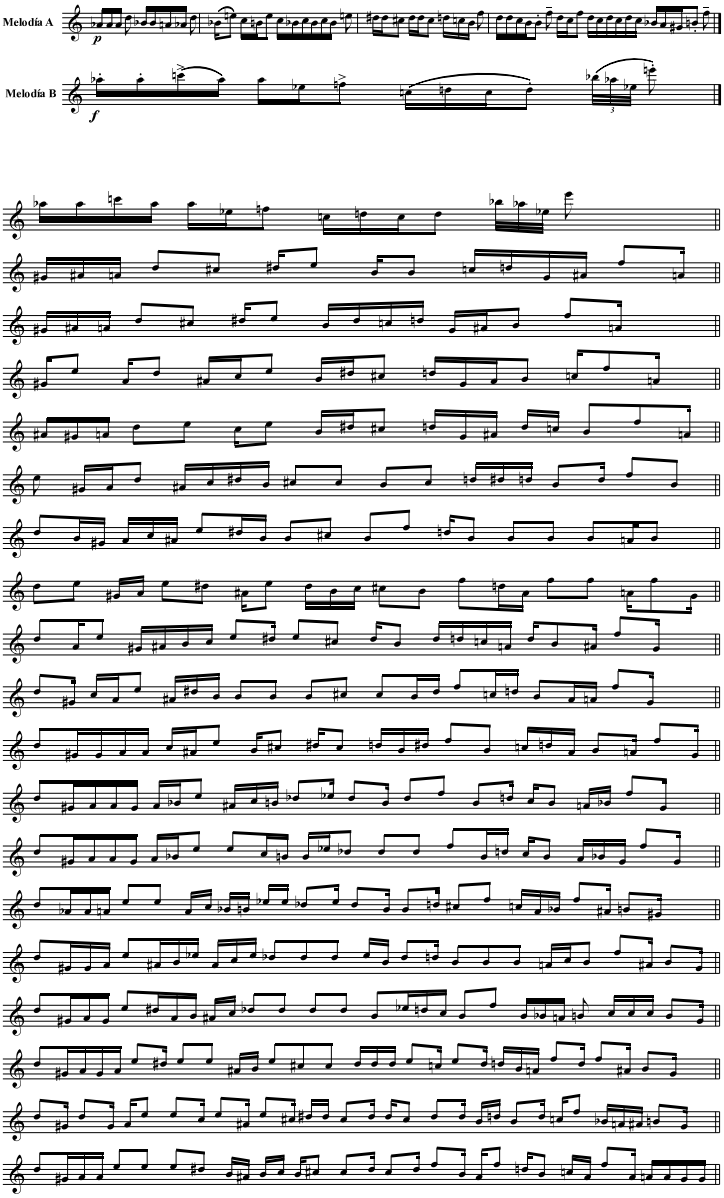


Figura 4.16: Estadios intermedios finales en la transición completa de $\mathscr{M}^B$ hacia $\mathscr{M}^A$.

Ejemplo 4.4.8 *Transición completa realizada mediante el algoritmo FCT entre dos cantos de pájaros transcritos por O. Messiaen (Messiaen, 1956). La melodía A se corresponde al canto de la alondra y la melodía B procede del canto del mirlo. Se han utilizado los parámetros $\lambda = 3$, vecindad gaussiana y métrica euclidiana.*



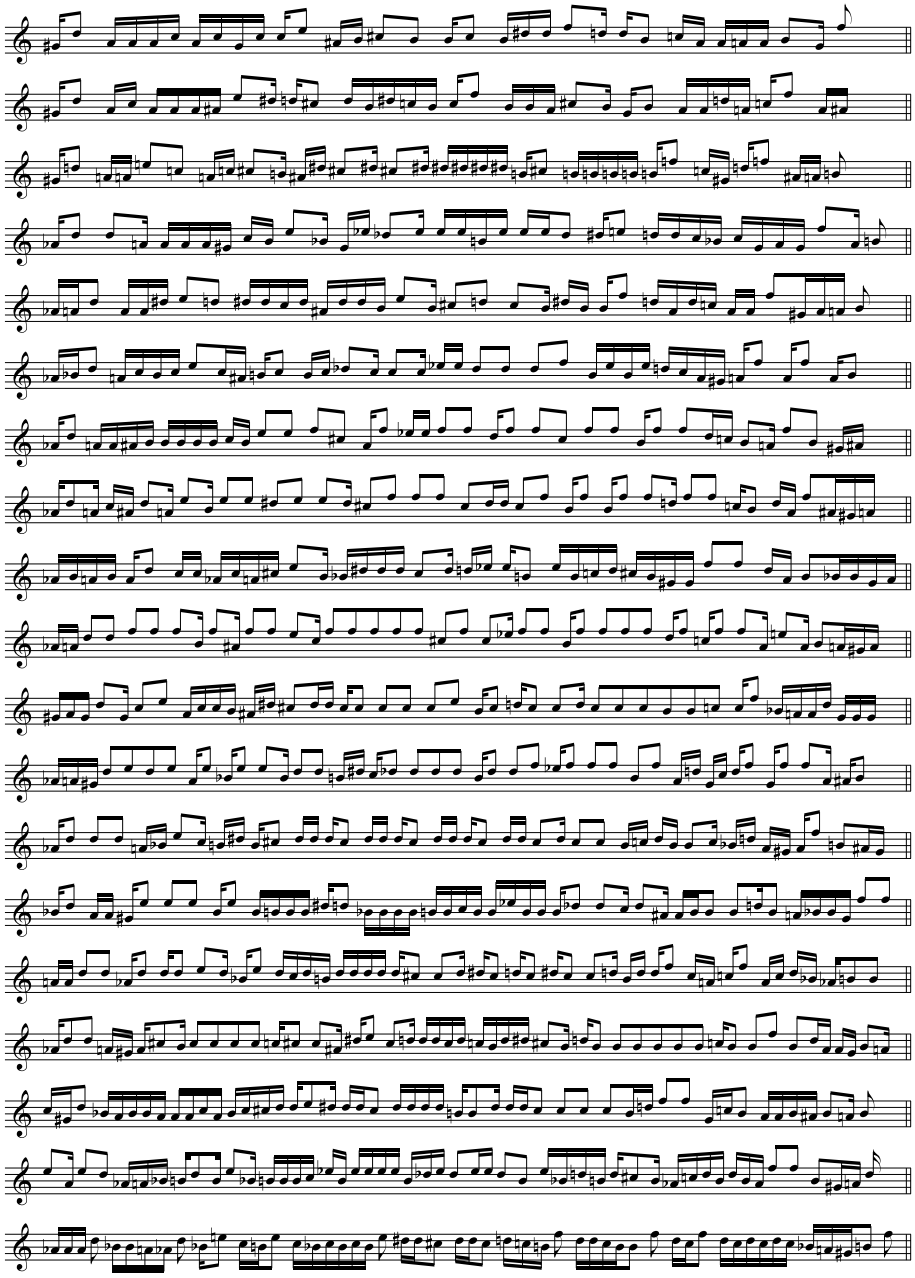


Figura 4.17: Estadios intermedios finales en la transición completa del canto del mirlo en el canto de la alondra.

Ejemplo 4.4.9 *Transición completa realizada mediante el algoritmo FCT entre dos cantos de pájaros transcritos por O. Messiaen, en los que se han incorporado silencios, acentos y staccati. Se han utilizado los parámetros* $\lambda = 30$*, vecindad exponencial estrecha y métrica Chord.*

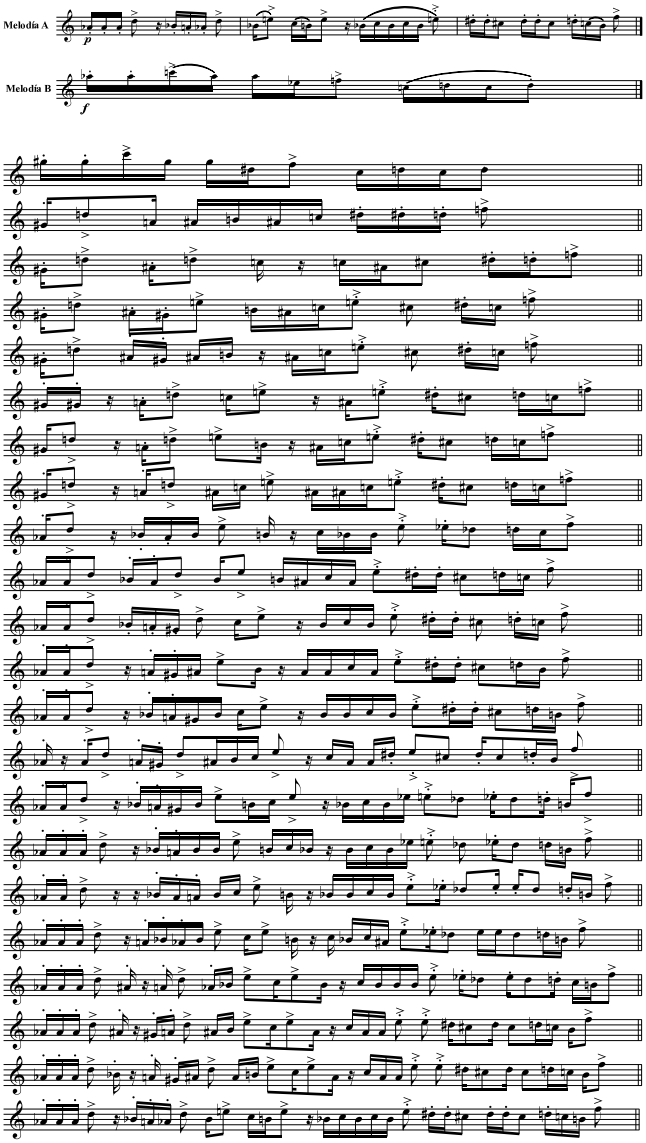


Figura 4.18: Estadios intermedios finales con silencios, acentos y staccati.

4.5 El ritmo

A continuación, de forma análoga a como se realizó en el punto anterior, definiremos una medida de la disimilitud media ordenada entre dos ritmos y estableceremos un proceso de generación de nuevo material rítmico mediante el algoritmo FOCM.

4.5.1 Definición de ritmo

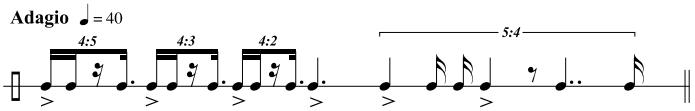
De igual forma que se definió el concepto de nota musical en 4.4.1, podemos definir el concepto de *elemento rítmico*, que será la mínima unidad constituyente de cualquier figuración rítmica.

Definición 4.5.1 Elemento rítmico. *Un elemento rítmico consiste en una tripla de observables $x \in \mathbb{R}^3$ definida mediante las siguientes* 3 *características musicales: coeficiente de duración* $]0, \infty[$ *(véase 4.6), silencio* $[0, 1]$ *y acento* $[0, 1]$.

Una vez que hemos establecido las 3 características mínimas necesarias para definir el concepto de elemento rítmico, podemos definir el concepto de ritmo como una sucesión ordenada de n elementos rítmicos, siendo cada nota un vector en el espacio métrico tridimensional $\mathbb{R}^3$.

Definición 4.5.2 Ritmo. *Un ritmo $\mathscr{R}$ es una secuencia ordenada $\mathscr{R} = \{x_i\}_{i=1}^{n}$, donde cada $x_i \in \mathbb{R}^3$ es un elemento rítmico definido mediante las características de coeficiente de duración* $]0, \infty[$*, silencio* $[0, 1]$ *y acento* $[0, 1]$.

Ejemplo 4.5.1 *Represéntese el siguiente ritmo según la definición anterior en la que se describe el ritmo como una tripla de coeficiente de duración* (δ)*, silencio y acento.*



Nota	Coef. duración (δ)	Silencio	Acento
1	0,078125	0	1
2	0,078125	0	0
3	0,078125	1	0
4	0,1171875	0	0
5	0,046875	0	1
6	0,046875	0	0
7	0,046875	1	0
8	0,0703125	0	0
9	0,03125	0	1
10	0,03125	0	0
11	0,03125	1	0
12	0,046875	0	0
13	0,375	0	1
14	0,2	0	1
15	0,05	0	0
16	0,05	0	0
17	0,2	0	1
18	0,1	1	0
19	0,35	0	0
20	0,05	0	0

4.5.2 La disimilitud rítmica

Consideremos dos ritmos $\mathscr{R}^A$ y $\mathscr{R}^B$, ambos pertenecientes a un espacio métrico tridimensional. Es posible definir una distancia entre ellos utilizando la expresión de la disimilitud media ordenada 3.77.

Definición 4.5.3 Disimilitud media entre dos ritmos. *Sean* $\mathscr{R}^A = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\} \subset \mathbb{R}^3$ *y* $\mathscr{R}^B = \{\mathbf{y_1}, \ldots, \mathbf{y_m}\} \subset \mathbb{R}^3$ *dos ritmos, donde* $n > m$. *Sea* $d : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$ *una función distancia. Sean* u_{ij} *los coeficientes de pertenencia finales calculados con el algoritmo FCM. La disimilitud media* $\mathscr{D}$ *entre* $\mathscr{R}^A$ *y* $\mathscr{R}^B$ *se define como*

$$\mathscr{D}(\mathscr{R}^A, \mathscr{R}^B) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} u_{ij} \cdot d(\mathbf{x_i}, \mathbf{y_j}) \ . \tag{4.10}$$

Nuevamente, $\mathscr{D}$ no considera el orden natural de la secuencia de elementos rítmicos existente en cada una de los ritmos. Podemos definir una disimilitud media ordenada de forma que el orden sea introducido en el cálculo final.

Definición 4.5.4 Disimilitud media ordenada entre dos ritmos. *Sean* $\mathscr{R}^A = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\} \in \mathbb{R}^3$ *y* $\mathscr{R}^B = \{\mathbf{y_1}, \ldots, \mathbf{y_m}\} \in \mathbb{R}^3$ *dos ritmos de distinto número de elementos rítmicos. Sea* $d : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$ *una función distancia. Sean* $\hat{u}_{ij}$ *los coeficientes de pertenencia finales calculados mediante el algoritmo FOCM de* $\mathscr{R}^B$ *sobre* $\mathscr{R}^A$*. La disimilitud media ordenada* $\widehat{\mathscr{D}}$ *de* $\mathscr{R}^A$ *sobre* $\mathscr{R}^B$ *se define como*

$$\widehat{\mathscr{D}}(\mathscr{R}^A, \mathscr{R}^B) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} \hat{u}_{ij} \cdot d(\mathbf{x_i}, \mathbf{y_j}). \tag{4.11}$$

Ejemplo 4.5.2 *Disimilitud media ordenada los siguientes ritmos* $\mathscr{R}^A$ *y* $\mathscr{R}^B$*, procedentes de la tabla de 120 deci-talas hindúes recopilados por Johnson (1989) (simhanandana y miçra varna, respectivamente), y estados intermedios de* $\mathscr{R}^B$ *en cada una de las iteraciones del algoritmo FOCM, ejecutado con los parámetros* $\lambda = 3$*, función de vecindad gaussiana y métrica de Canberra.*

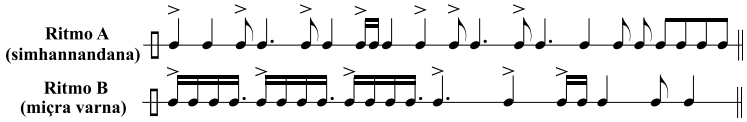


Figura 4.19: Ritmos *simhanandana* y *miçra varna*.

La disimilitud media ordenada entre los dos ritmos, calculada con los parámetros anteriormente descritos, es $\widehat{\mathscr{D}}(\mathscr{R}^A, \mathscr{R}^B) = 0,0318895271$. Los estadios intermedios en los que el ritmo *miçra varna* $\mathscr{R}^B$ se modula hacia el ritmo $\mathscr{R}^A$ son los siguientes:

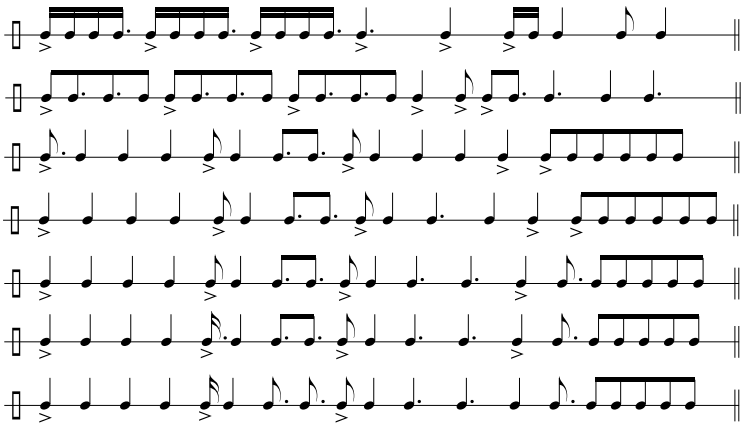


Figura 4.20: Estados intermedios del ritmo $\mathscr{R}^B$.

4.5.3 Resultados computacionales: transiciones rítmicas

Ejemplo 4.5.3 *Transición completa realizada mediante el algoritmo FCT entre los ritmos caccarî y simhavikridita, procedentes de los 120 deci-talas hindúes. Se han utilizado los parámetros $\lambda = 5$, vecindad exponencial y métrica Chord.*

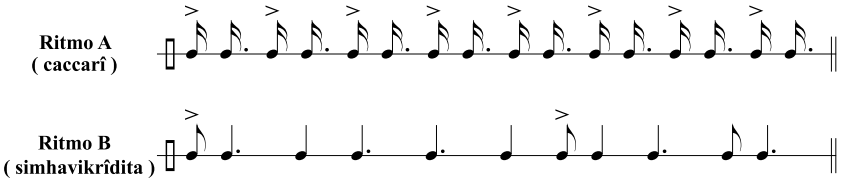


Figura 4.21: Ritmos caccarî y simhavikridita.

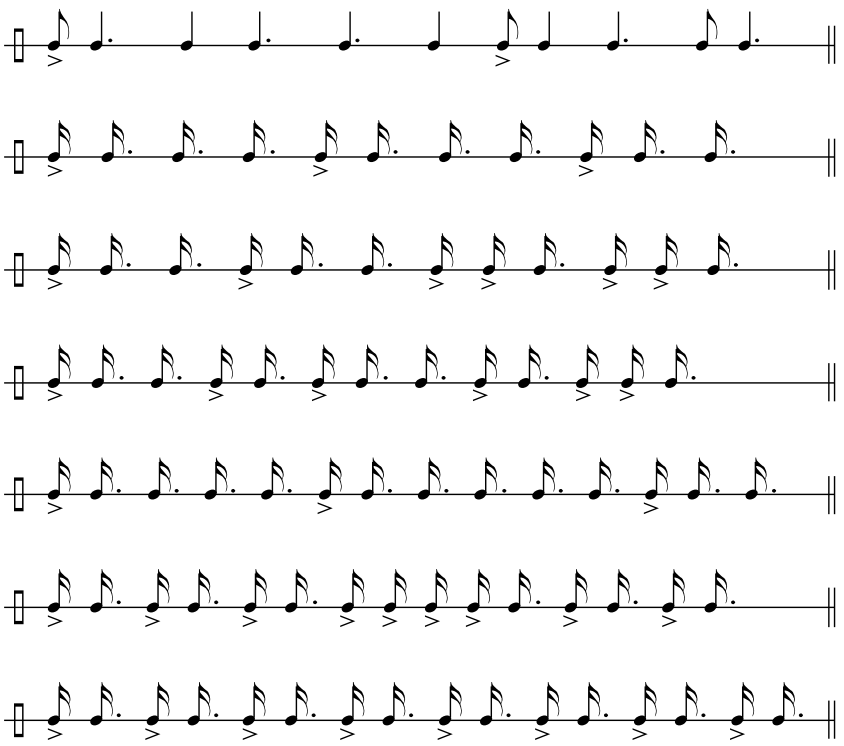


Figura 4.22: Estadios intermedios finales de la transición del ritmo simhavikridita hacia el caccarî.

Ejemplo 4.5.4 *Transición completa realizada mediante el algoritmo FCT entre el ritmo A, calculado mediante el Time Point System de Milton Babbit (Babbitt, 1962) con la serie* $(0, 11, 10, 2, 1, 3, 6, 4, 7, 5, 9, 8)$ *y el ritmo B, de carácter no retrogradable. Se han utilizado los parámetros* $\lambda = 3$*, vecindad gaussiana estrecha y métrica Chord.*

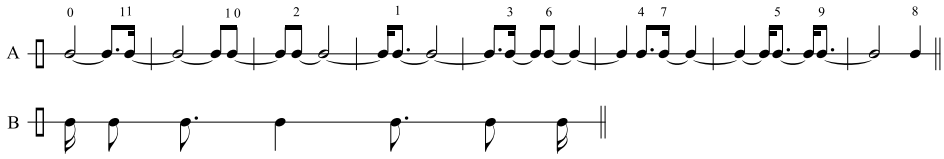


Figura 4.23: Ritmos A y B.

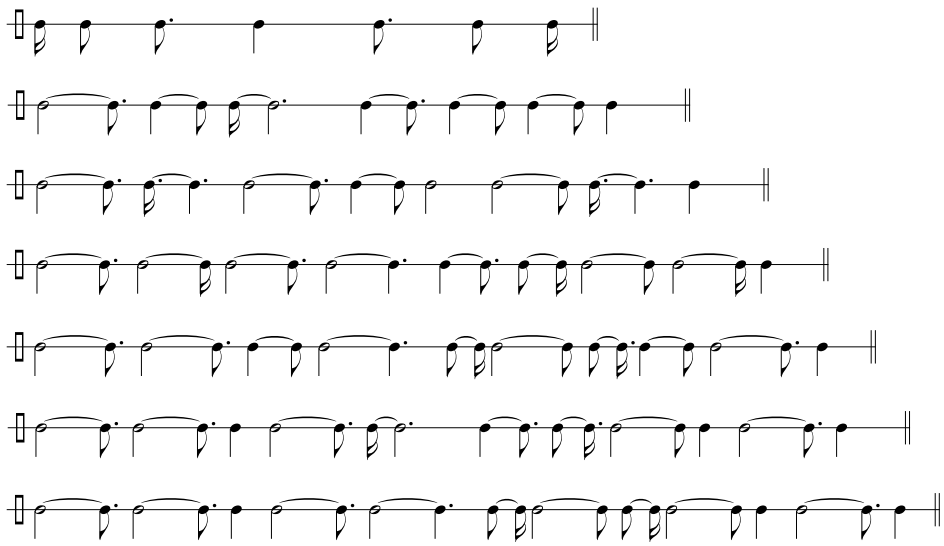


Figura 4.24: Estadios intermedios finales de la transición del ritmo B hacia el A.

Ejemplo 4.5.5 *Transición completa realizada mediante el algoritmo FCT entre el ritmo A, representado por la serie* $(0, 11, 10, 2, 1, 3, 6, 4, 7, 5, 9, 8)$ *de doce valores rítmicos (Šimundža, 1987) y el ritmo B, de carácter no retrogradable. Se han utilizado los parámetros* $\lambda = 3$*, vecindad gaussiana estrecha y métrica Chord.*

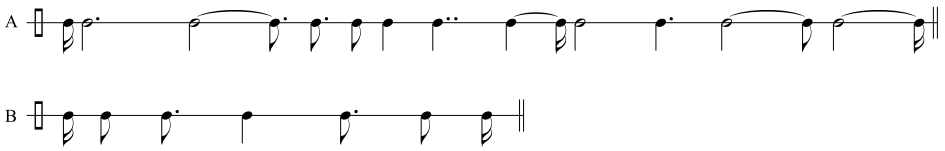


Figura 4.25: Ritmos A y B.

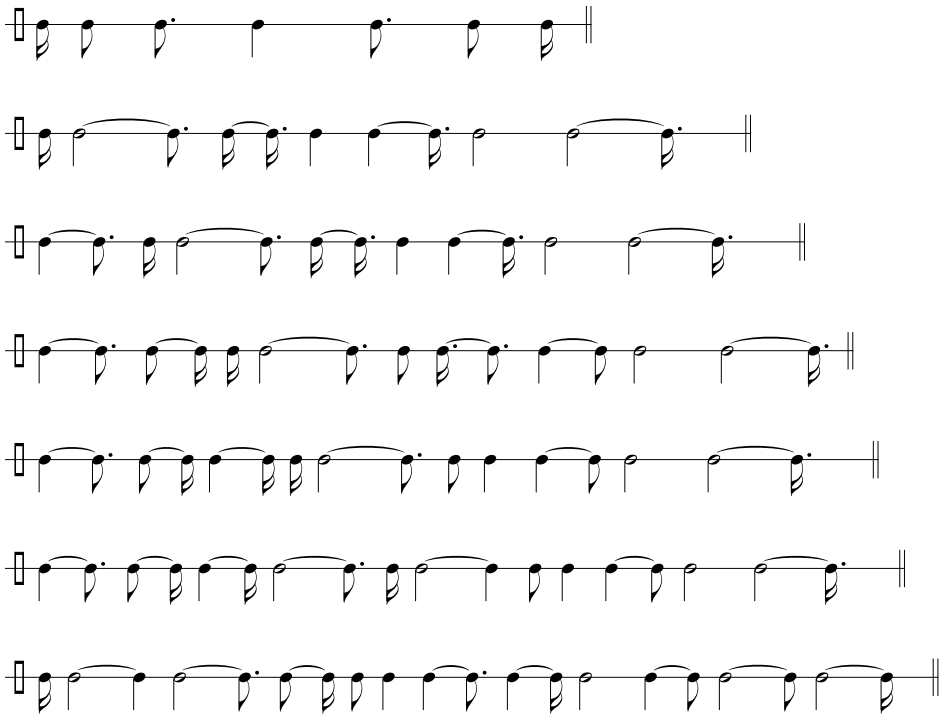


Figura 4.26: Estadios intermedios finales de la transición del ritmo B hacia el A.

4.6 La armonía

Podemos aplicar todo lo descrito anteriormente a las sucesiones armónicas. La armonía puede entenderse como una sucesión de acordes en la que tradicionalmente se establece un número de voces. Para cada voz, los movimientos melódicos vienen dictaminados por los diferentes encadenamientos de acordes. En esta sección, para formular la armonía prescindiremos de toda información que no sea referente a la altura (MIDI Pitch); definiremos un acorde como un conjunto de alturas y una determinada armonía como una sucesión de acordes. De forma análoga a como lo hemos realizado en la secciones anteriores, podremos establecer una medida de la disimilitud media ordenada entre dos armonías y por último generar nuevo material armónico.

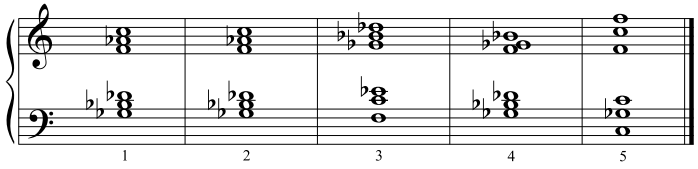
4.6.1 Definición de armonía

De igual forma que se definió el concepto de nota musical en 4.4.1, podemos definir el concepto de *acorde*, que será la mínima unidad constituyente de cualquier armonía. En un acorde sólo se incluirá información relativa a las alturas de las notas, prescindiendo por tanto de duraciones, intensidades, así como de cualquier otra información musical.

Definición 4.6.1 Acorde. *Un acorde consiste en una q-tupla* $\boldsymbol{x} \in \mathbb{R}^q$ *definida mediante q valores de MIDI pitch* $\in [1, 127]$.

Definición 4.6.2 Armonía. *Una armonía* $\mathscr{H}$ *es una secuencia ordenada* $\mathscr{H} = \{\boldsymbol{x}_i\}_{i=1}^n$, *donde cada* $\boldsymbol{x}_i \in \mathbb{R}^q$ *es un acorde caracterizado por q valores de MIDI pitch* $\in [1, 127]$.
Nótese que según la definición anterior, todos los acordes de una armonía $\mathscr{H}$ tienen que tener el mismo número q de valores de MIDI pitch.

Ejemplo 4.6.1 *Considérese la siguiente progresión armónica formada por cinco acordes de seis notas y represéntese según la definición anterior en la que se describe la armonía cono una sucesión de n-tuplas de MIDI pitch.*



Acorde	MIDI Pitch					
1	54	58	61	65	68	72
2	54	58	61	65	68	72
3	53	60	63	66	70	73
4	54	58	61	65	66	70
5	48	54	60	65	72	77

4.6.2 La disimilitud armónica

Consideremos dos armonías $\mathscr{H}^A$ y $\mathscr{H}^B$, ambas pertenecientes a un espacio métrico q-dimensional. Es posible definir una distancia entre ellos utilizando la expresión de la disimilitud media ordenada 3.77:

Definición 4.6.3 Disimilitud media entre dos armonías. *Sean* $\mathscr{H}^A = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\} \subset \mathbb{R}^q$ *y* $\mathscr{H}^B = \{\mathbf{y_1}, \ldots, \mathbf{y_m}\} \subset \mathbb{R}^q$ *dos armonías, donde* $n > m$. *Sea* $d : \mathbb{R}^q \times \mathbb{R}^q \to \mathbb{R}$ *una función distancia. Sean* u_{ij} *los coeficientes de pertenencia finales calculados con el algoritmo FCM. La disimilitud media* $\mathscr{D}$ *entre* $\mathscr{H}^A$ *y* $\mathscr{H}^B$ *se define como*

$$\mathscr{D}(\mathscr{H}^A, \mathscr{H}^B) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} u_{ij} \cdot d(\mathbf{x_i}, \mathbf{y_j}) \; . \tag{4.12}$$

Como $\mathscr{D}$ no considera el orden natural de la secuencia de acordes de cada armonía, es necesarios definir una disimilitud media ordenada de forma que el orden sea introducido en el cálculo final.

Definición 4.6.4 Disimilitud media ordenada entre dos armonías. *Sean* $\mathscr{H}^A = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\} \in \mathbb{R}^q$ *y* $\mathscr{H}^B = \{\mathbf{y_1}, \ldots, \mathbf{y_m}\} \in \mathbb{R}^q$ *dos armonías de distinto número de acordes. Sea* $d : \mathbb{R}^q \times \mathbb{R}^q \to \mathbb{R}$ *una función distancia. Sean* $\hat{u}_{ij}$ *los coeficientes de pertenencia finales calculados mediante el algoritmo FOCM de* $\mathscr{H}^B$ *sobre* $\mathscr{H}^A$. *La disimilitud media ordenada* $\widehat{\mathscr{D}}$ *de* $\mathscr{H}^A$ *sobre* $\mathscr{H}^B$ *se define como*

$$\widehat{\mathscr{D}}(\mathscr{H}^A, \mathscr{H}^B) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} \hat{u}_{ij} \cdot d(\mathbf{x_i}, \mathbf{y_j}). \tag{4.13}$$

Ejemplo 4.6.2 *Disimilitud media ordenada entre dos armonías* $\mathscr{H}^A$ *y* $\mathscr{H}^B$, *procedentes de los corales número 257 y 217 de J. S. Bach (Bach, 1996) respectivamente, y estado intermedios de* $\mathscr{H}^B$ *en cada una de las iteraciones del algoritmo*

FOCM, ejecutado con los parámetros $\lambda = 2$, función de vecindad gaussiana y distancia euclidiana.

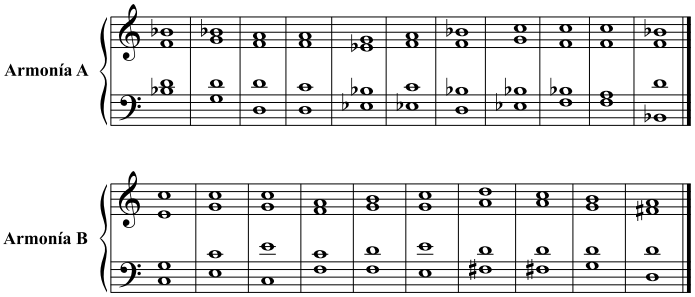


Figura 4.27: Armonías procedentes de los corales 257 y 217 de J. S. Bach. (Bach, 1996)

La disimilitud media ordenada calculada entre las dos armonías es $\widehat{\mathscr{D}}(\mathscr{H}^A, \mathscr{H}^B) = 0,5759481303$. Los estadios intermedios en los que la armonía $\mathscr{H}^B$ se modula hacia la armonía $\mathscr{H}^A$ son los siguientes:

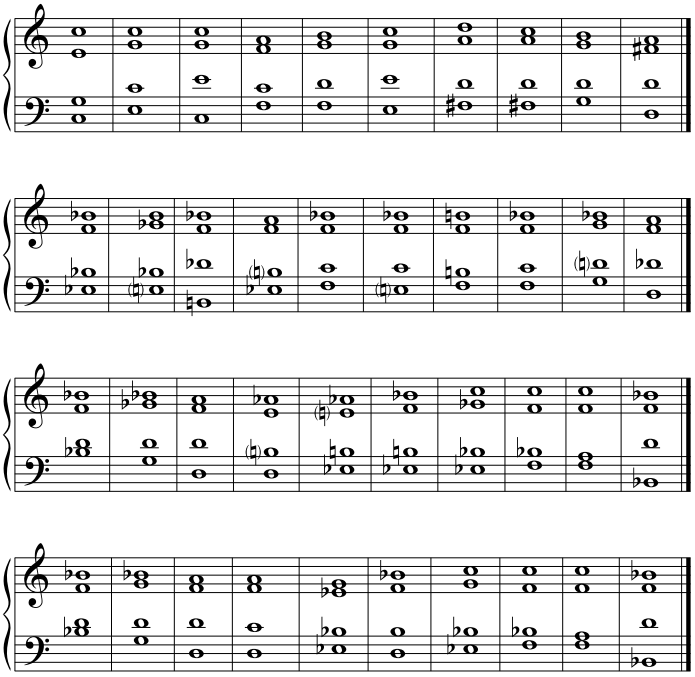
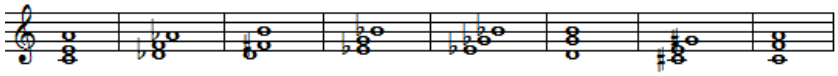


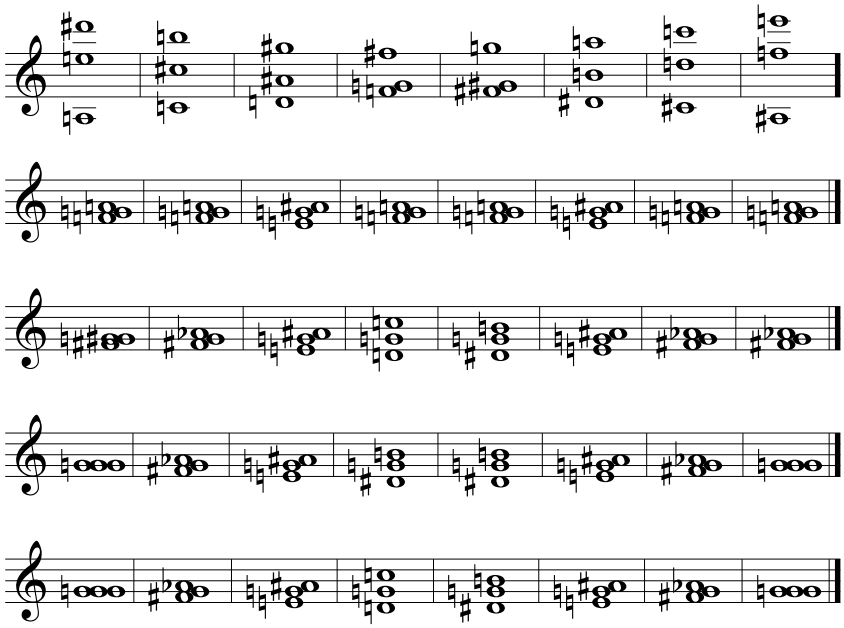
Figura 4.28: Estados intermedios de la armonía $\mathscr{H}^B$.

4.6.3 Resultados computacionales: transiciones armónicas

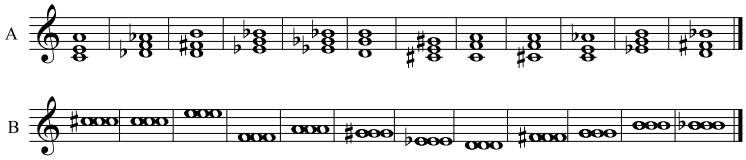
Ejemplo 4.6.3 *Transición armónica entre dos progresiones armónicas distintas utilizando FOCM con $\lambda = 2$, función de vecindad de tipo Gaussiana estrecha y métrica de Manhattan.*



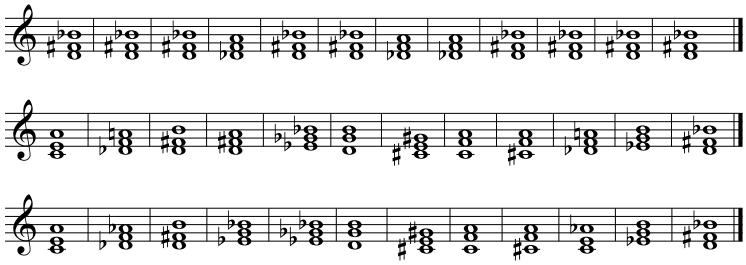
Obtenemos los siguientes estados intermedios



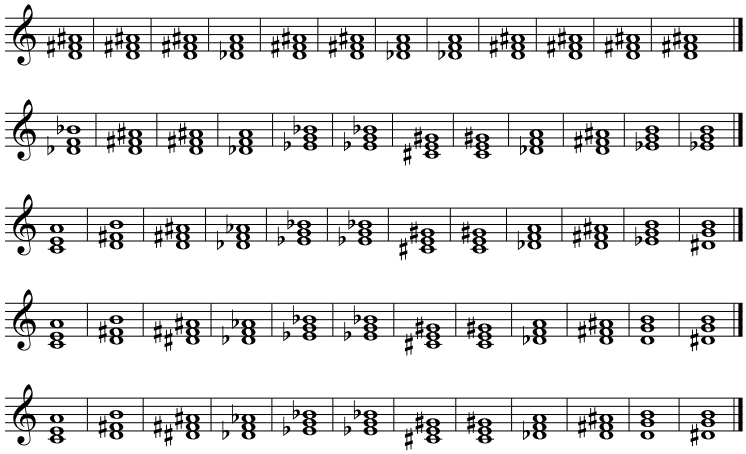
Ejemplo 4.6.4 *Comparación de las transiciones generadas por FOCM y FCM entre las triadas procedentes de la serie dodecafónica utilizada en el Op. 41 de A. Schoenberg (Brindle, 1969, pág.10) y la propia serie*[3]*, con* $\lambda = 1{,}55$*, función vecindad Gaussiana muy estrecha y métrica euclidiana.*



Obtenemos los siguientes estados intermedios:

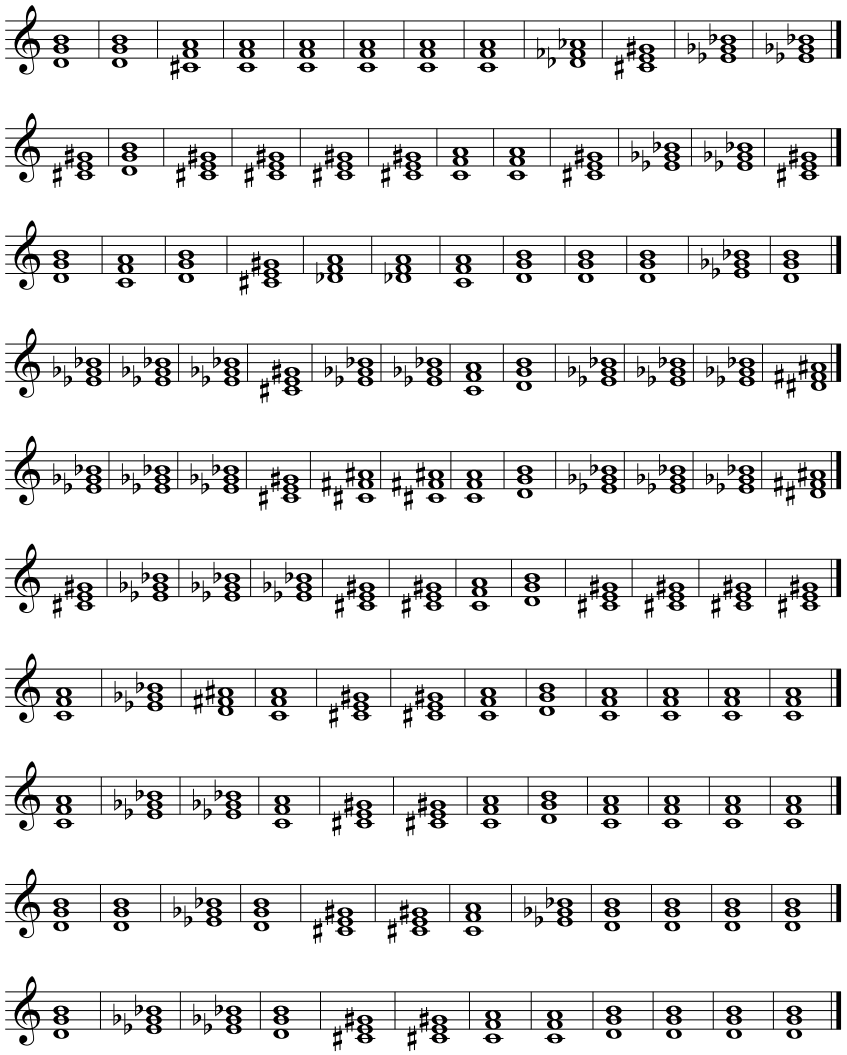


Calculando con una función de vecindad de tipo gaussiana estándar obtenemos:

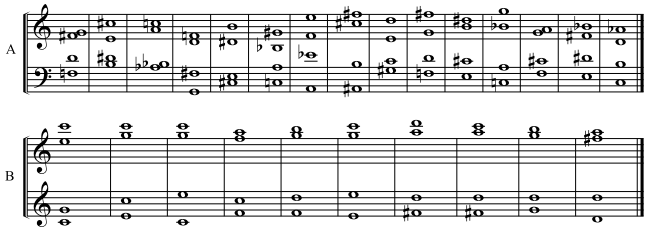


[3]Nótese que la serie se tiene que introducir triplicando las notas para que ésta pueda ser convertida en una transición armónica.

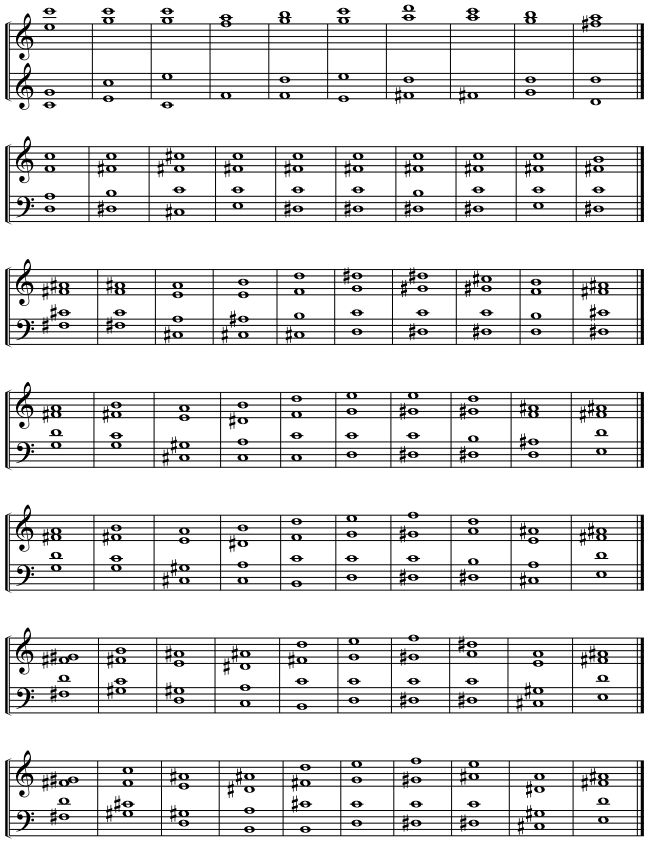
Si repetimos el proceso pero esta vez utilizando el algoritmo FCM, sin introducir la ordenación, con $\lambda = 2$ y métrica de Chebyshev obtenemos los siguientes resultados:



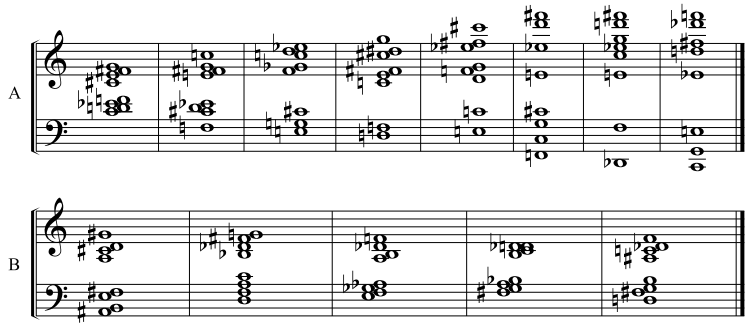
Ejemplo 4.6.5 *Transición armónica entre cuatríadas generadas por la serie utilizada en el Op. 26 de A. Webern (Brindle, 1969, pág.81) y las armonías del coral 217 de J. S. Bach, utilizando FOCM con* $\lambda = 2{,}0$*, función vecindad de tipo Gaussiana estándar y métrica geodésica.*



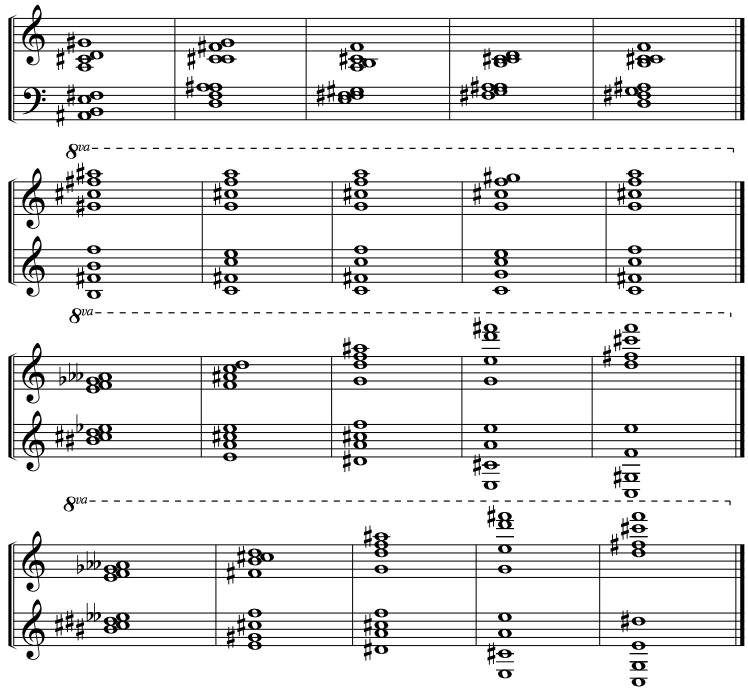
Resultados obtenidos:



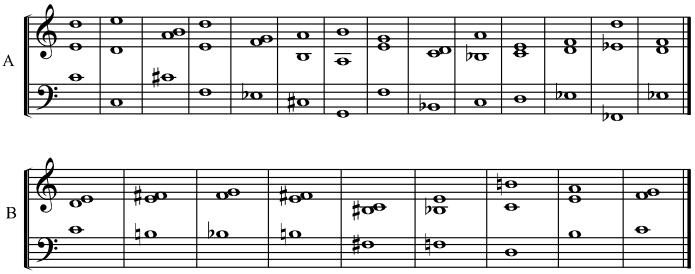
Ejemplo 4.6.6 *Transición armónica entre el campo armónico de poliacordes compuestos por ocho notas (Brindle, 1969, pág.77), utilizando FOCM con* $\lambda = 2{,}0$*, función vecindad de tipo Gaussiana estrecha y métrica euclidiana.*



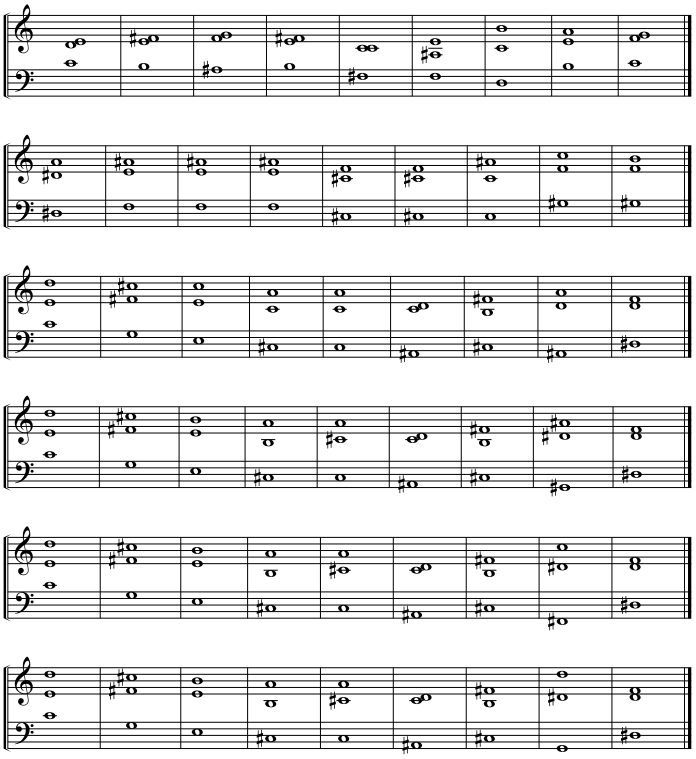
Obtenemos los siguientes estados intermedios:



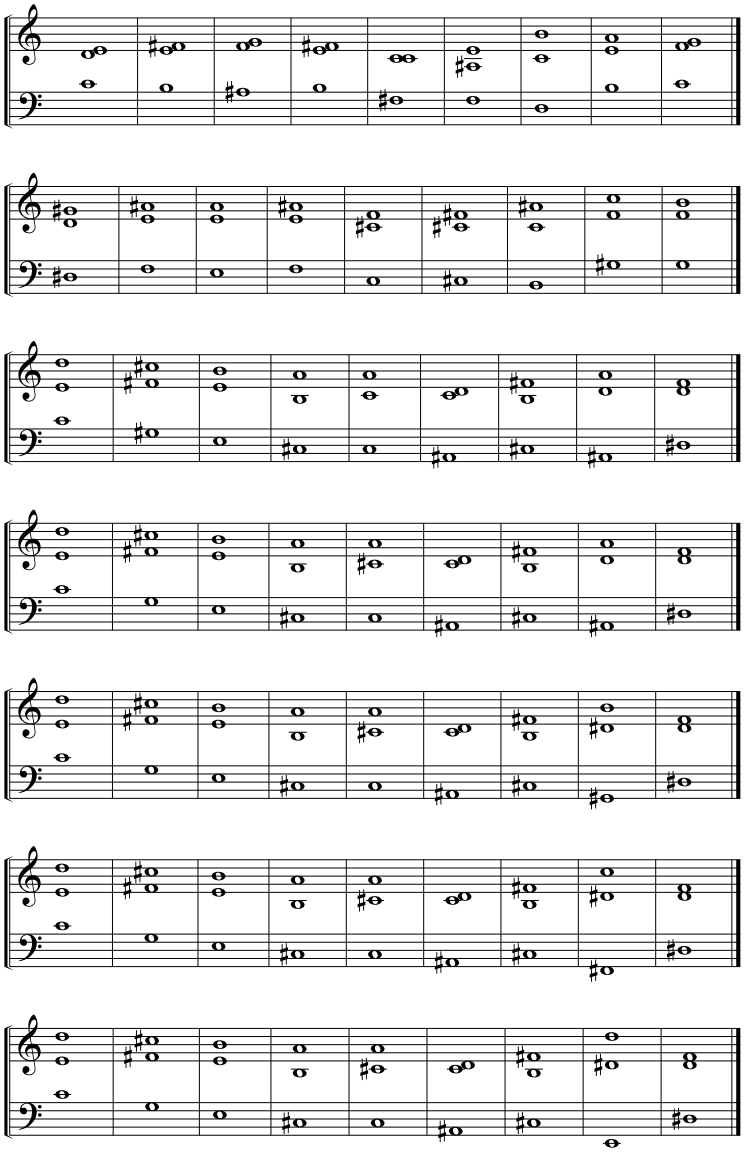
Ejemplo 4.6.7 *Comparación de transiciones armónicas entre triadas generadas por intervalos de segunda (Persichetti, 1989, pág.126), utilizando FOCM con* $\lambda = 2{,}0$*, función de vecindad de tipo Gaussiana muy estrecha y métricas euclidiana, Manhattan y Canberra.*



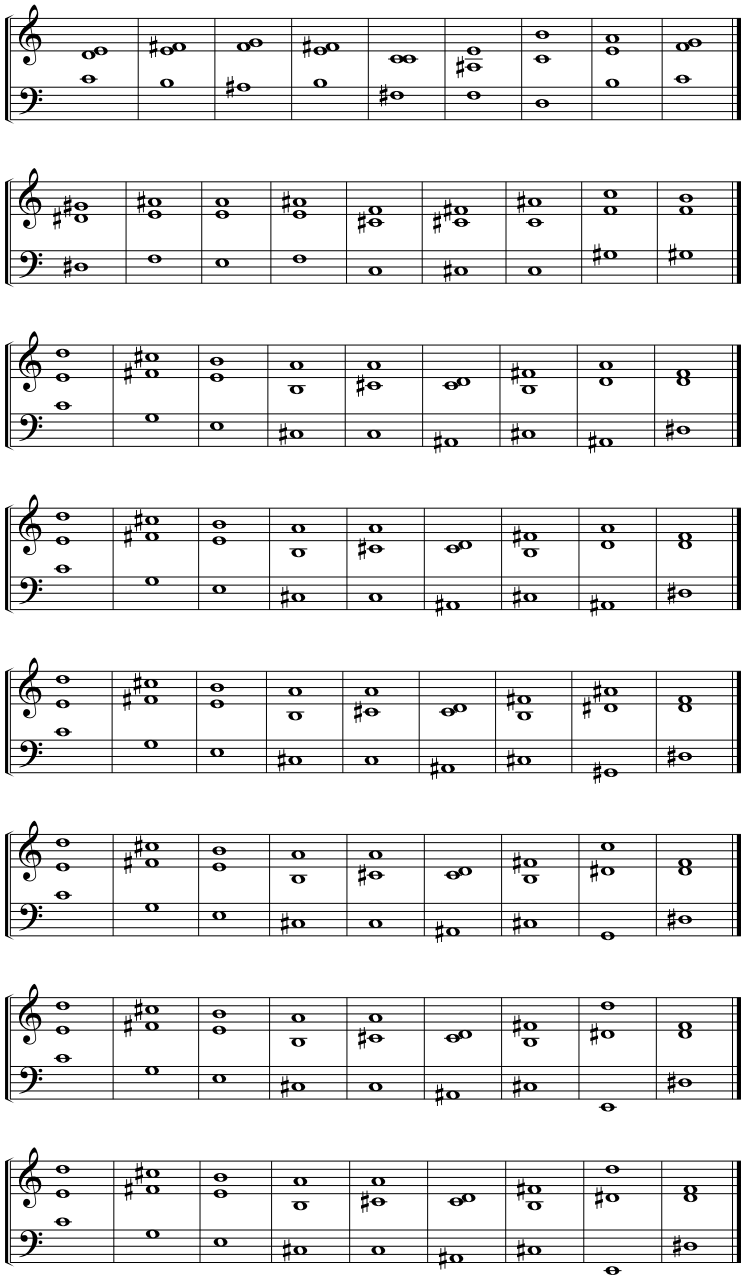
Resultados para la métrica euclidiana:



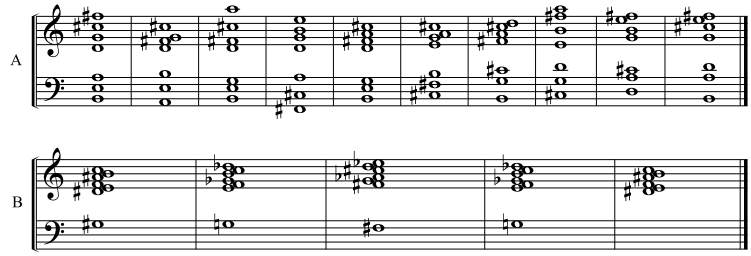
Resultados para la métrica manhattan:



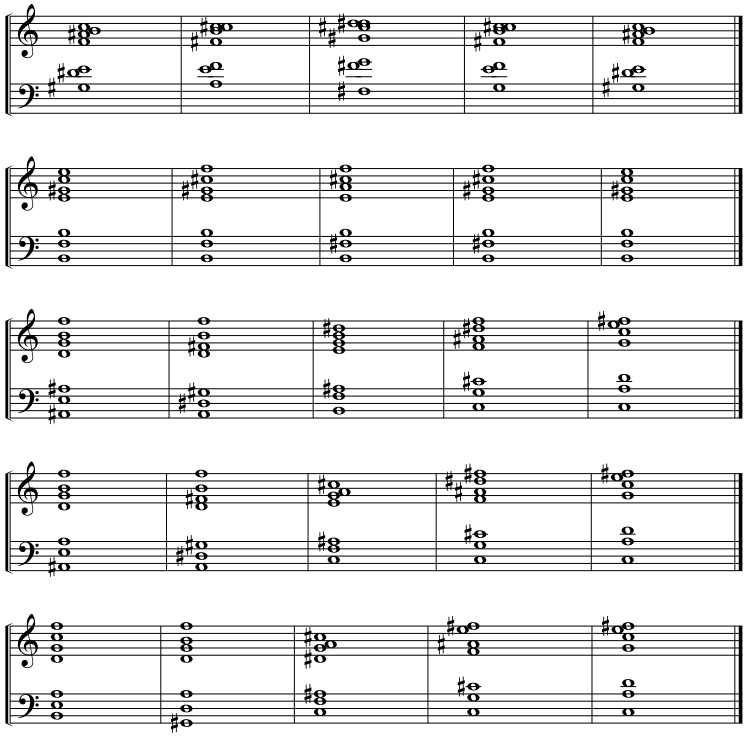
Resultados para la métrica de Canberra:



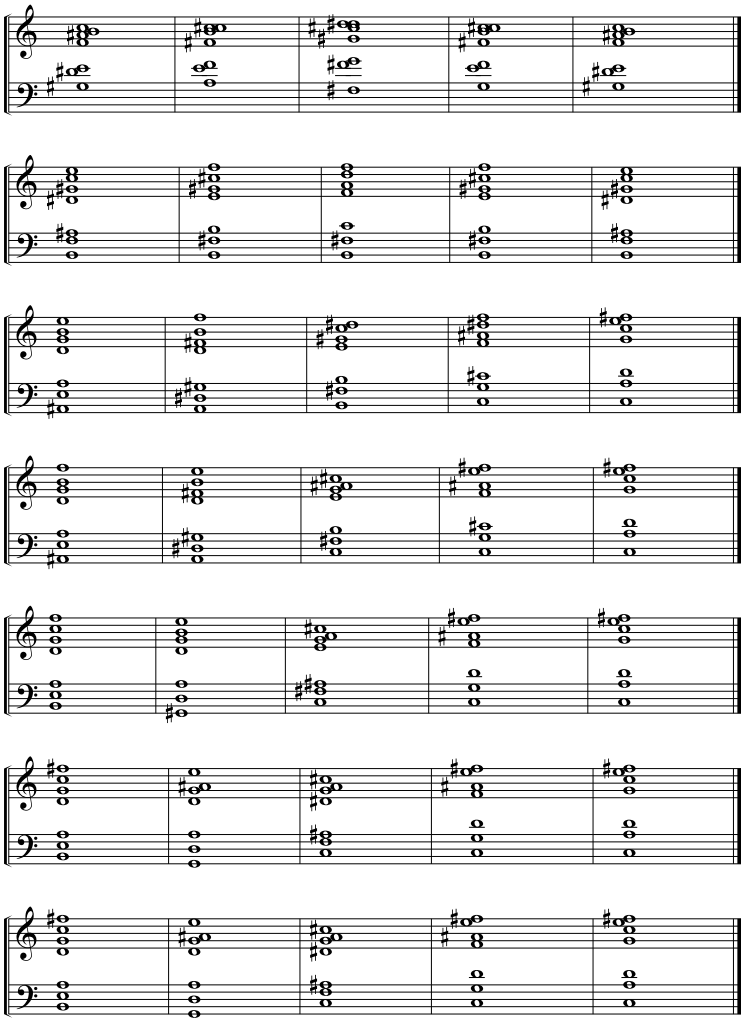
Ejemplo 4.6.8 *Comparación de transiciones armónicas entre dos armonías a siete partes, de carácter pandiatónico (Persichetti, 1989, pág.227), utilizando FOCM con* $\lambda = 2{,}0$*, función de vecindad de tipo Gaussiana y métricas euclidiana, manhattan y Canberra.*



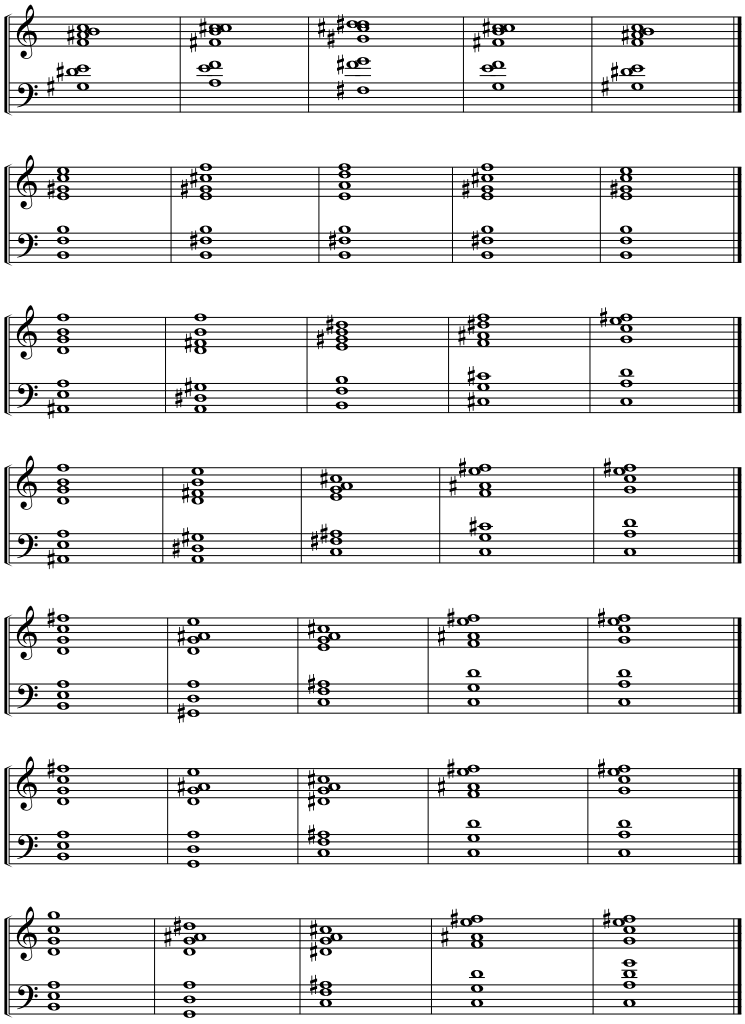
Resultados para la métrica euclidiana:



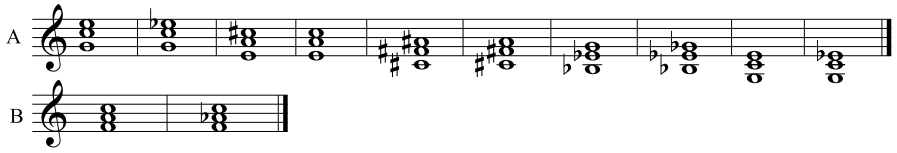
Resultados para la métrica manhattan:



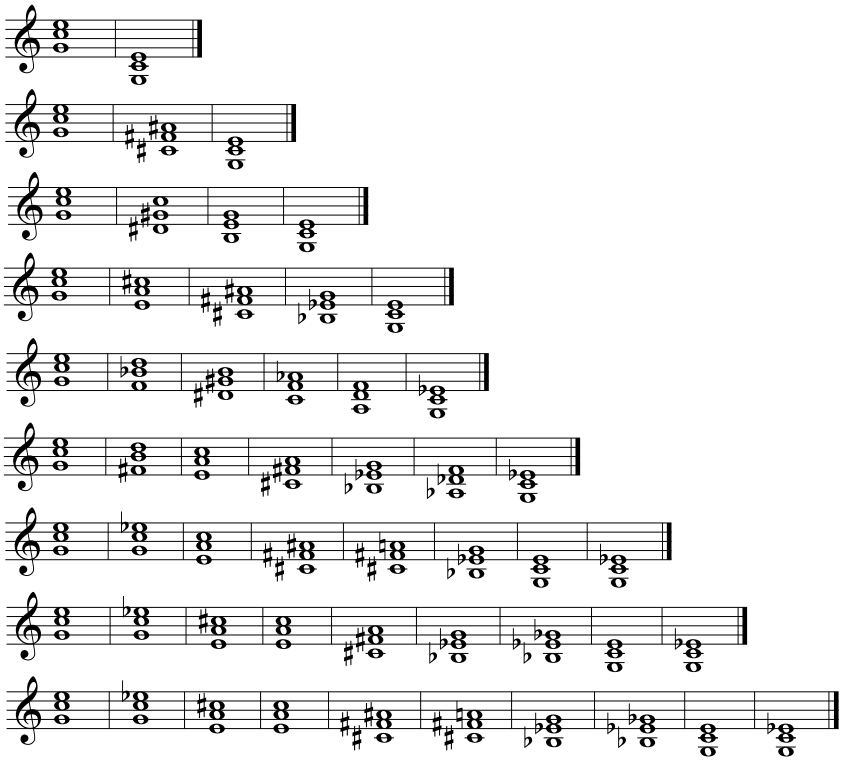
Resultados para la métrica de Canberra:



Ejemplo 4.6.9 *Transición armónica completa entre dos progresiones armónicas distintas utilizando el algoritmo FCT con $\lambda = 2$, función vecindad de tipo triangular estrecha y función distancia Chord.*



Obtenemos la siguiente transición completa de la armonía B hacia la A



4.7 El timbre

Desde un punto de vista físico el timbre puede ser descrito mediante la caracterización de cada uno de los sonidos parciales que lo forman. Tradicionalmente dicha caracterización se realiza mediante el análisis de Fourier, utilizando la Transformada Rápida de Fourier como método computacional para obtener el espectro energético de frecuencias de una señal en un instante determinado. Dicho espectro energético puede tener una dependencia temporal; la representación en forma de espectrograma nos permite visualizar la dependencia temporal del espectro energético. Como ejemplo mostraremos a continuación el espectrograma de la nota La_3 emitida por un saxofón alto

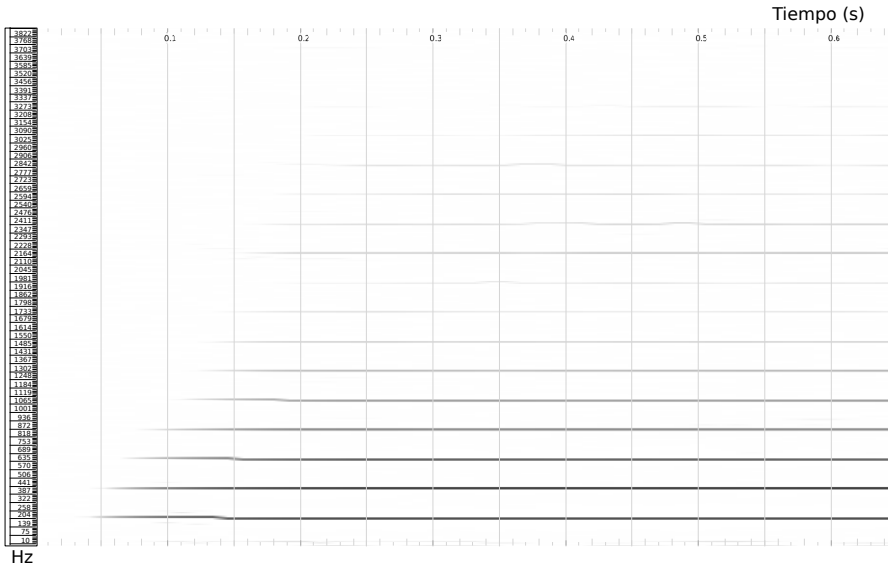


Figura 4.29: Espectrograma de una nota La_3 emitida por un saxofón alto.

En el presente estudio realizaremos una aproximación en el modelado del timbre, despreciando la dependencia temporal y considerando que éste queda caracterizado únicamente por el espectro de frecuencias, de forma estática. De esta manera, el timbre de la nota anteriormente mencionada se reduciría al siguiente espectro energético:

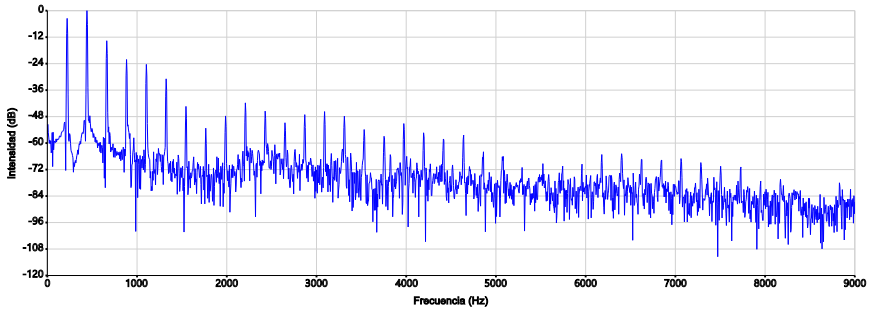


Figura 4.30: Espectro energético de una nota La_3 emitida por un saxofón alto.

Si mediante diferentes técnicas como el *Cepstrum* o el *Harmonic Product Spectrum (HPS)* (Park, 2009, pág.350) realizamos un filtrado del espectro energético anterior, simplificándolo a los n picos más intensos (los n máximos locales de mayor intensidad) obtenemos que podemos definir el espectro energético como la sucesión de las duplas formadas por las n primeras frecuencias junto con su respectivas intensidades relativas, expresadas en dB.

4.7.1 Definición de espectro energético

De manera análoga a como se ha realizado en los apartados anteriores, definiremos el concepto de *sobretono*, que será la mínima unidad constituyente de cualquier espectro energético. En un espectro energético únicamente consideraremos la información relativa a las frecuencias de los sobretonos y sus intensidades relativas expresadas en decibelios.

Definición 4.7.1 Sobretono. *Un sobretono consiste en la dupla* $\boldsymbol{x} \in \mathbb{R}^2$ *definida mediante los valores de frecuencia e intensidad.*

Definición 4.7.2 Espectro. *Una espectro* $\mathscr{F}$ *es una secuencia ordenada* $\mathscr{F} = \{\boldsymbol{x}_i\}_{i=1}^{n}$, *donde cada* $\boldsymbol{x}_i \in \mathbb{R}^2$ *es un sobretono caracterizado por los valores de frecuencia e intensidad.*

Ejemplo 4.7.1 *Considérese el siguiente espectro energético* $\mathscr{F}^{La_3} = \{\boldsymbol{x}_i\}_{i=1}^{39}$, *procedente de realizar un filtrado de las 39 frecuencias más intensas del espectro mostrado en la figura 4.31*

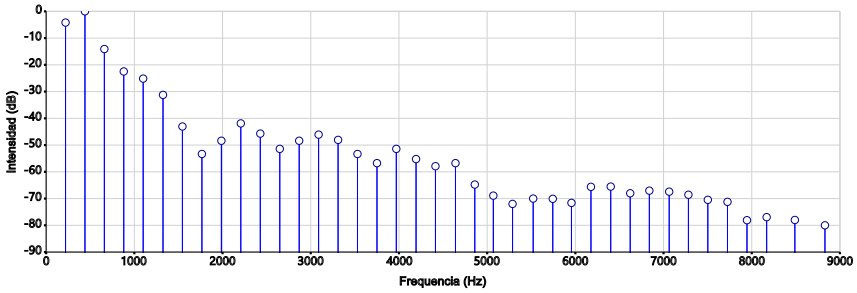


Figura 4.31: Espectro energético $\mathscr{F}^{La_3}$.

Tabla 4.2: Secuencia de parciales formada por las duplas $\mathbf{x}_i$ de frecuencia (Hz) e intensidad (dB) del espectro $\mathscr{F}^{La_3}$.

Nº	Hz	dB	Nº	Hz	dB	Nº	Hz	dB
1	220	-4,19	**14**	3089	-46,09	**27**	5956	-71,61
2	440	00,00	**15**	3310	-48,06	**28**	6177	-65,62
3	660	-14,09	**16**	3530	-53,33	**29**	6401	-65,52
4	880	-22,48	**17**	3751	-56,76	**30**	6622	-68,02
5	1100	-25,14	**18**	3970	-51,42	**31**	6839	-67,04
6	1325	-31,24	**19**	4194	-55,23	**32**	7064	-67,42
7	1545	-43,05	**20**	4414	-57,90	**33**	7282	-68,57
8	1766	-53,33	**21**	4641	-56,76	**34**	7502	-70,47
9	1987	-48,38	**22**	4860	-64,76	**35**	7725	-71,23
10	2207	-41,90	**23**	5070	-68,90	**36**	7946	-78,09
11	2428	-45,71	**24**	5288	-72,00	**37**	8169	-76,95
12	2649	-51,42	**25**	5522	-70,00	**38**	8489	-78,00
13	2869	-48,38	**26**	5744	-70,09	**39**	8830	-80,00

4.7.2 La disimilitud espectral

Consideremos dos espectros $\mathscr{F}^A$ y $\mathscr{F}^B$, ambos pertenecientes a un espacio métrico 2-dimensional. Es posible definir una distancia entre ellos utilizando la expresión de la disimilitud media ordenada 3.77:

Definición 4.7.3 Disimilitud media entre dos espectros. *Sean* $\mathscr{F}^A = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\} \subset \mathbb{R}^2$ *y* $\mathscr{F}^B = \{\mathbf{y_1}, \ldots, \mathbf{y_m}\} \subset \mathbb{R}^2$ *dos espectros energéticos, donde* $n > m$. *Sea* $d : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ *una función distancia. Sean* u_{ij} *los coeficientes de pertenencia finales calculados con el algoritmo FCM. La disimilitud media* $\mathscr{D}$ *entre* $\mathscr{F}^A$ *y* $\mathscr{F}^B$ *se define como*

$$\mathscr{D}(\mathscr{F}^A, \mathscr{F}^B) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} u_{ij} \cdot d(\mathbf{x_i}, \mathbf{y_j}) \ . \tag{4.14}$$

Nuevamente, como $\mathscr{F}$ no considera el orden natural de la secuencia de sobretonos de cada espectro, es necesarios definir una disimilitud media ordenada de forma que el orden sea introducido en el cálculo final.

Definición 4.7.4 Disimilitud media ordenada entre dos espectros. *Sean* $\mathscr{F}^A = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\} \in \mathbb{R}^2$ *y* $\mathscr{F}^B = \{\mathbf{y_1}, \ldots, \mathbf{y_m}\} \in \mathbb{R}^2$ *dos espectros energéticos de distinto número de sobretonos. Sea* $d : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ *una función distancia. Sean* $\hat{u}_{ij}$ *los coeficientes de pertenencia finales calculados mediante el algoritmo FOCM de* $\mathscr{F}^B$ *sobre* $\mathscr{F}^A$. *La disimilitud media ordenada* $\widehat{\mathscr{D}}$ *de* $\mathscr{F}^A$ *sobre* $\mathscr{F}^B$ *se define como*

$$\widehat{\mathscr{D}}(\mathscr{F}^A, \mathscr{F}^B) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} \hat{u}_{ij} \cdot d(\mathbf{x_i}, \mathbf{y_j}). \tag{4.15}$$

Ejemplo 4.7.2 *Compárese la Disimilitud media ordenada entre el espectro de la nota La_3 emitida por el saxo alto y el espectro de la misma nota emitida por el violín y por el piano, utilizando el algoritmo FOCM, ejecutado con los parámetros $\lambda = 3{,}0$, función de vecindad gaussiana y distancia euclidiana.*

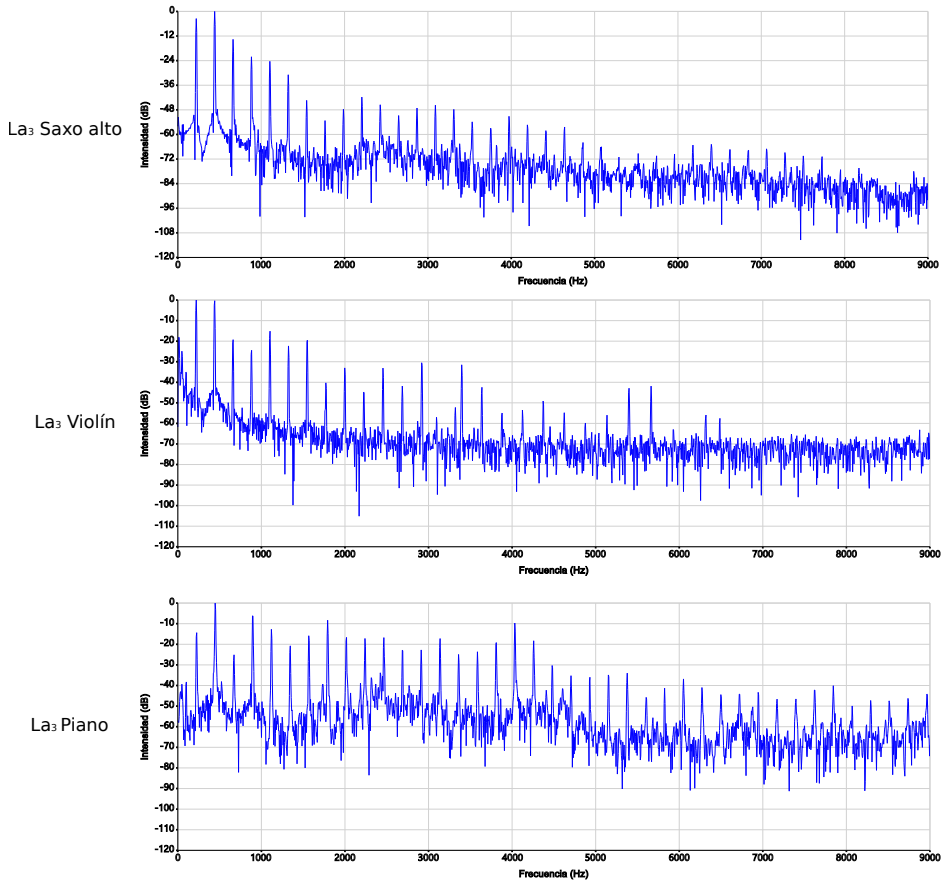


Figura 4.32: Espectro de frecuencias de la nota La_3 en el saxo alto, violín y piano respectivamente.

La disimilitud media ordenada calculada entre el espectro del saxo y el del violín es $\widehat{\mathscr{D}}(\mathscr{F}^{\text{saxo}}, \mathscr{F}^{\text{violín}}) = 4,4931291135$. Podemos observar los siguientes estadios intermedios en el proceso de convergencia del algoritmo:

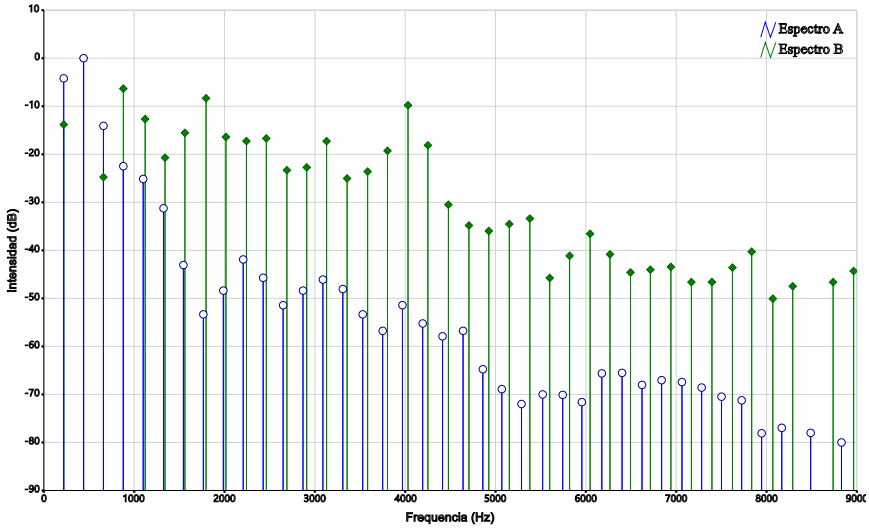


Figura 4.33: Estado inicial en el algoritmo entre los espectros $\mathscr{F}^{\text{saxo}}, \mathscr{F}^{\text{violín}}$.

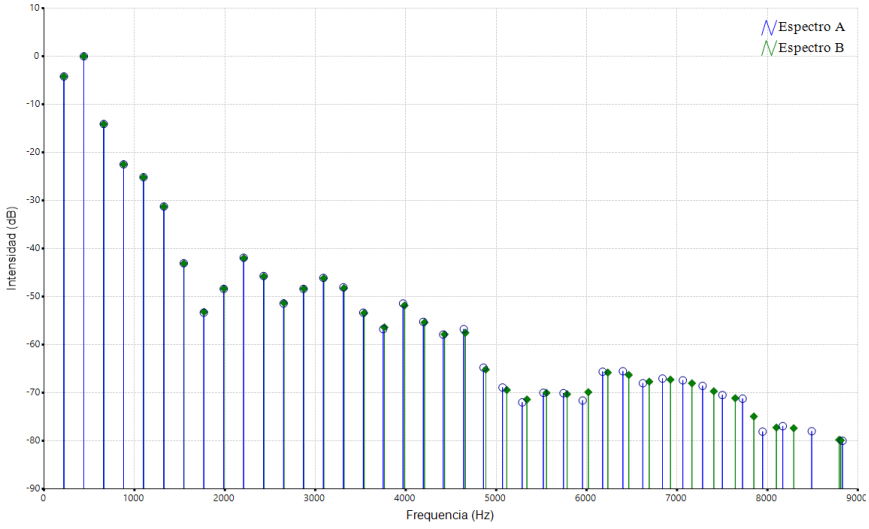


Figura 4.34: Primera iteración entre los espectros $\mathscr{F}^{\text{saxo}}, \mathscr{F}^{\text{violín}}$.

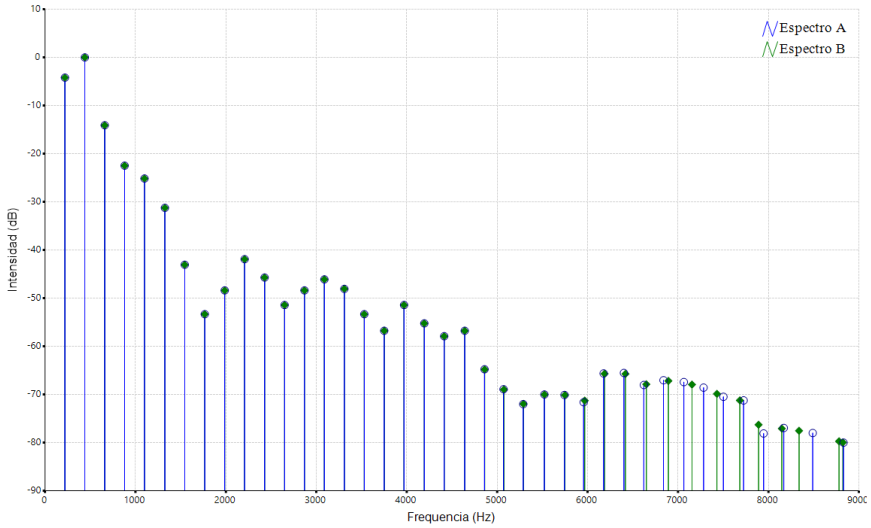


Figura 4.35: Segunda iteración entre los espectros $\mathscr{F}^{\text{saxo}}, \mathscr{F}^{\text{violín}}$.

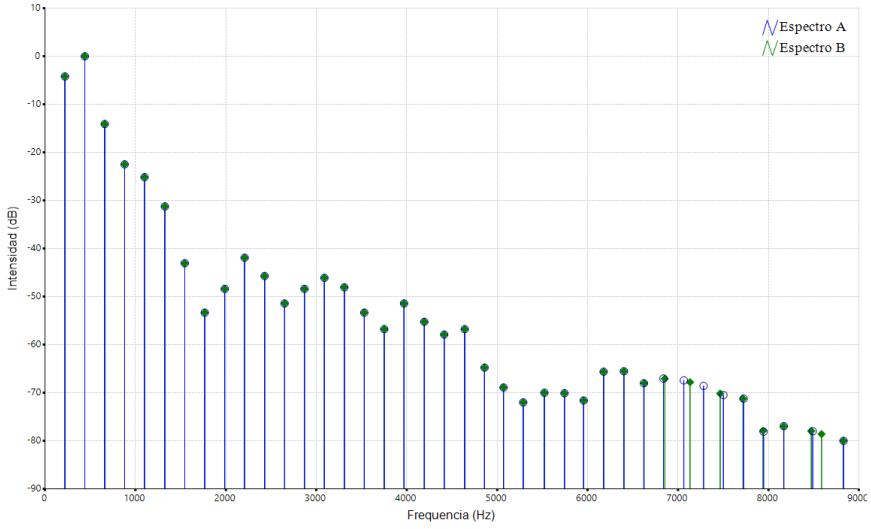


Figura 4.36: Tercera iteración entre los espectros $\mathscr{F}^{\text{saxo}}, \mathscr{F}^{\text{violín}}$.

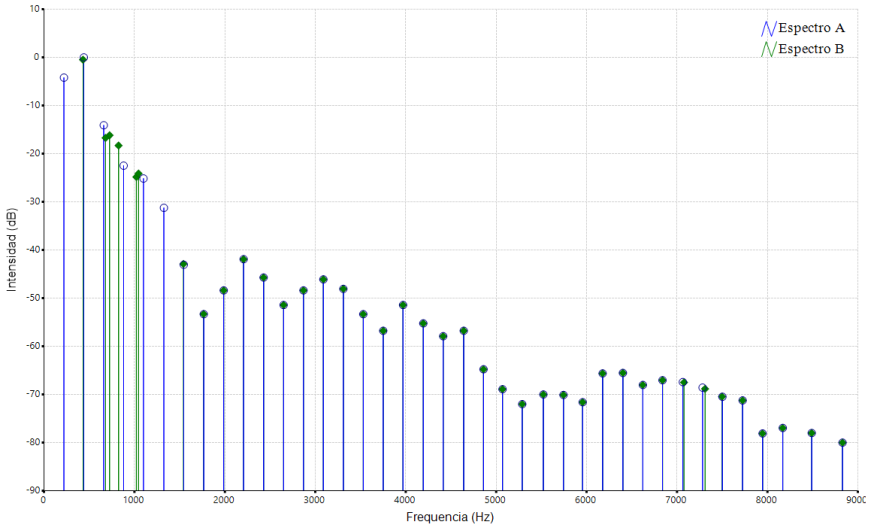


Figura 4.37: Quinta iteración entre los espectros $\mathscr{F}^{\text{saxo}}, \mathscr{F}^{\text{violín}}$.

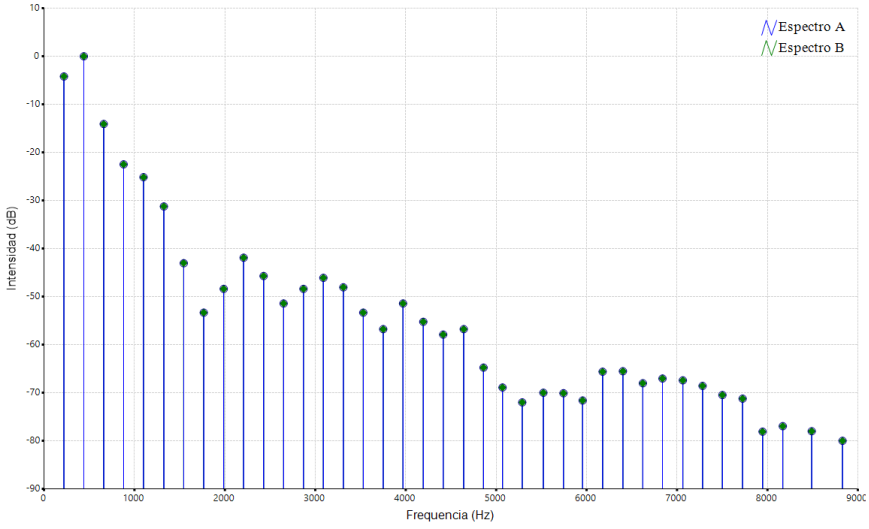


Figura 4.38: Estado final entre los espectros $\mathscr{F}^{\text{saxo}}, \mathscr{F}^{\text{violín}}$.

Calcularemos a continuación la disimilitud media ordenada calculada entre el espectro del saxo y el del piano es $\widehat{\mathscr{D}}(\mathscr{F}^{\text{saxo}}, \mathscr{F}^{\text{piano}}) = 12,6277192553$. Por tanto el resultado que obtenemos es que el espectro del saxo se encuentra más cercano al del violín que al del piano. Podemos observar los siguientes estadios intermedios en el proceso de convergencia del algoritmo:

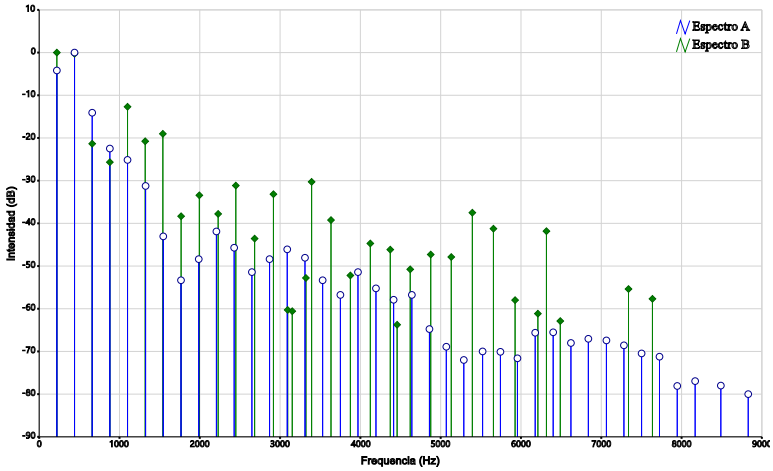


Figura 4.39: Estado inicial entre los espectros $\mathscr{F}^{\text{saxo}}, \mathscr{F}^{\text{piano}}$.

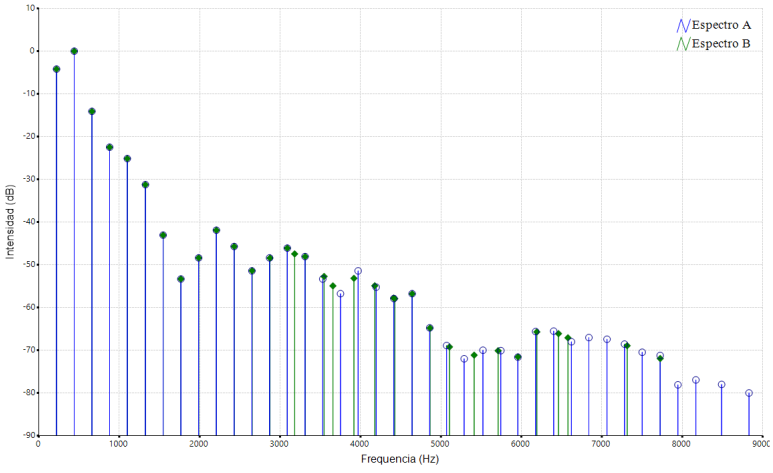


Figura 4.40: Primera iteración entre los espectros $\mathscr{F}^{\text{saxo}}, \mathscr{F}^{\text{piano}}$.

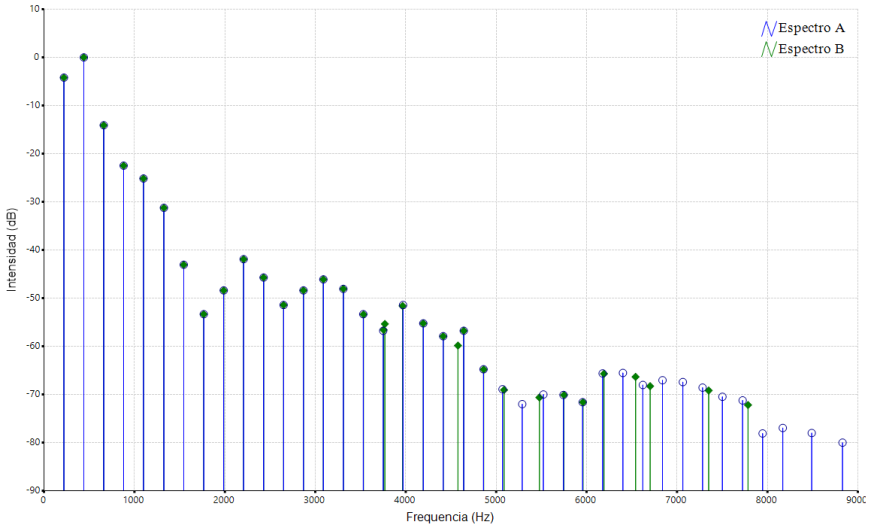


Figura 4.41: Segunda iteración entre los espectros $\mathscr{F}^{\text{saxo}}$, $\mathscr{F}^{\text{piano}}$.

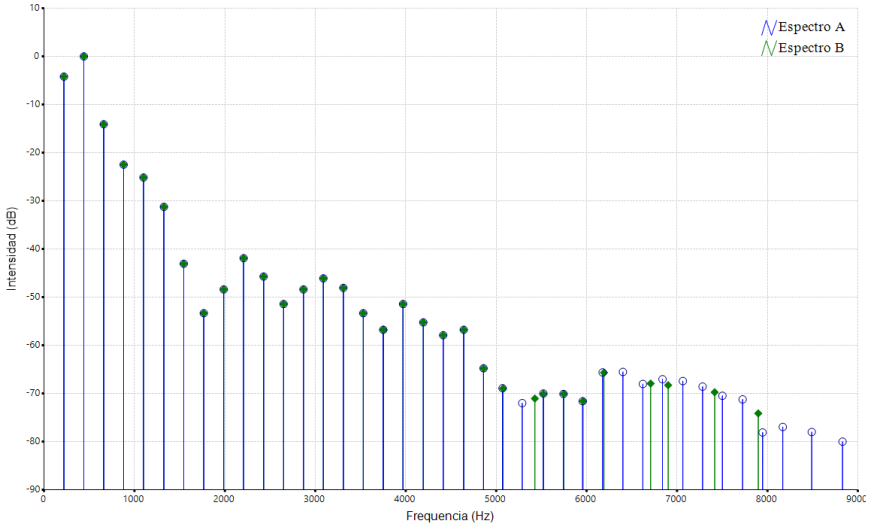


Figura 4.42: Tercera iteración entre los espectros $\mathscr{F}^{\text{saxo}}$, $\mathscr{F}^{\text{piano}}$.

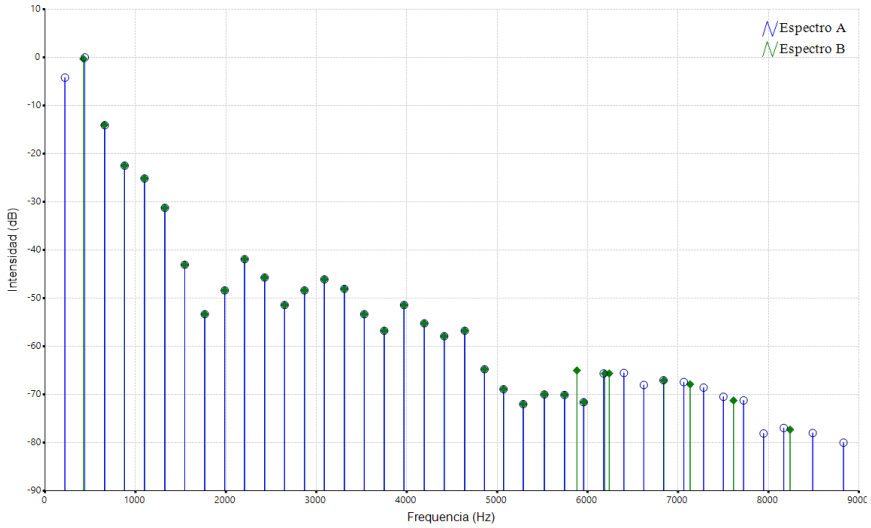


Figura 4.43: Séptima iteración entre los espectros $\mathscr{F}^{\text{saxo}}, \mathscr{F}^{\text{piano}}$.

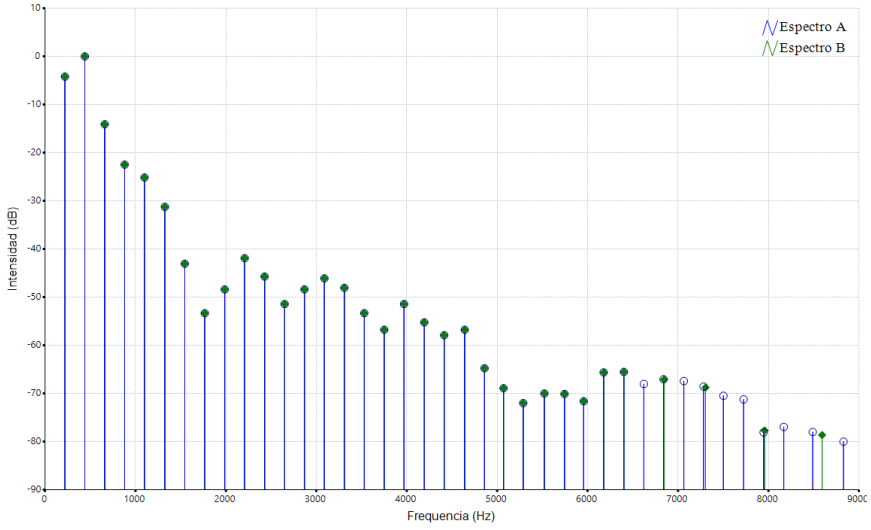


Figura 4.44: Estado final entre los espectros $\mathscr{F}^{\text{saxo}}, \mathscr{F}^{\text{piano}}$.

4.8 Resumen

En este capítulo hemos presentado el software MERCURY, en el que se ha realizado una implementación informática de estos nuevos procedimientos compositivos derivados de las transiciones difusas propuestas en el capítulo anterior. El programa nos ofrece distintos métodos algorítmicos para la generación de variaciones y transiciones entre material musical de carácter melódico, rítmico, armónico o tímbrico de manera accesible para la labor del usuario compositor. Desde la interfaz de usuario, el programa permite configurar los distintos parámetros de los algoritmos FCM, FOCM, o FCT, posibilitando de esta manera generar una amplia diversidad de resultados. El programa permite importar y exportar el material musical inicial y los resultados obtenidos en formato *MusicXML*.

Los ejemplos que se incluyen en el capítulo muestran distintos resultados de experimentos computacionales realizados íntegramente mediante el software MERCURY*, y ponen de manifiesto la potencialidad y aplicabilidad de los métodos empleados en la composición musical asistida por ordenador, ofreciendo al creador del siglo XXI una nueva herramienta compositiva con la que enriquecer el catálogo de recursos compositivos algorítmicos.*

Capítulo 5

Comparación y transiciones entre sistemas de afinación

«De la misma manera que el número es al número, también el espacio lo es al espacio; y en conformidad con los requisitos de la música conocidos, es necesario que la proporción entre espacio y espacio sea la misma que la existente entre sonido y sonido.»

«Unde sicut numerus ad numerum, sic etiam spatium ad spatium; iuxta illud postulatu in Música notissimim, Quae est spacii ad spacium, eandem soni ad sonum proportionoem esse necesse est. Quare ab aequalibus numeris aequalia spatia, a spatiis aequalibus aequales orientur soni.»

(Salinas, 1577, libro II, cap. IV, pág. 49)

5.1 Introducción

En el presente capítulo aglutina las fases finales nuestra investigación en las cuales abordaremos el estudio de los sistemas de afinación, generalizando su definición matemática y proponiendo diversos métodos para estimar tanto la compatibilidad entre un conjunto de notas medidas experimentalmente y un determinado sistema de afinación, como la disimilitud entre dos sistemas de afinación cualesquiera. Por último, procediendo de forma similar a como se hizo anteriormente, se aplicarán los algoritmos FOCM y FCT a la generación de transiciones entre distintos sistemas de afinación. Ampliaremos la implementación informática realizada en Mer-

CURY para contemplar los métodos propuestos en este capítulo y mostraremos los resultados computacionales resultantes de la experimentación con distintas transiciones entre sistemas de afinación, así como los resultados numéricos de diversos cálculos de compatibilidades y disimilitudes.

5.2 Nociones preliminares

5.2.1 El intervalo musical

Según Latham (2009), un intervalo musical queda establecido por la distancia entre la altura de dos notas. Sin embargo, la múltiples disquisiciones teóricas sobre el concepto de intervalo musical se encuentran relacionadas con los distintos planteamientos formales relativos a la construcción de los sistemas de afinación y de escalas y se pueden remontar hasta la aparición de las primeras fuentes sobre teoría musical que han perdurado en nuestros días. La tradición musical de la escuela pitagórica se considera como punto de partida para el estudio del intervalo musical y de la afinación, al menos en la tradición occidental:

> *Hasta ahora no ha habido un consenso general sobre si el hombre primitivo llegó a una escala instrumental siguiendo uno u otro principio, o varios principios simultáneamente, o ningún principio en absoluto. Dado que este es el caso, hay poco que ganar si decidiésemos comenzar nuestro estudio antes de la época de Pitágoras, cuyo sistema de afinación ha tenido una profunda influencia tanto en el mundo antiguo como en el moderno.*[1] *(Barbour, 2004, pág. 1)*

Las primeras discrepancias sobre el concepto de intervalo las encontramos en la dicotomía existente entre dos de las grandes teorías musicales griegas: la tradición pitagórica frente la concepción aristogénica. Pitágoras presenta una definición del intervalo musical basada en las proporciones entre las longitudes de diferentes cuerdas, partiendo con los intervalos de referencia de octava y quinta; sin embargo, Aristógenes basa su sistema en "*leyes naturales de la música que hay que descubrir mediante un buen oído y el uso de la memoria*"(Pajares, 2012, pág. 21), proponiendo un método para la determinación del intervalo basado en la tensión de la cuerda, entendiéndolo como la diferencia de tensión existente entre dos notas (Goldáraz, 1989).

> *Los intervalos, aunque se dan en la melodía de golpe, son numéricamente medibles en la tradición pitagórica; sólo sensorialmente en la aristogénica. (Goldáraz, 1989, pág. 10)*

[1] So far there has been no general agreement as to whether primitive man arrived at an instrumental scale by following one or another principie, several principies simultaneously, or no principie at all. Since this is the case, there is little to be gained by starting our study prior to the time of Pythagoras, whose system of tuning has had so profound an influence upon both the ancient and the modern world. (Barbour, 2004, pág. 1).

de esta forma podemos entender la discrepancia fundamental de la concepción interválica de Pitágoras frente a la de Aristógenes, que se verá retomada en la obra de los grandes tratadistas del renacimiento:

> *[...]la diferencia fundamental ya comentada entre la tradición pitagórica y la aristogénica y que afecta a la terminología. Cuando los teóricos renacentistas empleen términos como «espacio» o «relación entre espacios» para referirse a nota e intervalo respectivamente están siguiendo la tradición pitagórica; cuando usen «tensión» o «distancia-diferencia entre notas», siguen la aristogénica. (Goldáraz, 1989, pág. 11)*

La teoría musical de Aristógenes pasa totalmente desapercibida hasta el siglo XVI a causa de su elevado nivel técnico, de su discrepancia con la teoría pitagórica y de su incompletitud. Tan sólo algunos tratadistas continúan con esta tradición: Cleónides (siglo II d.C.) en su *Harmonica introductio* se muestra como el más importante transmisor de la teoría aristogénica; Aristides Quintilianus (siglos III-IV) se muestra partidario de aristógenes en su tratado *De Música*, y por último Gudentuis (siglo II o IV d.C.) propone en su *Harmonica introductio* una ampliación de las teoría de Aristógenes. Al contrario, la tradición pitagórica será continuada por numerosos autores como Ptolomeo de Alejandría (*Harmonica*, siglo II), Theron de Smyrna (*Expositio rerum mathematicarum ad legendum Platonem utilium*, siglo II), San Agustín (*De música*, 379-389 d.C.), y Boecio (De institucione música, ca.500 d.C.), quien asentará definitivamente la tradición pitagórica en la teoría musical de la Edad Media(Pajares, 2012, pág. 16).
Ya en pleno renacimiento, Zarlino se mostrará partidario de la concepción aristogénica para la definición de los intervalos mientras que adoptará la pitagórica para la medición de los mismos. Salinas directamente adoptará la concepción pitagórica, definiendo un intervalo como una proporción entre números y por tanto entre distancias. Galilei, sin embargo, adoptará una posición en gran parte aristogénica (Goldáraz, 1989, pág. 28).

La razón interválica

Continuando con la concepción pitagórica, nuestra definición actual del intervalo musical se expresa mediante la *proporción* (*razón*, *relación* o *ratio*) entre las frecuencias fundamentales de las dos notas que lo conforman. Como la relación existente entre la frecuencia n producida por una cuerda depende linealmente con la longitud L de la misma, para una tensión T, densidad ρ y diámetro d constantes resulta equivalente hablar de relación de frecuencias y relación de longitudes (Tipler, 1985).

$$f_n = \frac{n}{L}\sqrt{\frac{T}{\rho \cdot \pi \cdot d^2}} \tag{5.1}$$

Entendemos por tanto el concepto de *razón* entre dos notas musicales como el cociente entre sus dos frecuencias fundamentales. Esta definición resulta en una comparación expresada términos relativos, y no absolutos, que resulta invariante frente a transposiciones de octava.

Definición 5.2.1 Razón. *Sean* $f_1, f_2 \in]0, \infty[$ *las frecuencias correspondientes a dos notas cualesquiera. La razón del intervalo musical formado desde la nota* f_1 *hasta la nota* f_2 *se define como*

$$r = \frac{f_2}{f_1}, \quad f_1, f_2 > 0 \tag{5.2}$$

Nótese que $r \in]0, \infty[$ puede tomar como valor cualquier número real positivo mayor que cero. La razón carece de unidades y por tanto será un número adimensional, representando una proporción entre dos frecuencias.

$$[r] = \frac{[f_2]}{[f_1]} = \frac{\cancel{\text{Hz}}}{\cancel{\text{Hz}}} = 1 \tag{5.3}$$

Ejemplo 5.2.1 *Calcúlese la razón del intervalo formado por dos notas cuyas frecuencias son 261,62Hz y 294,39Hz*
Tomando la definición 5.2

$$r = \frac{294{,}39}{261{,}62} = \frac{9}{8} = 1{,}125$$

Ejemplo 5.2.2 *Calcúlese la razón del intervalo formado por dos notas cuyas frecuencias son 261,62Hz y 293,66Hz*

$$r = \frac{293{,}66}{261{,}62} = 1{,}122467701$$

Es fácil comprobar como no siempre será posible escribir mediante una fracción simplificada el valor de una razón cualquiera, de igual forma que no pueden escribirse como fracciones los números irracionales. La razón podrá escribirse como número fraccionario sólo cuando su valor sea un número racional.

El unísono y la octava

Definición 5.2.2 Unísono. *Dos notas de frecuencias $f_1, f_2 \in]0, \infty[$ se encontrarán en relación de unísono siempre que la razón del intervalo que forman tenga valor de 1.*

Ejemplo 5.2.3 *Calcúlese la razón del intervalo formado por dos notas cuyas frecuencias son 261,62Hz y 261,62Hz*

$$r = \frac{261{,}62}{261{,}62} = 1$$

Podemos afirmar que ambas notas forman un intervalo de unísono.

Si $r > 1$ se comprueba trivialmente que $f_2 > f_1$. De hecho, la definición 5.2.1 se construye de modo que la razón tenga valores superiores a la unidad siempre que la segunda frecuencia sea superior a la primera; siendo f_1 la frecuencia de referencia, las razones superiores a la unidad indicarán intervalos *positivos*, es decir, ascendentes, que alcanzan una frecuencia más aguda que f_1. Sin embargo nada impide que la razón pueda tener un valor comprendido en el intervalo $]0, 1[$, es decir, que sea mayor que cero y menor que uno. Si esto sucede se tiene que interpretar que en este caso $f_2 < f_1$, es decir, la segunda frecuencia será inferior a la primera y por tanto el intervalo es *negativo* (descendente), remitiendo f_2 a una frecuencia más grave que f_1.

Definición 5.2.3 Octava. *Dos notas de frecuencias $f_1, f_2 \in]0, \infty[$ se encontrarán en relación de octava siempre que la razón del intervalo que forman tenga valor de 2.*

Ejemplo 5.2.4 *Calcúlese la razón del intervalo formado por dos notas cuyas frecuencias son 261,62Hz y 523,24Hz*

$$r = \frac{523{,}24}{261{,}62} = 2$$

Consecuentemente ambas notas forman un intervalo de octava. Pero...¿qué sucedería en el caso en que $f_1 = 261{,}62$Hz y $f_2 = 130{,}81$Hz? La razón valdría $r = 0{,}5$ y ambas notas estarían relacionadas por un intervalo de octava descendente.

Ejemplo 5.2.5 *Calcúlese la razón del intervalo formado por la frecuencia $f_1 = 261{,}62$Hz y las siguientes frecuencias: 523,24Hz, 1046,48Hz, 2092,96Hz*

$$r = \frac{523{,}24}{261{,}62} = 2 \quad r = \frac{1046{,}48}{261{,}62} = 4 \quad r = \frac{2092{,}96}{261{,}62} = 8$$

Es fácil entender como en el primer caso las dos frecuencias están a distancia de octava ($2 \times 261{,}62$), en el segundo caso se encuentran a octava de la octava ($2 \times 2 \times 261{,}62 = 4 \times 261{,}62$), y en el tercer caso a tres octavas de diferencia ($2 \times 2 \times 2 \times 261{,}62 = 8 \times 261{,}62$). Sobre una misma nota f_1 de referencia la octava de la octava se calculará multiplicando la frecuencia inicial por dos. La enésima octava de la octava se calculará multiplicando por dos dicha frecuencia un total de n veces. Este mismo ejercicio se podría plantear realizando transposiciones de octava descendentes, y por tanto dividiendo entre 2.

En la tradición musical occidental, dos notas separadas por un intervalo de octava se consideran homónimas y por tanto equivalentes. La diferencia en su denominación radicará en su correspondiente índice acústico de octava[2]. Podemos considerar esta equivalencia de octava como uno de los principios básicos de nuestra concepción musical, de forma que el conjunto de notas equivalentes a una nota dada cualquiera se extiende infinitamente hacia el registro agudo y hacia el registro grave mediante sucesivas transposiciones de octava (Lerdahl y Jackendoff, 2003, pág.326).

Principio 5.2.1 Equivalencia de octava*. Dos notas de frecuencias* $f_1, f_2 \in]0, \infty[$ *son equivalentes de octava si se cumple*

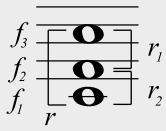
$$\exists n \in \mathbb{Z} \Big/ r = \frac{f_2}{f_1} = 2^n \tag{5.4}$$

donde n representa el número total de octavas de diferencia entre las dos frecuencias. Si $n > 0$, la frecuencia f_2 será más aguda que f_1; si $n < 0$, la frecuencia f_2 será más grave que f_1; por último, si $n = 0$, la frecuencia f_2 es igual a f_1 y se considera que ambas notas forman un intervalo de unísono.

[2]El índice acústico de octavas, (o índice de registro) es el símbolo o conjunto de símbolos usado para representar la altura de las notas prescindiendo del pentagrama. Las nomenclaturas más comunes son las de Riemann, Helmholtz, francesa, inglesa, científica, MIDI, etc. (Latham, 2009, pág. 761).

Operaciones básicas con razones interválicas

Proposición 5.2.1 Suma. *Sean $r_1 = {}^{f_2}/_{f_1}$ y $r_2 = {}^{f_3}/_{f_2}$ dos razones interválicas. La razón resultante de la suma de ambas es $r = r_1 \cdot r_2$*



$$
\begin{aligned}
r_1 = \frac{f_2}{f_1} \quad &\rightarrow \quad f_1 = \frac{f_2}{r_1} \\
r_2 = \frac{f_3}{f_2} \quad &\rightarrow \quad f_3 = r_2 \cdot f_2 \\
r = \frac{f_3}{f_1} = \frac{r_2 \cdot f_2}{{}^{f_2}/_{r_1}} &= \frac{r_2 \cdot f_2 \cdot r_1}{f_2} \\
r = r_1 \cdot r_2 \qquad & (5.5)
\end{aligned}
$$

No resulta complicado extender este resultado para demostrar que la razón de la suma de tres razones interválicas r_1, r_2 y r_3 vale $r = r_1 \cdot r_2 \cdot r_3$. Generalizando, la razón de la suma de un total de n razones $r_1, r_2, \ldots, r_n$ vale

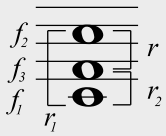
$$r = \prod_{i=1}^{n} r_i \tag{5.6}$$

Además, la operación de suma de razones interválicas cumple con las propiedades conmutativa y asociativa.

$$r = (r_1 \cdot r_2) \cdot r_3 = r_1 \cdot (r_2 \cdot r_3)$$

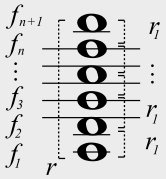
$$r = r_1 \cdot r_2 \cdot r_3 = r_2 \cdot r_1 \cdot r_3 = r_2 \cdot r_3 \cdot r_1 = r_3 \cdot r_2 \cdot r_1 = r_3 \cdot r_1 \cdot r_2$$

Proposición 5.2.2 Resta. *Sean $r_1 = {}^{f_2}/_{f_1}$ y $r_2 = {}^{f_3}/_{f_1}$ dos razones interválicas. La razón resultante de la resta de la primera menos la segunda es $r = {}^{r_1}/_{r_2}$*



$$
\begin{aligned}
r_1 = \frac{f_2}{f_1} \quad &\rightarrow \quad f_2 = r_1 \cdot f_1 \\
r_2 = \frac{f_3}{f_1} \quad &\rightarrow \quad f_3 = r_2 \cdot f_1 \\
r = \frac{f_2}{f_3} &= \frac{r_1 \cdot f_1}{r_2 \cdot f_1} \\
r = \frac{r_1}{r_2} \qquad & (5.7)
\end{aligned}
$$

Proposición 5.2.3 Multiplicación. *Sea $r_1 = {}^{f_2}/_{f_1}$ una razón interválica cualquiera. La razón resultante de la suma de n intervalos iguales r_1 es $r = r_1^n$*

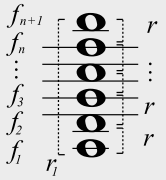


$$
\begin{aligned}
f_2 &= f_1 \cdot r_1 \\
f_3 &= f_2 \cdot r_1 = f_1 \cdot r_1^2 \\
f_4 &= f_3 \cdot r_1 = f_1 \cdot r_1^3 \\
&\vdots \\
f_n &= f_1 \cdot r_1^{n-1} \\
f_{n+1} &= f_1 \cdot r_1^n \\
r = \frac{f_{n+1}}{f_1} &= \frac{f_1 \cdot r_1^n}{f_1} = r_1^n \\
r &= r_1^n
\end{aligned}
\tag{5.8}
$$

Llegamos al mismo resultado mediante la expresión 5.6 sumando n veces el mismo intervalo r_1

$$
r = \prod_{i=1}^{n} r_1 = r_1^n \tag{5.9}
$$

Proposición 5.2.4 División. *Sea r_1 una razón interválica cualquiera. La razón resultante dividirla en n intervalos iguales es $r = \sqrt[n]{r_1}$*



$$
\begin{aligned}
f_2 &= f_1 \cdot r \\
f_3 &= f_2 \cdot r = f_1 \cdot r^2 \\
f_4 &= f_3 \cdot r = f_1 \cdot r^3 \\
&\vdots \\
f_n &= f_1 \cdot r^{n-1} \\
f_{n+1} &= f_1 \cdot r^n \\
r_1 = \frac{f_{n+1}}{f_1} &= \frac{f_1 \cdot r^n}{f_1} = r^n \\
r &= \sqrt[n]{r_1}
\end{aligned}
\tag{5.10}
$$

Proposición 5.2.5 Logaritmación. *Sean r_1 y r_2 dos razones interválicas cualesquiera. El número de veces n que el intervalo r_2 está contenido en el intervalo r_1 es $n = \log_{r_2} r_1$*

$$r_2^n = r_1; \quad \log r_2^n = \log r_1; \quad n \cdot \log r_2 = \log r_1$$

$$n = \frac{\log r_1}{\log r_2} = \log_{r_2} r_1 \tag{5.11}$$

Razón interválica circunscrita al ámbito de una octava

En las próximas secciones veremos que para calcular las razones de las notas en distintos sistemas de afinación resulta necesario reducir dicha razón al ámbito de una octava[3], esto es, restringir su valor al intervalo $[1, 2[$. ¿Cómo es posible realizar dicha operación? Supongamos que tenemos una razón cualquiera r, que ha sido calculada mediante un determinado sistema de afinación y que tiene un valor superior a la octava, por tanto $r > 2$. La forma más rudimentaria de reducir dicho intervalo al ámbito de la octava es dividiendo sucesivamente la razón r entre 2 (es decir, restar octavas sucesivamente) hasta que la razón sea menor que 2.

Ejemplo 5.2.6 *Reduzca la razón $r = 656{,}84$ al intervalo de octava. ¿Cuántas octavas se han restado?*

$$\frac{656{,}84}{2} = 328{,}42; \ \frac{328{,}42}{2} = 164{,}21; \ \frac{164{,}21}{2} = 82{,}105; \ \frac{82{,}105}{2} = 41{,}0525;$$
$$\frac{41{,}0525}{2} = 20{,}52625; \ \frac{20{,}52625}{2} = 10{,}263125; \ \frac{10{,}263125}{2} = 5{,}1315625;$$
$$\frac{5{,}1315625}{2} = 2{,}56578125; \ \frac{2{,}56578125}{2} = 1{,}282890625;$$

La razón queda $r = 1{,}282890625$. Se han restado un total de 9 octavas.

¿Cómo hemos de proceder si, en caso contrario, nuestra razón r es menor que la unidad y por tanto se trata de un intervalo *descendente*? En este caso tendremos que sumar octavas sucesivas hasta que la razón del intervalo sea mayor que 1, tal y como se muestra en el siguiente ejemplo:

[3] Véase el principio de equivalencia de octava 5.2.1.

Ejemplo 5.2.7 *Circunscríbase la razón $r = 0{,}315$ al intervalo de octava. ¿Cuántas octavas se han sumado?*

$$0{,}315 \cdot 2 = 0{,}63;\ 0{,}63 \cdot 2 = 1{,}26;$$

La razón queda $r = 1{,}26$. Se han sumado un total de 2 octavas.

Basándonos en una expresión similar a la propuesta por Liern (2005, pág.41)

$$f^* \coloneqq \log_2(\frac{f}{f_0}) - \lfloor \log_2(\frac{f}{f_0}) \rfloor; \quad f^* \in [0, 1[\quad \text{(Liern, 2005)}$$

Como $r = \frac{f}{f_0}$ y además $r^* \coloneqq 2^{f^*}$, podemos definir una transformación sobre cualquier razón r de tal manera que su valor quede limitado al intervalo $[1, 2[$.

$$\begin{aligned} f^* &\coloneqq \log_2 r - \lfloor \log_2 r \rfloor \\ r^* &\coloneqq 2^{f^* \coloneqq \log_2 r - \lfloor \log_2 r \rfloor} \\ r^* &= 2^{\log_2 r} \cdot 2^{-\lfloor \log_2 r \rfloor} \\ r^* &= r \cdot 2^{-\lfloor \log_2 r \rfloor}; \quad r^* \in [1, 2[\end{aligned} \tag{5.12}$$

Donde el r^* representa la transformada de la razón, circunscrita al intervalo $[1, 2[$ y $\lfloor x \rfloor$ es la *función suelo* de x. Es fácil comprobar como el número de octavas $m \in \mathbb{Z}$ que hay que sumar o restar a r para limitarla a $[1, 2[$ es $-\lfloor \log_2 r \rfloor$

$$m = \log_2 \frac{r^*}{r} = \log_2 2^{-\lfloor \log_2 r \rfloor} = -\lfloor \log_2 r \rfloor;$$

5.2.2 Unidades interválicas logarítmicas

Como hemos visto, la definición del intervalo musical como razón, es decir, como cociente entre dos frecuencias, implica la utilización de la multiplicación y de la división para calcular la operación suma y resta de razones, respectivamente. Además, la multiplicación de un intervalo implica la potenciación de razones y al contrario, la división de un intervalo en partes iguales, requiere del uso de raíces. Estas operaciones, aunque exactas, no resultan cómodas ni proporcionan unos resultados rápidamente inteligibles; sin embargo, gracias a la introducción de las unidades interválicas logarítmicas resultará mucho más fácil realizar las operaciones interválicas básicas.

Conceptualmente resulta sencillo el planteamiento general de estas unidades de medida; supongamos que nuestro intervalo de referencia es el semitono del temperamento igual. ¿Cuántos semitonos contiene un intervalo de octava? La respuesta es inmediata: 12. ¿Cuántos semitonos contiene un intervalo de doble octava? La respuesta es trivial: exactamente el doble, es decir, 24. La idea subyacente en el funcionamiento de todas las medidas logarítmicas es la misma: primero se definirá un intervalo mínimo de referencia a y se calculará su razón r_a. Posteriormente se averiguará, mediante la utilización de logaritmos, cuántos intervalos de referencia r_a están contenidos en un intervalo r cualquiera tal que

$$(r_a)^n = r; \quad n = \frac{\log r}{\log r_a}$$

donde r_a es la razón de nuestro intervalo de referencia, r es el intervalo que queremos expresar en términos del intervalo de referencia a, y n es el número de intervalos de referencia a que están contenidos en r. La utilización de logaritmos nos permite trabajar directamente con el exponente (en el caso de la ecuación anterior, el número n) de la ecuación, gracias a la propiedad $\log x^y = y \cdot \log x$. De esta manera conseguiremos simplificar las operaciones interválicas, especialmente si seleccionamos un intervalo r_a adecuado para nuestro propósito, ya que no trabajaremos con las razones, habitualmente fracciones o irracionales, sino que utilizaremos un número que especificará el total de intervalos de referencia r_a contenidos en el intervalo a caracterizar r. Los pioneros en el uso de logaritmos para el cálculo de los tamaños interválicos fueron Bonaventura Cavalieri[4], Juan Caramuel de Lobkowitz[5] y Lemme Rossi[6]. También Christiaan Huygens[7] utiliza estos métodos logarítmicos en su sistema de afinación (Coul, 2015).

[4]Bonaventura Cavalieri (Milán, 1598 - Bolonia, 1647), matemático italiano perteneciente a la orden de los Jesuatos, alumno de Galileo Galilei. Está considerado como uno de los precursores del cálculo infinitesimal moderno así como pionero en Italia en la utilización de logaritmos. Es célebre su teoría de los *indivisibles*, en la que estudia magnitudes geométricas discretas compuestas a partir de un número infinito de elementos indivisibles (Alexander, 2015).

[5]Juan Caramuel de Lobkowitz (Madrid, 1606 - Vigevano, 1682), filósofo, matemático, lógico, lingüista y monje cisterciense español. Gran erudito y prototipo del hombre renacentista, se interesó sobre lengua, literatura, teatro, poesía, pedagogía, criptografía, filosofía, teología, historia, política, música, pintura, escultura, arquitectura, matemáticas, física, astronomía, etc. Su obra *Mathesis biceps* contiene la primera descripción impresa del sistema binario (Knuth, 1969, pág.183), explicando los principios generales de los número en cualquier base n y poniendo de manifiesto las ventajas que en ocasiones tiene el realizar cambios de base para resolver determinados problemas. Fue también el primero en publicar en españa tablas de logaritmos (Fernández, 1919).

[6]Lemme Rossi (- 1673†), músico y teórico italiano. Fue el primero en publicar una propuesta de un temperamento igual consistente en dividir la octava en 31 notas, anticipándose a la misma idea propuesta por Christiaan Huyghens (Wardhaugh, 2017, pág.37).

[7]Christiaan Huygens (La Haya, 1629 - ibidem, 1695), astrónomo, físico y matemático neerlandés. Su contribución más importante en el terreno de la música consiste en su temperamento igual de 31 notas por escala (Meyer, 1951).

El cent

El cent es un microintervalo musical resultante de dividir una octava en 1200 partes iguales. Fue propuesto por Alexander John Ellis [8] en 1884, presentado tanto en su artículo On the musical scales of various nations como en un apéndice de su traducción del tratado de Von Helmholtz *Die Lehre von den Tonempfindungen als physiologische Grundlage für die Theorie der Musik*. Ellis, considerado uno de los padres de la etnomusicología, pretendía utilizar el cent para el estudio de escalas musicales alejadas de la tradición occidental, sin embargo esta unidad pronto se instituyó como la unidad de medida de intervalos estandarizada, ya que proporciona valores muy cómodos para denominar a los intervalos (especialmente a los intervalos del temperamento igual) y además, a efectos prácticos, es suficiente con redondear al cent más próximo, no siendo necesario trabajar con decimales. La razón correspondiente a un cent es la siguiente:

$$r_{1cent} = \sqrt[1200]{2} = 2^{1/1200} = 1{,}0057779\ldots \tag{5.13}$$

Por tanto podemos calcular el número de cents contenidos en un intervalo r cualquiera, ya que

$$\begin{gathered}(r_{1cent})^n = r \\ (2^{1/1200})^n = 2^{n/1200} = r \\ \log_2 2^{n/1200} = \log_2 r \\ \frac{n}{1200} \cdot \log_2 2 = \log_2 r \\ n = 1200 \cdot \log_2 r\end{gathered}$$

Si a la variable n la denominamos razón en cents*, tenemos*

$$r_{cents} = 1200 \cdot \log_2 r \tag{5.14}$$

Efectuando un sencillo cambio de base en el logaritmo, podemos aproximar la expresión anterior para calcular la razón en cents mediante logaritmos en base diez

$$\begin{aligned} r_{cents} &= 1200 \cdot \log_2 r \\ r_{cents} &= \frac{1200}{\log 2} \cdot \log r \end{aligned}$$

[8] Alexander John Ellis, (Middlesex, 1814 - Londres, 1890) matemático y filólogo inglés.

Podemos redondear a dos decimales el valor de $^{1200}/_{\log 2} = 3986{,}313714\ldots$ resultando la siguiente expresión

$$r_{cents} \approx 3986{,}31 \cdot \log r \tag{5.15}$$

que nos permite calcular de forma suficientemente precisa el número de cents r_{cent} contenidos en un intervalo cualquiera expresado mediante la razón r. Despejando de la ecuación 5.14 obtenemos la expresión para calcular el valor de la razón r asociada a un número de cents cualquiera r_{cents}

$$\begin{gathered} r_{cents} = 1200 \cdot \log_2 r \\ \frac{r_{cents}}{1200} = \log_2 r \\ 2^{r_{cents}/1200} = 2^{\log_2 r} = r \\ \\ r = 2^{r_{cents}/1200} \end{gathered} \tag{5.16}$$

O bien despejando de la ecuación 5.15

$$\begin{gathered} r_{cents} = 3986{,}31 \cdot \log r \\ \frac{r_{cents}}{3986{,}31} = \log r \\ 10^{r_{cents}/3986{,}31} = 10^{\log r} = r \\ \\ r = 10^{r_{cents}/3986{,}31} \end{gathered} \tag{5.17}$$

Ejemplo 5.2.8 *Calcúlese el número de cents contenidos en el intervalo de octava (razón $r = 2$).*

$$r_{cents} = 3986{,}31 \cdot \log 2 = 1199{,}9989 \approx 1200\text{cents}$$

Ejemplo 5.2.9 *Calcúlese el número de cents contenidos en el intervalo formado por las frecuencias 679,45Hz y 882,14Hz.*

$$r_{cents} = 3986{,}31 \cdot \log \frac{882{,}14}{679{,}45} \approx 451{,}97\text{cents}$$

Comentario 5.2.1 *Nótese como los intervalos expresados en cents pueden ser sumados y restados directamente*[9].

$$r_1 = \frac{f_2}{f_1}; \quad r_2 = \frac{f_3}{f_2}; \quad r = r_1 \cdot r_2$$
$$1200 \cdot \log_2 r = 1200 \cdot \log_2 (r_1 \cdot r_2)$$
$$1200 \cdot \log_2 r = 1200 \cdot \log_2 r_1 + 1200 \cdot \log_2 r_2$$
$$r^{cents} = r_1^{cents} + r_2^{cents}$$

Ejemplo 5.2.10 *Súmese estos tres intervalos expresados en cents: 386 cents, 1200 cents, 316 cents y calcúlese la razón del intervalo resultante:*

$$r_{cents} = 386 + 1200 + 316 = 1902 \quad \text{cents}$$
$$r = 10^{1902/3986,31} = 3{,}000081049 \approx 3$$

Ejemplo 5.2.11 *Divídase el intervalo de octava ($r = 2$), expresado en cents, en doce partes iguales.*

$$r_{cents} = \frac{1}{12} \cdot 1200 \cdot log_2 2 = 100 \quad \text{cents}$$

El Savart

Se define como 1/301 de parte de la octava por Joseph Sauveur[10] in 1696 bajo el nombre de *Eptaméride*, una séptima parte de la *Méride*. Posteriormente en el siglo XX está unidad será denominada como *Savart* por el físico francés Félix Savart[11] (Coul, 2015).

Méride y Heptaméride

Se define por Joseph Sauveur como 1/43 de la parte de una octava. La méride y la heptaméride (1/7 de la parte de la méride) fueron las primeras medidas logarítmicas de intervalos propuestas (Coul, 2015).

[9] Comentario válido para cualquier unidad de medida de intervalos logarítmica.
[10] Joseph Sauveur (1653-1716), matemático y físico francés.
[11] Félix Savart (1791-1841), físico francés.

El Grad

Propuesto por Andreas Werckmeister [12], se define como 1/12 parte de la coma pitagórica, y es por tanto la diferencia existente entre una quinta pitagórica (3/2) y una quinta del temperamento igual ($2^{7/12}$). Su razón es $3 \cdot 2^{-19/12}$, equivalente a 1.95500087 cents, aunque eventualmente se aproxima 2 (Coul, 2015).

El Schisma

El Schisma es la diferencia existente entre la coma pitagórica ($3^{12}/2^{19}$) y la coma sintónica ($\frac{3^4}{5 \cdot 2^4}$). Fue introducido por Boecio a principios del siglo VI en su tratado *De institutione música*. Su razón es $\frac{3^8 \cdot 5}{2^{15}}$, equivalente a 1.953720788 cents. Aunque muy parecidos en valor, no debe confundirse el schisma con el grad. El grad es ligeramente superior (0,00128 cents) (Coul, 2015).

5.2.3 El temperamento igual

El temperamento igual es el sistema de afinación utilizado actualmente de forma sistemática y estandarizada en la tradición musical occidental. Supone una división de la octava en doce partes iguales, denominadas semitonos (segunda menor). La razón de un semitono según el temperamento igual será la siguiente:

$$r_{2^{\text{a}}m} = \sqrt[12]{2} = 2^{1/12} = 1{,}059463094\ldots \tag{5.18}$$

Es fácil comprobar utilizando la expresión 5.14 como un semitono según el temperamento igual contiene exactamente 100 cents:

$$r_{2^{\text{a}}m} = 1200 \cdot \log_2 2^{1/12} = 100\,\text{cents} \tag{5.19}$$

No es extraño este resultado ya que Ellis, el creador del cent, establece un maridaje perfecto entre cents y temperamento igual; la definición del cent contempla 1200 cents en una octava, por tanto un semitono contiene una doceava parte de estos cents (1200/12 = 100cents). A diferencia de otras unidades de medida logarítmica de intervalos, el cent está específicamente diseñado para el temperamento igual. Como en este sistema de afinación todos los semitonos son iguales, todos los tonos (formados por dos semitonos) también serán iguales. De hecho todas las terceras menores (tres semitonos) también serán iguales, y lo mismo sucederá con terceras mayores, cuartas, etc. Para el cálculo de la razón de un intervalo será únicamente necesario conocer el número de semitonos que éste contiene.

[12] Andreas Werckmeister (1645-1706), compositor y teórico musical alemán.

$$
\begin{aligned}
r_{2^{\underline{a}}m} &= (2^{1/12})^1 = 2^{1/12} \\
r_{2^{\underline{a}}M} &= (2^{1/12})^2 = 2^{2/12} \\
r_{3^{\underline{a}}m} &= (2^{1/12})^3 = 2^{3/12} \\
r_{3^{\underline{a}}M} &= (2^{1/12})^4 = 2^{4/12} \\
&\vdots \\
r_n &= 2^{n/12} \qquad (5.20)
\end{aligned}
$$

donde n es el número de semitonos que contiene el intervalo.

Nótese que si $n = 0, r_0 = 1$ (unísono) y si $n = 12, r_{12} = 2$ (octava). Podemos extender este resultado: si $n = 24, r_{24} = 4$(quinceava o doble octava) y si $n = -12, r_{-12} = 0{,}5$(octava inferior). El resultado anterior expresado en cents es aún más sencillo:

$$
\begin{aligned}
r_n^{cents} &= 1200 \cdot \log_2 2^{n/12} = {}^{1200}\!/_{12} \cdot n \\
r_n^{cents} &= 100 \cdot n \qquad (5.21)
\end{aligned}
$$

En la siguiente tabla se muestran las razones y el número de cents de cada uno de los doce intervalos distintos existentes en el temperamento igual. Los intervalos enarmónicos tienen igual razón y número de cents[13].

Tabla 5.1: Razones y cents de los doce primeros intervalos según el temperamento igual.

Intervalo	Razón	Cents	Intervalo	Razón	Cents
$2^{\underline{a}}$ menor	$2^{1/12}$	100	$5^{\underline{a}}$ Justa	$2^{7/12}$	700
$2^{\underline{a}}$ Mayor	$2^{2/12}$	200	$6^{\underline{a}}$ menor	$2^{8/12}$	800
$3^{\underline{a}}$ menor	$2^{3/12}$	300	$6^{\underline{a}}$ Mayor	$2^{9/12}$	900
$3^{\underline{a}}$ Mayor	$2^{4/12}$	400	$7^{\underline{a}}$ menor	$2^{10/12}$	1000
$4^{\underline{a}}$ Justa	$2^{5/12}$	500	$7^{\underline{a}}$ Mayor	$2^{11/12}$	1100
$4^{\underline{a}}$ Aumentada	$2^{6/12}$	600	Octava	$2^{12/12}$	1200

[13] i.e. una cuarta aumentada tiene el mismo número de semitonos que una quinta disminuida, un total de seis, por tanto 600 cents.

5.2.4 La serie armónica

Una nota producida por un instrumento musical no está constituida únicamente por su frecuencia fundamental; existen muchas otras frecuencias, denominadas *frecuencias armónicas* (también parciales, sobretonos, o armónicos), que se producen al mismo tiempo que la frecuencia fundamental y que coexisten en la totalidad del sonido. La contribución de las intensidades relativas de cada una de estas frecuencias armónicas constituye, desde el punto de vista de la acústica, lo que en música se entiende por timbre. El análisis de Fourier, mediante algoritmos como la *Fast Fourier Transform* (FFT), es capaz de descomponer el timbre de cualquier sonido mostrando tanto las frecuencias constituyentes como sus intensidades relativas. Estas frecuencias armónicas se producen como consecuencia de la resonancia de tipo débil, es decir, en el seno de un medio con bajos coeficientes de amortiguamiento (Marion, 1975, pág. 135), de los distintos modos propios de vibración que generan ondas estacionarias sobre dicho medio. En el caso de los instrumentos musicales, la existencia y amplificación de estas frecuencias estacionarias es un factor clave para determinar la calidad tímbrica y la sensación de altura definida del sonido, jugando por tanto un papel clave en el diseño y construcción de instrumentos musicales. Los medios típicos de estudio de la acústica musical son las cuerdas y tubos (abiertos o cerrados, cilíndricos o cónicos). En dichos medios se producirán ondas estacionarias cuyas intensidades determinarán el espectro de frecuencias característico de cada instrumento. El estudio básico de la acústica musical se centra únicamente en la determinación, mediante la introducción de determinadas restricciones geométricas, de aquellas frecuencias armónicas que son permitidas porque producen interferencia constructiva y por tanto generan ondas estacionarias (Tipler, 1985). El análisis de las intensidades relativas características de cada frecuencia armónica se reserva para un análisis más profundo de la acústica de los instrumentos musicales.
La expresión general de las frecuencias armónicas en el caso en el que el medio oscilador sea una cuerda, el aire contenido por un tubo cilíndrico abierto, o el aire contenido en un tubo cónico cerrado es la siguiente (Calvo-Manzano, 1991, pág.51):

$$f_n = \frac{n \cdot v}{2L} \tag{5.22}$$

donde v es la velocidad del sonido en el medio, L es la longitud del medio y $n \in [1, 2, 3, \ldots, \infty[$ es el número de armónico. Observamos que la nota fundamental, en términos de acústica, se denomina *primer armónico* y además que, desde el punto de vista teórico, se pueden producir infinitas frecuencias armónicas[14].

[14]Desde el punto de vista teórico, existen infinitos armónicos ya que n puede coger infinitos valores enteros mayores que cero, sin embargo, esto no significa que cualquier frecuencia pueda ser una frecuencia armónica.

Es fácil comprobar como todas las frecuencias armónicas son múltiplos de la fundamental, esto es, múltiplos del primer armónico, con independencia de cual sea la frecuencia de éste:

$$
\begin{aligned}
f_n &= \frac{n \cdot v}{2L} \\
f_1 &= \frac{1 \cdot v}{2L} = \frac{v}{2L} \\
f_2 &= \frac{2 \cdot v}{2L} = 2 \cdot f_1 \\
&\vdots \\
f_n &= \frac{n \cdot v}{2L} = n \cdot f_1 \\
f_n &= n \cdot f_1 \qquad (5.23)
\end{aligned}
$$

Este resultado es sumamente importante ya que nos permitirá estudiar los intervalos musicales existentes entre los distintos armónicos, desde un punto de vista completamente teórico, aplicable a cualquier instrumento musical que cumpla con la condición 5.23, sin necesidad de conocer los valores exactos de las frecuencias de los armónicos que estamos estudiando.

Ejemplo 5.2.12 *Calcúlese la razón del intervalo formado por las frecuencias armónicas número 5 y número 4 de la serie armónica de la cuarta cuerda del violín. Exprese el resultado en cents. (Tómese* $La_4 = 442Hz$*).*

No necesitamos conocer las frecuencias de los armónicos 5 y 4 de la cuerda Sol del violín, ya que:

$$r = \frac{f_5}{f_4} = \frac{5 \cdot f_1}{4 \cdot f_1} = \frac{5}{4}$$

Expresamos ahora el resultado en cents:

$$r_{cents} = 3986{,}31 \cdot \log \frac{5}{4} = 386{,}31 \,\text{cents}$$

Los intervalos naturales

La serie armónica nos proporciona una infinita variedad de intervalos y microintervalos cuyas razones pueden ser calculadas de forma muy sencilla, tal y como se muestra en el ejemplo 5.2.12. Sin embargo, estos intervalos se encuentran circunscritos a un sistema de afinación propio: el sistema natural, perteneciente a la física ondulatoria, la acústica musical y la resonancia. A excepción del intervalo de octava, ninguno de ellos coincide con los intervalos calculados según el temperamento igual, sistema de afinación que utilizamos de forma estandarizada actualmente. En la siguiente figura podemos ver una representación aproximada[15] de la notación que tendrían las primeras 16 frecuencias armónicas de la nota Do_2 sobre el pentagrama:

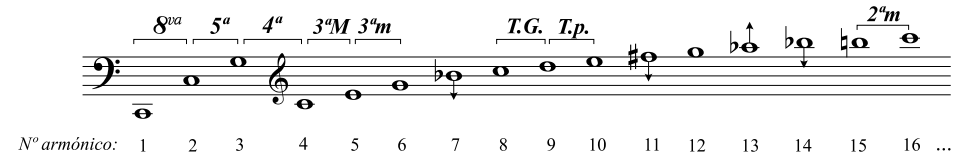


Figura 5.1: Primeros 16 armónicos de la nota Do_2 representados de forma aproximada en nuestra notación actual.

A continuación definiremos algunos intervalos naturales que han jugado un papel relevante en la tradición histórica del estudio de los sistemas de afinación occidentales.

Tabla 5.2: Tabla de intervalos naturales.

Intervalo	Razón	Cents	Intervalo	Razón	Cents
Octava	2/1	1200,00	Unísono	1/1	0,00
Quinta	3/2	701,955	Cuarta	4/3	498,045
3ª Mayor	5/4	386,314	6ª menor	8/5	813,686
3ª menor	6/5	315,641	6ª Mayor	5/3	884,359
Tono Grande	9/8	203,910	7ª menor pequeña	16/9	996,090
Tono pequeño	10/9	182,404	7ª menor Grande	9/5	1017,596
2ª menor	16/15	111,731	7ª Mayor	15/8	1088,269

[15] Nótese que con respecto a su nota equivalente calculada mediante temperamento igual y tomando como referencia la nota Do, los armónicos número 7 y 14 se quedan 31.17cents bajos, el armónico 11 se queda 48.68 cents bajo, y el armónico 13 se queda 40.53 cents alto. En la ilustración, las flechas ascendentes o descendentes representan dicha desviación. En el resto de armónicos, a excepción de la octava y sus múltiplos, esta discrepancia de los valores de las frecuencias naturales con las del temperamento igual sigue existiendo pero no es tan acusada.

Es fácil demostrar como cada intervalo formado por dos armónicos consecutivos es cada vez más pequeño. Por orden de aparición, el intervalo de octava es el primero, encontrándose formado entre los armónicos 2 y 1; el intervalo de quinta justa natural entre los armónicos 3 y 2; el intervalo de cuarta justa natural entre los armónicos 4 y 3; la tercera Mayor natural entre los armónicos 5 y 4; la tercera menor natural entre los armónicos 6 y 5. Las terceras que aparecen a continuación, involucrando el armónico 7 tradicionalmente han sido desechadas. Aparecen dos tipos de tonos: el tono Grande, formado por los armónicos 9 y 8, y el tono pequeño, formado por los armónicos 10 y 9. Nuevamente, el tono entre el armónico 11 y 10 es desechado, así como el resto de intervalos que aparecen hasta llegar a la segunda menor natural, formada por los armónicos 16 y 15. Estos intervalos aparecen infinitas veces en armónicos superiores, como por ejemplo el intervalo de octava que aparece nuevamente entre los armónicos 4:2, 8:4, etc.

5.3 Los sistemas de afinación

5.3.1 Los sistemas de afinación

Basándonos en la definición expuesta por Liern (2015, pág.206), proponemos a continuación una definición matemática para generalizar el concepto de sistema de afinación.

Definición 5.3.1 Sistema de afinación. *Sea* $\mathcal{B} = \{\beta_i\}_{i=1}^{k}$, *con* $\beta_i \in [1, 2[$ *un conjunto de* k *razones interválicas. Sea* $\mathcal{F} = \{f_i\}_{i=1}^{k}$ *un conjunto funciones* $f_i \colon \mathbb{Z} \to \mathbb{Z}$, $i = \{1, 2, \ldots, k\}$. *Un sistema de afinación formado por los intervalos* $\mathcal{B}$ *y las funciones* $\mathcal{F}$ *se define como el conjunto*

$$\mathcal{S}_{\mathcal{B}}^{\mathcal{F}} \coloneqq \left\{ r_n \,\middle|\, r_n = \left[\prod_{i=1}^{k} \beta_i^{f_i(n)} \right]^*, n \in \mathbb{Z} \right\} \tag{5.24}$$

donde $*$ es la transformación definida en 5.12 y por tanto $r_n \in [1, 2[$, $\forall n \in \mathbb{Z}$.

Ejemplo 5.3.1 *El Sistema Pitagórico abierto. Tal y como propone Liern (2015), sea* f_0 *una frecuencia de referencia, una frecuencia* f *está afinada según el sistema pitagórico si existen* $n, m \in \mathbb{Z}$ *tales que* $\beta^n \cdot 2^m \cdot f_0 = f$, *donde* $\beta = 3/2$ *es la razón de un intervalo de quinta natural. Exprese el sistema de afinación Pitagórico utilizando la definición .*

Sean $\mathcal{B} = \{\beta_1 = 3/2\}$, y $\mathcal{F} = \{f_1(n) = n\}$. Según la definición 5.3.1, el sistema de afinación pitagórico es el conjunto

$$\mathcal{S}_{pit} = \{r_n = [\beta_1^n]^*\}, \quad n \in \mathbb{Z}, \quad \beta_1 = 3/2 \tag{5.25}$$

La razón del intervalo pitagórico comprendido entre una nota f y otra nota de referencia f_0 (usualmente el *Do*) vendrá determinada únicamente por el número $n \in \mathbb{Z}$ de quintas naturales β con las que dicho intervalo se construye (positivas o negativas) contando por la espiral de quintas desde la nota de referencia f_0 hasta la nota f. Además, utilizando la transformación 5.12 podemos calcular la razón y circunscribirla al intervalo $[1, 2[$

$$r_{pit}^*(n) = \beta_1^n \cdot 2^{-\lfloor \log_2 \beta_1^n \rfloor} \tag{5.26}$$

La diferencia entre $r^*_{pit}(0)$ y $r^*_{pit}(12)$ es la famosa *coma pitagórica*.

$$r_{cp} = \frac{r^*_{pit}(12)}{r^*_{pit}(0)} = \frac{(3/2)^{12} \cdot 2^{-\lfloor \log_2 (3/2)^{12} \rfloor}}{(3/2)^0 \cdot 2^0} = \frac{3^{12}}{2^{19}} \quad (23{,}4600104 \text{ cents})$$

La diferencia entre $r^*_{pit}(4)$ y el intervalo de tercera mayor natural (5/4) es la no menos famosa *coma sintónica*.

$$r_{cs} = \frac{r^*_{pit}(4)}{5/4} = \frac{3^4 \cdot 2^2 \cdot 2^{-2}}{5 \cdot 2^4} = 2^{-4} \cdot 3^4 \cdot 5^{-1} = 81/80 \quad (21{,}5063896 \text{ cents})$$

Tabla 5.3: Cálculo de las 12 primeras notas del sistema pitagórico abierto.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot 2^0$	0,00	Fa♯	6	$\beta_1^6 \cdot 2^{-3}$	611,73
Do♯	7	$\beta_1^7 \cdot 2^{-4}$	113,69	Sol	1	$\beta_1^1 \cdot 2^0$	701,96
Re	2	$\beta_1^2 \cdot 2^{-1}$	203,91	Sol♯	8	$\beta_1^8 \cdot 2^{-4}$	815,64
Mi♭	-3	$\beta_1^{-3} \cdot 2^2$	294,13	La	3	$\beta_1^3 \cdot 2^{-1}$	905,87
Mi	4	$\beta_1^4 \cdot 2^{-2}$	407,82	Si♭	-2	$\beta_1^{-2} \cdot 2^2$	996,09
Fa	-1	$\beta_1^{-1} \cdot 2^1$	498,04	Si	5	$\beta_1^5 \cdot 2^{-2}$	1109,78

En el artículo de Liern (2015) encontramos un interesante método de calcular las distintas razones de las notas en el sistema pitagórico, así como determinar sus correspondientes alteraciones. Podemos expresar la ecuación 5.26 de la siguiente manera:

$$a_n = \frac{3^{n-1}}{2^{\lfloor \log_2 3^{n-1} \rfloor}}, \quad n \in \mathbb{Z} \tag{5.27}$$

en tal caso, de a_0 a a_6 obtendríamos las notas sin alteraciones (*Fa-Do-Sol-Re-La-Mi-Si*); de a_7 a a_{13} estas notas aparecerían alteradas con un sostenido (*Fa♯-Do♯-Sol♯-Re♯-La♯-Mi♯-Si♯*); de a_{14} a a_{20} aparecerían alteradas con doble sostenido, etc. Los términos con valores de n negativos se corresponden con bemoles: de a_{-7} a a_{-1} las notas aparece con un bemol (*Fa♭-Do♭-Sol♭-Re♭-La♭-Mi♭-Si♭*), de a_{-14} a a_{-8} las notas aparece con doble bemol, y así sucesivamente (Liern, 2015).

Definición 5.3.2 Sistema de afinación cerrado. *Sea $\mathcal{S}_{\mathcal{B}}^{\mathcal{F}}$ un sistema de afinación formado por los intervalos $\mathcal{B}$ y las funciones $\mathcal{F}$. Sea el $T \in \mathbb{N}^+$ período de repetición. Un sistema de afinación es T-cerrado si*

$$r_n = r_{n+T},\ \forall n \in \mathbb{Z} \tag{5.28}$$

en caso contrario, el sistema de afinación será abierto, como le sucede al Sistema Pitagórico definido en 5.25, ya que

$$r^*_{pit}(n) \neq r^*_{pit}(n+T),\ \forall n \in \mathbb{Z},\ \forall T \in \mathbb{N}^+$$

Para cerrar la espiral infinita de quintas es necesario modificar restándole el valor de la coma pitagórica. Esta quinta se denomina *Quinta del Lobo* y tradicionalmente se dispone entre las notas *Sol*♯ y *Mi*♭.

$$r_{5Lobo} = \frac{3/2}{3^{12}/2^{19}} = \frac{2^{18}}{3^{11}} \quad (678{,}49499048 \text{ cents})$$

Ejemplo 5.3.2 *Sistema de Afinación Pitagórico cerrado con quinta del lobo. Sean $\mathcal{B} = \{\beta_1 = \frac{3}{2}, \beta_2 = \frac{2^{18}}{3^{11}}\}$ los intervalos de quinta natural y de quinta del lobo pitagórica. Sean las funciones $\mathcal{F} = \{f_1, f_2\}$ tal que*

$$f_1(n) = n - f_2(n) \quad f_2(n) = \lfloor \frac{n+3}{12} \rfloor$$

el Sistema de Afinación Pitagórico cerrado es el conjunto

$$\mathcal{S}_{pitCerr} = \{r_n = \left[\beta_1^{n-f_2(n)} \cdot \beta_2^{f_2(n)}\right]^*\}, n \in \mathbb{Z} \tag{5.29}$$

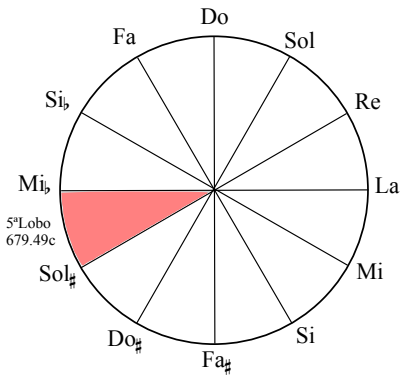


Figura 5.2: Ciclo pitagórico con quinta del lobo.

5.3.2 La justa entonación

Definición 5.3.3 Sistema de Zarlino. *Sean* $\mathcal{B} = \{\beta_1 = {}^3\!/_2, \beta_2 = {}^{27}\!/_{40}\}$ *los intervalos de quinta natural y quinta sintónica respectivamente. Sean las funciones* $\mathcal{F} = \{f_1, f_2\}$

$$f_1(n) = n - \lfloor \frac{n+4}{7} \rfloor - \lfloor \frac{n+1}{7} \rfloor$$

$$f_2(n) = \lfloor \frac{n+4}{7} \rfloor + \lfloor \frac{n+1}{7} \rfloor$$

el sistema de afinación de Zarlino es el conjunto

$$\mathcal{S}_{zar} = \{r_n = \left[\beta_1^{f_1(n)} \cdot \beta_2^{f_2(n)}\right]^*\}, n \in \mathbb{Z} \tag{5.30}$$

Para entenderla construcción de estas funciones hay que considerar lo siguiente: la expresión $\lfloor {}^{n+4}\!/_7 \rfloor$ calcula el número de quintas sintónicas β_2 que atraviesan la posición *Re-La*, empezando a contar desde la nota *Do*. La expresión $\lfloor {}^{n+1}\!/_7 \rfloor$ calcula el número de quintas sintónicas β_2 que atraviesan la posición *Fa-Si*♭, contando desde *Do*. Por tanto la suma de ambas expresiones $f_2(n)$ calcula el total de quintas sintónicas contenidas en la espiral de Zarlino para una posición n, empezando a contar desde *Do*. El número de quintas naturales $f_1(n)$ será el número total de quintas naturales menos el número de quintas sintónicas.

Tabla 5.4: Cálculo de las 19 notas del sistema de la escala de Zarlino.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot \beta_2^0 \cdot 2^0$	0.00	Sol♭	-6	$\beta_1^{-4} \cdot \beta_2^{-2} \cdot 2^4$	631.28
Do♯	7	$\beta_1^5 \cdot \beta_2^2 \cdot 2^{-4}$	70.67	Sol	1	$\beta_1^1 \cdot \beta_2^0 \cdot 2^0$	701.96
Re♭	-5	$\beta_1^{-3} \cdot \beta_2^{-2} \cdot 2^3$	133.24	Sol♯	8	$\beta_1^6 \cdot \beta_2^2 \cdot 2^{-4}$	772.63
Re	2	$\beta_1^2 \cdot \beta_2^0 \cdot 2^{-1}$	203.91	La♭	-4	$\beta_1^{-3} \cdot \beta_2^{-1} \cdot 2^3$	813.69
Re♯	9	$\beta_1^7 \cdot \beta_2^2 \cdot 2^{-5}$	274.58	La	3	$\beta_1^2 \cdot \beta_2^1 \cdot 2^{-1}$	884.36
Mi♭	-3	$\beta_1^{-2} \cdot \beta_2^{-1} \cdot 2^2$	315.64	La♯	10	$\beta_1^7 \cdot \beta_2^3 \cdot 2^{-5}$	955.03
Mi	4	$\beta_1^3 \cdot \beta_2^1 \cdot 2^{-2}$	386.31	Si♭	-2	$\beta_1^{-1} \cdot \beta_2^{-1} \cdot 2^2$	1017.60
Mi♯	11	$\beta_1^8 \cdot \beta_2^3 \cdot 2^{-6}$	456.99	Si	5	$\beta_1^4 \cdot \beta_2^1 \cdot 2^{-2}$	1088.27
Fa	-1	$\beta_1^{-1} \cdot \beta_2^0 \cdot 2^1$	498.04	Si♯	12	$\beta_1^9 \cdot \beta_2^3 \cdot 2^{-6}$	1158.94
Fa♯	6	$\beta_1^4 \cdot \beta_2^2 \cdot 2^{-3}$	568.72				

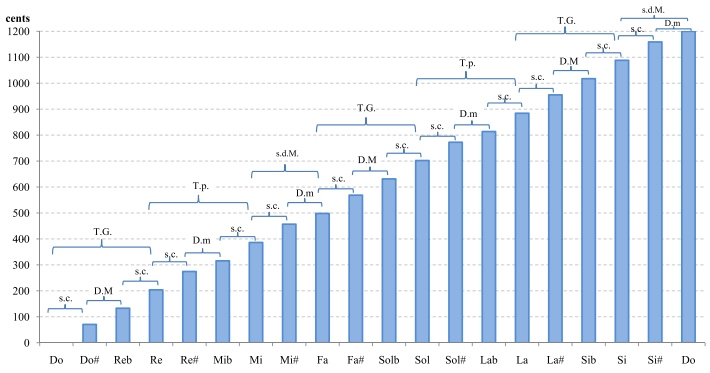


Figura 5.3: Número de cents para cada nota de la escala de Zarlino.

La expresión de la *díesis Mayor* (D.M) es $\beta_1^{-8} \cdot \beta_2^{-4} \cdot 2^7 = {}^{648}/_{625}$. La expresión de la *díesis menor* (D.m) es $\beta_1^{-9} \cdot \beta_2^{-3} \cdot 2^7 = {}^{128}/_{125}$. La expresión del *tono Grande* (T.G.) es $\beta_1^2 \cdot 2^{-1} = {}^{9}/_{8}$. La expresión del *tono pequeño*(T.p.) es $\beta_1 \cdot \beta_2 \cdot 2^{-1} = {}^{10}/_{9}$. La expresión del *semitono diatónico menor* (s.d.m.) es $\beta_1^{-4} \cdot \beta_2^{-1} \cdot 2^3 = {}^{16}/_{15}$. La expresión del *semitono diatónico Mayor* (s.d.M.) es $\beta_1^{-3} \cdot \beta_2^{-2} \cdot 2^3 = {}^{27}/_{25}$. Al no ser una fracción superparticular no cuadra con el sistema filosófico neoplatónico de Zarlino. La expresión del *semitono cromático* (s.c) es $\beta_1^5 \cdot \beta_2^2 \cdot 2^{-4} = {}^{25}/_{24}$.

Es remarcable señalar tal y como se muestra en la siguiente figura que el sistema de Zarlino de 19 notas es muy similar a la división de la octava en 19 partes iguales. Puede comprobarse esta afirmación mediante los resultados computacionales ejemplificados en la tabla 5.22.

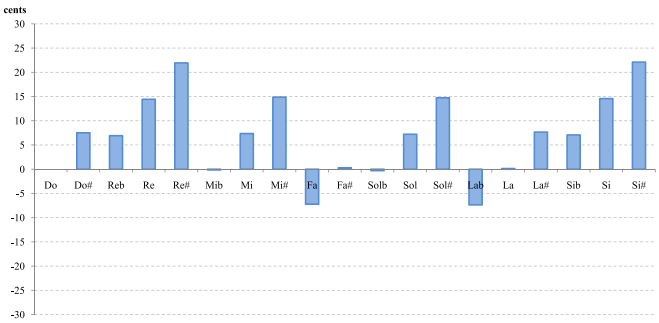


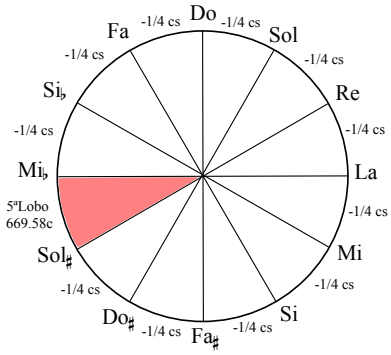
Figura 5.4: Desviación en cents de cada nota de la escala de Zarlino con su equivalente en el temperamento igual de 19 notas por octava.

5.3.3 Temperamentos abiertos

Utilizaremos la expresión propuesta por Liern (2015, pág.203) para calcular de forma general la quinta temperada β_1 perteneciente a cualquier sistema mesotónico de a/b de comma sintónica, donde $a, b \in \mathbb{N}^*$

$$\beta_1 = \sqrt[b]{\frac{2^{4\cdot a-b} \cdot 5^a}{3^{4\cdot a-b}}} \tag{5.31}$$

Mesotónico de 1/4 de Pietro Aaron



Propuesto por Pietro Aaron en (P. Aaron, 1 523, II, 4 1) (Link, 1963). Consiste en restar a cada quinta un cuarto de coma sintónica, de tal manera que cada cuatro quintas hemos reducido una coma sintónica completa. El valor de esta quinta reducida es:

$$\beta_1 = \frac{3/2}{\sqrt[4]{81/80}} = \sqrt[4]{\frac{2^{4-4} \cdot 5}{3^{4-4}}} = 5^{\frac{1}{5}} \quad (696{,}58 \text{ cents})$$

La quinta del lobo vale $\beta_1^{-11} \cdot 2^7 = 5^{-11/4} \cdot 2^7$ (737.64 cents), aumentando aproximadamente en 2.7 comas sintónicas con respecto a la quinta del lobo de la afinación pitagórica.

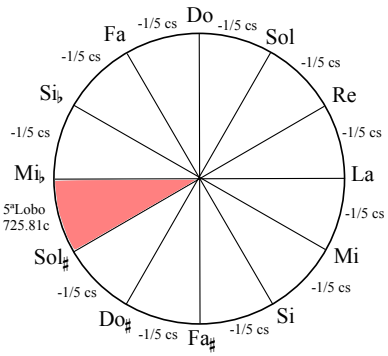
Definición 5.3.4 Temperamento mesotónico de 1/4. *Sean* $\mathcal{B} = \{\beta_1 = 5^{\frac{1}{5}}\}$, *y* $\mathcal{F} = \{f_1(n) = n\}$. *Según la definición 5.3.1, el temperamento mesotónico de un cuarto de coma sintónica es el conjunto*

$$\mathcal{S}_{1/4} = \{r_n = [\beta_1^n]^*\}, \quad n \in \mathbb{Z}, \quad \beta_1 = 5^{1/4} \tag{5.32}$$

Tabla 5.5: Cálculo de las 12 notas del temperamento mesotónico de 1/4.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot 2^0$	0	Fa♯	6	$\beta_1^6 \cdot 2^{-3}$	579,47
Do♯	7	$\beta_1^7 \cdot 2^{-4}$	76,05	Sol	1	$\beta_1^1 \cdot 2^0$	696,58
Re	2	$\beta_1^2 \cdot 2^{-1}$	193,16	Sol♯	8	$\beta_1^8 \cdot 2^{-4}$	772,63
Mi♭	-3	$\beta_1^{-3} \cdot 2^2$	310,26	La	3	$\beta_1^3 \cdot 2^{-1}$	889,74
Mi	4	$\beta_1^4 \cdot 2^{-2}$	386,31	Si♭	-2	$\beta_1^{-2} \cdot 2^2$	1006,84
Fa	-1	$\beta_1^{-1} \cdot 2^1$	503,42	Si	5	$\beta_1^5 \cdot 2^{-2}$	1082,89

Mesotónico de 1/5 de Sauveur



Consiste en restar a cada quinta un quinto de coma sintónica, de tal manera que cada cinco quintas hemos reducido una coma sintónica completa. El valor de esta quinta reducida es el siguiente:

$$\beta_1 = \frac{3/2}{\sqrt[5]{81/80}} = \left(\frac{3\cdot 5}{2}\right)^{1/5} \text{ (697,65 cents)}$$

La quinta del lobo vale $\beta_1^{-11}\cdot 2^7 = 2^{46/5}\cdot 3^{-11/5}\cdot 5^{-11/5}$ (725,81 cents).

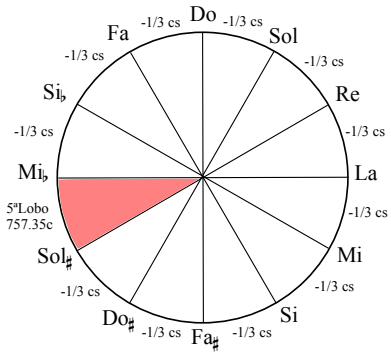
Definición 5.3.5 Temperamento mesotónico de 1/5. *Sean* $\mathcal{B}=\{\beta_1\}$, *y* $\mathcal{F}=\{f_1(n)=n\}$. *Según la definición 5.3.1, el temperamento mesotónico de un quinto de coma sintónica es el conjunto*

$$\mathcal{S}_{1/5} = \{r_n = [\beta_1^n]^*\}, \quad n\in\mathbb{Z}, \quad \beta_1 = \left(\frac{3\cdot 5}{2}\right)^{1/5} \tag{5.33}$$

Tabla 5.6: Cálculo de las 12 notas del temperamento mesotónico de 1/5.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0\cdot 2^0$	0	Fa♯	6	$\beta_1^6\cdot 2^{-3}$	585,92
Do♯	7	$\beta_1^7\cdot 2^{-4}$	83,58	Sol	1	$\beta_1^1\cdot 2^0$	697,65
Re	2	$\beta_1^2\cdot 2^{-1}$	195,31	Sol♯	8	$\beta_1^8\cdot 2^{-4}$	781,23
Mi♭	-3	$\beta_1^{-3}\cdot 2^2$	307,04	La	3	$\beta_1^3\cdot 2^{-1}$	892,96
Mi	4	$\beta_1^4\cdot 2^{-2}$	390,61	Si♭	-2	$\beta_1^{-2}\cdot 2^2$	1004,69
Fa	-1	$\beta_1^{-1}\cdot 2^1$	502,35	Si	5	$\beta_1^5\cdot 2^{-2}$	1088,27

Mesotónico de 1/3 de Salinas



Consiste en restar a cada quinta un tercio de coma sintónica, de tal manera que cada tres quintas hemos reducido una coma sintónica completa. Las terceras menores (o sextas mayores) serán todas naturales, excepto las que pasen por la quinta del lobo. El valor de esta quinta reducida es el siguiente:

$$\beta_1 = \frac{3/2}{\sqrt[3]{81/80}} = \sqrt[3]{10/3}\,(694{,}79 \text{ cents})$$

La quinta del lobo vale $\beta_1^{-11}\cdot 2^7 = 2^{10/3}\cdot 3^{11/3}\cdot 5^{-11/3}$ (757,35 cents).

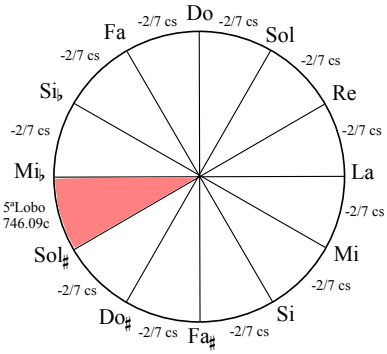
Definición 5.3.6 Temperamento mesotónico de 1/3. *Sean* $\mathcal{B} = \{\beta_1\}$, *y* $\mathcal{F} = \{f_1(n) = n\}$. *Según la definición 5.3.1, el temperamento mesotónico de un tercio de coma sintónica es el conjunto*

$$\mathcal{S}_{1/3} = \{r_n = [\beta_1^n]^*\}, \quad n \in \mathbb{Z}, \quad \beta_1 = \sqrt[3]{10/3} \tag{5.34}$$

Tabla 5.7: Cálculo de las 12 notas del temperamento mesotónico de 1/3.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot 2^0$	0	Fa♯	6	$\beta_1^6 \cdot 2^{-3}$	568,72
Do♯	7	$\beta_1^7 \cdot 2^{-4}$	63,5	Sol	1	$\beta_1^1 \cdot 2^0$	694,79
Re	2	$\beta_1^2 \cdot 2^{-1}$	189,57	Sol♯	8	$\beta_1^8 \cdot 2^{-4}$	758,29
Mi♭	-3	$\beta_1^{-3} \cdot 2^2$	315,64	La	3	$\beta_1^3 \cdot 2^{-1}$	884,36
Mi	4	$\beta_1^4 \cdot 2^{-2}$	379,14	Si♭	-2	$\beta_1^{-2} \cdot 2^2$	1010,43
Fa	-1	$\beta_1^{-1} \cdot 2^1$	505,21	Si	5	$\beta_1^5 \cdot 2^{-2}$	1073,93

Mesotónico de 2/7 de Zarlino



Consiste en restar a cada siete quintas, un total de dos comas sintónicas, por tanto a cada quinta hay que restarle $2/7$ de coma. El valor de esta quinta reducida es el siguiente:

$$\beta_1 = \sqrt[7]{\frac{2\cdot 5^2}{3}}\,(695{,}81 \text{ cents})$$

La quinta del lobo vale 757,35 cents.

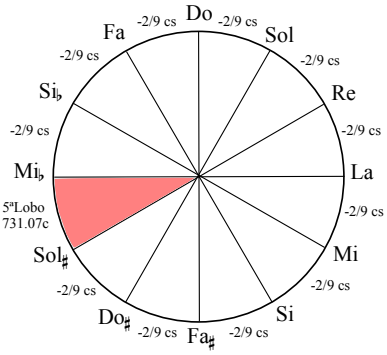
Definición 5.3.7 Temperamento mesotónico de $2/7$. *Sean* $\mathcal{B} = \{\beta_1\}$, *y* $\mathcal{F} = \{f_1(n) = n\}$. *Según la definición 5.3.1, el temperamento mesotónico de* $2/7$ *de coma sintónica es el conjunto*

$$\mathcal{S}_{2/7} = \{r_n = [\beta_1^n]^*\}, \quad n \in \mathbb{Z}, \quad \beta_1 = \sqrt[7]{\frac{2\cdot 5^2}{3}} \tag{5.35}$$

Tabla 5.8: Cálculo de las 12 notas del temperamento mesotónico de 2/7.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot 2^0$	0	Fa♯	6	$\beta_1^6 \cdot 2^{-3}$	574,86
Do♯	7	$\beta_1^7 \cdot 2^{-4}$	70,67	Sol	1	$\beta_1^1 \cdot 2^0$	695,81
Re	2	$\beta_1^2 \cdot 2^{-1}$	191,62	Sol♯	8	$\beta_1^8 \cdot 2^{-4}$	766,48
Mi♭	-3	$\beta_1^{-3} \cdot 2^2$	312,57	La	3	$\beta_1^3 \cdot 2^{-1}$	887,43
Mi	4	$\beta_1^4 \cdot 2^{-2}$	383,24	Si♭	-2	$\beta_1^{-2} \cdot 2^2$	1008,38
Fa	-1	$\beta_1^{-1} \cdot 2^1$	504,19	Si	5	$\beta_1^5 \cdot 2^{-2}$	1079,05

Mesotónico de 2/9 de Lemme Rossi



Consiste en restar a cada nueve quintas, un total de dos comas sintónicas, por tanto a cada quinta hay que restarle 2/9 de coma. El valor de esta quinta reducida es el siguiente:

$$\beta_1 = \sqrt[9]{\frac{3\cdot 5^2}{2}}\,(697{,}18 \text{ cents})$$

La quinta del lobo vale 731,07 cents.

Definición 5.3.8 Temperamento mesotónico de 2/9. *Sean* $\mathcal{B} = \{\beta_1\}$, *y* $\mathcal{F} = \{f_1(n) = n\}$. *Según la definición 5.3.1, el temperamento mesotónico de* 2/9 *de coma sintónica es el conjunto*

$$\mathcal{S}_{2/9} = \{r_n = [\beta_1^n]^*\}, \quad n \in \mathbb{Z}, \quad \beta_1 = \sqrt[9]{\frac{3\cdot 5^2}{2}} \tag{5.36}$$

Tabla 5.9: Cálculo de las 12 notas del temperamento mesotónico de 2/9.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot 2^0$	0	Fa♯	6	$\beta_1^6 \cdot 2^{-3}$	583,05
Do♯	7	$\beta_1^7 \cdot 2^{-4}$	80,23	Sol	1	$\beta_1^1 \cdot 2^0$	697,18
Re	2	$\beta_1^2 \cdot 2^{-1}$	194,35	Sol♯	8	$\beta_1^8 \cdot 2^{-4}$	777,41
Mi♭	-3	$\beta_1^{-3} \cdot 2^2$	308,47	La	3	$\beta_1^3 \cdot 2^{-1}$	891,53
Mi	4	$\beta_1^4 \cdot 2^{-2}$	388,7	Si♭	-2	$\beta_1^{-2} \cdot 2^2$	1005,65
Fa	-1	$\beta_1^{-1} \cdot 2^1$	502,82	Si	5	$\beta_1^5 \cdot 2^{-2}$	1085,88

5.3.4 Temperamentos cerrados regulares

Temperamento igual de 12 notas

Consiste cerrar la espiral de quintas naturales restando una doceava parte de la coma pitagórica ($\sqrt[12]{\frac{3^{12}}{2^{19}}}$) a cada una de ellas, de tal manera que la quinta temperada resultante es el siguiente:

$$\beta_1 = \frac{3/2}{\sqrt[12]{\frac{3^{12}}{2^{19}}}} = 2^{7/12}\,(700{,}00 \text{ cents})$$

Definición 5.3.9 Temperamento igual de 12 notas por octava. *Sean* $\mathcal{B} = \{\beta_1\}$, *y* $\mathcal{F} = \{f_1(n) = n\}$. *Según la definición 5.3.1, el temperamento igual de doce notas por octava es el conjunto*

$$\mathcal{S}_{T12} = \{r_n = [\beta_1^n]^*\}, \quad n \in \mathbb{Z}, \quad \beta_1 = 2^{7/12} \tag{5.37}$$

Tabla 5.10: Cálculo de las 12 notas del temperamento igual de 12 notas.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot 2^0$	0	Fa♯	6	$\beta_1^6 \cdot 2^{-3}$	600,00
Do♯	7	$\beta_1^7 \cdot 2^{-4}$	100,00	Sol	1	$\beta_1^1 \cdot 2^0$	700,00
Re	2	$\beta_1^2 \cdot 2^{-1}$	200,00	Sol♯	8	$\beta_1^8 \cdot 2^{-4}$	800,00
Mi♭	-3	$\beta_1^{-3} \cdot 2^2$	300,00	La	3	$\beta_1^3 \cdot 2^{-1}$	900,00
Mi	4	$\beta_1^4 \cdot 2^{-2}$	400,00	Si♭	-2	$\beta_1^{-2} \cdot 2^2$	1000,00
Fa	-1	$\beta_1^{-1} \cdot 2^1$	500,00	Si	5	$\beta_1^5 \cdot 2^{-2}$	1100,00

Se puede demostrar fácilmente como en el temperamento igual de doce notas, la razón de cualquier intervalo r_n *formado por un total de* $n \in \mathbb{R}$ *semitonos se calcula mediante la expresión*

$$r_n = \sqrt[12]{2^n}, \forall n \in \mathbb{R} \tag{5.38}$$

Temperamento igual de 19 notas

Consiste dividir la octava en 19 partes iguales. El intervalo resultante ($\sqrt[19]{2}$ o 63,1578947 cents) es un ligeramente superior al cuarto de tono del temperamento igual de 12 notas. El ciclo se cierra de manera natural mediante la concatenación de 19 intervalos β_1 tal que:

$$\beta_1 = 2^{1/19}\,(63{,}1578947 \text{ cents})$$

Definición 5.3.10 Temperamento igual de 19 notas por octava. *Sean* $\mathcal{B} = \{\beta_1\}$, *y* $\mathcal{F} = \{f_1(n) = n\}$. *Según la definición 5.3.1, el temperamento igual de diecinueve notas por octava es el conjunto*

$$\mathcal{S}_{T19} = \{r_n = [\beta_1^n]^*\}, \quad n \in \mathbb{Z}, \quad \beta_1 = 2^{1/19} \tag{5.39}$$

Tabla 5.11: Cálculo de las notas del temperamento igual de 19 notas.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot 2^0$	0,00	Sol♭	10	$\beta_1^{10} \cdot 2^0$	631,58
Do♯	1	$\beta_1^1 \cdot 2^0$	63,16	Sol	11	$\beta_1^{11} \cdot 2^0$	694,74
Re♭	2	$\beta_1^2 \cdot 2^0$	126,32	Sol♯	12	$\beta_1^{12} \cdot 2^0$	757,89
Re	3	$\beta_1^3 \cdot 2^0$	189,47	La♭	13	$\beta_1^{13} \cdot 2^0$	821,05
Re♯	4	$\beta_1^4 \cdot 2^0$	252,63	La	14	$\beta_1^{14} \cdot 2^0$	884,21
Mi♭	5	$\beta_1^5 \cdot 2^0$	315,79	La♯	15	$\beta_1^{15} \cdot 2^0$	947,37
Mi	6	$\beta_1^6 \cdot 2^0$	378,95	Si♭	16	$\beta_1^{16} \cdot 2^0$	1.010,53
Mi♯	7	$\beta_1^7 \cdot 2^0$	442,11	Si	17	$\beta_1^{17} \cdot 2^0$	1.073,68
Fa	8	$\beta_1^8 \cdot 2^0$	505,26	Si♯	18	$\beta_1^{18} \cdot 2^0$	1.136,84
Fa♯	9	$\beta_1^9 \cdot 2^0$	568,42				

Se puede demostrar fácilmente como en el temperamento igual de diecinueve notas, la razón r_n *de cualquier nota* $n \in \mathbb{N}^+$ *se calcula mediante la expresión*

$$r_n = \sqrt[19]{2^n}, \forall n \in \mathbb{N}^+ \tag{5.40}$$

Temperamento igual de 24 notas

Consiste dividir la octava en 24 partes iguales. El intervalo resultante ($\sqrt[24]{2}$ o 50,00 cents) es un cuarto de tono del temperamento igual de 12 notas. El ciclo se cierra de manera natural mediante 24 intervalos β_1 equivalentes a media quinta temperada (tercera menor del temperamento igual más un cuarto de tono).

$$\beta_1 = 2^{7/24}\,(350{,}00 \text{ cents})$$

Definición 5.3.11 Temperamento igual de 24 notas por octava. *Sean* $\mathcal{B} = \{\beta_1\}$, *y* $\mathcal{F} = \{f_1(n) = n\}$. *Según la definición 5.3.1, el temperamento igual de veinticuatro notas por octava es el conjunto*

$$\mathcal{S}_{T24} = \{r_n = [\beta_1^n]^*\}, \quad n \in \mathbb{Z}, \quad \beta_1 = 2^{7/24} \tag{5.41}$$

Tabla 5.12: Cálculo de las notas del temperamento igual de 24 notas.

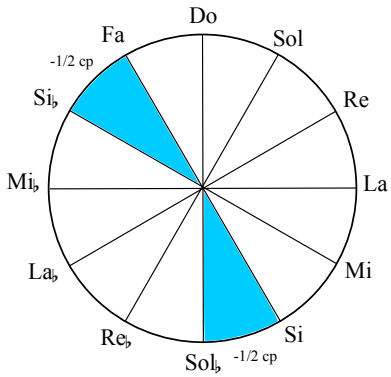
Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot 2^0$	0,00	Fa♯	12	$\beta_1^{12} \cdot 2^{-3}$	600,00
Do𝄲	7	$\beta_1^7 \cdot 2^{-2}$	50,00	Fa𝄪	19	$\beta_1^{19} \cdot 2^{-5}$	650,00
Do♯	14	$\beta_1^{14} \cdot 2^{-4}$	100,00	Sol	2	$\beta_1^2 \cdot 2^0$	700,00
Do𝄪	21	$\beta_1^{21} \cdot 2^{-6}$	150,00	Sol𝄲	9	$\beta_1^9 \cdot 2^{-2}$	750,00
Re	4	$\beta_1^4 \cdot 2^{-1}$	200,00	Sol♯	16	$\beta_1^{16} \cdot 2^{-4}$	800,00
Re𝄲	11	$\beta_1^{11} \cdot 2^{-3}$	250,00	Sol𝄪	23	$\beta_1^{23} \cdot 2^{-6}$	850,00
Re♯	18	$\beta_1^{18} \cdot 2^{-5}$	300,00	La	6	$\beta_1^6 \cdot 2^{-1}$	900,00
Re𝄪	1	$\beta_1^1 \cdot 2^0$	300,00	La𝄲	13	$\beta_1^{13} \cdot 2^{-3}$	950,00
Mi	8	$\beta_1^8 \cdot 2^{-2}$	400,00	La♯	20	$\beta_1^{20} \cdot 2^{-5}$	1000,00
Mi𝄲	15	$\beta_1^{15} \cdot 2^{-4}$	450,00	La𝄪	3	$\beta_1^3 \cdot 2^0$	1050,00
Fa	22	$\beta_1^{-1} \cdot 2^1$	500,00	Si	10	$\beta_1^5 \cdot 2^{-2}$	1100,00
Fa𝄲	5	$\beta_1^0 \cdot 2^0$	550,00	Si𝄲	17	$\beta_1^0 \cdot 2^0$	1150,00

Se puede demostrar fácilmente como en el temperamento igual de veinticuatro notas, la razón de cualquier intervalo r_n *formado por un total de* $n \in \mathbb{R}$ *cuartos de tono se calcula mediante la expresión*

$$r_n = \sqrt[24]{2^n}, \forall n \in \mathbb{R} \tag{5.42}$$

5.3.5 Temperamentos cerrados irregulares

H. Schreiber (1518)



Propone repartir la mitad de la coma pitagórica en dos quintas (Si-Sol♭ y Fa-Si♭) de valor $\beta_2 = 2^{17/2} \cdot 3^{-5}$ (690,23 cents).

Definición 5.3.12 Temperamento Schreiber. *Sean* $\mathcal{B} = \{\beta_1 = 3/2,\ \beta_2 = 2^{17/2} \cdot 3^{-5}\}$ *y* $\mathcal{F} = \{f_1, f_2\}$ *tal que*

$$f_1(n) = n - f_2(n)$$
$$f_2(n) = \lfloor \frac{n+6}{12} \rfloor + \lfloor \frac{n+1}{12} \rfloor$$

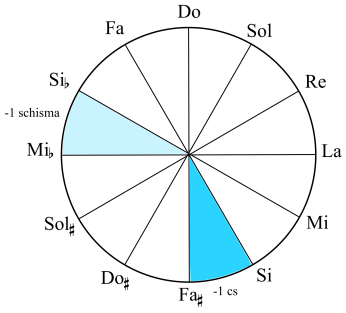
el Temperamento Schreiber es el conjunto

$$\mathcal{S}_{Sch} = \{r_n = \left[\beta_1^{n-f_2(n)} \cdot \beta_2^{f_2(n)}\right]^*\}, n \in \mathbb{Z} \tag{5.43}$$

Tabla 5.13: Cálculo de las 12 notas del temperamento Schreiber.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot \beta_2^0 \cdot 2^0$	0	Fa♯	6	$\beta_1^5 \cdot \beta_2^1 \cdot 2^{-3}$	600
Do♯	7	$\beta_1^6 \cdot \beta_2^1 \cdot 2^{-4}$	101,96	Sol	1	$\beta_1^1 \cdot \beta_2^0 \cdot 2^0$	701,96
Re	2	$\beta_1^2 \cdot \beta_2^0 \cdot 2^{-1}$	203,91	Sol♯	8	$\beta_1^7 \cdot \beta_2^1 \cdot 2^{-4}$	803,91
Mi♭	-3	$\beta_1^{-2} \cdot \beta_2^{-1} \cdot 2^2$	305,87	La	3	$\beta_1^3 \cdot \beta_2^0 \cdot 2^{-1}$	905,87
Mi	4	$\beta_1^4 \cdot \beta_2^0 \cdot 2^{-2}$	407,82	Si♭	-2	$\beta_1^{-1} \cdot \beta_2^{-1} \cdot 2^2$	1007,82
Fa	-1	$\beta_1^{-1} \cdot \beta_2^0 \cdot 2^1$	498,04	Si	5	$\beta_1^5 \cdot \beta_2^0 \cdot 2^{-2}$	1109,78

M. Agrícola (1539)



Propone restar una coma sintónica a la quinta dispuesta entre Si y Fa♯. Para cerrar el ciclo, le resta un Schisma a la quinta comprendida entre Si♭ y Mi♭ ($\beta_3 = 2^{14} \cdot 3^{-7} \cdot 5^{-1}$ de 700,0012801 cents). Nótese que esta quinta β_3 resultante no es exactamente una quinta temperada, a causa de la diferencia entre la Schisma y el Grad. Existen cuatro 3ªM justas (las que atraviesan la quinta sintónica), cuatro 3ªM pitagóricas y cuatro 3ªM un Schisma (1,95 cents) más pequeñas que las pitagóricas).

Definición 5.3.13 Temperamento Agrícola. *Sean* $\mathcal{B} = \{\beta_1 = 3/2,\ \beta_2 = 2^{40/27},\ \beta_3 = 2^{14} \cdot 3^{-7} \cdot 5^{-1}\}$ *y* $\mathcal{F} = \{f_1, f_2, f_3\}$ *tal que*

$$f_1(n) = n - f_2(n) - f_3(n) \quad f_2(n) = \lfloor \frac{n+6}{12} \rfloor \quad f_3(n) = \lfloor \frac{n+2}{12} \rfloor$$

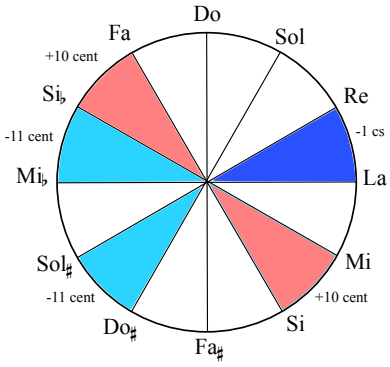
el Temperamento Agrícola es el conjunto

$$\mathcal{S}_{Agr} = \{r_n = \left[\beta_1^{n-f_2(n)-f_3(n)} \cdot \beta_2^{f_2(n)} \cdot \beta_3^{f_3(n)}\right]^*\}, n \in \mathbb{Z} \tag{5.44}$$

Tabla 5.14: Cálculo de las 12 notas del temperamento Agrícola.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^0$	0	Fa♯	6	$\beta_1^5 \cdot \beta_2^1 \cdot \beta_3^0 \cdot 2^{-3}$	590,22
Do♯	7	$\beta_1^6 \cdot \beta_2^1 \cdot \beta_3^0 \cdot 2^{-4}$	92,18	Sol	1	$\beta_1^1 \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^0$	701,96
Re	2	$\beta_1^2 \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^{-1}$	203,91	Sol♯	8	$\beta_1^7 \cdot \beta_2^1 \cdot \beta_3^0 \cdot 2^{-4}$	794,13
Mi♭	-3	$\beta_1^{-2} \cdot \beta_2^0 \cdot \beta_3^{-1} \cdot 2^2$	296,09	La	3	$\beta_1^3 \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^{-1}$	905,87
Mi	4	$\beta_1^4 \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^{-2}$	407,82	Si♭	-2	$\beta_1^{-2} \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^2$	996,09
Fa	-1	$\beta_1^{-1} \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^1$	498,04	Si	5	$\beta_1^5 \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^{-2}$	1109,78

S. Ganassi (1543)



Propone restar una coma sintónica a la quinta dispuesta entre Re y La ($\beta_2 = 40/27$). Para cerrar el ciclo, resta a dos quintas 11 cents ($\beta_3 = 2^{-119/120} \cdot 3$ de 711,96 cents) y suma a otras dos quintas 10 cents ($\beta_4 = 2^{-1211/1200} \cdot 3$ de 690,96 cents), aproximando el valor del Schisma necesario para restar en total, el valor de una coma pitagórica ($-1cs + 10 - 11 - 11 + 10 = -1cs - 2 \approx -1cp$). Nótese que este sistema, desde un estricto punto de vista teórico no es cerrado a causa de las aproximaciones realizadas. Si que lo es desde el punto de vista práctico.

Definición 5.3.14 Temperamento Ganassi. *Sean* $\mathcal{B} = \{\beta_1 = 3/2,\ \beta_2 = 2^{40/27},\ \beta_3 = 2^{-119/120} \cdot 3,\ \beta_4 = 2^{-1211/1200} \cdot 3\}$ *y* $\mathcal{F} = \{f_1, f_2, f_3, f_4\}$ *tal que*

$$f_1(n) = n - f_2(n) - f_3(n) - f_4(n)$$
$$f_2(n) = \lfloor \frac{n+9}{12} \rfloor$$
$$f_3(n) = \lfloor \frac{n+6}{12} \rfloor + \lfloor \frac{n+1}{12} \rfloor$$
$$f_4(n) = \lfloor \frac{n+4}{12} \rfloor + \lfloor \frac{n+2}{12} \rfloor$$

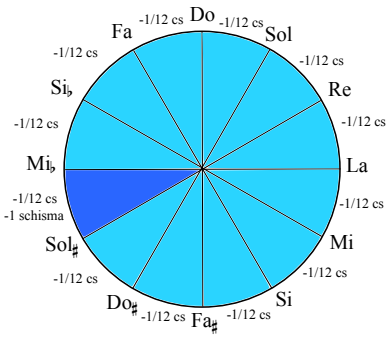
el Temperamento Ganassi es el conjunto

$$\mathcal{S}_{Gns} = \{r_n = \left[\beta_1^{n-f_2(n)-f_3(n)-f_4(n)} \cdot \beta_2^{f_2(n)} \cdot \beta_3^{f_3(n)} \cdot \beta_4^{f_4(n)}\right]^*\}, n \in \mathbb{Z} \qquad (5.45)$$

Tabla 5.15: Cálculo de las 12 notas del temperamento Ganassi.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0\beta_2^0\beta_3^0\beta_4^0 2^0$	0	Fa♯	6	$\beta_1^4\beta_2^1\beta_3^1\beta_4^0 2^{-3}$	600,22
Do♯	7	$\beta_1^5\beta_2^1\beta_3^1\beta_4^0 2^{-4}$	102,18	Sol	1	$\beta_1^1\beta_2^0\beta_3^0\beta_4^0 2^0$	701,96
Re	2	$\beta_1^2\beta_2^0\beta_3^0\beta_4^0 2^{-1}$	203,91	Sol♯	8	$\beta_1^5\beta_2^1\beta_3^1\beta_4^1 2^{-4}$	793,13
Mi♭	-3	$\beta_1^{-1}\beta_2^0\beta_3^{-1}\beta_4^{-1} 2^2$	295,13	La	3	$\beta_1^2\beta_2^1\beta_3^0\beta_4^0 2^{-1}$	884,36
Mi	4	$\beta_1^3\beta_2^1\beta_3^0\beta_4^0 2^{-2}$	386,31	Si♭	-2	$\beta_1^{-1}\beta_2^0\beta_3^{-1}\beta_4^0 2^2$	986,09
Fa	-1	$\beta_1^{-1}\beta_2^0\beta_3^0\beta_4^0 2^1$	498,04	Si	5	$\beta_1^4\beta_2^1\beta_3^0\beta_4^0 2^{-2}$	1088,27

J. Bermudo (1555)



Propone temperar todas las quintas restándoles una doceava parte de la coma sintónica (resultando quintas temperadas $\beta_1 = 2^{-2/3} \cdot 3^{-2/3} \cdot 5^{1/12}$ de 700,16 cents, muy próximas a las quintas de 700 cents del temperamento igual). Para cerrar el ciclo necesita restar un Schisma, realizándolo en la quinta comprendida entre Sol♯y Mi♭($\beta_2 = 2^{43/3} \cdot 3^{-22/3} \cdot 5^{-11/12}$ de 698,21 cents). Obtiene un total de ocho 3ªM muy parecidas a las del temperamento igual (400,16 cents). Las cuatro 3ªM que atraviesan el Schisma son un poco más grandes (402,6 cents).

Definición 5.3.15 Temperamento Bermudo. *Sean* $\mathcal{B} = \{\beta_1 = 2^{-2/3} \cdot 3^{-2/3} \cdot 5^{1/12}, \beta_2 = 2^{43/3} \cdot 3^{-22/3} \cdot 5^{-11/12}\}$ *y* $\mathcal{F} = \{f_1, f_2\}$ *tal que*

$$f_1(n) = n - f_2(n)$$
$$f_2(n) = \lfloor \frac{n+3}{12} \rfloor$$

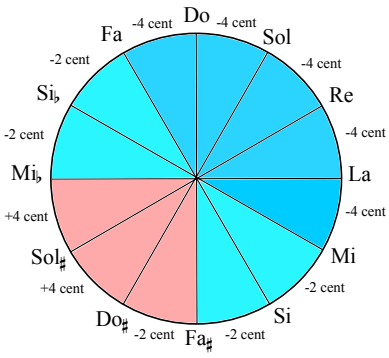
el Temperamento Bermudo es el conjunto

$$\mathcal{S}_{Ber} = \{r_n = \left[\beta_1^{n-f_2(n)} \cdot \beta_2^{f_2(n)}\right]^*\}, n \in \mathbb{Z} \tag{5.46}$$

Tabla 5.16: Cálculo de las 12 notas del temperamento Bermudo.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot \beta_2^0 \cdot 2^0$	0	Fa♯	6	$\beta_1^6 \cdot \beta_2^0 \cdot 2^{-3}$	600,98
Do♯	7	$\beta_1^7 \cdot \beta_2^0 \cdot 2^{-4}$	101,14	Sol	1	$\beta_1^1 \cdot \beta_2^0 \cdot 2^0$	700,16
Re	2	$\beta_1^2 \cdot \beta_2^0 \cdot 2^{-1}$	200,33	Sol♯	8	$\beta_1^8 \cdot \beta_2^0 \cdot 2^{-4}$	801,3
Mi♭	-3	$\beta_1^{-3} \cdot \beta_2^0 \cdot 2^2$	299,51	La	3	$\beta_1^3 \cdot \beta_2^0 \cdot 2^{-1}$	900,49
Mi	4	$\beta_1^4 \cdot \beta_2^0 \cdot 2^{-2}$	400,65	Si♭	-2	$\beta_1^{-2} \cdot \beta_2^0 \cdot 2^2$	999,67
Fa	-1	$\beta_1^{-1} \cdot \beta_2^0 \cdot 2^1$	499,84	Si	5	$\beta_1^5 \cdot \beta_2^0 \cdot 2^{-2}$	1100,81

A. Schlick (1511)



Reconstrucción según Barbour (Barbour, 2004, pág.137). Propone tres tipos de quintas diferentes; el primero una quinta natural a la que se le han restado 4 cents ($\beta_1 = 2^{-301/300} \cdot 3$, de 697,96 cents); el segundo una quinta natural a la que se le han restado 2 cents ($\beta_2 = 2^{-601/600} \cdot 3$, de 699,96 cents, y es a efecto prácticos equivalente a una quinta del temperamento igual; el tercer tipo es una quinta natural a la que se le han sumado 4 cents ($\beta_3 = 2^{-299/300} \cdot 3$), de 705,96 cents.

Definición 5.3.16 Temperamento Schlick. *Sean* $\mathcal{B} = \{\beta_1 = 2^{-301/300} \cdot 3,\ \beta_2 = 2^{-601/600} \cdot 3,\ \beta_3 = 2^{-299/300} \cdot 3\}$ *y* $\mathcal{F} = \{f_1, f_2, f_3\}$ *tal que*

$$f_1(n) = \lfloor \frac{n}{12} \rfloor + \lfloor \frac{n+11}{12} \rfloor + \lfloor \frac{n+10}{12} \rfloor + \lfloor \frac{n+9}{12} \rfloor + \lfloor \frac{n+8}{12} \rfloor + \lfloor \frac{n+7}{12} \rfloor$$

$$f_2(n) = \lfloor \frac{n+6}{12} \rfloor + \lfloor \frac{n+5}{12} \rfloor + \lfloor \frac{n+2}{12} \rfloor + \lfloor \frac{n+2}{12} \rfloor$$

$$f_3(n) = \lfloor \frac{n+4}{12} \rfloor + \lfloor \frac{n+3}{12} \rfloor$$

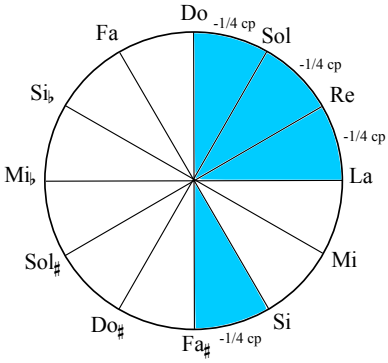
el Temperamento Schlick es el conjunto

$$\mathcal{S}_{Sck} = \{r_n = \left[\beta_1^{f_1(n)} \cdot \beta_2^{f_2(n)} \cdot \beta_3^{f_3(n)}\right]^*\}, n \in \mathbb{Z} \tag{5.47}$$

Tabla 5.17: Cálculo de las 12 notas del temperamento Bermudo.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^0$	0	Fa♯	6	$\beta_1^5 \cdot \beta_2^1 \cdot \beta_3^0 \cdot 2^{-3}$	589,73
Do♯	7	$\beta_1^5 \cdot \beta_2^2 \cdot \beta_3^0 \cdot 2^{-4}$	89,69	Sol	1	$\beta_1^1 \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^0$	697,96
Re	2	$\beta_1^2 \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^{-1}$	195,91	Sol♯	8	$\beta_1^5 \cdot \beta_2^2 \cdot \beta_3^1 \cdot 2^{-4}$	795,64
Mi♭	-3	$\beta_1^{-1} \cdot \beta_2^{-2} \cdot \beta_3^0 \cdot 2^2$	302,13	La	3	$\beta_1^3 \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^{-1}$	893,87
Mi	4	$\beta_1^4 \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^{-2}$	391,82	Si♭	-2	$\beta_1^{-1} \cdot \beta_2^{-1} \cdot \beta_3^0 \cdot 2^2$	1002,09
Fa	-1	$\beta_1^{-1} \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^1$	502,04	Si	5	$\beta_1^5 \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^{-2}$	1089,78

A. Werckmeister III (1691)



Este temperamento irregular propone la utilización de un tipo de quinta reducida en un cuarto de coma pitagórica $\beta_2 = 2^{15/4} \cdot 3^{-2}$ (696,09 cents), de tal manera que las tres primeras quintas (Do-Sol, Sol-Re, Re-La) y la séptima (Si-Fa♯) son reemplazadas por esta quinta más pequeña.

Definición 5.3.17 Temperamento Werckmeister III. *Sean* $\mathcal{B} = \{\beta_1 = 3/2,\ \beta_2 = 2^{15/4} \cdot 3^{-2}\}$ *y* $\mathcal{F} = \{f_1, f_2\}$ *tal que*

$$f_1(n) = n - f_2(n)$$
$$f_2(n) = \lfloor \frac{n+11}{12} \rfloor + \lfloor \frac{n+10}{12} \rfloor + \lfloor \frac{n+9}{12} \rfloor + \lfloor \frac{n+6}{12} \rfloor$$

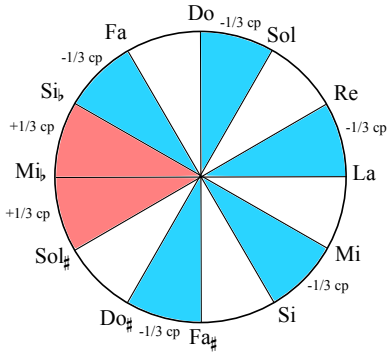
el Temperamento Werckmeister III es el conjunto

$$\mathcal{S}_{WIII} = \{r_n = \left[\beta_1^{n-f_2(n)} \cdot \beta_2^{f_2(n)}\right]^*\}, n \in \mathbb{Z} \tag{5.48}$$

Tabla 5.18: Cálculo de las 12 notas del temperamento Werckmeister III.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot \beta_2^0 \cdot 2^0$	0	Fa♯	6	$\beta_1^2 \cdot \beta_2^4 \cdot 2^{-3}$	588,27
Do♯	7	$\beta_1^3 \cdot \beta_2^4 \cdot 2^{-4}$	90,22	Sol	1	$\beta_1^0 \cdot \beta_2^1 \cdot 2^0$	696,09
Re	2	$\beta_1^0 \cdot \beta_2^2 \cdot 2^{-1}$	192,18	Sol♯	8	$\beta_1^4 \cdot \beta_2^4 \cdot 2^{-4}$	792,18
Mi♭	-3	$\beta_1^{-3} \cdot \beta_2^0 \cdot 2^2$	294,13	La	3	$\beta_1^0 \cdot \beta_2^3 \cdot 2^{-1}$	888,27
Mi	4	$\beta_1^1 \cdot \beta_2^3 \cdot 2^{-2}$	390,22	Si♭	-2	$\beta_1^{-2} \cdot \beta_2^0 \cdot 2^2$	996,09
Fa	-1	$\beta_1^{-1} \cdot \beta_2^0 \cdot 2^1$	498,04	Si	5	$\beta_1^2 \cdot \beta_2^3 \cdot 2^{-2}$	1092,18

A. Werckmeister IV(II) (1691)



Propone un tipo de quinta reducida en un tercio de coma pitagórica $\beta_2 = 2^{16/3} \cdot 3^{-3}$ (694,13 cents) y otro tipo de quinta aumentada en un tercio de coma pitagórica $\beta_3 = 2^{-22/3} \cdot 3^5$ (709,78 cents). El valor de la quinta reducida ($\beta_2 \approx 40{,}317/27$) es muy cercano a la quinta sintónica (40/27).

Definición 5.3.18 Temperamento Werckmeister IV(II). *Sean* $\mathcal{B} = \{\beta_1 = {}^3\!/_2,\ \beta_2 = 2^{16/3} \cdot 3^{-3},\ \beta_3 = 2^{-22/3} \cdot 3^5\}$ *y* $\mathcal{F} = \{f_1, f_2, f_3\}$ *tal que*

$$f_1(n) = n - f_2(n) - f_3(n)$$

$$f_2(n) = \lfloor \frac{n+11}{12} \rfloor + \lfloor \frac{n+9}{12} \rfloor + \lfloor \frac{n+7}{12} \rfloor + \lfloor \frac{n+5}{12} \rfloor + \lfloor \frac{n+1}{12} \rfloor$$

$$f_3(n) = \lfloor \frac{n+2}{12} \rfloor + \lfloor \frac{n+3}{12} \rfloor$$

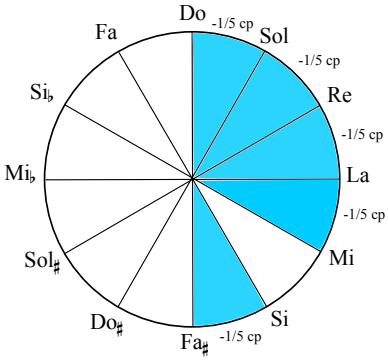
el Temperamento Werckmeister IV(II) es el conjunto

$$\mathcal{S}_{WIVI} = \{r_n = \left[\beta_1^{n-f_2(n)-f_3(n)} \cdot \beta_2^{f_2(n)} \cdot \beta_3^{f_3(n)}\right]^*\}, n \in \mathbb{Z} \qquad (5.49)$$

Tabla 5.19: Cálculo de las 12 notas del temperamento Werckmeister IV(II).

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^0$	0	Fa♯	6	$\beta_1^3 \cdot \beta_2^3 \cdot \beta_3^0 \cdot 2^{-3}$	588,27
Do♯	7	$\beta_1^3 \cdot \beta_2^4 \cdot \beta_3^0 \cdot 2^{-4}$	82,4	Sol	1	$\beta_1^0 \cdot \beta_2^1 \cdot \beta_3^0 \cdot 2^0$	694,13
Re	2	$\beta_1^1 \cdot \beta_2^1 \cdot \beta_3^0 \cdot 2^{-1}$	196,09	Sol♯	8	$\beta_1^4 \cdot \beta_2^4 \cdot \beta_3^0 \cdot 2^{-4}$	784,36
Mi♭	-3	$\beta_1^{-1} \cdot \beta_2^{-1} \cdot \beta_3^{-1} \cdot 2^2$	294,13	La	3	$\beta_1^1 \cdot \beta_2^2 \cdot \beta_3^0 \cdot 2^{-1}$	890,22
Mi	4	$\beta_1^2 \cdot \beta_2^2 \cdot \beta_3^0 \cdot 2^{-2}$	392,18	Si♭	-2	$\beta_1^{-1} \cdot \beta_2^{-1} \cdot \beta_3^0 \cdot 2^2$	1003,91
Fa	-1	$\beta_1^{-1} \cdot \beta_2^0 \cdot \beta_3^0 \cdot 2^1$	498,04	Si	5	$\beta_1^2 \cdot \beta_2^3 \cdot \beta_3^0 \cdot 2^{-2}$	1086,31

Kellner-Bach



Propone un tipo de quinta reducida en un quinto de coma pitagórica $\beta_2 = \sqrt[5]{2^{14}/3^7}$ (697,26 cents).

Definición 5.3.19 Temperamento Kellner-Bach. *Sean* $\mathcal{B} = \{\beta_1 = 3/2, \beta_2 = \sqrt[5]{2^{14}/3^7}$ *y* $\mathcal{F} = \{f_1, f_2\}$ *tal que*

$$f_1(n) = n - f_2(n)$$
$$f_2(n) = \lfloor\frac{n+11}{12}\rfloor + \lfloor\frac{n+10}{12}\rfloor + \lfloor\frac{n+9}{12}\rfloor + \lfloor\frac{n+8}{12}\rfloor + \lfloor\frac{n+6}{12}\rfloor$$

el Temperamento Kellner-Bach es el conjunto

$$\mathcal{S}_{KB} = \{r_n = \left[\beta_1^{n-f_2(n)} \cdot \beta_2^{f_2(n)}\right]^*\}, n \in \mathbb{Z} \tag{5.50}$$

Tabla 5.20: Cálculo de las 12 notas del temperamento Kellner-Bach.

Nota	n	r^*	cents	Nota	n	r^*	cents
Do	0	$\beta_1^0 \cdot \beta_2^0 \cdot 2^0$	0	Fa♯	6	$\beta_1^1 \cdot \beta_2^5 \cdot 2^{-3}$	588,27
Do♯	7	$\beta_1^2 \cdot \beta_2^5 \cdot 2^{-4}$	90,22	Sol	1	$\beta_1^0 \cdot \beta_2^1 \cdot 2^0$	697,26
Re	2	$\beta_1^0 \cdot \beta_2^2 \cdot 2^{-1}$	194,53	Sol♯	8	$\beta_1^3 \cdot \beta_2^5 \cdot 2^{-4}$	792,18
Mi♭	-3	$\beta_1^{-3} \cdot \beta_2^0 \cdot 2^2$	294,13	La	3	$\beta_1^0 \cdot \beta_2^3 \cdot 2^{-1}$	891,79
Mi	4	$\beta_1^0 \cdot \beta_2^4 \cdot 2^{-2}$	389,05	Si♭	-2	$\beta_1^{-2} \cdot \beta_2^0 \cdot 2^2$	996,09
Fa	-1	$\beta_1^{-1} \cdot \beta_2^0 \cdot 2^1$	498,04	Si	5	$\beta_1^1 \cdot \beta_2^4 \cdot 2^{-2}$	1091,01

J. G. Neidhardt (1724-1732)

Propone un tipo de quinta reducida en un sexto de coma pitagórica $\beta_2 = 2^{13/6} \cdot 3^{-1}$ (698,05 cents) y otro tipo de quinta reducida en un doceavo de coma pitagórica $\beta_3 = 2^{7/12}$ (700 cents). Nótese que esta última quinta reducida coincide con una temperada según el temperamento igual de 12 notas.

Definición 5.3.20 Temperamento Neidhardt. *Sean* $\mathcal{B} = \{\beta_1 = 3/2,\ \beta_2 = 2^{13/6} \cdot 3^{-1},\ \beta_3 = 2^{7/12}\}$ *y* $\mathcal{F} = \{f_1, f_2, f_3\}$ *tal que*

$$f_1(n) = n - f_2(n) - f_3(n)$$
$$f_2(n) = \lfloor \frac{n+11}{12} \rfloor + \lfloor \frac{n+10}{12} \rfloor + \lfloor \frac{n+9}{12} \rfloor + \lfloor \frac{n+8}{12} \rfloor$$
$$f_3(n) = \lfloor \frac{n+7}{12} \rfloor + \lfloor \frac{n+6}{12} \rfloor + \lfloor \frac{n+3}{12} \rfloor + \lfloor \frac{n+3}{12} \rfloor$$

el Temperamento Neidhardt es el conjunto

$$\mathcal{S}_N = \{r_n = \left[\beta_1^{n-f_2(n)-f_3(n)} \cdot \beta_2^{f_2(n)} \cdot \beta_3^{f_3(n)}\right]^*\}, n \in \mathbb{Z} \tag{5.51}$$

Valloti (1754)

Propone un tipo de quinta reducida en un sexto de coma pitagórica $\beta_2 = 2^{13/6} \cdot 3^{-1}$ (698,05 cents).

Definición 5.3.21 Temperamento Valloti. *Sean* $\mathcal{B} = \{\beta_1 = 3/2,\ \beta_2 = 2^{13/6} \cdot 3^{-1}\}$ *y* $\mathcal{F} = \{f_1, f_2\}$ *tal que*

$$f_1(n) = n - f_2(n)$$
$$f_2(n) = \lfloor \frac{n+11}{12} \rfloor + \lfloor \frac{n+10}{12} \rfloor + \lfloor \frac{n+9}{12} \rfloor + \lfloor \frac{n+8}{12} \rfloor + \lfloor \frac{n+7}{12} \rfloor + \lfloor \frac{n}{12} \rfloor$$

el Temperamento Valloti es el conjunto

$$\mathcal{S}_V = \{r_n = \left[\beta_1^{n-f_2(n)} \cdot \beta_2^{f_2(n)}\right]^*\}, n \in \mathbb{Z} \tag{5.52}$$

5.4 Compatibilidad entre notas y sistemas de afinación

Tradicionalmente la manera más sencilla de definir una nota perteneciente a un sistema de afinación determinado ha sido mediante la cuantificación de su frecuencia. Dicha frecuencia puede ser expresada en términos absolutos gracias a magnitudes físicas como los Hertzios, o en términos relativos mediante la utilización de escalas logarítmicas que indiquen el número de cents, savarts, o cualquier otro microintervalo de referencia contenido en el intervalo formado entre la frecuencia de la nota que queremos cuantificar y una frecuencia cualquiera de referencia denominada *diapasón*.
Sin embargo, observamos que la realidad musical proporciona una gran diversidad de frecuencias que pueden estar asociadas a la ejecución de una misma nota; ligeras variaciones en la posición de los dedos sobre el diapasón en el caso de un instrumento de cuerda; imperceptibles cambios en la presión de los labios sobre la boquilla en el caso de un instrumento de viento; ligeros desplazamientos en la colocación del instrumento; alteraciones en la presión o dirección de la columna de aire; fluctuaciones de temperatura; progresivas pérdidas de tensión de las cuerdas, etc. Todos estos factores determinan un cierto grado de incertidumbre asociado a la ejecución de una misma nota musical, sin que por ello necesariamente tengamos que decir que dicha nota se encuentra desafinada o fuera del sistema de afinación referencial.

Ejemplo 5.4.1 *Consideremos un violinista que durante el transcurso de su concierto ejecuta una nota* Si_4 *en la primera posición de la segunda cuerda del violín, resultando una frecuencia* $f_1 = 495{,}84Hz$*. Pocos compases después repite la ejecución de la misma nota* Si_4*, empleando la misma cuerda, posición, ataque e intensidad pero resultando esta vez una frecuencia* $f_2 = 496{,}575Hz$*. ¿Será perceptible el cambio sucedido en la afinación de la nota? ¿Cuál de las dos frecuencias* f_1 *o* f_2 *es la correcta?*

Si calculamos el número de cents existentes en el intervalo formado por esas dos frecuencias, obtenemos que

$$r_{cents}(f_1, f_2) = 1200 \times \left| \log_2 \left(\frac{495{,}84}{496{,}57} \right) \right| = 2{,}547 \,\text{cents}$$

Existe una diferencia de aproximadamente 2.5 cents entre las dos ejecuciones de la misma nota. En condiciones ideales, se conoce que la diferencia mínima de afinación entre dos notas que un oído humano bien entrenado musicalmente es capaz de distinguir (Wood, 2008) reside en aproximadamente 4 cents; por tanto dichas notas a pesar de no tener exactamente la misma frecuencia serán asimila-

das como *idénticas* ya que su variación en frecuencia es menor que la precisión del oído humano. ¿Cuál de las dos notas será la correcta? Si nuestra frecuencia de referencia es un La_4 de 442Hz, la frecuencia de un Si_4 según el temperamento igual tiene que ser

$$f_{\mathrm{Si}_4} = 2^{\frac{2}{12}} \cdot 442 = 496{,}128\,\mathrm{Hz}$$

Observamos que ninguna de las dos notas Si_4 realizadas por el instrumentista coincide con el valor exacto calculado desde el punto de vista teórico; sin embargo las dos serán consideradas como *correctamente* afinadas según el temperamento igual, ya que cada una de las notas tiene una desviación con respecto al valor teórico menor que la resolución en frecuencias de la percepción auditiva humana. Si continuásemos nuestro experimento, midiendo la frecuencia de las siguientes notas Si_4 que nuestro intérprete imaginario pudiera ejecutar durante el transcurso de su concierto, es fácil imaginar que obtendríamos un sinfín de frecuencias distintas entre sí. ¿Cómo podemos entonces distinguir cuáles de estas frecuencias se encuentran correctamente afinadas según nuestro sistema de afinación de aquellas que no lo están? Necesitamos una nueva definición del concepto de *nota musical* que incorpore la posibilidad de que cada nota lleve asociado un intervalo de frecuencias válidas, a diferencia de la concepción tradicional en la que una nota lleva asociada tan solo una frecuencia.

Sobre esta cuestión destacan las aportaciones realizadas por Liern (2005) en su artículo *Fuzzy tuning systems: the mathematics of musicians*, donde el autor introduce de una manera muy natural el formalismo de la lógica *fuzzy* en la definición de *nota musical*, ya que la describe como una banda de frecuencias posibles en torno a una frecuencia central. Ahora una nota vendrá caracterizada por un conjunto *fuzzy*. Veamos a continuación con un poco más de detalle la propuesta de Liern.

5.4.1 Notas como números fuzzy

Hemos visto anteriormente como para trabajar con escalas relativas de medida de frecuencia es necesario definir previamente un valor de referencia f_0, también llamado *diapasón*. A partir de aquí, obtendremos el valor del intervalo formado por la frecuencia de la nota y la frecuencia del diapasón. Usualmente se establece el diapasón en el $La_4 = 442$Hz, pero para nuestro estudio posterior sobre sistemas de afinación resultarás más conveniente situar el diapasón sobre la nota Do_4. Por tanto, considerando que nuestro sistema de afinación es el temperamento igual y que existen nueve semitonos iguales entre ambas notas, podemos establecer la

frecuencia del diapasón con la siguiente expresión:

$$f_o = f_{\mathrm{Do}_4} = 2^{-\frac{8}{12}} \cdot f_{\mathrm{La}_4} = 2^{-\frac{3}{4}} \cdot 442 = 262{,}81\,\mathrm{Hz}$$

Basándonos en la ecuación propuesta por Liern, pag.41 (2005), podemos definir una transformación para convertir cualquier frecuencia f en un número de cents comprendido entre $[0, 1200[$, tomando por tanto como referencia la frecuencia del diapasón f_0 e introduciendo además el principio de equivalencia de octava[16]. La expresión de dicha transformación quedaría de la siguiente manera

$$f^* = 1200 \left(log_2 \left(\frac{f}{f_0} \right) - \left\lfloor log_2 \left(\frac{f}{f_0} \right) \right\rfloor \right) \tag{5.53}$$

donde el uso de la función suelo nos asegura que $0 \leq f^* < 1200$ (cents) aunque f esté en relación mayor o menor a la octava con respecto a f_0.

Ejemplo 5.4.2 *Tomando la frecuencia $f_0 = 262{,}81Hz$ como diapasón, aplíquese la transformación 5.53 sobre los siguiente intervalos: unísono, octava, quinceava, quinta natural, quinta natural (octava baja), quinta natural (más quinceava), cuarta natural, cuarta natural (octava baja), cuarta natural (más quinceava), formados a partir de la frecuencia f_0.*

Intervalo	f(Hz)	razón	f^*(cents)
unísono	442,00	1 : 1	0,00
octava	884,00	2 : 1	0,00
quinceava	1768,00	4 : 1	0,00
5ª natural	663,00	3 : 2	701,96
5ª nat. (octava baja)	331,50	3 : 4	701,96
5ª nat. (más quinceava)	2652,00	6 : 1	701,96
4ª natural	589,33	4 : 3	498,04
4ª nat. (octava baja)	294,67	4 : 6	498,04
4ª nat. (más quinceava)	2357,33	16 : 3	498,04

Un grado musical es, en términos generales, la nota de la escala musical a la que se hace referencia (Latham, 2009). En el análisis armónico tonal tradicional algunos grados adquieren denominaciones muy características (e.g. tónica, supertónica, mediante, subdominante, dominante, submediante, sensible). Sin embargo nada

[16] Véase el apartado 5.2.1.

nos impide tener escalas musicales de un número de grados distinto de siete; de hecho es un suceso muy corriente en la música. Las escalas octófonas (mayor y menor), los modos pentatónicos (mayor, menor, slendro, pelog, hirahoshi), el modo cromático, los modos de transposiciones limitadas de Olivier Messiaen, por poner tan solo algunos ejemplos, son escalas en las que el número de grados puede diferir con respecto a los siete grados de los modos mayor y menor. Multitud de escalas pueden ser generadas a partir de la ordenación de las notas calculadas según un determinado sistema de afinación. Sin embargo, en lo sucesivo denominaremos *grado* musical a una determinada nota definida por un sistema de afinación cualquiera. Partiremos de la afirmación de que dado un sistema de afinación, podemos calcular un conjunto finito de m notas que, convenientemente ordenadas y recluidas en el ámbito de una octava, formen una determinada escala musical construida por m grados. Basándonos en las aportaciones realizadas por Liern (2005), realizaremos a continuación una serie de definiciones previas al estudio de los sistemas de afinación.

Definición 5.4.1 f-grado *musical. Sea* $\delta \in [1, 50]$ *el nivel de tolerancia expresado en cents. Sea* f_P *la frecuencia de un determinado grado musical calculado mediante un sistema de afinación cualquiera. Un* f-grado *musical* $\tilde{P}$ *se define como el número* fuzzy *triangular*

$$\tilde{P} := (f_P^*, f_P^* - \delta, f_P^* + \delta) = (f_P^*, \delta)$$

cuya función de pertenencia es

$$\mu_{\tilde{P}}(x) = \begin{cases} 1 - \dfrac{|f_P^* - x^*|}{\delta} & \text{si } |f_P^* - x^*| < \delta \\ 0 & \text{si } |f_P^* - x^*| \geq \delta \end{cases}$$

Por tanto, una nota *fuzzy* cualquiera $\tilde{N}$ vendrá determinada por dos parámetros; el primero será la transformada de su frecuencia f_N^*, el segundo será su coeficiente $\mu_{\tilde{P}}(f_N)$ que nos indicará el grado de pertenencia de la frecuencia de la nota al *f-grado* $\tilde{P}$.

$$\tilde{N} = (f_N^*, \mu_{\tilde{P}}(f_N))$$

Es interesante reseñar que una nota con una frecuencia f^* alejada un valor superior a δ de f_P^* tendrá un coeficiente de pertenencia igual a cero.

Ejemplo 5.4.3 *Sea* $\tilde{P}_7 = (700, 50)$ *el* f-grado *número siete de un sistema de afinación. Sea el diapasón* $f_0 = 262{,}81Hz$. *Calcule los distintos coeficientes de pertenencia de las siguientes frecuencias asociadas a dicho* f-grado*: 668,74Hz, 652,17Hz, 695,43Hz.*

$$f_1 = 668{,}74\text{Hz} \to f_1^* = 716{,}8789 \to \mu_{\tilde{P}_7}(f_1) = 1 - \frac{|700 - 716{,}8789|}{50} = 0{,}662422$$

$$f_2 = 652{,}17\text{Hz} \to f_2^* = 673{,}4421 \to \mu_{\tilde{P}_7}(f_2) = 1 - \frac{|700 - 673{,}4421|}{50} = 0,468842$$

$$f_3 = 668{,}74\text{Hz} \to f_3^* = 784{,}6307 \to |700 - 784{,}6307| > 50 \to \mu_{\tilde{P}_7}(f_3) = 0$$

Definición 5.4.2 f-sistema de afinación. *Un sistema de afinación* fuzzy *se define como una secuencia de* m f-grados *distintos*

$$\tilde{S}(\delta) = \{\tilde{P}_1, \tilde{P}_2, \ldots, \tilde{P}_m\} = \{\tilde{P}_j\}_{j=1}^m$$

Definición 5.4.3 f-nota *musical. Sea* $\tilde{S}(\delta)$ *un* f-sistema *de afinación de un número* m *de* f-grados *distintos. Una* f-nota *musical se define como*

$$\tilde{N} = \left(f_N^*; \{\mu_{\tilde{P}_j}(f_N)\}_{j=1}^m\right) \tag{5.54}$$

Es decir, al introducir un *f-sistema* de afinación, de forma teórica una *f-nota* podría *pertenecer* a varios *f-grados* de ese sistema; por tanto la definición de la nota musical requiere de la tupla de valores formada por su frecuencia y los coeficientes de pertenencia de esa nota a cada uno de los grados de la escala.

Ejemplo 5.4.4 *Sea* $\tilde{S}^Z(\delta)$ *un* f-sistema de afinación *basado en el sistema de Zarlino de 19 grados por octava. Obtenga todas las* f-notas *correspondiente a la frecuencia* $f_N = 842{,}51Hz$ *que tengan al menos un coeficiente de pertenencia distinto de cero. Tómese como diapasón* $f_0 = 262{,}81Hz$ *y como nivel de tolerancia* $\delta = 50$.
En la siguiente tabla vemos los valores de cents correspondientes a los distintos grados de la escala de Zarlino:

Nota	Cents	Nota	Cents	Nota	Cents	Nota	Cents
Do	0	Mi♭	316	Sol♭	631	La♯	955
Do♯	71	Mi	386	Sol	702	Si♭	1018
Re♭	133	Mi♯	457	Sol♯	773	Si	1088
Re	204	Fa	498	La♭	814	Si♯	1159
Re♯	275	Fa♯	569	La	884	Do	1200

Transformemos la frecuencia $f_N^*(842{,}51Hz) = 816{,}81$ Observamos que en el intervalo $(816{,}81 - 50, 816{,}81 + 50)$ existen dos posibles *f-grados*: *La♭* y *Sol♭*. Calcularemos los respectivos coeficientes de pertenencia de esos dos *f-grados* a f_N

$$\mu_{\tilde{La\flat}}(842{,}51) = 1 - \frac{|814 - 816{,}81|}{50} = 0{,}9438$$

$$\mu_{\tilde{Sol\flat}}(842{,}51) = 1 - \frac{|773 - 816{,}81|}{50} = 0{,}1238$$

Por tanto, la *f-nota* queda configurada de la siguiente manera, prescindiendo de los coeficientes de pertenencia con valor igual a cero:

$$\tilde{N} = \big(816{,}81; \mu_{\tilde{La\flat}} = 0{,}9438; \mu_{\tilde{Sol\flat}} = 0{,}1238\big)$$

5.4.2 Compatibilidad de una serie de notas con un sistema de afinación

Supongamos que tenemos un conjunto de n frecuencias medidas experimentalmente. ¿Cómo podríamos determinar cuál es el sistema de afinación que mejor se adapta a ellas? Hemos expuesto que la propia naturaleza del fenómeno musical asocia una indeterminación a todas y cada una de estas mediciones, por tanto no existe una correspondencia exacta entre valores teóricos extraídos de la definición del sistema de afinación y mediciones efectuadas; la realidad impone un error tanto en la medida como en la propia ejecución del gesto musical que finalmente se traduce en un error sobre la frecuencia. Necesitamos definir una distancia entre nuestro conjunto de frecuencias medidas experimentalmente y el conjunto de grados pertenecientes al sistema de afinación. Con esta distancia podremos determinar de entre todos los posibles cuál es el sistema de afinación que mejor se adapta, es decir, que menor distancia tiene, a nuestro conjunto de frecuencias observadas. Para la definición de dicha distancia nos basaremos en la definición propuesta en el capítulo 3.

Definición 5.4.4 Distancia media de afinación. *Sea* $F = \{f_1, f_2, \ldots, f_n\}$ *un conjunto de n frecuencias observadas experimentalmente. Sea* $\tilde{S}(\delta) = \{\tilde{P}_j\}_{j=1}^m$ *un* f-sistema *de afinación de referencia constituido por un número m de* f-grados *distintos. Sea* $\{\tilde{N}_1, \tilde{N}_2, \ldots, \tilde{N}_n\}$ *el conjunto de las n* f-notas *construidas a partir de* $\tilde{S}(\delta)$*, la expresión 5.54 y cada una de las frecuencias existentes en el conjunto F. La distancia media de afinación es*

$$d(F,\tilde{S}) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} \mu_{\tilde{P}_j}(f_{N_i}) \cdot |f^*_{N_i} - f^*_{P_j}|$$

Nuevamente podemos construir la matriz de pertenencia

$$\mathbf{U} = \begin{pmatrix} \mu_{\tilde{P}_1}(f_{N_1}) & \cdots & \mu_{\tilde{P}_m}(f_{N_1}) \\ \vdots & \ddots & \vdots \\ \mu_{\tilde{P}_1}(f_{N_n}) & \cdots & \mu_{\tilde{P}_m}(f_{N_n}) \end{pmatrix}$$

Podemos exigir que la suma de los coeficientes de pertenencia de una *f-nota* a todos los *f-grados* sea igual a 1, obteniendo por tanto que la suma de los valores de cada fila de la matiz U tiene que ser igual a la unidad, tal y como se muestra en la siguiente condición de normalización

$$\sum_{j=1}^{m} \mu_{\tilde{P}_j}(f_{N_i}) = 1, \quad 1 \leq i \leq n.$$

donde u_{ij} es el elemento (i,j) de la matriz de pertenencia $\mathbf{U}$. Podemos definir por tanto unos nuevos coeficientes de pertenencia normalizados $\hat{u}_{ij}$ de la siguiente manera

$$\hat{u}_{ij} = \frac{\mu_{\tilde{P}_j}(f_{N_i})}{\sum_{k=1}^{m} \mu_{\tilde{P}_k}(f_{N_i})} \tag{5.55}$$

Por último, reescribiendo la expresión 5.4.4 con los coeficientes normalizados obtendremos la expresión de la distancia normalizada

$$\hat{d}(F,\tilde{S}) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} \hat{u}_{ij} \cdot |f^*_{N_i} - f^*_{P_j}| \tag{5.56}$$

Ejemplo 5.4.5 *Supongamos un instrumento musical que interpreta una melodía determinada. Mediante el análisis de una grabación digital realizada de dicha interpretación, y utilizando el método propuesto en 5.56, determínese qué sistema de afinación es más compatible con las notas de la melodía interpretada por dicho instrumento. Tómese como referencia la frecuencia del* $\mathrm{La}_4 = 440Hz$ *y nivel de tolerancia* $\delta = 50$.

En primer lugar, hemos de determinar las frecuencias fundamentales de cada una de las notas que interpreta el instrumento. Para ello dividiremos la grabación digital en numerosas ventanas temporales discretas, compuestas por 2^{14} muestras, aplicando a cada una de ellas una función ventana de tipo *Hamming*. Posteriormente obtendremos el espectro energético de cada ventana mediante la aplicación del algoritmo de *Fast Fourier Transform* propuesto por *Cooley-Turkey* (véase Chu (2008) y Dos Passos (2016)). Encontraremos computacionalmente la frecuencia fundamental de cada ventana temporal mediante la aplicación del método *Harmonic Product Spectrum* (véanse André, Khelf y Leclere (2017) y Noll (1970)). La función ventana Hamming viene determinada por la siguiente expresión (Enochson y Otnes, 1968, pág. 142):

$$w(n) = 0{,}54 - 0{,}46 \cdot cos\left(\frac{2\pi n}{M-1}\right) \tag{5.57}$$

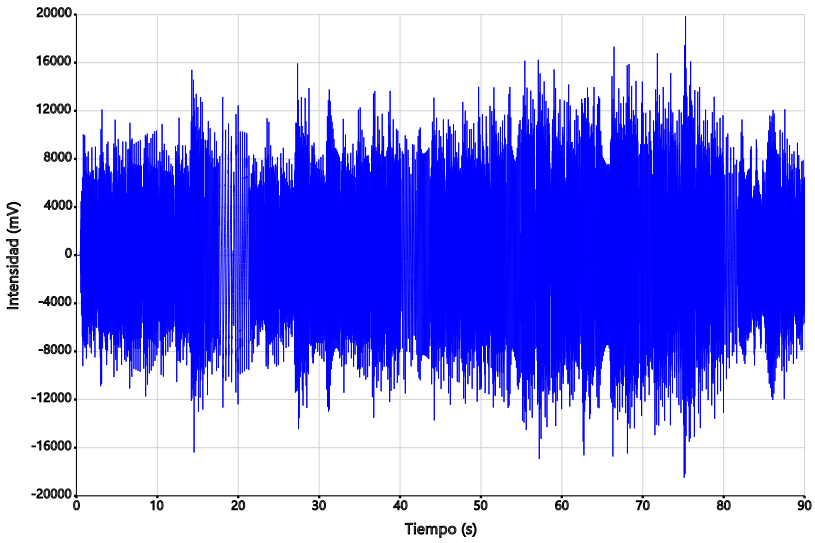


Figura 5.5: Grabación de la melodía a analizar.

En la siguiente figura podemos ver el ejemplo de una ventana temporal construida con un total de 2^{14} muestras sobre la que se ha aplicado la función ventana 5.57.

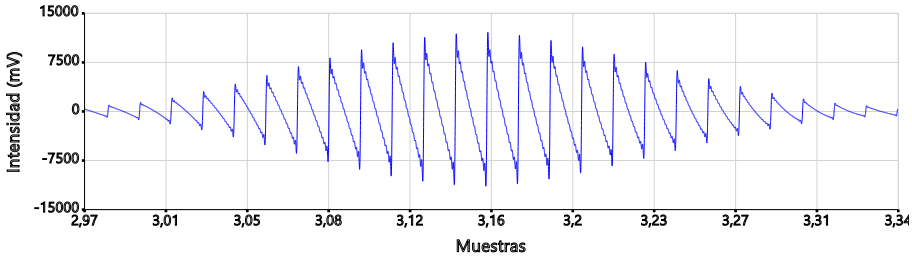


Figura 5.6: Ventana temporal número ocho de la grabación analizada.

El análisis FFT de la ventana anterior nos arroja el siguiente espectro energético, en el que podemos comprobar como el timbre del instrumento grabado se compone únicamente de quince armónicos.

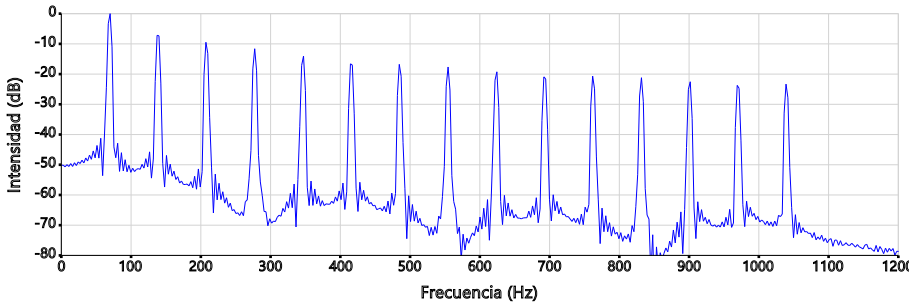


Figura 5.7: Espectro energético correspondiente a la ventana temporal número ocho.

Mediante el método HPS, propuesto por Noll (1970), podemos determinar la frecuencia fundamental de dicho espectro. El funcionamiento de dicho método aprovecha la propiedad de los armónicos musicales gracias a la cual el armónico número n es igual a $n \cdot f_0$, donde f_0 es la frecuencia fundamental. Sumando varias veces el espectro original con compresiones de ratio $1 : k$ del mismo espectro conseguimos anular todos los componentes espectrales que no sean armónicos. La frecuencia fundamental a determinar f_0 será el primero (y usualmente el más intenso de los armónicos que hayan sobrevivido al proceso HPS). En la siguiente figura podemos ver resumido su funcionamiento:

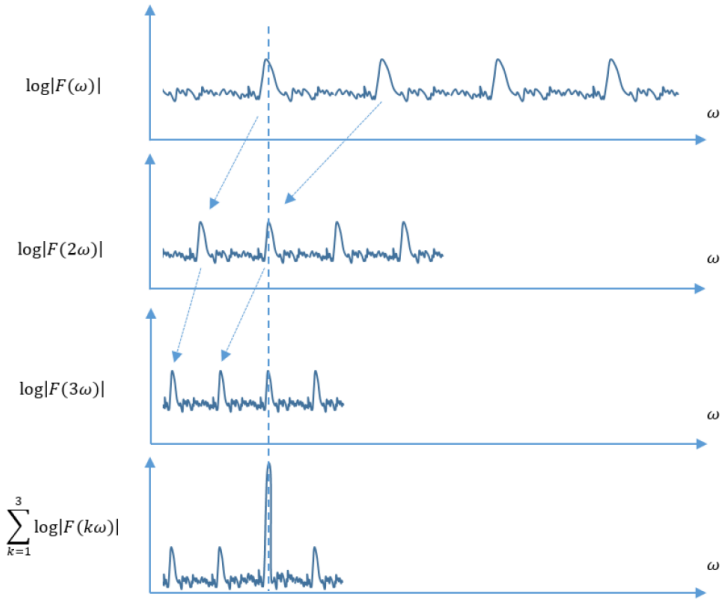


Figura 5.8: Ejemplo del funcionamiento del Harmonic Product Spectrum como el resultado de una suma de k espectros comprimidos (André, Khelf y Leclere, 2017, pág. 2).

El resultado de aplicar el método HPS al conjunto de todas las ventanas de la grabación nos devuelve un histograma de aparición de determinadas frecuencias, que circunscritas al intervalo Do_4-Do_5 resulta de la siguiente manera:

f_0(Hz)	Nº	f_0(Hz)	Nº	f_0(Hz)	Nº	f_0(Hz)	Nº
262,53	4	326,97	10	369,78	5	439,49	9
273,37	4	328,30	5	385,93	5	460,81	6
276,06	10	332,30	5	389,96	4	463,48	10
278,63	6	343,03	5	391,09	10	466,16	5
289,45	5	347,02	5	407,23	1	490,47	4
292,14	10	348,42	10	412,67	10	491,61	5
294,68	6	353,69	1	413,87	5	493,03	10
308,80	1	364,48	5	417,95	4	514,56	6
309,51	4	367,01	5	436,71	5	519,94	10
310,77	15	368,50	5	437,97	4	521,14	5

Tabla 5.21: Resumen de las frecuencias fundamentales f_0 encontradas en la grabación mediante el método HPS, acompañadas del número de veces que aparecen.

Utilizando un nivel de tolerancia $\delta = 50$ podemos calcular el coeficiente de pertenencia $\mu_{\tilde{P}_j}(f_{N_i})$ de cada f-nota f_{N_i} a cada f-grado $\tilde{P}_j$. Mediante expresión 5.56 podemos calcular la distancia media de afinación normalizada del conjunto de notas de la tabla 5.21 con diversos sistemas de afinación, y de esta manera encontrar cuál es el más compatible.

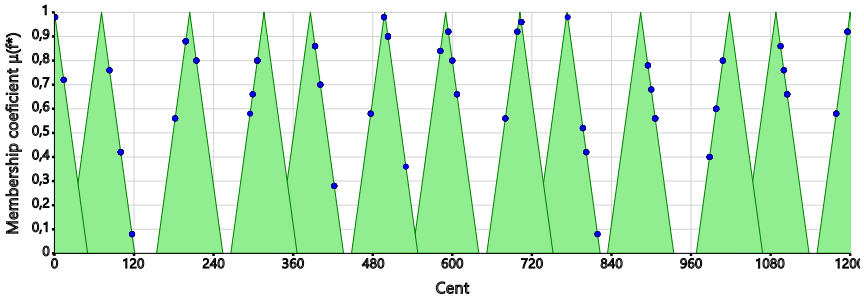


Figura 5.9: Notas de la tabla 5.21 superpuestas sobre la Justa Entonación.

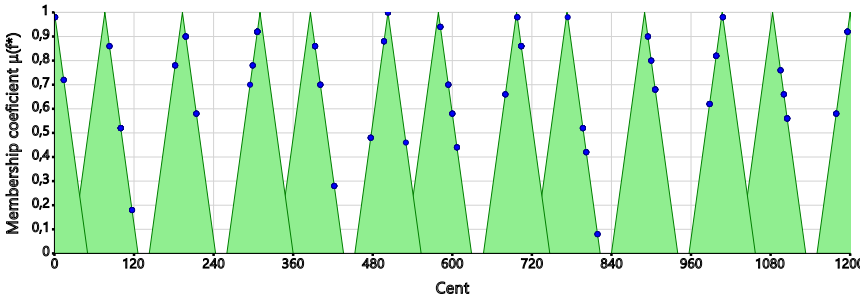


Figura 5.10: Notas de la tabla 5.21 superpuestas sobre el Mesotónico1/4.

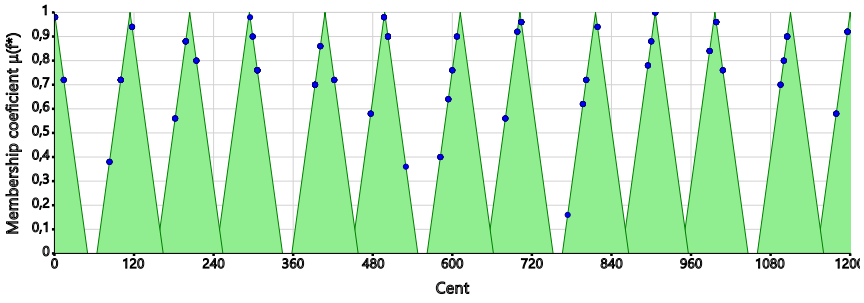


Figura 5.11: Notas de la tabla 5.21 superpuestas sobre el sistema Pitagórico.

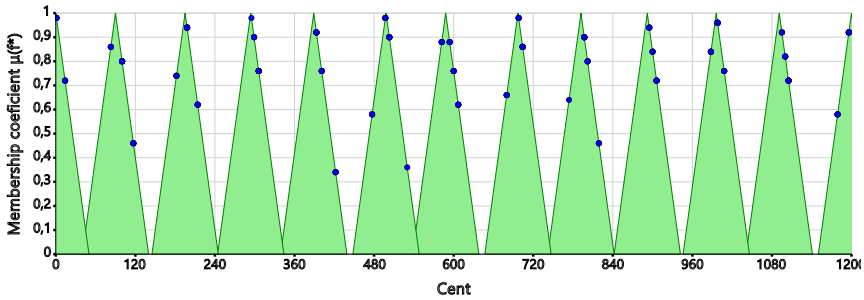


Figura 5.12: Notas de la tabla 5.21 superpuestas sobre el temperamento Kellner-Bach.

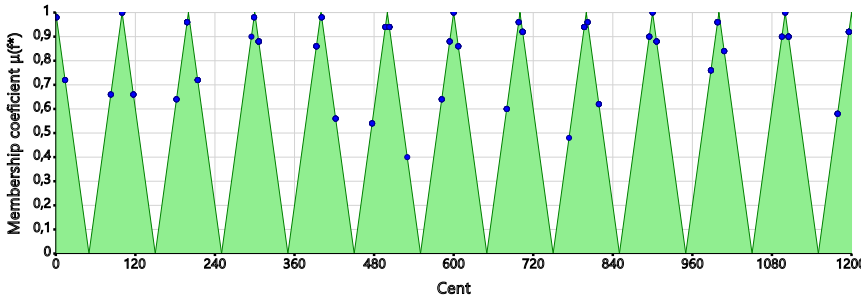


Figura 5.13: Notas de la tabla 5.21 superpuestas sobre el Temperamento Igual.

Los resultados computacionales obtenidos para la distancia media de afinación normalizada entre el conjunto de notas de la tabla 5.21 y los diferentes sistemas de afinación son los siguientes:

Sistema de afinación	Distancia
Justa Entonación	1,9753793532
Mesotónico 1/4	1,8113336683
Pitagórico cerrado	1,2553954370
Kellner-Bach	1,2089606993
Temperamento igual	**0,8495630336**

Por lo que podemos concluir que las notas interpretadas son más compatibles con el Temperamento Igual de doce notas por octava, por lo que podemos concluir que el instrumento estaba afinado en ese sistema.

5.5 Similitud entre sistemas de afinación

5.5.1 Comparación de sistemas de afinación de igual número de grados

Si consideramos dos sistemas de afinación con el mismo número n de grados, tal y como han sido definidos en 5.3.1 quedarían finalmente expresados mediante dos conjuntos de número reales $\mathcal{S}^A = \left\{r_i^A\right\}_{i=1}^n$ y $\mathcal{S}^B = \left\{r_j^B\right\}_{j=1}^n$. Representados en un espacio métrico 1-dimensional, podemos medir una distancia entre ellos acumulando la distancia parcial existente entre cada pareja de grados r_i^A, r_j^B, con $i, j \geq 1 \leq n$, respetando el orden establecido por la secuencia de grados de cada sistema de afinación.

Definición 5.5.1 Distancia media entre dos sistemas de afinación. *Sean* $\mathcal{S}^A = \{r_1^A, \ldots, r_n^A\}$ *y* $\mathcal{S}^B = \{r_1^B, \ldots, r_n^B\}$ *dos sistemas de afinación, ambos con n grados, pertenecientes a un espacio métrico* 1*-dimensional* $\mathbb{R}$. *Sea* $d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ *una función distancia, la distancia media entre* $\mathcal{S}^A$ *and* $\mathcal{S}^B$ *se define como*

$$\bar{D}(\mathcal{S}^A, \mathcal{S}^B) = \frac{1}{n} \sum_{i=1}^{n} d(r_i^A, r_i^B). \tag{5.58}$$

5.5.2 Medición de la disimilitud entre sistemas de afinación mediante el fuzzy clustering

De manera análoga a como realizamos en el capítulo anterior, si queremos comparar sistemas de afinación de diferente número de grados, la ecuación 5.58 tiene que ser generalizada. Proponemos dos medidas de la disimilitud, basadas en la utilización de los algoritmos FCM y FOCM, definidas en la sección 3.9 mediante las expresiones 3.76 y 3.77. Consideremos dos sistemas de afinación $\mathcal{S}^A$ y $\mathcal{S}^B$ con un número distinto de grados. Realicemos una partición *fuzzy* de los grados de $\mathcal{S}^A$ con los centroides iniciales proporcionados por los grados de $\mathcal{S}^B$, y apliquemos el algoritmo FCM (véase 3.6) un número de l veces, hasta que el criterio de parada sea satisfecho. Una vez que el proceso de particionado ha sido completado, podemos utilizar la función de disimilitud presentada en 3.76 para definir una disimilitud entre $\mathcal{S}^A$ y $\mathcal{S}^B$ utilizando los coeficientes finales de pertenencia u_{ij} calculados en la matriz U y los centroides originales.

Definición 5.5.2 Disimilitud media entre dos sistemas de afinación. *Sean* $\mathcal{S}^A = \{r_1^A, \ldots, r_n^A\} \subset \mathbb{R}$ *y* $\mathcal{S}^B = \{r_1^B, \ldots, r_m^B\} \subset \mathbb{R}$ *dos sistemas de afinación, donde*

$n > m$. Sea $d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ una función distancia. Sean u_{ij} los coeficientes de pertenencia finales calculados con el algoritmo FCM. La disimilitud media $\mathscr{D}$ entre $\mathcal{S}^A$ y $\mathcal{S}^B$ se define como

$$\mathscr{D}(\mathcal{S}^A, \mathcal{S}^B) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} u_{ij} \cdot d(r_i^A, r_j^B) \; . \tag{5.59}$$

Definición 5.5.3 Disimilitud media ordenada entre dos sistemas de afinación. *Sean $\mathcal{S}^A = \{r_1^A, \ldots, r_n^A\} \subset \mathbb{R}$ y $\mathcal{S}^B = \{r_1^B, \ldots, r_m^B\} \subset \mathbb{R}$ dos sistemas de afinación donde $n > m$. Sea $d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ una función distancia. Sean $\hat{u}_{ij}$ los coeficientes de pertenencia finales calculados mediante el algoritmo FOCM de $\mathcal{S}^A$ sobre $\mathcal{S}^B$. La disimilitud media ordenada $\widehat{\mathscr{D}}$ entre $\mathcal{S}^A$ y $\mathcal{S}^B$ se define como*

$$\widehat{\mathscr{D}}(\mathcal{S}^A, \mathcal{S}^B) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} \hat{u}_{ij} \cdot d(r_i^A, r_j^B) \; . \tag{5.60}$$

Ejemplo 5.5.1 *Calcúlese la disimilitud media ordenada entre el sistema $\mathcal{S}^A$ de afinación de Zarlino, de 19 notas (véase 5.4) con el Temperamento Igual de 12 notas (sistema $\mathcal{S}^B$) y con el Temperamento igual de 19 notas por octava ($\mathcal{S}^C$), utilizando diferentes funciones distancia y diferentes funciones de vecindad. Tómese el valor del grado de fuzzyness $\lambda = 2$.*

Los resultados computacionales obtenidos se muestran en la siguiente tabla. Observamos como el sistema de Zarlino es mucho más similar al Temperamento igual de 19 notas por octava.

Tabla 5.22: Resultados del cálculo de la disimilitud media ordenada entre los sistemas de afinación de Zarlino ($\mathcal{S}^A$) con el Temperamento Igual de 12 notas por octava ($\mathcal{S}^B$) y Temperamento igual de 19 notas por octava ($\mathcal{S}^C$).

Func. Vecindad	Func. Distancia	$\widehat{\mathscr{D}}(\mathcal{S}^A, \mathcal{S}^B)$	$\widehat{\mathscr{D}}(\mathcal{S}^A, \mathcal{S}^C)$
Gaussiana	Euclidiana	2,3748359358	0,4486382847
Gaussiana	Canberra	0,0057975191	0,0006609388
Gaussiana	Discreta	0,0818340606	0,0498614958
Exponencial	Euclidiana	2,2230760291	0,4486382847
Exponencial	Canberra	0,0035681603	0,0006609388
Exponencial	Discreta	0,0811489144	0,0498614958
Triangular	Euclidiana	2,4348853123	0,4486382847
Triangular	Canberra	0,0042923987	0,0006609388
Triangular	Discreta	0,0819931774	0,0498614958

5.5.3 Transiciones entre sistemas de afinación

De forma análoga a como se mostró en el capítulo 4 para los casos de la melodía, ritmo, armonía y timbre, podemos realizar transiciones completas entre dos sistemas de afinación $\mathcal{S}^A$ y $\mathcal{S}^B$ cualesquiera.

Ejemplo 5.5.2 *Estados intermedios finales de la transición completa entre los sistemas de afinación de Zarlino, de 19 notas ($\mathcal{S}^A$) (véase 5.4) y el Temperamento Igual de 12 notas (sistema $\mathcal{S}^B$). Se han utilizado los parámetros $\lambda = 1{,}5$, vecindad exponencial estrecha y métrica de Canberra.*

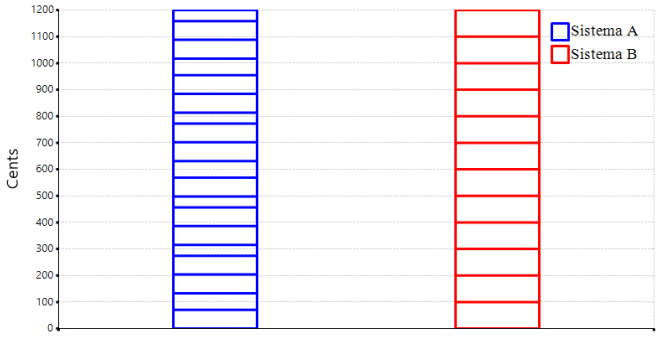


Figura 5.14: Estado inicial del sistema $\mathcal{S}^B$ (Temperamento igual de 12 notas).

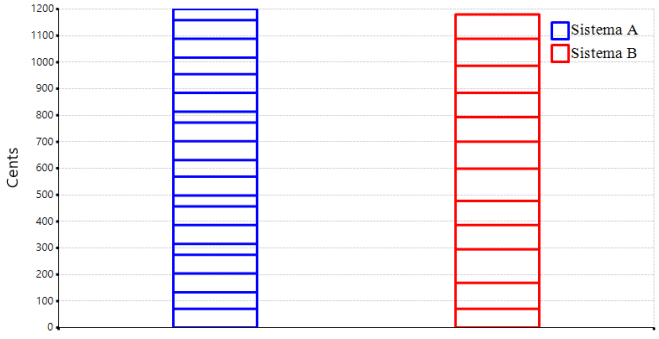


Figura 5.15: Estado 1 de la transición de $\mathcal{S}^B$ hacia $\mathcal{S}^A$.

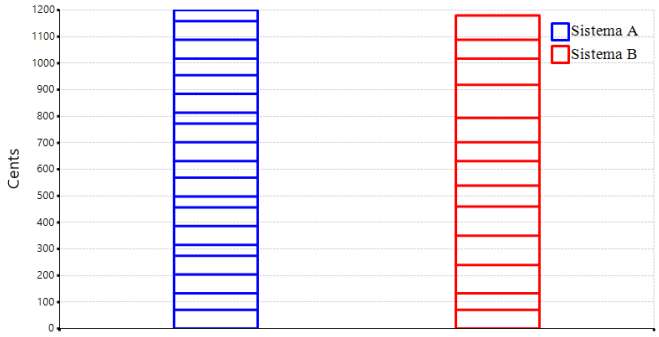


Figura 5.16: Estado 2 de la transición de $\mathcal{S}^B$ hacia $\mathcal{S}^A$.

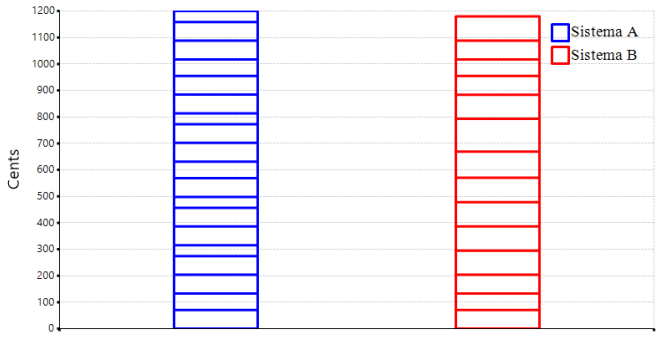


Figura 5.17: Estado 3 de la transición de $\mathcal{S}^B$ hacia $\mathcal{S}^A$.

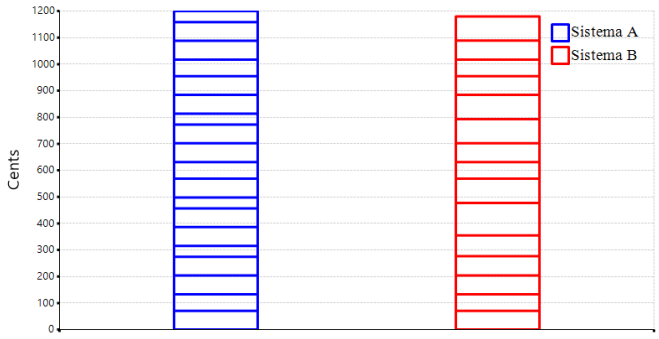


Figura 5.18: Estado 4 de la transición de $\mathcal{S}^B$ hacia $\mathcal{S}^A$.

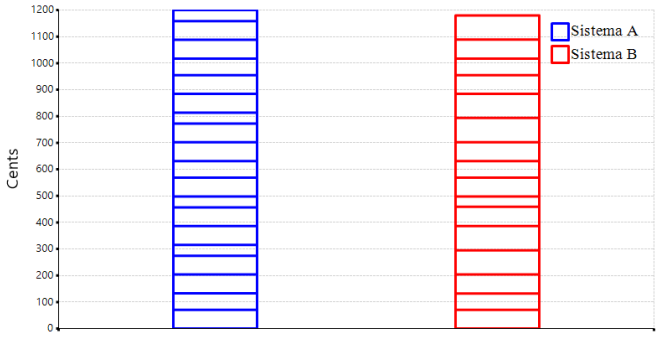


Figura 5.19: Estado 5 de la transición de $\mathcal{S}^B$ hacia $\mathcal{S}^A$.

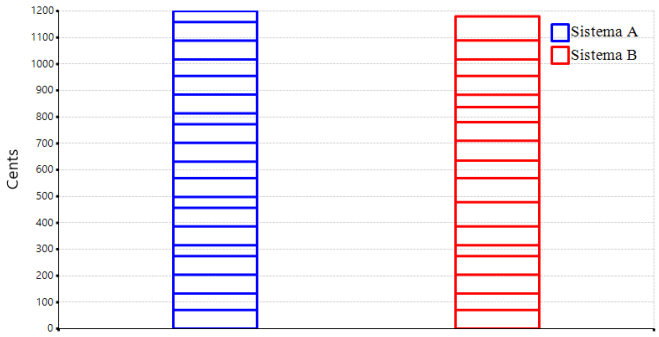


Figura 5.20: Estado 6 de la transición de $\mathcal{S}^B$ hacia $\mathcal{S}^A$.

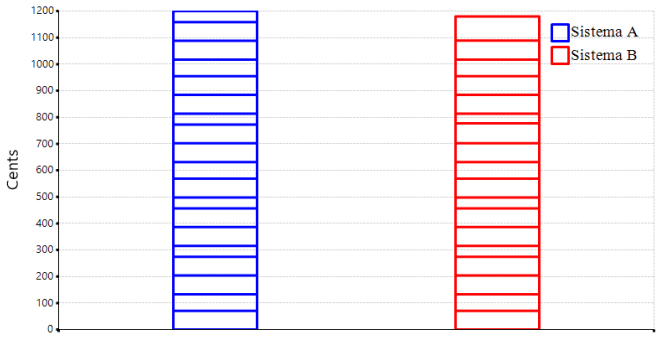


Figura 5.21: Estado 7 de la transición de $\mathcal{S}^B$ hacia $\mathcal{S}^A$.

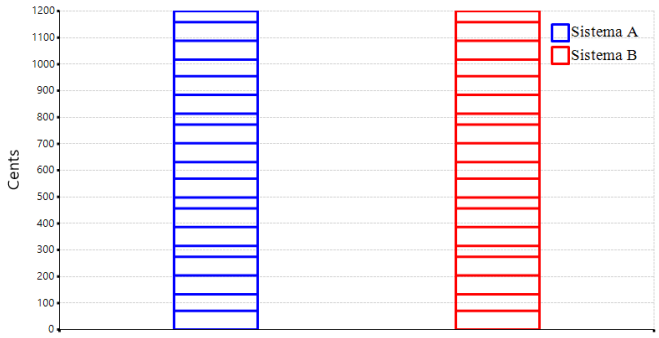


Figura 5.22: Estado final de la transición de $\mathcal{S}^B$ hacia $\mathcal{S}^A$.

En la tabla 5.23 podemos ver los resultados numéricos, expresados en cents, de los diferentes estados de la transición completa entre los dos sistemas de afinación.

Inicial	1	2	3	4	5	6	7	8
100,00	70,71	133,24	133,24	133,24	70,67	70,67	70,67	70,67
200,00	168,66	70,69	203,94	203,94	132,10	133,24	133,24	133,24
300,00	294,98	239,30	70,67	276,80	203,95	203,94	203,94	203,94
400,00	386,27	350,45	295,11	70,67	295,08	274,58	274,58	274,58
500,00	477,52	460,10	386,44	355,21	386,28	315,64	315,64	315,64
600,00	598,46	538,75	477,75	477,40	459,19	386,38	386,31	386,31
700,00	700,84	631,12	570,70	568,67	498,04	477,63	457,04	456,99
800,00	793,14	702,20	669,38	631,28	568,42	568,71	498,04	498,04
900,00	884,54	793,53	792,66	701,98	631,28	634,75	568,72	568,72
1000,00	986,72	918,85	883,97	793,17	702,41	710,42	631,35	631,28
1100,00	1088,44	1017,19	955,03	884,35	793,69	780,25	701,96	701,96
1200,00	1179,76	1088,55	1017,60	955,03	884,36	837,46	777,04	772,63
		1179,86	1088,64	1017,60	955,03	884,04	813,69	813,69
			1179,95	1088,72	1017,60	954,98	884,36	884,36
				1180,01	1088,79	1017,60	955,03	955,03
					1180,06	1088,83	1017,60	1017,6
						1180,09	1088,85	1088,27
							1180,11	1158,94
								1200,00

Tabla 5.23: Valores expresados en cents del estado inicial de $\mathcal{S}^B$ y estados intermedios en su transición completa hasta $\mathcal{S}^A$.

5.6 Resumen

En este capítulo hemos estudiado los sistemas de afinación, realizando una revisión de los conceptos básicos necesarios previos a su definición matemática genérica en base a las propuestas realizadas por Liern (2005). Se ha aplicado dicha definición a los principales sistemas de afinación y temperamentos existentes en la tradición musical occidental, mostrando tanto los intervalos como las funciones necesarias para la producción de todas sus notas musicales y ejemplificando los resultados obtenidos para cada uno de ellos.

> *Se ha definido un método para el cálculo de la compatibilidad entre un conjunto de notas medidas experimentalmente y un sistema de afinación, definiendo las notas en términos* fuzzy *y adaptando las definiciones de disimilitud propuestas en el capítulo 3. Se ha extendido la disimilitud media ordenada definida en capítulos anteriores al caso de la disimilitud entre sistemas de afinación, utilizándose para ello el algoritmo FOCM (véase 3.7). Se han definido, además, transiciones completas entre distintos sistemas de afinación mediante la aplicación del algoritmo FCT (véase 3.8.1).*

Además se ha incluido en MERCURY dichos cálculos, realizándose una implementación informática en este programa capaz de detectar las frecuencias fundamentales desde grabaciones musicales monofónicas en formato *wav*, gracias al algoritmo de Cooley-Turkey para la *Fast Fourier Transform* y el método *Harmonic Product Spectrum* (HPS) para la determinación de la frecuencia fundamental. A lo largo del capítulo se han mostrado diferentes resultados procedentes de experimentos en los que se ha calculado diferentes compatibilidades, disimilitudes y transiciones completas entre distintos sistemas de afinación.

Capítulo 6

Conclusiones

«[...]Aschenbach había escrito expresamente, en un pasaje poco conocido de sus obras, que casi todas las cosas grandes que existen son grandes porque se han creado contra algo, a pesar de algo: a pesar de dolores y tribulaciones, de pobreza y abandono; a pesar de la debilidad corporal, del vicio, de la pasión. Eso era algo más que una observación: era el resultado de una experiencia íntimamente vivida por él, la fórmula de su vida y de su gloria, la clave de su obra.»

(Thomas Mann. La muerte en Venecia.)

6.1 Conclusiones

En esta tesis hemos propuesto distintos métodos para generar variaciones y transiciones musicales, basados en los métodos de clasificación difusa. Dichas técnicas nos permiten realizar transiciones, parciales o completas, entre dos melodías, ritmos o armonías. Nos permiten, además, establecer una medida de la disimilitud entre dos secuencias musicales cualesquiera, en la que el orden de los elementos es tenida en cuenta. Dicha disimilitud nos ha permitido extender los resultados obtenidos a los sistemas de afinación, definiendo una medida de la compatibilidad entre un conjunto de notas medidas experimentalmente y un sistema de afinación cualquiera, así como la definición de una disimilitud entre dos sistemas de afinación. Se han implementado computacionalmente estas técnicas en el software

MERCURY, facilitándonos ilustrar con numerosos ejemplos la utilización de las técnicas propuestas en la composición musical asistida por ordenador.

6.1.1 Sobre los objetivos

A continuación justificamos brevemente el grado de consecución de los objetivos que nos planteamos al iniciar esta tesis:

1. Se ha definido una nueva medida de la disimilitud melódica entre una melodía inicial y otra final, basada en algoritmos de agrupamiento difuso. Se han realizado numerosos experimentos con material melódico de distinta índole, en los que se comprueba como dicha disimilitud es fácilmente calculable mediante su implementación computacional. Durante el proceso de cálculo, se obtienen diferentes pasos que pueden ser interpretados como estados intermedios de la convergencia de la melodía inicial hacia la melodía final. Gracias a la aplicación reiterada de esta técnica, en la que se añade una nueva nota a la melodía inicial cada vez que ésta finaliza, se ha propuesto un algoritmo capaz de construir transiciones melódicas completas entre dos melodías dadas.

2. Se ha generalizado la medida de la disimilitud a cualquier otro elemento musical que sea parametrizable en términos vectoriales, como por ejemplo el ritmo, la armonía o el timbre. Como consecuencia de esto, se han generalizado la construcción de transiciones completas para el caso de dos ritmos, dos secuencias de acordes, o dos espectros tímbricos cualesquiera. Por último, se han extendido estas técnicas al estudio de la compatibilidad entre notas y sistemas de afinación, proporcionando una generalización tanto de la medida de la disimilitud entre una serie de notas y un sistema de afinación, como de la disimilitud entre dos sistemas de afinación cualesquiera.

3. Se han implementado computacionalmente en el software MERCURY todas las técnicas propuestas en la presente memoria de investigación, proporcionando al compositor herramientas con las que generar, de manera sencilla, variaciones y transiciones, ya sean de carácter melódico, rítmico, armónico o tímbrico. Se amplía de esta manera el catálogo de técnicas relativas al ámbito de la composición musical asistida por ordenador y se ofrece, al mismo tiempo, un nuevo paradigma práctico para el problema de la creatividad musical computarizada. Se ha utilizado MERCURY para la generación del material musical empleado en la creación de la composición *Transiciones difusas*, adjunta como apéndice.

6.1.2 Sobre el estado de la cuestión

Se ha realizado una revisión bibliográfica que abarca desde los inicios de la computación musical, liderados por los pioneros experimentos de Lejaren Hiller y otros investigadores, hasta nuestros días. Tal y como se ha explicado en la sección de metodología, el proceso se ha realizado consultando las publicaciones de las principales revistas y congresos relacionados con nuestro ámbito de investigación, así como distintos libros, tesis y páginas web. Las conclusiones que podemos extraer de dicho proceso de búsqueda son las siguientes:

1. Las líneas básicas de investigación existentes en el ámbito de la composición musical asistida por ordenador son las siguientes: la utilización de sistemas estocásticos, caóticos o autosemejantes, la implementación de reglas compositivas y satisfacción de restricciones, la utilización de autómatas celulares, el uso de algoritmos genéticos; los sistemas de agentes, la utilización de gramáticas formales; los modelos basados en cadenas de Markov, la aplicación de técnicas propias del *machine learning* como las redes neuronales o los modelos de *n-gramas*.

2. No hemos encontrado en la literatura científica ningún ejemplo de utilización de algoritmos o técnicas de *clustering* (k-means, c-means, etc.) aplicados directamente a la composición musical asistida por ordenador. Existen, sin embargo, algunos ejemplos de su uso en el campo del análisis musical asistido por ordenador (reconocimiento de estilos, clasificación de compositores, clasificación de modos melódicos, etc.). Por tanto, consideramos que nuestra aportación constituye un enfoque original en cuanto al uso de algoritmos de agrupamiento aplicados a la generación de variaciones o transiciones entre materiales musicales.

6.1.3 Sobre el marco teórico

Hemos descrito la clasificación y funcionamiento de los distintos algoritmos de agrupamiento (*clustering*) existentes, haciendo énfasis sobre los de tipo *hard*, como por ejemplo el *k-means*, así como su versión difusa denominada *fuzzy c-means*, en el que cada punto del conjunto de datos puede pertenecer a más de un grupo o categoría. De esta manera, la pertenencia de cada punto a cada grupo deja de estar representada por una variable binaria, para convertirse en un coeficiente real positivo, comprendido entre cero y uno, que indica el grado de pertenencia de cada punto a cada uno de los grupos. Es posible, por tanto, que cada uno de los puntos sea clasificado simultáneamente como perteneciente en mayor o menor grado a cada uno de los grupos. La matriz de partición fuzzy nos permite ahora

introducir una normalización de sus elementos determinada por las funciones de vecindad que, aplicadas sobre el algoritmo *c-means*, introducirán en el algoritmo la dependencia entre el orden de la secuencia de datos y el orden de la secuencia de centroides y por tanto el proceso de agrupamiento se realizará ahora teniendo en consideración el orden de los elementos. Dicho algoritmo, denominado *Fuzzy Ordered c-Means* (FOCM), constituye una de nuestras aportaciones principales presentadas en esta memoria de investigación.
Utilizando este algoritmo, hemos definido una disimilitud media ordenada para dos secuencias de datos de distinta longitud, en la que el resultado depende del orden de los elementos constituyentes de cada cadena. Los experimentos realizados con dicha disimilitud sobre la melodía, el ritmo, la armonía, el timbre y los sistemas de afinación, arrojan resultados coherentes y muestran su viabilidad para comparar secuencias de datos de distinta longitud, no únicamente en el terreno musical sino en cualquier situación en que se requiera expresar determinada información como una sucesión finita de puntos pertenecientes a un espacio métrico de dimensión finita.
Por último, hemos presentado el algoritmo *fuzzy complete transitions*(FCT), una propuesta para realizar transiciones musicales completas, ya sean de carácter melódico, rítmico, armónico o tímbrico, que constituye la principal aportación sobre la que se articula la presente investigación. Gracias a este algoritmo, podemos realizar variaciones y transiciones desde un material musical inicial hacia otro final.

6.1.4 Sobre la implementación

Se ha realizado la implementación computacional de los algoritmos Fuzzy Ordered c-Means (FOCM) y Fuzzy Complete Transitions (FCT) en el software Mercury®. La introducción de datos, ya sean melodías, ritmos o armonías se puede realizar cargando archivos con formato *MusicXML*. El programa extrae y procesa la información que necesita para cada caso, mostrando por la pantalla los resultados. Mediante una interfaz gráfica, el usuario puede configurar los distintos parámetros bajo los que se ejecutarán las transiciones y experimentar con los resultados obtenidos, que serán representados gráficamente en forma de notación musical convencional. Dichos resultados pueden ser guardados en archivo, reproducidos a través del MIDI o exportados en formato MusicXML para su posterior edición con cualquier programa de notación musical (*Sibelius*, *Finale*, *MuseScore*, etc.). Además, permite la inclusión manual o a través del portapapeles de los datos de diferentes espectros de frecuencia y sistemas de afinación, para realizar cálculos de disimilitud o transiciones en estos casos. Los resultados pueden exportarse nuevamente a través del portapapeles o de forma visual mediante la generación

de distintos tipos de gráficas.
Los ejemplos que se muestran en la presente memoria relativos al cálculo de la disimilitud y transiciones musicales han sido generados utilizando nuestro software MERCURY. En todos los casos, la implementación se ha mostrado robusta, eficaz en términos de convergencia y eficiente, requiriendo tiempos computacionales reducidos incluso para la generación de transiciones entre secuencias musicales largas. Los resultados han podido ser guardados y exportados correctamente en formato MusicXML, siendo posteriormente abiertos y maquetados en el editor de partituras Finale 2015. Ha resultado muy ergonómica la posibilidad que ofrece la interfaz gráfica de MERCURY para trabajar con múltiples pestañas simultáneamente, dividiendo el área de trabajo y posibilitando la comparación de los resultados obtenidos bajo distintos parámetros. El material generado por el programa depende sensiblemente de la configuración inicial seleccionada por el usuario, configurada por el coeficiente de *fuzzyness*, el criterio de parada, la función de vecindad o la función distancia. Esto se traduce en una enorme cantidad variaciones y transiciones posibles. Los algoritmos que utiliza el programa son deterministas: siempre que se utilicen los mismos parámetros sobre el mismo material musical, se encontrarán idénticos resultados, ya que no se utiliza durante el cálculo ningún elemento estocástico.

6.1.5 Sobre la composición musical

Se ha compuesto de la obra *Transiciones difusas*, para cuarteto de cuerda, adjunta como Apéndice II, en la que encontramos un ejemplo completo de composición musical realizada utilizando transiciones y variaciones musicales generadas mediante MERCURY. Esta composición consta de tres movimientos, y cada uno responde a la aplicación compositiva de un tipo de transición musical diferente.

En el primer movimiento, *Andante*, se han utilizado las transiciones de tipo melódico. El material musical procede de la transición melódica realizada desde un sencillo motivo, construido con interválicas tritonales, hasta una melodía construida con una serie dodecafónica. Los estadios intermedios resultantes de esta transición se escuchan distribuidos en cada una de las cuatro voces, sucediéndose uno tras otro de manera ordenada. Progresivamente las melodías acaban superponiéndose hasta que se alcanza la melodía final.

El segundo movimiento, *Corale, adagio molto*, constituye un ejemplo sobre la utilización de transiciones armónicas, a un total de cuatro voces, generadas mediante nuestro programa. Se han utilizado diversas progresiones armónicas, procedentes de la verticalización de la serie dodecafónica utilizada en el primer movimiento.

Por último, el tercer movimiento, *Presto con fuoco* muestra una compleja polifonía rítmica realizada mediante la superposición de los diferentes ritmos intermedios calculados mediante transiciones entre diversos *deci-talas* hindúes.

En nuestra opinión, la experiencia compositiva con MERCURY ha sido satisfactoria. El programa resulta cómodo y su interfaz de usuario es ágil y versátil, haciendo posible la experimentación entre distintos tipos de material musical y distintas configuraciones de los parámetros. La búsqueda de material musical ha de hacerse de manera exploratoria, ya que no es posible de antemano predecir las características de los resultados obtenidos por nuestro sistema.

Los métodos presentados en esta memoria de investigación amplían el catálogo de técnicas relativas a la composición musical asistida por ordenador, proporcionando nuevas herramientas compositivas capaces de generar variaciones y transiciones musicales y abriendo, al mismo tiempo, una línea de investigación basada en la utilización de métodos de clasificación difusa aplicados a la "creatividad artificial".

6.2 Futuras líneas de investigación

A lo largo de la realización de esta tesis, han ido surgiendo nuevas preguntas de investigación, tanto desde el punto de vista de la computación como musicales. A continuación resumimos algunas:

1. Hemos observado que los algoritmos implementados convergen muy rápidamente, es decir, en las primeras iteraciones se produce un gran distanciamiento de los centroides con respecto a su posición inicial. Este hecho, deseable en términos de la eficiencia computacional, se traduce musicalmente en transiciones demasiado abruptas. La transición musical, tradicionalmente entendida como un proceso de variación orgánica, consiste en un equilibrio entre repetición y cambio progresivo. Por tanto, resultaría interesante modificar dichos algoritmos para obtener una menor eficiencia de los mismos, que se traduzca en unas transiciones más suaves.

2. Las transiciones completas se realizan aplicando el algoritmo Fuzzy Ordered c-Means de forma reiterada, añadiendo un punto nuevo cada vez que éste finaliza. Dicho punto se localiza en la posición intermedia de la pareja de centroides consecutivos cuya distancia es máxima; por tanto los puntos se tienden a localizar en las posiciones intermedias localizadas donde los centroides están más separados. Sería interesante introducir nuevas reglas o posibilidades para la localización de dichos nuevos puntos: posiciones aleatorias, entre la pareja de centroides consecutiva de menor distancia, entre posiciones intermedias de centroides y datos, etc. Todos estas posibilidades proporcionarían aún más variedad al conjunto de resultados obtenidos. La creación de nuevos puntos en posiciones aleatorias introduciría a su vez un comportamiento no determinista en el algoritmo.

3. En esta primera versión de la implementación informática se ha restringido, por motivos de simplicidad, las líneas melódicas a una única voz. No es posible, por tanto, realizar transiciones melódicas polifónicas. Sin embargo, no existe limitación teórica que justifique dicha restricción, por lo que en una futura versión, el software Mercury podría ser capaz de realizar comparaciones y transiciones melódicas de forma polifónica.

4. En el caso de las transiciones armónicas, su implementación computacional se ha limitado a secuencias de acordes con igual número de voces. Tal y como se explica en Martínez y Liern (2017), dicha limitación podría ser superada utilizando el mínimo común múltiplo del número de voces de ambas secuencias.

5. Las transiciones entre timbres, caracterizadas como transiciones entre espectros de frecuencias estáticos, pueden ser de interés para la composición electrónica (Supper, 2004) o la composición espectral (Garant, 2011) realizando una síntesis aditiva cuyos parciales e intensidades van progresivamente cambiando con el tiempo según los resultados que arroja el algoritmo. Sin embargo, la realidad del análisis espectral de los instrumentos musicales acústicos sugiere que el timbre de éstos contiene una dependencia temporal que ha de ser tenida en cuenta si se pretende alcanzar sonidos con una verdadera riqueza sonora. Técnicas como la síntesis FM, propuesta por Chowning (1973), permiten alcanzar dicha riqueza temporal con poco gasto computacional. Podría ser una interesante línea de investigación en el ámbito de la síntesis sonora combinar las posibilidades de la síntesis FM con las transiciones espectrales aquí presentes, generando transiciones de timbres variables con el tiempo, creados mediante la caracterización vectorial de parámetros típicos de esta síntesis como el índice de modulación o la armonicidad.

6. Las transiciones entre distintos sistemas de afinación nos abren la puerta a pensar en composiciones en las cuales el sistema de afinación no sea constante, sino que puede cambiar de un sistema inicial a otro final, atravesando diferentes estados intermedios generados mediante el algoritmo FCT. Podemos encontrar un interesante ejemplo en la obra *Strandlines* para guitarra y electrónica, del compositor americano Richard Karpen, estrenada por el guitarrista Stefan Östersjö (Coessens y Östersjö, 2014; Östersjö, 2008), en la cual la guitarra varía en múltiples ocasiones, mediante *scordaturas*, su sistema de afinación.

7. Hemos comprobado experimentalmente, gracias a numerosos ejemplos, la convergencia de nuestros algoritmos FOCM y FCT, sin encontrar ningún contraejemplo a la convergencia de los mismos, siempre que se establezca un criterio de parada óptimo. Sin embargo, esto no constituye una demostración matemática, quedando pendiente su realización para futuros investigadores.

8. Enfocando su uso hacia la música electrónica y la síntesis sonora, se podría definir la nota musical de manera no simbólica, incluyendo en las características que la definen, además de la frecuencia y la intensidad, los parámetros relativos a la envolvente del sonido de tipo *ADSR* (*Attack, Decay, sustain, Release*), para la generación de transiciones en las que también cambiarán los aspectos relativos a la envolvente del sonido.

9. En la presente investigación nos hemos centrado en la notación musical convencional, utilizando el temperamento igual de doce notas como sistema de afinación de referencia. y dejando a un lado numerosas manifestaciones mu-

sicales que hacen uso de otros sistemas, como por ejemplo las escalas con microinterválica propias de la música folklórica[1], la música microtonal[2], la música de carácter no simbólico (sirvan de ejemplo la *Musique concrete*[3] o la *Elektronische Musik*[4]), o músicas de vanguardia en las que se utilizan nuevas grafías para una notación musical no convencional (técnicas extendidas, multifónicos, *slaps*, etc.). Estas realidades musicales pueden ser paulatinamente parametrizadas de forma matemática e incorporadas en los algoritmos propuestos en la presente tesis.

10. MERCURY ha sido desarrollado para el sistema operativo *Windows* bajo *.NET Framework 4.5*. Siendo conscientes de la actual evolución hacia el uso de aplicaciones móviles y sistemas multiplataforma, queda pendiente una futura implementación del programa en dispositivos móviles. Gracias al sistema de capas utilizado en su desarrollo y el bajo nivel de acoplamiento entre los distintos módulos, el módulo de cálculo es fácilmente exportable a otro entorno de desarrollo o lenguaje de programación.

11. La música se constituye sobre unos elementos básicos: la melodía, el ritmo, la armonía, el timbre, y la afinación. En numerosas ocasiones el compositor ha de realizar una elección que favorece uno de estos aspectos en detrimento de algunos otros, buscando una suerte de mínimo local. La definición de unas transiciones musicales completas podría abordarse como un problema de decisión multicriterio (Kahraman, 2008) en el que podrían incorporarse los métodos propuestos en la presente investigación.

12. Se debería investigar un algoritmo de transiciones inversas entre dos secuencias, en el que la secuencia de mayor número de elementos es la que se modifica progresivamente mediante una aplicación reiterada del algoritmo FOCM. Se podría suprimir un punto cada vez que el algoritmo finalice, hasta que se alcance la secuencia final.

[1] Véase los artículos *Toward a quarter-tone syntax: Analyses of selected works by Blackwood, Haba, Ives, and Wyschnegradsky* de Skinner (2007), y La originalidad musical del flamenco: libertad creativa y sometimiento a cánones de Berlanga (2009).

[2] Véase *Toward a quarter-tone syntax: Analyses of selected works by Blackwood, Haba, Ives, and Wyschnegradsky* de Skinner (2007).

[3] Véase el *Traité des objets musicaux* de Schaeffer (1966).

[4] Véase el texto *Texte zur Musik* de Stockhausen y col. (1963).

Bibliografía

Abel, J. F. (1981). *Computer Composition of Melodic Deep Structures*. Ann Arbor, Michigan: Michigan Publishing, University of Michigan Library (vid. pág. 32).

Agostini, A. (2012). *Bach: An environment for computer-aided composition in max*. Ann Arbor, Michigan: Michigan Publishing, University of Michigan Library (vid. pág. 62).

Agustín-Aquino, O. A., Junod, J. y Mazzola, G. (2015). *Computational Counterpoint Worlds. Mathematical Theory, Software, and Experiments*. Computational Music Science. Heidelberg: Springer (vid. pág. 66).

Albini, G (2018). Combinatorics, Probability and Choice in Music Composition: Towards an Aesthetics of Composing Systems for Non-Musicians. *Proceedings of Bridges 2018: Mathematics, Art, Music, Architecture, Education, Culture*. Stockholm, Sweden (vid. pág. 68).

Alexander, A. (2015). *Infinitesimal. How a Dangerous Mathematical Theory Shaped the Modern World*. New York: One World (vid. pág. 185).

Aloupis, G., Fevens, T., Langerman, S., Matsui, T., Mesa, A., Nuñez, Y., Rappaport, D. y Toussaint, G. (2006). Algorithms for computing geometric measures of melodic similarity. *Computer Music Journal*, 30(3), págs. 67-76 (vid. pág. 4).

Ames, C. (1982). Protocol: Motivation, design, and production of a composition for solo piano. *Journal of New Music Research*, 11(4), págs. 213-238 (vid. pág. 33).

— (1987). Automated composition in retrospect. *Leonardo*, págs. 169-185 (vid. pág. 16).

André, H., Khelf, I. y Leclere, Q. (2017). Harmonic Product Spectrum revisited and adapted for rotating machine monitoring based on IAS. *Proceedings of the 9th International Conference Surveillance*. Fes, Morocco (vid. págs. 224, 226).

Computer Generated Orchestration: Towards Using Musical Timbre in Composition (2017). Strasbourg, France: Université de Strasbourg (vid. pág. 67).

Ariza, C. (2011). Two pioneering projects from the early history of computer-aided algorithmic composition. *Computer Music Journal*, 35(3), págs. 40-56 (vid. págs. 16-18).

Assayag, G., Rueda, C., Laurson, M., Agon, C. y Delerue, O. (1999). Computer-assisted composition at IRCAM: From PatchWork to OpenMusic. *Computer Music Journal*, 23(3), págs. 59-72 (vid. págs. 40-42).

Babbitt, M. (1962). Twelve-tone rhythmic structure and the electronic medium. *Perspectives of New Music*, págs. 49-79 (vid. pág. 145).

Bach, J. S. (1996). *371 Corales para órgano o armonio, dividido en dos tomos*. Vol. 2. Barcelona: Boileau (vid. págs. 148, 149).

Bain, R. (1990). Algorithmic Composition: Quantum Mechanics and the Musical Domain. *Proceedings of the 1990 International Computer Music Conference*. Glasgow, Scotland, págs. 276-279 (vid. pág. 39).

Bales, W. K. (1978). *Computer-aided Composition and Performance with AMUS*. Ann Arbor, Michigan: Michigan Publishing, University of Michigan Library (vid. págs. 28, 29).

Ball, P. (2012). Computer science: Algorithmic rapture. *Nature*, 488, págs. 458-458 (vid. pág. 3).

Barbour, J. M. (2004). *Tuning and temperament: A historical survey*. New York: Dover Publications (vid. págs. 176, 212).

Bemman, B. y Meredith, D. (2016). Generating Milton Babbitt all-partition arrays. *Journal of New Music Research*, 45(2), págs. 184-204 (vid. pág. 68).

— (2018). Generating new musical works in the style of Milton Babbitt. *Computer Music Journal*, 42(1), págs. 60-79 (vid. pág. 68).

Benward, B. (2014). *Music in Theory and Practice*. Vol. 1. New Yotk: McGraw-Hill Higher Education (vid. pág. 37).

Berlanga, M. Á. (2009). La originalidad musical del flamenco: libertad creativa y sometimiento a cánones. *La Nueva Alboreá*, 10, págs. 32-34 (vid. pág. 245).

Beyls, P. (1989). The musical universe of cellular automata. *Proceedings of the 1989 International Computer Music Conference*. Ohio, USA, págs. 34-41 (vid. pág. 38).

— (2003). Selectionist musical automata: Integrating explicit instruction and evolutionary algorithms. *Proceedings of the IX Brazilian Symposium on Computer Music*. Campinas, Brasil (vid. pág. 39).

Bezdek, J. C. (1973). *Fuzzy mathematics in pattern classification*. Tesis doct. Applied Math. Center, Cornell University, New York, USA (vid. págs. 89, 90).

— (1981). *Pattern recognition with fuzzy objective function algorithms*. New York: Plenum Press (vid. págs. 13, 74, 75, 77, 84, 85, 89, 110).

Bezdek, J. C., Keller, J., Krisnapuram, R. y Pal, N. (1999). *Fuzzy models and algorithms for pattern recognition and image processing*. Boston, London, Dordrecht: Kluwer Academic Publishers (vid. pág. 74).

Biles, J. A. y col. (1994). GenJam: A genetic algorithm for generating jazz solos. *Proceedings of the 1994 International Computer Music Conference*. Copenhagen, Denmark, págs. 131-137 (vid. pág. 45).

Biletskyy, A. (2000). Doctor Webern: A visual environment for computer-assisted composition based on linear thematism. *Computer Music Journal*, 24(3), págs. 34-37 (vid. pág. 51).

Bolognesi, T. (1983). Automatic composition: Experiments with self-similar music. *Computer Music Journal*, 7(1), págs. 25-36 (vid. págs. 35, 36).

Boulanger, R. C. (2000). *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming*. Cambridge, Massachusetts: The MIT press (vid. pág. 37).

Brindle, R. S. (1969). *Serial composition*. Oxford: Oxford University Press (vid. págs. 151, 153, 154).

Brooks Jr, F. P., Hopkins Jr, A. L., Neumann, P. G. y Wright, W. V. (1957). *An experiment in musical composition*. Cambridge, Massachusetts: The MIT Press, págs. 23-40 (vid. págs. 18, 25).

Buhmann, J. (1995). Data clustering and learning. En: *The Handbook of Brain Theory and Neural Networks*. Ed. por M. A. Arbib. Cambridge, Massachusetts: The MIT Press, págs. 278-281 (vid. pág. 72).

Bumgardner, J. (2009). Kircher's Mechanical Composer: A Software Implementation. *Proceedings of Bridges 2009: Mathematics, Music, Art, Architecture, Culture*. Banff, Canada, págs. 21-28 (vid. pág. 1).

Busoni, F. (1911). *A New Esthetic of Music*. New York: G. Schirmer (vid. pág. 1).

Buteau, C. (2006). Melodic Clustering Within Topological Spaces of Schumann's Träumerei. *Proceedings of 2006 International Computer Music Conference*. New Orleans, USA (vid. pág. 55).

Cage, J. (1961a). *Silence: lectures and writings*. New york: Wesleyan University Press (vid. pág. 71).

— (1961b). To Describe the Process of Composition Used in Music of Changes and Imaginary Landscape No. 4. En: *Silence: lectures and writings*. New york: Wesleyan University Press, págs. 57-59 (vid. pág. 4).

Calvo-Manzano, A. (1991). *Acústica físico-musical*. Madrid: Real Musical (vid. pág. 191).

Carpentier, G. y Bresson, J. (2010). Interacting with symbol, sound, and feature spaces in orchidée, a computer-aided orchestration environment. *Computer Music Journal*, 34(1), págs. 10-27 (vid. pág. 59).

Carretero, A. (2013). El proceso de composición musical a través las técnicas bio-inspiradas de inteligencia artificial: investigación desde la creación musical. Tesis doct. Universidad Rey Juan Carlos, Madrid, España (vid. pág. 65).

Cherla, S., Purwins, H. y Marchini, M. (2013). Automatic phrase continuation from guitar and bass melodies. *Computer Music Journal*, 37(3), págs. 68-81 (vid. pág. 63).

Chomsky, N. (1957). Syntactic Structures. *Lectures on Government and Binding*, 6, págs. 1-52 (vid. pág. 64).

— (1963). Formal properties of grammars. En: *Handbook of Mathematical Psychology*. Ed. por R. Duncan, R. R. Bush y E. Galanter. Vol. 2, págs. 328-418 (vid. pág. 28).

Chowning, J. M. (1973). Synthesis of complex audio spectra by means of frequency modulation. *Journal of audio engineering society*, 21(7), págs. 526-534 (vid. pág. 244).

Chu, E. (2008). *Discrete and continuous fourier transforms: analysis, applications and fast algorithms*. Boca Raton, London, New York: CRC Press (vid. pág. 224).

Chuan, C.-H. y Chew, E. (2011). Generating and evaluating musical harmonizations that emulate style. *Computer Music Journal*, 35(4), págs. 64-82 (vid. pág. 61).

Cipriani, A. y Giri, M. (2010). *Electronic music and sound design*. Roma: ConTempoNet (vid. pág. 49).

Clatworthy, J., Buick, D., Hankins, M., Weinman, J. y Horne, R. (2005). The use and reporting of cluster analysis in health psychology: A review. *British journal of health psychology*, 10(3), págs. 329-358 (vid. pág. 72).

Codina, L. (2017). *Revisiones sistematizadas y cómo llevarlas a cabo con garantías*. URL: `https://www.lluiscodina.com/revision-sistematica-salsa-framework/` (visitado 20-04-2017) (vid. pág. 8).

Coessens, K. y Östersjö, S. (2014). Kairos in the Flow of Musical Intuition. En: *Artistic Experimentation in Music: An Anthology*. Ed. por D. Crispin y B. Gilmore. Leuven University Press, págs. 323-332 (vid. pág. 244).

Collins, N. (2012). Automatic composition of electroacoustic art music utilizing machine listening. *Computer Music Journal*, 36(3), págs. 8-23 (vid. pág. 63).

Comaniciu, D. y Meer, P. (1999). Mean shift analysis and applications. *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. Kerkyra, Greece, págs. 1197-1203 (vid. pág. 72).

Conway, J. (1970). The game of life. *Scientific American*, 223(4), pág. 4 (vid. pág. 44).

Cope, D. (2004). A musical learning algorithm. *Computer Music Journal*, 28(3), págs. 12-27 (vid. pág. 54).

Coul, M. O. de (2015). *Huygens-Fokker Foundation centre for microtonal music*. URL: `http://www.huygens-fokker.org/docs/measures.html` (visitado 27-12-2017) (vid. págs. 185, 188, 189).

Courtot, F. (1990). A constraint-based logic program for generating polyphonies. *Proceedings of the 1990 International Computer Music Conference*. Glasgow, Scotland, págs. 292-294 (vid. pág. 40).

— (1992). CARLA: Knowledge acquisition and induction for computer assisted composition. *Journal of New Music Research*, 21(3), págs. 191-217 (vid. pág. 40).

Cuthbert, M. S. y Ariza, C. (2010). music21: A toolkit for computer-aided musicology and symbolic music. *Proceedings of the 11th International Society for Music Information Retrieval Conference*. Utrecht, Netherlands, págs. 637-642 (vid. pág. 58).

Cuthbert, M. S., Ariza, C. y Friedland, L. (2011). Feature Extraction and Machine Learning on Symbolic Music using the music21 Toolkit. *Proceedings of*

the 12th International Society for Music Information Retrieval Conference. Miami, Florida, págs. 387-392 (vid. pág. 58).

Czekanowski, J. (1962). The theoretical assumptions of Polish anthropology and morphological facts. *Current Anthropology*, 3(5), págs. 481-494 (vid. pág. 82).

De León, P. J. P., Rizo, D., Ramirez, R. e Iñesta, J. M. (2008). Melody characterization by a genetic fuzzy system. *Proceedings of the 5th Sound and Music Computing Conference*. Berlin, Germay, pág. 15 (vid. pág. 56).

Deal, S. y Sanchez, J. (2013). Integration of machine learning algorithms in the computer-acoustic composition Goldstream Variations. *Proceedings of the 2013 International Computer Music Conference*. Perth, Australia (vid. pág. 64).

Déchelle, F., Borghesi, R., Cecco, M. D., Maggi, E., Rovan, B. y Schnell, N. (1999). jMax: an environment for real-time musical applications. *Computer Music Journal*, 23(3), págs. 50-58 (vid. págs. 48, 49).

Dobrian, C. (1993). *Music and artificial intelligence*. URL: `http://music.arts.uci.edu/dobrian/CD.music.ai.htm` (visitado 04-02-2019) (vid. pág. 2).

Dodge, C. (1988). Profile: A Musical Fractal. *Computer Music Journal*, 12(3), págs. 10-14 (vid. pág. 38).

Dos Passos, W. (2016). *Numerical Methods, Algorithms and Tools in C#*. Boca Raton, London, New York: CRC Press (vid. pág. 224).

Duda, R. O., Hart, P. E. y Stork, D. G. (1973). *Pattern classification*. New York: Wiley (vid. págs. 75, 85).

Dunn, J. C. (1973). A fuzzy relative of the ISODATA process and its use in detecting compact well-separated clusters. *Journal of Cybernetics*, 3(3), págs. 32-57 (vid. pág. 89).

Duran, B. S. y Odell, P. L. (1974). *Cluster analysis, a survey*. Vol. 100. Lectures notes in economics and mathematical systems. Berlin: Springer (vid. págs. 107, 108).

Ellis, A. J. (1885). On the musical scales of various nations. *Nature*, 31, págs. 488-490 (vid. págs. 186, 189).

Enochson, L. D. y Otnes, R. K. (1968). Programming and analysis for digital time series data. *Defense Technical Information Center* (vid. pág. 224).

Erickson, R. F. (1975). The Darms project: A status report. *Computers and the Humanities*, 9(6), págs. 291-298 (vid. pág. 4).

Falkenstein, J. T. von y Tlalim, T. (2009). Alter Ego: A Generative Music Creation System. *Proceedings of the 2009 International Computer Music Conference*. Ann Arbor, Michigan (vid. pág. 57).

Farbood, M. y Schöner, B. (2001). Analysis and Synthesis of Palestrina-Style Counterpoint Using Markov Chains. *Proceedings of 2001 International Computer Music Conference*. Havana, Cuba (vid. págs. 53, 54).

Fernández, D. (1919). Un matemático español del siglo XVII: Juan Caramuel. *Revista matemática hispano-americana*, págs. 121-127 (vid. pág. 185).

Fernández, J. D. y Vico, F. (2013). AI methods in algorithmic composition: A comprehensive survey. *Journal of Artificial Intelligence Research*, 48, págs. 513-582 (vid. pág. 3).

Fisch, R., Gravner, J. y Griffeath, D. (1991). Cyclic cellular automata in two dimensions. En: *Spatial Stochastic Processes*. Ed. por K. Alexander y J. Watkins. Birkhäuser, Boston: Springer, págs. 171-185 (vid. pág. 44).

Forte, A. (1973). *The structure of atonal music*. New Haven: Yale University Press (vid. pág. 35).

Gan, G., Ma, C. y Wu, J. (2007). *Data clustering: theory, algorithms, and applications*. Philadelphia: SIAM (vid. págs. 72-74, 79, 84, 86).

Garant, D. (2011). *Tristan Murail: les objets sonores complexes: analyse de l'esprit des dunes*. Paris: Editions L'Harmattan (vid. pág. 244).

Gimenes, M. y Miranda, E. R. (2008). An A-Life Approach to Machine Learning of Musical Worldviews for Improvisation Systems. *NICS Reports*, 1, págs. 52-70 (vid. pág. 56).

Goldáraz, J. J. (1989). Aristógenes en la teoría musical del renacimiento: Fundamentos de la ciencia armónica y medición de intervalos. *Revista de Musicología*, 12(1), págs. 23-46 (vid. págs. 176, 177).

Grant, M. J. y Booth, A. (2009). A typology of reviews: an analysis of 14 review types and associated methodologies. *Health Information & Libraries Journal*, 26(2), págs. 91-108 (vid. pág. 8).

Green, M (1980). PROD: A grammar-based computer composition program. *Proceedings of the 1980 International Computer Music Conference*. San Francisco, California, págs. 101-110 (vid. pág. 31).

Gross, D. (1975). *A set of computer programs to aid in music analysis*. Tesis doct. Indiana University, Bloomington, Indiana, USA (vid. pág. 4).

Harley, J. (2004). *Xenakis: his life in music*. New York: Routledge (vid. pág. 23).

Hartmann, P. (1990). Selection of musical identities. *Proceedings of the 1990 International Computer Music Conference*. Glasgow, Scotland, págs. 234-236 (vid. pág. 46).

Hild, H., Feulner, J. y Menzel, W. (1992). HARMONET: A neural net for harmonizing chorales in the style of J.S. Bach. En: *Advances in Neural Information Processing Systems*. Ed. por J. E. Moody, S. J. Hanson y R. P. Lippmann. Burlington, Massachusetts: Morgan-Kaufmann, págs. 267-274 (vid. pág. 47).

Hiller, L. (1959). Computer music. *Scientific American*, 201(6), págs. 109-121 (vid. pág. 20).

— (1968). *Music composed with computers: an historical survey*. 18. Urbana-Champaign: University of Illinois (vid. págs. 16, 17).

— (1982). *Stochastic Generation of Note Parameters for Music Composition*. Ann Arbor, Michigan: Michigan Publishing, University of Michigan Library (vid. pág. 35).

Hiller, L. y Baker, R. A. (1964). Computer Cantata: A study in compositional method. *Perspectives of New Music*, págs. 62-90 (vid. pág. 20).

Hiller, L. e Isaacson, L. (1959). *Experimental music: composition with an electronic computer*. New York: McGraw-Hill (vid. págs. 2, 15-17, 19, 20, 33).

Holmes, T. (2002). *Electronic and experimental music. Pioneers in technology and composition*. New York: Routledge (vid. pág. 16).

— (2012). *Electronic and experimental music: technology, music, and culture*. New York: Routledge (vid. págs. 16, 19, 21, 24).

Honingh, A. y Bod, R. (2011). Clustering and classification of music by interval categories. En: *Mathematics and Computation in Music. MCM 2011*. Ed. por C. Agon, M. Andreatta, G. Assayag, E. Amiot, J. Bresson y J. Mandereau. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, págs. 346-349 (vid. pág. 62).

Horner, A. y Ayers, L. (1995). Harmonization of Musical Progressions with Genetic Algorithms. *Proceedings of the 1995 International Computer Music Conference*. Banff, Canada, págs. 483-484 (vid. pág. 45).

Horner, A. y Goldberg, D. E. (1991). Genetic algorithms and computer-assisted music composition. *Proceedings of the 1991 International Computer Music Conference*. Vol. 51. 61801. Montreal, Canada, págs. 479-482 (vid. págs. 4, 43).

Jacob, B. (1995). Composing with genetic algorithms. *Proceedings of the 1995 International Computer Music Conference*. Banff, Canada, págs. 452-455 (vid. pág. 46).

Jain, A. K. y Dubes, R. C. (1988). *Algorithms for clustering data*. New Jersey: Prentice-Hall, Inc. (vid. pág. 73).

Johnson, R. A., Wichern, D. W. y col. (2014). *Applied multivariate statistical analysis*. New Jersey: Prentice-Hall (vid. pág. 83).

Johnson, R. S. (1989). *Messiaen*. Los Angeles, California: University of California Press (vid. pág. 143).

Jones, D. E. (2000). A Computational Composer's Assistant for Atonal Counterpoint. *Computer Music Journal*, 24(4), págs. 33-43 (vid. págs. 51, 52).

Jones, K. J. (1980). Sound Generating Techniques on the ITT 2020 and Apple II Computers. *Proceedings of the Conference on Computer Music in Britain*. London, UK (vid. pág. 34).

— (1981). Compositional applications of stochastic processes. *Computer Music Journal*, 5(2), págs. 45-61 (vid. pág. 33).

Kahraman, C. (2008). *Fuzzy multi-criteria decision making: theory and applications with recent developments*. Vol. 16. Springer Optimization and Its Applications. New York, USA: Springer-Verlag (vid. pág. 245).

Kaufman, L. y Rousseeuw, P. J. (2009). *Finding groups in data: an introduction to cluster analysis*. Vol. 344. New Jersey: John Wiley & Sons (vid. pág. 83).

Keller, R. M. y Morrison, D. R. (2007). A grammatical approach to automatic improvisation. *Proceedings of the Fourth Sound and Music Conference*. Lefkada, Greece (vid. págs. 4, 56).

Kenkel, N. C. y Orlóci, L. (1986). Applying metric and nonmetric multidimensional scaling to ecological studies: some results. *Ecology*, 67(4), págs. 919-928 (vid. pág. 81).

Klein, M. (1957). Syncopation in Automation. *Radio-Electronics*, 1, págs. 36-37 (vid. págs. 16, 17).

Klügel, N. (2014). *FugueGenerator-Collaborative Melody Composition Based on a Generative Approach for Conveying Emotion in Music*. Ann Arbor, Michigan: Michigan Publishing, University of Michigan Library (vid. pág. 65).

Knuth, D. (1969). *The art of computer programming*. Vol. 2. Boston: Addison-wesley (vid. pág. 185).

Koenig, G. M. (1970a). Project 1. *Electronic Music Reports*, 2(970), págs. 32-44 (vid. págs. 24, 33).

— (1970b). Project 2: A programme for musical composition. *Electronic Music Reports*, 3(970), págs. 4-6 (vid. pág. 24).

Kreidler, J. (2009). *Programming electronic music in Pd*. Hofheim: Wolke Verlag (vid. pág. 49).

Krzyzaniak, M. (2018). Interactive Learning of Timbral Rhythms for Percussion Robots. *Computer Music Journal*, 42(2), págs. 35-51 (vid. pág. 68).

Laine, P. (1997). Generating musical patterns using mutually inhibited artificial neurons. *Proceedings of 1997 International Computer Music Conference*. Thessaloniki, Greece, págs. 422-425 (vid. pág. 47).

Laitinen, M. y Lemström, K. (2010). Geometric algorithms for melodic similarity. *Proceedings of the 2010 Music Information Retrieval Evaluation eXchange*. Utrecht, Nederlands (vid. pág. 5).

Latham, A. (2009). *Diccionario enciclopédico de la música*. México: Fondo de Cultura económica (vid. págs. 122, 176, 180, 219).

Laurson, M., Kuuskankare, M. y Norilo, V. (2009). An overview of PWGL, a visual programming environment for music. *Computer Music Journal*, 33(1), págs. 19-31 (vid. págs. 56, 57).

Legendre, P. y Legendre, L. F. (1983). *Numerical ecology*. Oxford: Elsevier (vid. págs. 81, 83).

León, T. y Liern, V. (2012). Mathematics and soft computing in music. En: *Soft Computing in Humanities and Social Sciences*. Ed. por R. Seising y V. Sanz. Studies in Fuzziness and Soft Computing. Berlin: Springer, págs. 451-465 (vid. pág. 122).

Lerdahl, F. y Jackendoff, R. (2003). *Teoría generativa de la música tonal*. Trad. por J. González-Castelao. Madrid: Ediciones Akal (vid. pág. 180).

Levy, D. (2005). *Robots unlimited: Life in a virtual age*. New York: AK Peters/CRC Press (vid. pág. 1).

Lewis, J. P. (1989). *Algorithms for music composition by neural nets: improved CBR paradigms*. Ann Arbor, Michigan: Michigan Publishing, University of Michigan Library (vid. pág. 39).

— (1991). Creation by refinement and the problem of algorithmic music composition. En: *Music and connectionism*. Ed. por P. Todd y G. Loy. Cambridge, Massachusetts: The MIT Press, págs. 212-228 (vid. pág. 39).

Lidov, D. y Gabura, J. (1973). A melody writing algorithm using a formal language model. *Computers in the Humanities*, 3(4), págs. 138-48 (vid. págs. 26, 27).

Liern, V. (2005). Fuzzy tuning systems: the mathematics of musicians. *Fuzzy Sets and Systems*, 150(1), págs. 35-52 (vid. págs. 10, 12, 14, 184, 218-220, 235).

— (2015). On the construction, comparison, and exchangeability of tuning systems. *Journal of Mathematics and Music*, 9(3), págs. 197-213 (vid. págs. 195, 196, 200).

Link, J. W. (1963). *Theory and Tuning: Aaron's Meantone Temperament*. Los Angeles, California: Tuners Supply Company (vid. pág. 200).

Lipschutz, M. M. (1970). *Teoría y problemas de geometría diferencial*. New York: McGraw-Hill (vid. pág. 38).

Long, C., Wong, R. C. y Sze, R. K. W. (2013). T-Music: A melody composer based on frequent pattern mining. *Proceedings of the 29th International Conference on Data Engineering*. Brisbane, Australia, págs. 1332-1335 (vid. pág. 64).

Lorrain, D. (1980). A panoply of stochastic cannons. *Computer Music Journal*, 4(1), págs. 53-81 (vid. pág. 33).

Loughran, R., McDermott, J. y Neil, M. (2015). Grammatical Evolution with Zipf Law Based Fitness for Melodic Composition. *Proceedings of the 12th Sound and Music Computing Conference*. Maynooth, Ireland (vid. pág. 66).

Louzeiro, P. (2017). Real-time compositional procedures for mediated soloist-ensemble interaction: the comprovisador. *Proceedings of 2017 International Conference on Mathematics and Computation in Music*. Ciudad de México, México (vid. pág. 67).

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Oakland, USA., págs. 281-297 (vid. págs. 84, 110).

Magnusson, T. (2010). ixi lang: a SuperCollider parasite for live coding. *Proceedings of the 2010 International Computer Music Conference*. Utrecht, Netherlands, págs. 503-506 (vid. pág. 60).

Magnusson, T. (2011). ixi lang: a SuperCollider parasite for live coding. *Proceedings of 2011 International Computer Music Conference*. Huddersfield, UK, págs. 503-506 (vid. pág. 50).

Maidín, D. O. (1998). A geometrical algorithm for melodic difference. *Computing in musicology*, 11, págs. 65-72 (vid. pág. 4).

Manaris, B., Stevens, B. y Brown, A. R. (2016). JythonMusic: An environment for teaching algorithmic music composition, dynamic coding and musical performativity. *Journal of Music, Technology & Education*, 9(1), págs. 33-56 (vid. pág. 67).

Manning, P. (2013). *Electronic and computer music*. Oxford: Oxford University Press (vid. pág. 16).

Marion, J. B. (1975). *Dinámica clásica de las partículas y sistemas*. Trad. por J. Vilardel. Barcelona: Reverté (vid. pág. 191).

Martin, J. H. y Jurafsky, D. (2009). *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*. Prentice Hall Series in Artificial Intelligence. New York: Pearson (vid. pág. 31).

Martínez, B. y Liern, V. (2017). A Fuzzy-Clustering Based Approach for Measuring Similarity Between Melodies. *Proceedings of the Sixth International Conference on Mathematics and Computation in Music*. Ciudad de México, México, págs. 279-290 (vid. págs. 94, 99, 126, 129, 132, 134, 243).

— (2019). Mercury: A software based on fuzzy clustering for computer-assisted composition. *Proceedings of the Seventh International Conference on Mathematics and Computation in Music*. Madrid, España, págs. 200-212 (vid. págs. 32, 115, 117-121).

Martínez, B. (2018). *Técnicas computacionales para el cálculo de la similitud melódica*. Madrid, España: Editorial Académica Española (vid. pág. 5).

— (2019). *Emulación de estilos musicales mediante modelos de Markov controlados por ontologías pesadas*. Madrid, España: Editorial Académica Española (vid. págs. 17-20, 28, 35, 47, 53, 54, 57, 58, 62-65).

Mathews, M. V. y Moore, F. R. (1970). GROOVE, a program to compose, store, and edit functions of time. *Communications of the ACM*, 13(12), págs. 715-721 (vid. pág. 22).

Mathews, M. V. y Rosler, L. (1968). Graphical language for the scores of computer-generated sounds. *Perspectives of New Music*, 6(2), págs. 92-118 (vid. pág. 22).

Matthews, J. (1978). Music 3150, A Fortran Program for Composing Music for Conventional Instruments. *Proceedings of the International Conference on Mathematics and Computing*. Urbana, Illinois (vid. págs. 29, 30).

Matthews, M. (1981). *Algorithms for Harmonic Pitch Structure Generation*. Ann Arbor, Michigan: Michigan Publishing, University of Michigan Library (vid. pág. 35).

Maxwell, J. B., Eigenfeldt, A. y Pasquier, P. (2012). ManuScore: Music Notation-Based Computer Assisted Composition. *Proceedings of the 2012 International Computer Music Conference*. Ljubljana, Slovenia (vid. pág. 63).

Mazón, J. M. (2011). *Cálculo diferencial: teoría y problemas*. València: Universitat de València (vid. pág. 79).

Mazzola, G., Zahorka, O. y Stange-Elbe, J. (1995). Analysis and Performance of a Dream. *Proceedings of the KTH Symposion on Grammars for Music Performance*. Stockholm, Sweden, págs. 59-68 (vid. pág. 55).

McCartney, J. (1996). SuperCollider: a new real time synthesis language. *Proceedings of the 1996 International Computer Music Conference*. Hong Kong, China, págs. 257-258 (vid. pág. 50).

— (2002). Rethinking the computer music language: SuperCollider. *Computer Music Journal*, 26(4), págs. 61-68 (vid. pág. 50).

McIntyre, R. A. (1994). Bach in a box: The evolution of four part baroque harmony using the genetic algorithm. *Proceedings of the First IEEE Conference on Evolutionary Computation*. Orlando, Florida, págs. 852-857 (vid. pág. 46).

McNabb, M. (1981). Dreamsong: The Composition. *Computer Music Journal*, 5(4), págs. 36-53 (vid. pág. 35).

McVicar, M., Fukayama, S. y Goto, M. (2014). AutoRhythmGuitar: Computer-aided composition for rhythm guitar in the tab space. *Proceedings of the 2014 International Computer Music Conference*. Athens, Greece (vid. pág. 65).

Meinecke, J. (1981). *Stochastic Melody Writing Procedures: An Analysis Based Approach*. Ann Arbor, Michigan: Michigan Publishing, University of Michigan Library (vid. pág. 35).

Messiaen, O. (1956). *The Technique of My Musical Language*. Vol. 1. Paris: Alphonse Leduc (vid. pág. 138).

Meyer, M. F. (1951). Fokker's Organ in Huygens' Tuning. *The Journal of the Acoustical Society of America*, 23(3), págs. 369-369 (vid. pág. 185).

Millen, D (1990). Cellular automata music. *Proceedings of the 1990 International Computer Music Conference*. San Francisco (vid. pág. 43).

— (1992). Generation of formal patterns for music composition by means of cellular automata. *Proceedings of the 1992 International Computer Music Conference*. San Jose, California, págs. 398-398 (vid. pág. 43).

— (2004). An Interactive Cellular Automata Music Application in Cocoa. *Proceedings of 2004 International Computer Music Conference*. Miami, USA (vid. pág. 54).

Miranda, E. R. (1990). *Cellular Automata Music Investigation*. Tesis doct. University of York, York, UK (vid. págs. 4, 44).

— (1993). Cellular automata music: An interdisciplinary project. *Journal of New Music Research*, 22(1), págs. 3-21 (vid. pág. 44).

Mongeau, M. y Sankoff, D. (1990). Comparison of musical sequences. *Computers and the Humanities*, 24(3), págs. 161-175 (vid. pág. 4).

Montemurro, M. A. (2001). Beyond the Zipf-Mandelbrot law in quantitative linguistics. *Physica A: Statistical Mechanics and its Applications*, 300(3), págs. 567-578 (vid. pág. 66).

Moorer, J. A. (1972). Music and computer composition. *Communications of the ACM*, 15(2), págs. 104-113 (vid. págs. 25, 26).

Mozer, M. C. (1994). Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing. *Connection Science*, 6(2), págs. 247-280 (vid. pág. 47).

Myhill, J. (1978). Some simplifications and improvements in the stochastic music program. *Proceedings of the 1978 International Conference on Mathematics and Computing*. Urbana, Illinois (vid. pág. 23).

— (1979). Controlled Indeterminacy: A First Step Towards a Semi Stochastic Music Language. *Computer Music Journal*, 3(3), págs. 12-14 (vid. pág. 23).

Navarro, M. (2017). *Sociedades Humano-Agente: Un Caso de Estudio en Creatividad Musical*. Tesis doct. Departamento de Informática y Automática, Universidad de Salamanca, Salamanca, España (vid. pág. 67).

Neuman, I. (2013). Generative Grammars for Interactive Composition Based on Schaeffer TARTYP. *Proceedings of the 2013 International Computer Music Conference*. Perth, Australia (vid. pág. 64).

Nierhaus, G. (2009). *Algorithmic composition: paradigms of automated music generation*. Vienna: Springer-Verlag (vid. págs. 16, 34).

Nilson, C. (2007). Live coding practice. *Proceedings of the 7th international conference on new interfaces for musical expression*. Ney York, USA, págs. 112-117 (vid. pág. 60).

Noll, A. M. (1970). Pitch determination of human speech by the harmonic product spectrum, the harmonic surn spectrum, and a maximum likelihood estimate. *Proceedings of Symposium on Computer Processing in Communication*. Vol. 19. New York, págs. 779-797 (vid. págs. 224, 225).

Nuanáin, C. O., Herrera, P. y Jorda, S. (2015). Target-based rhythmic pattern generation and variation with genetic algorithms. *Proceedings of the 12th Sound and Music Computing Conference*. Maynooth, Ireland (vid. págs. 65, 66).

Núñez, A. (1993). *Informática y electrónica musical*. Madrid: Editorial Paraninfo (vid. pág. 16).

Östersjö, S. (2008). *Shut Up'N'Play! Negotiating the Musical Work*. Tesis doct. Malmö Academy of Music, Malmö, Sweden (vid. pág. 244).

Padberg, H. A. (1964). *Computer-composed canon and free-fugue*. Tesis doct. Saint Louis University, Saint Louis, Misuri, USA (vid. págs. 17, 18).

Pajares, R. L. (2012). *Historia de la música en 6 bloques. Bloque 5: Altura y Duración.* Vol. 5. Madrid: Editorial Visión Libros (vid. págs. 176, 177).

Park, T. H. (2009). *Introduction to digital signal processing: Computer musically speaking*. New Jersey: World Scientific (vid. pág. 163).

Persichetti, V. (1989). *Armonía del siglo XX*. Madrid: Real Musical (vid. págs. 155, 158).

Phillips, S. (2002). Acceleration of k-means and related clustering algorithms. *Algorithm Engineering and Experiments*, págs. 61-62 (vid. pág. 85).

Pierce, J. R. (1961). Symbols, signals, and noise: The nature and process of communication. *Foundations of Language*, 9(1), págs. 150-151 (vid. pág. 16).

Prinz, D (1952). Introduction to Programming on the Manchester Electronic Digital Computer. *Ferranti* (vid. pág. 17).

Psenicka, D. (2009). Automatic Score Generation with FOMUS. *Proceedings of the 2009 International Computer Music Conference*. Ann Arbor, Michigan (vid. pág. 57).

Puckette, M. (1986). The patcher. *Proceedings of the 1986 International Computer Music Conference*. San Francisco, USA (vid. pág. 48).

— (1996). Pure data, another integrated computer music enviornment. *Proceedings of the Second Intercollege Computer Music Concerts*. Tachikawa, Japan, págs. 37-41 (vid. págs. 48, 49).

— (2001). Pd Repertory Project-History of Pluton. *Proceedings of the 2001 International Computer Music Conference*. Havana, Cuba, págs. 21-25 (vid. pág. 48).

— (2007). *The theory and technique of electronic music*. Singapur: World Scientific Publishing Company (vid. pág. 49).

Puente, A. O. de la, Alfonso, R. S. y Moreno, M. A. (2002). Automatic composition of music by means of grammatical evolution. *Proceedings of the 2002 conference on APL: array processing languages*. Madrid, Spain, págs. 148-155 (vid. pág. 54).

Quick, D. y Hudak, P. (2013). A temporal generative graph grammar for harmonic and metrical structure. *Proceedings of the 2013 International Computer Music Conference*. Perth, Australia (vid. pág. 64).

Roads, C. (1978). *Composing grammars*. San Francisco, California: Computer Music Association (vid. págs. 28, 33).

— (1985). Research in music and artificial intelligence. *ACM Computing Surveys*, 17(2), págs. 163-190 (vid. pág. 2).

Roads, C. y Mathews, M. (1980). Interview with Max Mathews. *Computer Music Journal*, 4(4), págs. 15-22 (vid. pág. 2).

Rowat, R. (2019). *Huawei used AI technology to complete Schubert's unfinished symphony*. URL: `https://www.cbcmusic.ca/posts/20900/huawei-ai-technology-schubert-unfinished-symphony` (visitado 04-02-2019) (vid. pág. 3).

Rumsey, F. y McCormick, T. (2012). *Sonido y grabación. Introducción a las técnicas sonoras*. Trad. por S. Adriá, V. Pineda y M. Willis. Madrid: Ediciones Omega (vid. pág. 16).

Ruspini, E. H. (1970). Numerical methods for fuzzy clustering. *Information Sciences*, 2(3), págs. 319-350 (vid. pág. 77).

Salinas, F. d. (1577). *De musica libri septem*. Vol. 2. Salamanca: Mathias Gastius (vid. pág. 175).

Sánchez, C., Moreno, F., Albarracín, D., Fernández, J. D. y Vico, F. J. (2013). Melomics: A case-study of AI in Spain. *AI Magazine*, 34(3), págs. 99-103 (vid. págs. 3, 64).

Sandred, Ö. (2010). PWMC, a constraint-solving system for generating music scores. *Computer Music Journal*, 34(2), págs. 8-24 (vid. pág. 61).

Saunders, J. A. (1980). Cluster analysis for market segmentation. *European Journal of marketing*, 14(7), págs. 422-435 (vid. pág. 72).

Schaeffer, P. (1966). *Traité des objets musicaux*. Paris: Seuil (vid. págs. 64, 245).

Schöenberg, A. (2001). *Fundamentos de la composición Musical*. Madrid: Real Musical (vid. pág. 4).

Seaman, B. y Rössler, O. E. (2011). *Neosentience: The benevolence engine*. Bristol, UK y Chicago, USA: Intellect Books (vid. pág. 1).

Selfridge-Field, E. (1997). *Beyond MIDI: the handbook of musical codes*. Cambridge, Massachusetts: The MIT press (vid. págs. 121, 122, 124).

Šimundža, M. (1987). Messiaen's Rhythmical Organisation and Classical Indian Theory of Rhythm. *International Review of the Aesthetics and Sociology of Music*, págs. 117-144 (vid. pág. 146).

Skinner, M. L. (2007). *Toward a quarter-tone syntax: Analyses of selected works by Blackwood, Haba, Ives, and Wyschnegradsky*. Buffalo, New York, USA: State University of New York (vid. pág. 245).

Sly, A., Agarwala, N. e Inoue, Y. (2017). Music composition using recurrent neural networks. *Natural Language Processing with Deep Learning*, 1, págs. 1-12 (vid. pág. 67).

Smith, B. D. y Garnett, G. E. (2012). Unsupervised Play: Machine Learning Toolkit for Max. *Proceedings of the 12th international conference on new interfaces for musical expression*. Ann Arbor, Michigan, págs. 112-117 (vid. pág. 64).

Smith, L. (1972). SCORE: A Musician's Approach to Computer Music. *Journal of the Audio Engineering Society*, 20(1), págs. 7-14 (vid. pág. 28).

Steedman, M. J. (1984). A generative grammar for jazz chord sequences. *Music Perception: An Interdisciplinary Journal*, 2(1), págs. 52-77 (vid. pág. 37).

Stockhausen, K. Momentform: Neue Zusammenhange zwischen AuffUhrungsdauer, Werkdauer und Moment. *Texte zur elektronischen und instrumentalen Musik*, 1 (), págs. 23-45 (vid. pág. 4).

Stockhausen, K., Schnebel, D., Blumröder, C. von y Misch, I. (1963). *Texte zur Musik*. Berlin: Stockhausen Verlag (vid. pág. 245).

Stravinski, I. (1983). *Poética musical: en forma de seis lecciones*. Trad. por E. Grau. Madrid: Taurus (vid. pág. 113).

Supper, M. (2004). *Música electrónica y música con ordenador: historia, estética, métodos, sistemas*. Trad. por A. Arteaga. Madrid: Alianza Editorial (vid. págs. 16, 244).

Tipei, S. (1975). MP1: a computer program for music composition. *Proceedings of the Second Music Computation Conference*. Urbana, Illionis, págs. 68-82 (vid. pág. 28).

Tipler, P. (1985). *Física*. Trad. por A. Bramón, J. Casas, J. Enric y F. López. Vol. 1. Barcelona: Reverté (vid. págs. 177, 191).

Todd, P. M. (1989). A connectionist approach to algorithmic composition. *Computer Music Journal*, 13(4), págs. 27-43 (vid. pág. 39).

Tokui, N., Iba, H. y col. (2000). Music composition with interactive evolutionary computation. *Proceedings of the 3rd international conference on generative art*. Vol. 17. 2. Milan, Italy, págs. 215-226 (vid. pág. 53).

Triviño-Rodriguez, J. L. y Morales-Bueno, R. (2001). Using multiattribute prediction suffix graphs to predict and generate music. *Computer Music Journal*, 25(3), págs. 62-79 (vid. pág. 53).

Truax, B. (1973). *The Computer Composition: Sound Synthesis Programs POD4, POD5 & POD6*. Utrecht: Utrecht State University, Institute of Sonology (vid. págs. 25, 33).

— (1977). The POD system of interactive composition programs. *Computer Music Journal*, 1(3), págs. 30-39 (vid. pág. 25).

Truax, B. (1984). *Acoustic communication*. Vol. 1. New Jersey: Ablex Publishing Corporation. (vid. pág. 25).

Tzimeas, D. y Mangina, E. E. (2007). A GA Tool for Computer Assisted Music Composition. *Proceedings of 2007 International Computer Music Conference*. Copenhagen, Denmark (vid. pág. 56).

Urbano, J. (2014). MelodyShape at MIREX 2014 Symbolic Melodic Similarity. *Proceedings of the 10th Music Information Retrieval Evaluation eXchange*. Taipei, Taiwan (vid. pág. 5).

Van Der Merwe, A. y Schulze, W. (2011). Music generation with Markov models. *IEEE MultiMedia*, 18(3), págs. 78-85 (vid. pág. 60).

Vercoe, B. (1986). *Csound: A manual for the audio processing system and supporting programs with tutorials*. Cambridge, Massachusetts: The MIT Press (vid. pág. 37).

Vico, F. J., Sanchez, C y Albarracín, D (2011). MELOMICS: Aportaciones de la informática y la biología a la musicoterapia receptiva. *Actas del III Congreso Nacional de Musicoterapia*. Vol. 3. Cádiz, España, págs. 83-93 (vid. pág. 3).

Von Helmholtz, H. (1870). *Die Lehre von den Tonempfindungen als physiologische Grundlage für die Theorie der Musik*. Wendhausen: F. Vieweg und sohn (vid. pág. 186).

Voss, R. F. y Clarke, J. (1978). 1/f noise in music: Music from 1/f noise. *The Journal of the Acoustical Society of America*, 63(1), págs. 258-263 (vid. pág. 36).

Wallis, I., Ingalls, T. y Campana, E. (2008). Computer-generating emotional music: The design of an affective music algorithm. *Proceedings of the Second International Conference on Digital Audio Effects*. Espoo, Finland (vid. pág. 65).

Wardhaugh, B. (2017). *Music, Experiment and Mathematics in England*. New York: Routledge (vid. pág. 185).

Weinberg, G., Godfrey, M. y Rae, A. (2007). A real-time genetic algorithm in human-robot musical improvisation. *International Symposium on Computer*

Music Modeling and Retrieval. Copenhagen, Denmark, págs. 351-359 (vid. pág. 55).

Wenker, J. (1979). A computer-aided analysis of Anglo-Canadian folktunes. *ACM SIGLASH Newsletter*, 12(4), págs. 9-10 (vid. pág. 4).

Whittaker, R. H. (1952). A study of summer foliage insect communities in the Great Smoky Mountains. *Ecological monographs*, 22(1), págs. 1-44 (vid. pág. 82).

Williams, W. y Lambert, J. t. (1966). Multivariate methods in plant ecology: similarity analyses and information-analysis. *The Journal of Ecology*, págs. 427-445 (vid. pág. 107).

Winsor, P. (1991). PAT-PROC: An Interactive, Pattern-Process, Algorithmic Composition Program. *Proceedings of the 1991 International Computer Music Conference*. Montreal, Canada, págs. 114-114 (vid. págs. 43, 44).

Wolfram, S. (2002). *A new kind of science*. Urbana-Champain, Illionis: Wolfram media (vid. pág. 54).

Wood, A. (2008). *The Physics of music*. London: Davies Press (vid. pág. 217).

Xenakis, I. (1971). *Formalized music: thought and mathematics in composition*. New York: Pendragon Press (vid. págs. 2, 23, 33).

— (2009). *Música de la Arquitectura*. Madrid: Ediciones Akal (vid. pág. 22).

Yeung, K. Y., Medvedovic, M. y Bumgarner, R. E. (2003). Clustering gene-expression data with repeated measurement. *Genome biology*, 4(5), págs. 34-37 (vid. pág. 72).

Yi, S. y Lazzarini, V. (2012). Csound for android. *Proceedings of the 2012 Linux Audio Conference*. Stanford, California (vid. pág. 37).

Yu, C. y Wong, R. C.-W. (2017). *A Melody Composer for Both Tonal and Non-Tonal Languages*. Ann Arbor, Michigan: Michigan Publishing, University of Michigan Library (vid. pág. 67).

Apéndices

Apéndice A

Implementación en Mercury

A.1 Mercury.Calculations.Algorithms

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;
using Mercury.Calculations.Functions.Neighborhood;
using Mercury.Calculations.Functions.Distance;

namespace Mercury.Calculations.Algorithms
{

    public class FuzzyCMeans
    {

        private double _m = 2; //Fuzzy coefficient
        public double M { get { return _m; } set { _m = value; } }

        private NeighborhoodBase _sortingFunction = null;
        public NeighborhoodBase SortingFunction { get { return _sortingFunction
            ; } set { _sortingFunction = value; } }
```

```
private double _stopCriterion = 0.1; // StopCriterion
public double StopCriterion { get { return _stopCriterion; } set {
    _stopCriterion = value; } }

public FuzzyCMeans()
{
}
public FuzzyCMeans(double fuzzyCoefficient)
{
   M = fuzzyCoefficient;
}
public FuzzyCMeans(double fuzzyCoefficient, NeighborhoodBase
    sortingFunction)
{
   SortingFunction = sortingFunction;
   M = fuzzyCoefficient;
}
public FuzzyCMeans(double fuzzyCoefficient, NeighborhoodBase
    sortingFunction, double stopCriterion)
{
   SortingFunction = sortingFunction;
   M = fuzzyCoefficient;
   StopCriterion = stopCriterion;
}

public List<double[,]> Calculate(double[,] points, double[,] centroids,
     int n, int c, int dimensions, ref double[,] U, NeighborhoodBase
    sortingFunction, DistanceBase distanceFunction)
{
   //Declarations
   int iterations = 0;
   double[,] U_buf = null; //Matrix of fuzzy coeficients (buffer)
   double[,] U_s = null; //Matrix of fuzzy coeficients SORTED
   double[,] S = null; //Matrix of sorting function
   List<double[,]> steps = new List<double[,]>(); //theme B steps of
       the algorithm's process

   //1º Initialize the matriz os fuzzy coeficients U and U_buf
   U = null; //Matrix of fuzzy coeficients
   U = Calculate_U(points, centroids, M, n, c, dimensions,
       distanceFunction);//U = Random initialization ??? Calculate_U(
       points, centroids);
   InitializeMatrix(ref U_buf, n, c);
   InitializeSortingMatrix(ref S, n, c, sortingFunction);
```

```
    while (ContinueIterating(U, U_buf, StopCriterion, n, c) && (
        iterations <= 150)) //4º Compare U with previous U_buff
    //while (ContinueIterating(U, U_buf, StopCriterion, n, c)) //4º
        Compare U with previous U_buff
    {
        U_buf = U;

        //2º Update centroids
        centroids = Update_Centroids(U, points, centroids, M, n, c,
            dimensions);

        //3º Recalculate U
        U = Calculate_U(points, centroids, M, n, c, dimensions,
            distanceFunction);

        //4º Modify the matriz U in case user selected a sortingFunction
             **** ONLY IF A SORTING TYPE IS SELECTED *****
        if (sortingFunction != null)
        {
            U_s = Calculate_U_s(U, S, n, c);
            U = U_s; // Assign the new normalizated fuzzy coefficients
                matrix
        }

        //Storage the centroids
        steps.Add(centroids);

        //Add one to iterations
        iterations = iterations + 1;
    }
    return steps;
}

public List<double[,]> CalculateAddPoints(double[,] points, double[,]
    centroids, int n, int c, int dimensions, ref double[,] U,
    NeighborhoodBase sortingFunction, DistanceBase distanceFunction,
    bool addPoints)
{
    //Declarations
    int iterations = 0;
    double[,] U_buf = null; //Matrix of fuzzy coeficients (buffer)
    double[,] U_s = null; //Matrix of fuzzy coeficients SORTED
    double[,] S = null; //Matrix of sorting function
    List<double[,]> steps = new List<double[,]>(); //theme B steps of
        the algorithm's process
```

```
try
{

    while (c <= n)
    {

        //1º Initialize the matriz os fuzzy coeficients U and U_buf
        U = null; //Matrix of fuzzy coeficients
        U_buf = null;
        S = null;
        U = Calculate_U(points, centroids, M, n, c, dimensions,
            distanceFunction);//U = Random initialization ???
            Calculate_U(points, centroids);
        InitializeMatrix(ref U_buf, n, c);
        if (c == n)
        {
            InitializeSortingMatrix(ref S, n, c, new
                DiscreteNeighborhood());
        }
        else
        {
            InitializeSortingMatrix(ref S, n, c, sortingFunction);
        }
        iterations = 0;
        while (ContinueIterating(U, U_buf, StopCriterion, n, c) && (
            iterations <= 300)) //4º Compare U with previous U_buff
        //while (ContinueIterating(U, U_buf, StopCriterion, n, c))
            //4º Compare U with previous U_buff
        {
            U_buf = U;

            //2º Update centroids
            centroids = Update_Centroids(U, points, centroids, M, n,
                 c, dimensions);


            //3º Recalculate U
            U = Calculate_U(points, centroids, M, n, c, dimensions,
                distanceFunction);

            //4º Modify the matriz U in case user selected a
                sortingFunction **** ONLY IF A SORTING TYPE IS
                SELECTED *****
            if (sortingFunction != null)
            {
               U_s = Calculate_U_s(U, S, n, c);
```

```
            U = U_s; // Assign the new normalizated fuzzy
                coefficients matrix
        }

        //Storage the centroids
        if (addPoints) {steps.Add(centroids); }

        //Add one to iterations
        iterations = iterations + 1;
    }
    //Storage only the last centroids
    if (!addPoints) { steps.Add(centroids); }

    //5º Add one extra centroid
    //Choose where to add the extra pont. Between the two points
         most far away
    int index = 0;
    double dbuf = 0;
    double dbuf2 = 0;
    double[] p1 = null;
    double[] p2 = null;
    double[] p3 = null;
    Assign(centroids, 0, ref p1, dimensions);
    Assign(centroids, 1, ref p2, dimensions);
    dbuf = distanceFunction.Calculate(p1, p2);
    for (int i = 0; i <= centroids.GetLength(0) - 2; i++)
    {
        Assign(centroids, i, ref p1, dimensions);
        Assign(centroids, i + 1, ref p2, dimensions);
        dbuf2 = distanceFunction.Calculate(p1, p2);
        if (dbuf2 > dbuf)
        {
            dbuf = dbuf2;
            index = i;
        }
    }



    //the new point is the average of the two selected points
    Assign(centroids, index, ref p1, dimensions);
    Assign(centroids, index + 1, ref p2, dimensions);
    p3 = new double[dimensions];
    for (int i = 0; i <= dimensions - 1; i++)
    {
        p3[i] = (p2[i] + p1[i]) / 2;
```

```
            }
            double[,] newCentroids = new double[centroids.GetLength(0) +
                 1, dimensions];
            for (int i = 0; i <= index - 1; i++)
            {
                for (int l = 0; l <= dimensions - 1; l++)
                {
                    newCentroids[i, l] = centroids[i, l];
                }
            }
            for (int l = 0; l <= dimensions - 1; l++)
            {
                newCentroids[index, l] = p3[l];
            }
            for (int i = index; i <= centroids.GetLength(0) - 1; i++)
            {
                for (int l = 0; l <= dimensions - 1; l++)
                {
                    newCentroids[i + 1, l] = centroids[i, l];
                }
            }
            centroids = newCentroids;
            c = centroids.GetLength(0);
        }


    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }

    return steps;
}


public List<double[,]> CalculateRemovePoints(double[,] points, double
     [,] centroids, int n, int c, int dimensions, ref double[,] U,
     NeighborhoodBase sortingFunction, DistanceBase distanceFunction,
     bool addPoints)
{
    //Declarations
    int iterations = 0;
    double[,] U_buf = null; //Matrix of fuzzy coeficients (buffer)
    double[,] U_s = null; //Matrix of fuzzy coeficients SORTED
    double[,] S = null; //Matrix of sorting function
```

```
List<double[,]> steps = new List<double[,]>(); //theme B steps of
    the algorithm's process

try
{

    while (c >= n)
    {

        //1º Initialize the matriz os fuzzy coeficients U and U_buf
        U = null; //Matrix of fuzzy coeficients
        U_buf = null;
        S = null;
        U = Calculate_U(points, centroids, M, n, c, dimensions,
            distanceFunction);//U = Random initialization ???
            Calculate_U(points, centroids);
        InitializeMatrix(ref U_buf, n, c);
        if (c == n)
        {
            InitializeSortingMatrix(ref S, n, c, new
                DiscreteNeighborhood());
        }
        else
        {
            InitializeSortingMatrix(ref S, n, c, sortingFunction);
        }
        iterations = 0;
        while (ContinueIterating(U, U_buf, StopCriterion, n, c) && (
            iterations <= 300)) //4º Compare U with previous U_buff
        //while (ContinueIterating(U, U_buf, StopCriterion, n, c))
            //4º Compare U with previous U_buff
        {
            U_buf = U;

            //2º Update centroids
            centroids = Update_Centroids(U, points, centroids, M, n,
                 c, dimensions);


            //3º Recalculate U
            U = Calculate_U(points, centroids, M, n, c, dimensions,
                distanceFunction);

            //4º Modify the matriz U in case user selected a
                sortingFunction **** ONLY IF A SORTING TYPE IS
                SELECTED *****
```

```
    if (sortingFunction != null)
    {
        U_s = Calculate_U_s(U, S, n, c);
        U = U_s; // Assign the new normalizated fuzzy
            coefficients matrix
    }

    //Storage the centroids
    if (addPoints) { steps.Add(centroids); }

    //Add one to iterations
    iterations = iterations + 1;
}
//Storage only the last centroids
if (!addPoints) { steps.Add(centroids); }

//5º Add one extra centroid
//Choose where to add the extra pont. Between the two points
     most far away
int index = 0;
int j = 0;
double dbuf = 0;
double dbuf2 = 0;
double[] p1 = null;
double[] p2 = null;
double[,] newCentroids = null;
Assign(centroids, 0, ref p1, dimensions);
Assign(centroids, 1, ref p2, dimensions);
dbuf = distanceFunction.Calculate(p1, p2);
for (int i = 0; i <= centroids.GetLength(0) - 2; i++)
{
    Assign(centroids, i, ref p1, dimensions);
    Assign(centroids, i + 1, ref p2, dimensions);
    dbuf2 = distanceFunction.Calculate(p1, p2);
    if (dbuf2 > dbuf)
    {
        dbuf = dbuf2;
        index = i;
    }
}

newCentroids = new double[centroids.GetLength(0) - 1,
    dimensions];
for (int i = 0; i <= centroids.GetLength(0) - 1; i++)
{
    if (i != index)
```

```
                {
                    for (int l = 0; l <= dimensions - 1; l++)
                    {
                        newCentroids[j, l] = centroids[i, l];
                    }
                    j = j + 1;
                }
            }
            centroids = newCentroids;
            c = centroids.GetLength(0);
        }


    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }

    return steps;
}




private double[,] Calculate_U(double[,] points, double[,] centroids,
    double m, int n, int c, int dimensions, DistanceBase
    distanceFunction)
{
    double[,] U = new double[n, c];
    double buf = 0;
    double buf1 = 0;
    double buf2 = 0;
    double[] Xi = new double[dimensions];
    double[] Cj = new double[dimensions];
    double[] Ck = new double[dimensions];

    for (int i = 0; i <= n - 1; i++) //all the points
    {
        for (int j = 0; j <= c - 1; j++) //all the centroids
        {
            //Calculate each coefficient
            buf=0;
            Assign(points, i, ref Xi, dimensions);
            Assign(centroids, j, ref Cj, dimensions);
```

```
            buf1 = distanceFunction.Calculate(Xi, Cj);
            for (int k = 0; k <= c - 1; k++)
            {
                //Assign
                Assign(centroids, k, ref Ck, dimensions);
                //Formula for calculating the fuzzy coefficients
                buf2 = distanceFunction.Calculate(Xi, Ck);
                buf = buf + Math.Pow(buf1 / buf2, (2 / (m - 1)));
            }

            if (buf != 0)
            {
                U[i, j] = 1 / buf;
            }
            else
            {
                U[i, j] = double.MaxValue;
            };
        }
    }

    return U;
}


private double[,] Update_Centroids(double[,] U, double[,] points,
    double[,] centroids, double m, int n, int c, int dimensions)
{
    double buf1 = 0;
    double buf2 = 0;
    double buf3 = 0;
    double[,] newCentroids = new double[c, dimensions];

    for (int j = 0; j <= c - 1; j++) //recorremos todos los centroides
    {

        //Calculate the inferior term of the expression
        buf2 = 0;
        buf3 = 0;
        for (int i = 0; i <= n - 1; i++)
        {
            buf2 = buf2 + Math.Pow(U[i, j], m);
            buf3 = buf3 + U[i, j];
        }
```

```
        for (int d = 0; d <= dimensions - 1; d++) //for all the
            dimensions
        {
            //calculate the upper term of the expression
            buf1 = 0;
            for (int i = 0; i <= n - 1; i++)
            {
                buf1 = buf1 + Math.Pow(U[i, j], m) * points[i, d];
            }

            //calculate the d-component of the centroid j
            if (buf2 == 0)
            {
                //Console.WriteLine("Division by zero on cluster " + j.
                    ToString());
            }
            else
            {
                newCentroids[j, d] = buf1 / buf2;
            }
            if (buf2 == 0 & buf1==0)
            {
                //Console.WriteLine("Zero divided by zero on cluster " +
                     j.ToString());
            }

        }
    }

    return newCentroids;
}



private bool ContinueIterating(double[,] U, double[,] U_buf, double
    epsilon, int n, int c)
{
    double max = 0;
    double buf = 0;

    // calculate the max difference for the whole matrix
    for (int i = 0; i <= n - 1; i++)
    {
        for (int j = 0; j <= c - 1; j++)
        {
            buf = Math.Abs(U[i, j] - U_buf[i, j]);
```

```
            if (buf > max) { max = buf; }
        }
    }
    return (max >= epsilon);
}

private void Assign(double[,] matrix, int i, ref double[] vector, int
    dimensions)
{
    vector = new double[dimensions];
    for (int j = 0; j <= dimensions - 1; j++)
    {
        vector[j] = matrix[i,j];
    }
}

private void InitializeMatrix(ref double[,] matrix, int n, int c)
{
    if (matrix == null) { matrix = new double[n, c]; };
    for (int i = 0; i <= n - 1; i++)
    {
        for (int j = 0; j <= c - 1; j++)
        {
            matrix[i, j] = 0;
        }
    }
}

private void InitializeSortingMatrix(ref double[,] S, int n, int c,
    NeighborhoodBase sortingFunction)
{
    if (S == null) { S = new double[n, c]; };
    // calculate each discrete value of sorting function for values i,j
    if (sortingFunction != null)
    {
        sortingFunction.M = c;
        sortingFunction.N = n;
        for (int i = 0; i <= n - 1; i++)
        {
            for (int j = 0; j <= c - 1; j++)
            {
                sortingFunction.j = j;
                S[i, j] = sortingFunction.Value(i);
            }
        }
    }
```

```
        }


        private double[,] Calculate_U_s(double[,] U, double[,] S, int n, int c)
        {
            double[,] U_s = new double[n, c]; //Sorted U matrix
            double[] V = new double[n]; //Vector of normalization
            double buf = 0;

            //Calculate the components of the normalization vector
            for (int i = 0; i <= n - 1; i++) //all the points
            {

                buf = 0;
                for (int j = 0; j <= c - 1; j++) //all the centroids
                {
                    buf = buf + U[i, j] * S[i, j];
                }
                V[i] = buf;
            }

            //Calculate the normalizated Sorted Fuzzy Coefficients Matrix
            for (int i = 0; i <= n - 1; i++) //all the points
            {
                for (int j = 0; j <= c - 1; j++) //all the centroids
                {
                    if (V[i] != 0)
                    {
                        U_s[i, j] = U[i, j] * S[i, j] / V[i];
                    }
                    else
                    {
                        U_s[i, j] = 0;
                    }
                }
            }

            return U_s;
        }
    }
}
```

A.2 Mercury.Calculations.Analysis

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Analysis
{
    public static class Derivative
    {
        public static double[] Calculate(double[] X, double[] Y)
        {
            if (X.Length != Y.Length) { throw new Exception("Inconsistent
                arrays to calculate the differenciation");}
            if (X.Length < 2) { throw new Exception("Array too small for
                differenciate");}
            double[] dYdX = new double[X.Length];

            for (int i = 0; i <= X.Length - 1; i++)
            {
                if (i != X.Length - 1)
                {
                    dYdX[i] = (Y[i + 1] - Y[i]) / (X[i + 1] - X[i]);
                }
                else
                {
                    //Last point
                    dYdX[i] = dYdX[i - 1];
                }
            }

            return dYdX;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Analysis
{
    public static class MaxMin
    {

        public static double GetMinimumValue(double[] data)
        {
            double buf = data[0];
            for (int i = 0; i <= data.Length - 1; i++)
            {
                if (data[i] < buf)
                {
                    buf = data[i];
                }
            }
            return buf;
        }

        public static double GetMinimumValue(double[,] data, int dimension)
        {
            double buf = data[0, dimension];
            for (int i = 0; i <= data.GetLength(0) - 1; i++)
            {
                if (data[i, dimension] < buf)
                {
                    buf = data[i, dimension];
                }
            }
            return buf;
        }

        public static double GetMaximumValue(double[] data)
        {
            double buf = data[0];
            for (int i = 0; i<= data.Length-1; i++)
            {
                if (data[i] > buf)
                {
                    buf = data[i];
                }
```

```
    }
    return buf;
}

public static double GetMaximumValue(double[,] data, int dimension)
{
    double buf = data[0, dimension];
    for (int i = 0; i <= data.GetLength(0) - 1; i++)
    {
        if (data[i, dimension] > buf)
        {
            buf = data[i, dimension];
        }
    }
    return buf;
}


public static SortedList<double, PointD> GetMaxMinPoints(double[] X,
     double[] Y, double[] dYdX)
{
    SortedList<double, PointD> minMax = new SortedList<double, PointD
        >();
    if (X.Length != dYdX.Length) { throw new Exception("Inconsistent
        arrays to calculate the differenciation"); }
    if (X.Length != Y.Length) { throw new Exception("Inconsistent
        arrays to calculate the differenciation"); }
    if (X.Length < 2) { throw new Exception("Array too small for
        differenciate"); }

    for (int i = 0; i <= dYdX.Length - 1; i++)
    {
        if (dYdX[i] == 0) {minMax.Add(Y[i], new PointD(X[i], Y[i]));}
        else
        {
            if (i != dYdX.Length - 1)
            {
                if (Math.Sign(dYdX[i]) != Math.Sign(dYdX[i+1])) { minMax
                    .Add(Y[i], new PointD(X[i], Y[i]));}
            }
            else
            {
                if (dYdX[i] == 0) {minMax.Add(Y[i], new PointD(X[i], Y[i
                    ]));}
            }
        }
```

```
            }

            return minMax;
        }

        public static SortedList<double, PointD> GetMaxPoints(double[] X,
            double[] Y, double[] dYdX)
        {
            SortedList<double, PointD> Max = new SortedList<double, PointD>();
            if (X.Length != dYdX.Length) { throw new Exception("Inconsistent
                arrays to calculate the differenciation"); }
            if (X.Length < 2) { throw new Exception("Array too small for
                differenciate"); }

            for (int i = 0; i <= dYdX.Length - 1; i++)
            {
                //Si la derivada cambia de positiva a negativa, es un máximo
                if (i != dYdX.Length - 1)
                {
                    if ((Math.Sign(dYdX[i]) == 1) && (Math.Sign(dYdX[i + 1]) ==
                        -1))
                    {
                        Max.Add(Y[i+1], new PointD(X[i+1], Y[i+1]));
                    }
                }
                else
                {
                    if ((Math.Sign(dYdX[i - 1]) == 1) && (Math.Sign(dYdX[i]) ==
                        -1)) { Max.Add(Y[i], new PointD(X[i], Y[i])); }
                }
            }
            return Max;
        }
    }
}
```

A.3 Mercury.Calculations.Difference

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;
using Mercury.Calculations.Algorithms;
using Mercury.Calculations.Functions.Neighborhood;
using Mercury.Calculations.Functions.Distance;

namespace Mercury.Calculations.Difference
{
    public class MeanDifference
    {

        private double _m = 2; //Fuzzy coefficient
        public double M { get { return _m; } set { _m = value; } }

        private NeighborhoodBase _sortingFunction = null;
        public NeighborhoodBase SortingFunction { get { return _sortingFunction
            ; } set { _sortingFunction = value; } }

        private DistanceBase _distanceFunction = null;
        public DistanceBase DistanceFunction { get { return _distanceFunction;
            } set { _distanceFunction = value; } }

        private double _stopCriterion = 0.1; // StopCriterion
        public double StopCriterion { get { return _stopCriterion; } set {
            _stopCriterion = value; } }

        public MeanDifference() { }

        public MeanDifference(NeighborhoodBase sortingFunction, DistanceBase
            distanceFunction)
        {
            SortingFunction = sortingFunction;
            DistanceFunction = distanceFunction;
        }

        public MeanDifference(double fuzzyCoefficient, NeighborhoodBase
            sortingFunction, DistanceBase distanceFunction)
        {
            SortingFunction = sortingFunction;
            DistanceFunction = distanceFunction;
```

```
    M = fuzzyCoefficient;
}
public MeanDifference(double fuzzyCoefficient, double stopCriterion,
    NeighborhoodBase sortingFunction, DistanceBase distanceFunction)
{
    SortingFunction = sortingFunction;
    DistanceFunction = distanceFunction;
    M = fuzzyCoefficient;
    StopCriterion = stopCriterion;
}

public double Calculate_FuzzyCMeansDifference(double[,] points, double
    [,] centroids, ref List<double[,]> steps, bool addExtraPoints,
    bool showAllStates)
{
    //Declarations
    int n = 0; //Number of Points
    int c = 0; //number of Centroids
    int d = 0; //Number of Dimensions

    //Previous checkings
    if ((points == null) || (points.GetLength(0) == 0) || (points.
        GetLength(1) == 0)) { throw new Exception("Invalid point array
        "); }
    if ((centroids == null) || (centroids.GetLength(0) == 0) || (
        centroids.GetLength(1) == 0)) { throw new Exception("Invalid
        centroid array"); }
    if (points.GetLength(1) != centroids.GetLength(1) ) { throw new
        Exception("Points and centroids don't have the same number of
        dimensions"); }

    //Init the list where storage the several changes of the centroids
    steps = new List<double[,]>();
    steps.Add(centroids); //Storage the initial centroids

    //Establish the number of points, centroids and dimensions
    n = points.GetLength(0);
    c = centroids.GetLength(0);
    d = points.GetLength(1);

    //Fuzzy C-Means Difference
    return CalculateFuzzyCMeansDifference(points, centroids, ref steps,
         n, c, d, SortingFunction, DistanceFunction, addExtraPoints,
        showAllStates);

}
```

```
public double Calculate_MeanDifference(double[,] points, double[,]
    centroids, ref List<double[,]> steps)
{
    //Declarations
    int n = 0; //Number of Points
    int c = 0; //number of Centroids
    int d = 0; //Number of Dimensions

    //Previous checkings
    if ((points == null) || (points.GetLength(0) == 0) || (points.
        GetLength(1) == 0)) { throw new Exception("Invalid point array
        "); }
    if ((centroids == null) || (centroids.GetLength(0) == 0) || (
        centroids.GetLength(1) == 0)) { throw new Exception("Invalid
        centroid array"); }
    if (points.GetLength(1) != centroids.GetLength(1)) { throw new
        Exception("Points and centroids don't have the same number of
        dimensions"); }

    //Init the list where storage the several changes of the centroids
    steps = new List<double[,]>();
    steps.Add(centroids); //Storage the initial centroids

    //Establish the number of points, centroids and dimensions
    n = points.GetLength(0);
    c = centroids.GetLength(0);
    d = points.GetLength(1);

    //To fuzzy or Not to Fuzzy
    if (n == c)
    {
        //Not fuzzy. Case of equal number of points and centroids
        return CalculateMeanDifference(points, centroids, ref steps, n,
            c, d, DistanceFunction);
    }
    else
    {
        //Error
        throw new Exception("The two melodies doesn't have the same
            number of notes.");
    };

}
```

```
private double CalculateFuzzyCMeansDifference(double[,] points, double
    [,] centroids, ref List<double[,]> steps, int n, int c, int
    dimensions, NeighborhoodBase sortingFunction, DistanceBase
    distanceFunction, bool addRemovePoints, bool showAllStates)
{
    //Declarations
    double distance = 0; //Final mean distance
    double[,] U = null; //Matrix of fuzzy coeficients

    FuzzyCMeans FCM = new FuzzyCMeans(this.M, sortingFunction, this.
        StopCriterion);
    if (addRemovePoints)
    {
        if (c < n)
        {
            steps.AddRange(FCM.CalculateAddPoints(points, centroids, n,
                c, dimensions, ref U, sortingFunction, distanceFunction,
                 showAllStates));
        }
        else
        {
            steps.AddRange(FCM.CalculateRemovePoints(points, centroids,
                n, c, dimensions, ref U, sortingFunction,
                distanceFunction, showAllStates));
        }
        distance = 0;
    }
    else
    {
        steps.AddRange(FCM.Calculate(points, centroids, n, c, dimensions
            , ref U, sortingFunction, distanceFunction));
        distance = Dt_FuzzyCMeansDifference(U, points, steps[0], n, c,
            dimensions, distanceFunction);
    }

    return distance;
}

private double CalculateMeanDifference(double[,] points, double[,]
    centroids, ref List<double[,]> steps, int n, int c, int dimensions
    , DistanceBase distanceFunction)
{
    //Case of equal number of points
    steps.Add(centroids);
    return Dt_MeanDifference(points, centroids, n, c, dimensions,
        distanceFunction);
```

```
}

public double Dt_FuzzyCMeansDifference(double[,] U, double[,] points,
    double[,] centroids, int n, int c, int dimensions, DistanceBase
    distanceFunction)
{
    double distance = 0;
    double[] Xi = null;
    double[] Cj = null;
    try
    {
        for (int j = 0; j <= c - 1; j++)
        {
            for (int i = 0; i <= n - 1; i++)
            {
                Assign(centroids, j, ref Cj, dimensions);
                Assign(points, i, ref Xi, dimensions);
                distance = distance + U[i, j] * distanceFunction.
                    Calculate(Xi, Cj);
            }
        }

    }
    catch(Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }

    return distance /c /n;
}

private double Dt_MeanDifference( double[,] points, double[,] centroids
    , int n, int c, int dimensions, DistanceBase distanceFunction)
{
    double distance = 0;
    double[] Xi = null;
    double[] Cj = null;
    if (n == c)
    {

        for (int i = 0; i <= n - 1; i++)
        {
            Assign(centroids, i, ref Cj, dimensions);
            Assign(points, i, ref Xi, dimensions);
            distance = distance + distanceFunction.Calculate(Xi, Cj);
        }
```

```
            }
            else
            {
                throw new Exception("Not able to calculate distance with equal
                    number of points");
            }

            return distance /c /n;
        }

        private void Assign(double[,] matrix, int i, ref double[] vector, int
            dimensions)
        {
            vector = new double[dimensions];
            for (int j = 0; j <= dimensions - 1; j++)
            {
                vector[j] = matrix[i, j];
            }
        }
    }
}
```

A.4 Mercury.Calculations.Duration

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Duration
{
    public static class DurationCalculator
    {
        public static double TotalDuration(int dots, int duration, int
            normalNotes, int actualNotes)
        {
            if (duration == 0)
            {
                return 0;
            }
            else
            {
                if (normalNotes==0) {normalNotes=1;}
                if (actualNotes==0) {actualNotes = 1;}
                double totalDuration = (Math.Pow(2, dots + 1) - 1 )/ Math.Pow(2,
                     dots) / duration * normalNotes / actualNotes;
                return totalDuration;
            }
        }

        public static double TotalDurationMS(double bpm, int dots, int duration
            , int normalNotes, int actualNotes)
        {
            if (duration == 0)
            {
                return 0;
            }
            else
            {
                double totalDuration = TotalDuration(dots, duration, normalNotes
                    , actualNotes);
                return 60*4*1000*totalDuration/bpm;
            }
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;


namespace Mercury.Calculations.Duration
{

    public class RhythmicFigure
        {
            public byte dots;
            public byte duration;
            public byte actualNotes;
            public byte normalNotes;
            public RhythmicFigure(byte Dots, byte Duration, byte ActualNotes,
                byte NormalNotes)
            {
                dots = Dots;
                duration = Duration;
                actualNotes = ActualNotes;
                normalNotes = NormalNotes;
            }
            public override string ToString()
            {
                return duration.ToString() + " " + dots.ToString() + " " +
                    actualNotes.ToString() + " " + normalNotes.ToString();
            }
            public RhythmicFigure Clone()
            {
                RhythmicFigure newFigure = new RhythmicFigure(dots, duration,
                    actualNotes, normalNotes);
                return newFigure;
            }
        }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;


namespace Mercury.Calculations.Duration
{

      public static class RhythmicFigureCalculator
      {
          public static SortedDictionary<double, List<RhythmicFigure>>
               CalculateAllPossibleDurations(byte[] DurationsRange, byte[]
               ActualNotesRange, byte[] NormalNotesRange, byte MaxDots, byte
               MaxTiedNotes)
          {
              //Faltan comprobaciones de si son vacions los arrys de bytes!!!!


              //byte[] durationsRange = new byte[] { 1, 2, 4, 8, 16, 32, 64,
                  128 };
              //byte[] actualNotesRange = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8,
                   9, 10, 11, 12 };
              //byte[] normalNotesRange = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8,
                   9, 10, 11, 12 };

              byte[] dotsRange = new byte[MaxDots + 1];
              for (byte i = 0; i <= MaxDots; i++) { dotsRange[i] = i; }

              SortedDictionary<double, List<RhythmicFigure>> durations = new
                  SortedDictionary<double, List<RhythmicFigure>>();
              List<RhythmicFigure> newList = new List<RhythmicFigure>();
              CalculateWithRecursivity(ref durations, newList, 0,
                  DurationsRange, ActualNotesRange, NormalNotesRange,
                  dotsRange, MaxTiedNotes, 0);

              return durations;
          }

          private static void CalculateWithRecursivity(ref SortedDictionary<
               double, List<RhythmicFigure>> durations, List<RhythmicFigure>
               PreviousFigures, double PreviousDuration, byte[] DurationsRange,
                byte[] ActualNotesRange, byte[] NormalNotesRange, byte[]
               DotsRange, byte MaxTiedNotes, byte currentTiedNote)
```

```
{
    double newDuration = 0;

    for (int duration = 0; duration <= DurationsRange.Length - 1;
        duration++)
    {
        for (int dots = 0; dots <= DotsRange.Length - 1; dots++)
        {
            for (int actualNotes = 0; actualNotes <=
                ActualNotesRange.Length - 1; actualNotes++)
            {
                for (int normalNotes = 0; normalNotes <=
                    NormalNotesRange.Length - 1; normalNotes++)
                {
                    if (((ActualNotesRange[actualNotes] ==
                        NormalNotesRange[normalNotes]) && (
                        ActualNotesRange[actualNotes] == 1)) || (
                        ActualNotesRange[actualNotes] !=
                        NormalNotesRange[normalNotes]))
                    {
                        newDuration = Calculations.Duration.
                            DurationCalculator.TotalDuration((int)
                            DotsRange[dots],
                            (int)DurationsRange[duration],
                            (int)NormalNotesRange[normalNotes],
                            (int)ActualNotesRange[actualNotes]);
                        if (!durations.ContainsKey(newDuration +
                            PreviousDuration))
                        {
                            //Create the new figure
                            RhythmicFigure newFigure = new
                                RhythmicFigure(DotsRange[dots],
                                DurationsRange[duration],
                                ActualNotesRange[actualNotes],
                                NormalNotesRange[normalNotes]);

                            //Create a new list of figures and clone
                                the previous list
                            List<RhythmicFigure> newList = new List<
                                RhythmicFigure>();
                            foreach (RhythmicFigure figure in
                                PreviousFigures)
                            {
                                newList.Add(figure.Clone());
                            }
```

```
                                    //Add new figure to the list
                                    newList.Add(newFigure);

                                    //Add this list with the duration to the
                                        durations list
                                    durations.Add(newDuration +
                                        PreviousDuration, newList);

                                    //Continue iterating with recursivity
                                    if (currentTiedNote < MaxTiedNotes)
                                    {
                                        byte nextTiedNote = (byte)(
                                            currentTiedNote + 1);
                                        CalculateWithRecursivity(ref durations
                                            , newList, (newDuration +
                                            PreviousDuration), DurationsRange,
                                             ActualNotesRange,
                                            NormalNotesRange, DotsRange,
                                            MaxTiedNotes, nextTiedNote);
                                    }//else stop this branche of recursivity

                                }
                            }
                        }
                    }
                }
            }

        public static List<RhythmicFigure> GetCloserFigure(double
            durationCoefficient, SortedDictionary<double, List<
            RhythmicFigure>> Figures)
        {
            int i = 0;
            double[] keys = Figures.Keys.ToArray();
            double buf = Math.Abs(durationCoefficient - keys[0]);
            while ((i<=keys.Length-1)&&(buf >= Math.Abs(durationCoefficient
                - keys[i])))
            {
                buf = Math.Abs(durationCoefficient - keys[i]);
                i = i + 1;
            }
            return Figures[keys[i-1]];
        }
    }
}
```

A.5 Mercury.Calculations.Functions.Distance

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Distance
{
    [Serializable]
    public class AssociationIndexDistance: DistanceBase
    {
        public AssociationIndexDistance(): base() { }
        public AssociationIndexDistance(string Name) : base(Name) { }
        public override double Calculate(double[] point1, double[] point2)
        {
            double buf = 0;
            double x = 0;
            double y = 0;
            double q = (double)point1.GetLength(0);

            x = 0;
            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                x = x + point1[i];
            }
            x = Math.Pow(x, 0.5);
            if (x == 0) { throw new DistanceException(); }

            y = 0;
            for (int i = 0; i <= point2.GetLength(0) - 1; i++)
            {
                y = y + point2[i];
            }
            y = Math.Pow(y, 0.5);
            if (y == 0) { throw new DistanceException(); }

            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                buf = buf + Math.Abs(point1[i] / x - point2[i] / y);
            }
            return Math.Max(0.5 * buf, this.Min);
        }
        public override DistanceBase Clone()
        {
```

```
            AssociationIndexDistance buf = new AssociationIndexDistance();
            buf.Name = this.Name;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Distance
{
    [Serializable]
    public class AverageDistance: DistanceBase
    {
        public AverageDistance(): base() { }
        public AverageDistance(string Name) : base(Name) { }
        public override double Calculate(double[] point1, double[] point2)
        {
            double buf = 0;
            double q = (double)point1.GetLength(0);
            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                buf = buf + Math.Pow(point1[i] - point2[i], 2);
            }
            return Math.Max(Math.Pow(buf / q, 0.5), this.Min);
        }
        public override DistanceBase Clone()
        {
            AverageDistance buf = new AverageDistance();
            buf.Name = this.Name;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Distance
{
    [Serializable]
    public class AverageManhattanDistance : DistanceBase
    {
        public AverageManhattanDistance() : base() { }
        public AverageManhattanDistance(string Name) : base(Name) { }
        public override double Calculate(double[] point1, double[] point2)
        {
            double buf = 0;
            double q = (double)point1.GetLength(0);
            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                buf = buf + Math.Abs(point1[i] - point2[i]);
            }
            return Math.Max(buf / q, this.Min);
        }
        public override DistanceBase Clone()
        {
            AverageManhattanDistance buf = new AverageManhattanDistance();
            buf.Name = this.Name;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Distance
{
    [Serializable]
    public class CanberraDistance: DistanceBase
    {
        public CanberraDistance(): base() { }
        public CanberraDistance(string Name) : base(Name) { }
        public override double Calculate(double[] point1, double[] point2)
        {
            double buf = 0;
            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                if ((Math.Abs(point1[i]) + Math.Abs(point2[i])) == 0)
                {
                    //throw new DistanceException();
                }
                else
                {
                    buf = buf + Math.Abs(point1[i] - point2[i]) / (Math.Abs(
                        point1[i]) + Math.Abs(point2[i]));
                }

            }
            return Math.Max(buf, this.Min);
        }
        public override DistanceBase Clone()
        {
            CanberraDistance buf = new CanberraDistance();
            buf.Name = this.Name;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Distance
{
    [Serializable]
    public class ChebyshevDistance: DistanceBase
    {
        public ChebyshevDistance(): base() { }
        public ChebyshevDistance(string Name) : base(Name) { }
        public override double Calculate(double[] point1, double[] point2)
        {
            double buf = 0;
            double max = Math.Abs(point1[0] - point2[0]);
            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                buf = Math.Abs(point1[i] - point2[i]);
                if (buf > max) { max = buf; }
            }
            return Math.Max(buf, this.Min);
        }
        public override DistanceBase Clone()
        {
            ChebyshevDistance buf = new ChebyshevDistance();
            buf.Name = this.Name;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Distance
{
    [Serializable]
    public class ChordDistance: DistanceBase
    {
        public ChordDistance(): base() { }
        public ChordDistance(string Name) : base(Name) { }
        public override double Calculate(double[] point1, double[] point2)
        {
            double buf = 0;
            double x = 0;
            double y = 0;
            double q = (double)point1.GetLength(0);

            x = 0;
            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                x = x + Math.Pow(point1[i], 2);
            }
            x = Math.Pow(x, 0.5);
            if (x == 0) { throw new DistanceException(); }

            y = 0;
            for (int i = 0; i <= point2.GetLength(0) - 1; i++)
            {
                y = y + Math.Pow(point2[i], 2);
            }
            y = Math.Pow(y, 0.5);
            if (y == 0) { throw new DistanceException(); }

            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                buf = buf + (point1[i] * point2[i]);
            }
            if (2.0 * buf / (x * y) <= 2)
            {
                return Math.Max(Math.Pow(2.0 - 2.0 * buf / (x * y), 0.5), this.
                    Min);
            }
            else
```

```
            {
                return Math.Max(0, this.Min);
            }
        }
        public override DistanceBase Clone()
        {
            ChordDistance buf = new ChordDistance();
            buf.Name = this.Name;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Distance
{
    [Serializable]
    public class DiscreteDistance: DistanceBase
    {
        public DiscreteDistance(): base() { }
        public DiscreteDistance(string Name) : base(Name) { }
        public override double Calculate(double[] point1, double[] point2)
        {
            //if x=y then d(x,y)=0, else d(x,y)=1
            double buf = 0;
            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                buf = buf + Math.Abs(point1[i] - point2[i]);
            }
            if (buf > 0) { buf = 1; }
            return Math.Max(buf, this.Min);
        }
        public override DistanceBase Clone()
        {
            DiscreteDistance buf = new DiscreteDistance();
            buf.Name = this.Name;
            return buf;
        }
    }
}
```

```
using System;
using System.Xml.Serialization;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Distance
{
    [XmlInclude(typeof(ChebyshevDistance))]
    [XmlInclude(typeof(DiscreteDistance))]
    [XmlInclude(typeof(EuclideanDistance))]
    [XmlInclude(typeof(ManhattanDistance))]
    [XmlInclude(typeof(MinkowskiDistance))]
    [XmlInclude(typeof(MinkowskiDistance_p5))]
    [XmlInclude(typeof(MinkowskiDistance_p15))]
    [XmlInclude(typeof(MinkowskiDistance_p4))]
    [Serializable]
    public abstract class DistanceBase
    {
        public string Name;
        public double Min = 0.0001;
        public DistanceBase() { }
        public DistanceBase(string Name) { this.Name = Name; }
        public abstract double Calculate(double[] point1, double[] point2);
        public override string ToString()
        {
            return this.Name;
        }
        public abstract DistanceBase Clone();
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Mercury.Domain.MusicalSymbol;
using Mercury.Calculations;
using Mercury.Domain.MusicalSymbol.Enumerations;
using Mercury.Calculations.Duration;

namespace Mercury.Calculations.Functions.Distance
{

    public class DistanceException : Exception { }

}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Distance
{
    [Serializable]
    public class DivergenceDistance: DistanceBase
    {
        public DivergenceDistance() : base() { }
        public DivergenceDistance(string Name) : base(Name) { }
        public override double Calculate(double[] point1, double[] point2)
        {
            double buf = 0;
            double q = (double)point1.GetLength(0);
            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                if ((point1[i] + point2[i]) != 0)
                {
                    buf = buf + Math.Pow((point1[i] - point2[i]) / (point1[i] +
                        point2[i]), 2);
                }
            }
            return Math.Max(Math.Pow(buf / q, 0.5), this.Min);
        }
        public override DistanceBase Clone()
        {
            DivergenceDistance buf = new DivergenceDistance();
            buf.Name = this.Name;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Distance
{
    [Serializable]
    public class EuclideanDistance: DistanceBase
    {
        public EuclideanDistance(): base() { }
        public EuclideanDistance(string Name) : base(Name) { }
        public override double Calculate(double[] point1, double[] point2)
        {
            double buf = 0;
            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                buf = buf + Math.Pow(point1[i] - point2[i], 2);
            }
            return Math.Max(Math.Pow(buf, 0.5), this.Min);
        }
        public override DistanceBase Clone()
        {
            EuclideanDistance buf = new EuclideanDistance();
            buf.Name = this.Name;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Distance
{
    [Serializable]
    public class GeodesicDistance: DistanceBase
    {
        public GeodesicDistance(): base() { }
        public GeodesicDistance(string Name) : base(Name) { }
        public override double Calculate(double[] point1, double[] point2)
        {
            double buf = 0;
            double x = 0;
            double y = 0;
            double q = (double)point1.GetLength(0);

            x = 0;
            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                x = x + Math.Pow(point1[i], 2);
            }
            x = Math.Pow(x, 0.5);
            if (x == 0) { throw new DistanceException(); }

            y = 0;
            for (int i = 0; i <= point2.GetLength(0) - 1; i++)
            {
                y = y + Math.Pow(point2[i], 2);
            }
            y = Math.Pow(y, 0.5);
            if (y == 0) { throw new DistanceException(); }

            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                buf = buf + (point1[i] * point2[i]);
            }

            if (2.0 * buf / (x * y) <= 2)
            {
                if (Math.Pow(2.0 - 2.0 * buf / (x * y), 0.5) <= 4)
                {
```

```
                return Math.Max(Math.Acos(1.0 - 0.5 * Math.Pow(2.0 - 2.0 *
                    buf / (x * y), 0.5)), this.Min);
            }
            else
            {
                return Math.Max(0, this.Min);
            }
        }
        else
        {
            return Math.Max(0, this.Min);
        }

    }
    public override DistanceBase Clone()
    {
        GeodesicDistance buf = new GeodesicDistance();
        buf.Name = this.Name;
        return buf;
    }
  }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Distance
{
    [Serializable]
    public class ManhattanDistance: DistanceBase
    {
        public ManhattanDistance(): base() { }
        public ManhattanDistance(string Name) : base(Name) { }
        public override double Calculate(double[] point1, double[] point2)
        {
            double buf = 0;
            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                buf = buf + Math.Abs(point1[i] - point2[i]);
            }
            return Math.Max(buf, this.Min);
        }
        public override DistanceBase Clone()
        {
            ManhattanDistance buf = new ManhattanDistance();
            buf.Name = this.Name;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Distance
{
    [Serializable]
    public abstract class MinkowskiDistance : DistanceBase
    {
        public MinkowskiDistance() : base() { }
        public MinkowskiDistance(string Name) : base(Name) { }
        public double epsilon = 0.01;
        public override double Calculate(double[] point1, double[] point2)
        {
            double p = Getp();
            double buf = 0;
            for (int i = 0; i <= point1.GetLength(0) - 1; i++)
            {
                buf = buf + Math.Pow(Math.Abs(point1[i] - point2[i]), p);
            }
            return Math.Max(Math.Pow(buf, 1 / p), this.Min);
        }
        protected abstract double Getp();
    }
    [Serializable]
    public class MinkowskiDistance_p3 : MinkowskiDistance
    {
        public MinkowskiDistance_p3() : base() { }
        public MinkowskiDistance_p3(string Name) : base(Name) { }
        public override DistanceBase Clone()
        {
            MinkowskiDistance_p3 buf = new MinkowskiDistance_p3();
            buf.Name = this.Name;
            return buf;
        }
        protected override double Getp()
        {
            return 3.0;
        }
    }
    [Serializable]
    public class MinkowskiDistance_p4 : MinkowskiDistance
    {
        public MinkowskiDistance_p4() : base() { }
```

```
        public MinkowskiDistance_p4(string Name) : base(Name) { }
        public override DistanceBase Clone()
        {
            MinkowskiDistance_p4 buf = new MinkowskiDistance_p4();
            buf.Name = this.Name;
            return buf;
        }
        protected override double Getp()
        {
            return 4.0;
        }
    }
    [Serializable]
    public class MinkowskiDistance_p5 : MinkowskiDistance
    {
        public MinkowskiDistance_p5() : base() { }
        public MinkowskiDistance_p5(string Name) : base(Name) { }
        public override DistanceBase Clone()
        {
            MinkowskiDistance_p5 buf = new MinkowskiDistance_p5();
            buf.Name = this.Name;
            return buf;
        }
        protected override double Getp()
        {
            return 5.0;
        }
    }
    [Serializable]
    public class MinkowskiDistance_p15 : MinkowskiDistance
    {
        public MinkowskiDistance_p15() : base() { }
        public MinkowskiDistance_p15(string Name) : base(Name) { }
        public override DistanceBase Clone()
        {
            MinkowskiDistance_p15 buf = new MinkowskiDistance_p15();
            buf.Name = this.Name;
            return buf;
        }
        protected override double Getp()
        {
            return 15.0;
        }
    }
}
```

A.6 Mercury.Calculations.Functions.Neighbourhood

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Neighborhood
{
    [Serializable]
    public class DiscreteNeighborhood : NeighborhoodBase
    {
        public DiscreteNeighborhood(double M, double N, double j)
        {
            base.M = M;
            base.N = N;
            base.j = j;
        }
        public DiscreteNeighborhood(string name)
        {
            this.Name = name;
        }
        public DiscreteNeighborhood() { }
        public override double Value(double i)
        {
            if (i == j)
            {
                return 1.0;
            }
            else
            {
                return 0.0;
            }
        }
        public override NeighborhoodBase Clone()
        {
            DiscreteNeighborhood buf = new DiscreteNeighborhood(this.Name);
            buf.Name = this.Name;
            buf.j = this.j;
            buf.M = this.M;
            buf.N = this.N;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Neighborhood
{
    [Serializable]
    public class ExponentialNeighborhood : NeighborhoodBase
    {
        public double Tau = 0.2;
        public double A = 1.0;
        public ExponentialNeighborhood(double M, double N, double j)
        {
            base.M = M;
            base.N = N;
            base.j = j;
            this.Tau = CalculateTau();
        }
        public ExponentialNeighborhood(string name)
        {
            this.Name = name;
            this.Tau = CalculateTau();
        }
        public ExponentialNeighborhood() { }
        public override double Value(double i)
        {
            Tau = CalculateTau();
            return A * Math.Exp(-Math.Abs(i - (N - 1) * j / (M - 1)) / Tau);
        }
        public virtual double CalculateTau()
        {
            return N / 8.0;
        }
        public override NeighborhoodBase Clone()
        {
            ExponentialNeighborhood buf = new ExponentialNeighborhood(this.Name
                );
            buf.Name = this.Name;
            buf.A = this.A;
            buf.j = this.j;
            buf.M = this.M;
            buf.N = this.N;
            return buf;
        }
```

```
}

[Serializable]
public class WideExponentialNeighborhood : ExponentialNeighborhood
{
    public WideExponentialNeighborhood(double M, double N, double j) : base
        (M, N, j) { }
    public WideExponentialNeighborhood(string name) : base(name) { }
    public WideExponentialNeighborhood() { }
    public override double CalculateTau()
    {
        return N / 3.0;
    }
    public override NeighborhoodBase Clone()
    {
        WideExponentialNeighborhood buf = new WideExponentialNeighborhood(
            this.Name);
        buf.Name = this.Name;
        buf.A = this.A;
        buf.j = this.j;
        buf.M = this.M;
        buf.N = this.N;
        return buf;
    }
}

[Serializable]
public class NarrowExponentialNeighborhood : ExponentialNeighborhood
{
    public NarrowExponentialNeighborhood(double M, double N, double j) :
        base(M, N, j) { }
    public NarrowExponentialNeighborhood(string name) : base(name) { }
    public NarrowExponentialNeighborhood() { }
    public override double CalculateTau()
    {
        return N / 12.0;
    }
    public override NeighborhoodBase Clone()
    {
        NarrowExponentialNeighborhood buf = new
            NarrowExponentialNeighborhood(this.Name);
        buf.Name = this.Name;
        buf.A = this.A;
        buf.j = this.j;
        buf.M = this.M;
        buf.N = this.N;
```

```
            return buf;
        }
    }

    [Serializable]
    public class ExtraNarrowExponentialNeighborhood : ExponentialNeighborhood
    {
        public ExtraNarrowExponentialNeighborhood(double M, double N, double j)
             : base(M, N, j) { }
        public ExtraNarrowExponentialNeighborhood(string name) : base(name) { }
        public ExtraNarrowExponentialNeighborhood() { }
        public override double CalculateTau()
        {
            return N / 24.0;
        }
        public override NeighborhoodBase Clone()
        {
            ExtraNarrowExponentialNeighborhood buf = new
                ExtraNarrowExponentialNeighborhood(this.Name);
            buf.Name = this.Name;
            buf.A = this.A;
            buf.j = this.j;
            buf.M = this.M;
            buf.N = this.N;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Neighborhood
{
    [Serializable]
    public class GaussNeighborhood : NeighborhoodBase
    {
        public double Sigma = 0.2;
        public double A = 1.0;
        public GaussNeighborhood(double M, double N, double j)
        {
            base.M = M;
            base.N = N;
            base.j = j;
            this.Sigma = CalculateSigma();
        }
        public GaussNeighborhood(string name)
        {
            this.Name = name;
            this.Sigma = CalculateSigma();
        }
        public GaussNeighborhood() { }
        public override double Value(double i)
        {
            Sigma = CalculateSigma();
            return A * Math.Exp(-0.5 * Math.Pow(i - (N - 1) * j / (M - 1), 2) /
                 Math.Pow(Sigma, 2));
        }
        public virtual double CalculateSigma()
        {
            return N / 6.0;
        }
        public override NeighborhoodBase Clone()
        {
            GaussNeighborhood buf = new GaussNeighborhood(this.Name);
            buf.Name = this.Name;
            buf.Sigma = this.Sigma;
            buf.A = this.A;
            buf.j = this.j;
            buf.M = this.M;
            buf.N = this.N;
            return buf;
```

```
    }
}

[Serializable]
public class WideGaussNeighborhood : GaussNeighborhood
{
    public WideGaussNeighborhood(double M, double N, double j):base(M,N,j)
        {}
    public WideGaussNeighborhood(string name): base (name) {}
    public WideGaussNeighborhood() { }
    public override double CalculateSigma()
    {
        return N / 3.0;
    }
    public override NeighborhoodBase Clone()
    {
        WideGaussNeighborhood buf = new WideGaussNeighborhood(this.Name);
        buf.Name = this.Name;
        buf.Sigma = this.Sigma;
        buf.A = this.A;
        buf.j = this.j;
        buf.M = this.M;
        buf.N = this.N;
        return buf;
    }
}

[Serializable]
public class NarrowGaussNeighborhood : GaussNeighborhood
{
    public NarrowGaussNeighborhood(double M, double N, double j) : base(M,
        N, j) { }
    public NarrowGaussNeighborhood(string name) : base(name) { }
    public NarrowGaussNeighborhood() { }
    public override double CalculateSigma()
    {
        return N / 12.0;
    }
    public override NeighborhoodBase Clone()
    {
        NarrowGaussNeighborhood buf = new NarrowGaussNeighborhood(this.Name
            );
        buf.Name = this.Name;
        buf.Sigma = this.Sigma;
        buf.A = this.A;
        buf.j = this.j;
```

```
            buf.M = this.M;
            buf.N = this.N;
            return buf;
        }
    }

    [Serializable]
    public class ExtraNarrowGaussNeighborhood : GaussNeighborhood
    {
        public ExtraNarrowGaussNeighborhood(double M, double N, double j) :
             base(M, N, j) { }
        public ExtraNarrowGaussNeighborhood(string name) : base(name) { }
        public ExtraNarrowGaussNeighborhood() { }
        public override double CalculateSigma()
        {
            return N / 36.0;
        }
        public override NeighborhoodBase Clone()
        {
            ExtraNarrowGaussNeighborhood buf = new ExtraNarrowGaussNeighborhood
                (this.Name);
            buf.Name = this.Name;
            buf.Sigma = this.Sigma;
            buf.A = this.A;
            buf.j = this.j;
            buf.M = this.M;
            buf.N = this.N;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Neighborhood
{
    [Serializable]
    public class InequalTriangleNeighborhood : NeighborhoodBase
    {
        public InequalTriangleNeighborhood(double M, double N, double j)
        {
            base.M = M;
            base.N = N;
            base.j = j;
        }
        public InequalTriangleNeighborhood(string name)
        {
            this.Name = name;
        }
        public InequalTriangleNeighborhood() { }
        public override double Value(double i)
        {
            double mu = (N-1)*j/(M-1);
            if (j == 0)
            {
                return 1- i / (N - 1);
            }
            else
            {
                if ((i >= 0) && (i <= mu))
                {
                    return i / mu;
                }
                else if ((i > mu) && (i <= (N - 1)))
                {
                    return 1 - (i - mu) / (N - 1 - mu);
                }
                else { return 0.0; }
            }

        }
        public override NeighborhoodBase Clone()
        {
```

```
            InequalTriangleNeighborhood buf = new InequalTriangleNeighborhood(
                this.Name);
            buf.Name = this.Name;
            buf.j = this.j;
            buf.M = this.M;
            buf.N = this.N;
            return buf;
        }
    }
}
```

```
using System;
using System.Xml.Serialization;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Mercury.Calculations.Fuctions;
namespace Mercury.Calculations.Functions.Neighborhood
{
    [XmlInclude(typeof(ExponentialNeighborhood))]
    [XmlInclude(typeof(WideExponentialNeighborhood))]
    [XmlInclude(typeof(NarrowExponentialNeighborhood))]
    [XmlInclude(typeof(ExtraNarrowExponentialNeighborhood))]
    [XmlInclude(typeof(GaussNeighborhood))]
    [XmlInclude(typeof(WideGaussNeighborhood))]
    [XmlInclude(typeof(NarrowGaussNeighborhood))]
    [XmlInclude(typeof(ExtraNarrowGaussNeighborhood))]
    [XmlInclude(typeof(InequalTriangleNeighborhood))]
    [XmlInclude(typeof(RectangularNeighborhood))]
    [XmlInclude(typeof(WideRectangularNeighborhood))]
    [XmlInclude(typeof(NarrowRectangularNeighborhood))]
    [XmlInclude(typeof(ExtraNarrowRectangularNeighborhood))]
    [XmlInclude(typeof(SigmoidalNeighborhood))]
    [XmlInclude(typeof(TrapezoidalNeighborhood))]
    [XmlInclude(typeof(TriangularNeighborhood))]
    [XmlInclude(typeof(WideTriangularNeighborhood))]
    [XmlInclude(typeof(NarrowTriangularNeighborhood))]
    [XmlInclude(typeof(ExtraNarrowTriangularNeighborhood))]
    [Serializable]
    public abstract class NeighborhoodBase: FunctionBase
    {
        public double M, N, j;
        public NeighborhoodBase(){}
        public NeighborhoodBase(double M, double N, double j)
        {
            this.M = M;
            this.N = N;
            this.j = j;
        }
        public virtual NeighborhoodBase Clone()
        {
            throw new Exception("Not implemented clone method");
            return null;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Neighborhood
{
    [Serializable]
    public class RectangularNeighborhood : NeighborhoodBase
    {
        public double L = 0.2;
        public RectangularNeighborhood(double M, double N, double j)
        {
            base.M = M;
            base.N = N;
            base.j = j;
            this.L = CalculateL();
        }
        public RectangularNeighborhood(string name)
        {
            this.Name = name;
            this.L = CalculateL();
        }
        public RectangularNeighborhood() { }
        public override double Value(double i)
        {
            double mu = (N-1)*j/(M-1);
            this.L = CalculateL();

            if ((i >= 0) && (i < mu-0.5*L))
            {
                return 0.0;
            }
            else if (((mu-0.5*L) <= i) && (i <= (mu+0.5*L)))
            {
                return 1.0;
            }
            else if (((mu + 0.5 * L) < i) && (i <= (N-1)))
            {
                return 0.0;
            }
            else
            {
                return 0.0;
            }
```

```
    }
    public virtual double CalculateL()
    {
        return (N - 1)/3;
    }
    public override NeighborhoodBase Clone()
    {
        RectangularNeighborhood buf = new RectangularNeighborhood(this.Name
            );
        buf.Name = this.Name;
        buf.L = this.L;
        buf.j = this.j;
        buf.M = this.M;
        buf.N = this.N;
        return buf;
    }
}

[Serializable]
public class WideRectangularNeighborhood : RectangularNeighborhood
{
    public WideRectangularNeighborhood(double M, double N, double j) : base
        (M, N, j) { }
    public WideRectangularNeighborhood(string name) : base(name) { }
    public WideRectangularNeighborhood() { }
    public override double CalculateL()
    {
        return (N - 1)/2;
    }
    public override NeighborhoodBase Clone()
    {
        WideRectangularNeighborhood buf = new WideRectangularNeighborhood(
            this.Name);
        buf.Name = this.Name;
        buf.L = this.L;
        buf.j = this.j;
        buf.M = this.M;
        buf.N = this.N;
        return buf;
    }
}

[Serializable]
public class NarrowRectangularNeighborhood : RectangularNeighborhood
{
```

```
        public NarrowRectangularNeighborhood(double M, double N, double j) :
            base(M, N, j) { }
        public NarrowRectangularNeighborhood(string name) : base(name) { }
        public NarrowRectangularNeighborhood() { }
        public override double CalculateL()
        {
            return (N - 1) / 6;
        }
        public override NeighborhoodBase Clone()
        {
            NarrowRectangularNeighborhood buf = new
                NarrowRectangularNeighborhood(this.Name);
            buf.Name = this.Name;
            buf.L = this.L;
            buf.j = this.j;
            buf.M = this.M;
            buf.N = this.N;
            return buf;
        }
    }

    [Serializable]
    public class ExtraNarrowRectangularNeighborhood : RectangularNeighborhood
    {
        public ExtraNarrowRectangularNeighborhood(double M, double N, double j)
             : base(M, N, j) { }
        public ExtraNarrowRectangularNeighborhood(string name) : base(name) { }
        public ExtraNarrowRectangularNeighborhood() { }
        public override double CalculateL()
        {
            return (N - 1) / 12;
        }
        public override NeighborhoodBase Clone()
        {
            ExtraNarrowRectangularNeighborhood buf = new
                ExtraNarrowRectangularNeighborhood(this.Name);
            buf.Name = this.Name;
            buf.L = this.L;
            buf.j = this.j;
            buf.M = this.M;
            buf.N = this.N;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Neighborhood
{
    [Serializable]
    public class SigmoidalNeighborhood : NeighborhoodBase
    {

        public SigmoidalNeighborhood(double M, double N, double j)
        {
            base.M = M;
            base.N = N;
            base.j = j;
        }
        public SigmoidalNeighborhood(string name)
        {
            this.Name = name;
        }
        public SigmoidalNeighborhood() { }
        public override double Value(double i)
        {
            double tau = 10;
            double a = -2*j/(M-1) + 1;
            double b = tau / (N - 1);
            double c = (N - 1) / 2;
            return 1/(1+ Math.Exp(a * b * (i - c)));
        }

        public override NeighborhoodBase Clone()
        {
            SigmoidalNeighborhood buf = new SigmoidalNeighborhood(this.Name);
            buf.Name = this.Name;
            buf.j = this.j;
            buf.M = this.M;
            buf.N = this.N;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Neighborhood
{
    [Serializable]
    public class TrapezoidalNeighborhood : NeighborhoodBase
    {
        public double L = 0.2;
        public TrapezoidalNeighborhood(double M, double N, double j)
        {
            base.M = M;
            base.N = N;
            base.j = j;
            this.L = CalculateL();
        }
        public TrapezoidalNeighborhood(string name)
        {
            this.Name = name;
            this.L = CalculateL();
        }
        public TrapezoidalNeighborhood() { }
        public override double Value(double i)
        {
            double mu = (N-1)*j/(M-1);
            this.L = CalculateL();

            if ((i >= 0) && (i < mu-0.5*L))
            {
                return Math.Abs(i/(mu-0.5*L));
            }
            else if (((mu-0.5*L) <= i) && (i <= (mu+0.5*L)))
            {
                return 1.0;
            }
            else if (((mu + 0.5 * L) < i) && (i <= (N-1)))
            {
                return 1 + Math.Abs((0.5*L+mu-i)/(N-mu-0.5*L-1));
            }
            else
            {
                return 0.0;
            }
```

```
        }
        public virtual double CalculateL()
        {
            return (N - 1)/3;
        }
        public override NeighborhoodBase Clone()
        {
            TrapezoidalNeighborhood buf = new TrapezoidalNeighborhood(this.Name
                );
            buf.Name = this.Name;
            buf.L = this.L;
            buf.j = this.j;
            buf.M = this.M;
            buf.N = this.N;
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Functions.Neighborhood
{
    [Serializable]
    public class TriangularNeighborhood : NeighborhoodBase
    {
        public double L = 0.2;
        public double A = 1.0;
        public TriangularNeighborhood(double M, double N, double j)
        {
            base.M = M;
            base.N = N;
            base.j = j;
            this.L = CalculateL();
        }
        public TriangularNeighborhood(string name)
        {
            this.Name = name;
            this.L = CalculateL();
        }
        public TriangularNeighborhood() { }
        public override double Value(double i)
        {
            double buf = 0;
            L = CalculateL();
            buf = 1 - Math.Abs((i - (N - 1) * j / (M - 1)) / (0.5 * L));
            if (buf < 0) { buf = 0; }
            return buf;

        }
        public virtual double CalculateL()
        {
            return (N - 1);
        }
        public override NeighborhoodBase Clone()
        {
            TriangularNeighborhood buf = new TriangularNeighborhood(this.Name);
            buf.Name = this.Name;
            buf.L = this.L;
            buf.A = this.A;
            buf.j = this.j;
```

```
        buf.M = this.M;
        buf.N = this.N;
        return buf;
    }
}

[Serializable]
public class WideTriangularNeighborhood : TriangularNeighborhood
{
    public WideTriangularNeighborhood(double M, double N, double j) : base(
        M, N, j) { }
    public WideTriangularNeighborhood(string name) : base(name) { }
    public WideTriangularNeighborhood() { }
    public override double CalculateL()
    {
        return (N - 1) * 2;
    }
    public override NeighborhoodBase Clone()
    {
        WideTriangularNeighborhood buf = new WideTriangularNeighborhood(
            this.Name);
        buf.Name = this.Name;
        buf.L = this.L;
        buf.A = this.A;
        buf.j = this.j;
        buf.M = this.M;
        buf.N = this.N;
        return buf;
    }
}

[Serializable]
public class NarrowTriangularNeighborhood : TriangularNeighborhood
{
    public NarrowTriangularNeighborhood(double M, double N, double j) :
        base(M, N, j) { }
    public NarrowTriangularNeighborhood(string name) : base(name) { }
    public NarrowTriangularNeighborhood() { }
    public override double CalculateL()
    {
        return (N - 1) / 2;
    }
    public override NeighborhoodBase Clone()
    {
        NarrowTriangularNeighborhood buf = new NarrowTriangularNeighborhood
            (this.Name);
```

```
            buf.Name = this.Name;
            buf.L = this.L;
            buf.A = this.A;
            buf.j = this.j;
            buf.M = this.M;
            buf.N = this.N;
            return buf;
        }
    }

    [Serializable]
    public class ExtraNarrowTriangularNeighborhood : TriangularNeighborhood
    {
        public ExtraNarrowTriangularNeighborhood(double M, double N, double j)
            : base(M, N, j) { }
        public ExtraNarrowTriangularNeighborhood(string name) : base(name) { }
        public ExtraNarrowTriangularNeighborhood() { }
        public override double CalculateL()
        {
            return (N - 1) / 4;
        }
        public override NeighborhoodBase Clone()
        {
            ExtraNarrowTriangularNeighborhood buf = new
                ExtraNarrowTriangularNeighborhood(this.Name);
            buf.Name = this.Name;
            buf.L = this.L;
            buf.A = this.A;
            buf.j = this.j;
            buf.M = this.M;
            buf.N = this.N;
            return buf;
        }
    }
}
```

A.7 Mercury.Calculations.Graphics.Scatter

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Graphics.Scatter
{
    public class CentMembershipScatter : Scatter2DBase
    {
        public CentMembershipScatter() { }
        public void AddCentMamberchip(double cent, double membership)
        {
            if (data == null) { data = new List<PointD>(); };
            data.Add(new PointD(cent, membership));
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Graphics.Scatter
{
    public class DurationMultiPitchRestScatter : ScatterNDBase
    {
        public DurationMultiPitchRestScatter() { }
        public void AddDurationPitches(double duration, double[] pitches, bool
            isRest)
        {
            if (data == null) { data = new List<double[]>(); };
            double[] buf = new double[pitches.GetLength(0) + 2];
            buf[0] = duration;
            for (int i = 1; i <= buf.GetLength(0) - 2; i++)
            {
                buf[i] = pitches[i - 1];
            }
            buf[buf.GetLength(0) - 1] = Convert.ToDouble(isRest);
            data.Add(buf);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Graphics.Scatter
{
    public class DurationPitchRestAccentScatter : ScatterNDBase
    {
        public DurationPitchRestAccentScatter() { }
        public void AddDurationPitch(double duration, double pitch, bool isRest
             , bool accent)
        {
            if (data == null) { data = new List<double[]>(); };
            data.Add(new double[] { duration, pitch, Convert.ToDouble(isRest),
                 Convert.ToDouble(accent)});
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Graphics.Scatter
{
    public class DurationPitchRestAccentStaccatoScatter : ScatterNDBase
    {
        public DurationPitchRestAccentStaccatoScatter() { }
        public void AddDurationPitch(double duration, double pitch, bool isRest
            , bool accent, bool stacatto)
        {
            if (data == null) { data = new List<double[]>(); };
            data.Add(new double[] { duration, pitch, Convert.ToDouble(isRest),
                Convert.ToDouble(accent), Convert.ToDouble(stacatto) });
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Graphics.Scatter
{
    public class DurationPitchRestScatter : ScatterNDBase
    {
        public DurationPitchRestScatter() { }
        public void AddDurationPitch(double duration, double pitch, bool isRest
            )
        {
            if (data == null) { data = new List<double[]>(); };
            data.Add(new double[] { duration, pitch, Convert.ToDouble(isRest)})
                ;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Graphics.Scatter
{
    public class DurationPitchScatter : Scatter2DBase
    {
        public DurationPitchScatter() { }
        public void AddDurationPitch(double duration, int pitch)
        {
            if (data == null) { data = new List<PointD>(); };
            data.Add(new PointD(duration,pitch));
        }
        public void AddDurationPitch(double duration, double pitch)
        {
            if (data == null) { data = new List<PointD>(); };
            data.Add(new PointD(duration, pitch));
        }

    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Graphics.Scatter
{
    public class HarmonicScatter : ScatterNDBase
    {
        public HarmonicScatter() { }
        public void AddChord(double[] pitches)
        {
            if (data == null) { data = new List<double[]>(); };
            data.Add(pitches);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Graphics.Scatter
{
    public class RhythmicalScatter : ScatterNDBase
    {
        public RhythmicalScatter() { }
        public void AddRhythimcFigure(double durationCoefficient, bool isRest)
        {
            if (data == null) { data = new List<double[]>(); };
            data.Add(new double[2] { durationCoefficient, Convert.ToDouble(
                isRest) });
        }
        public void AddRhythimcFigure(double durationCoefficient, bool isRest,
             bool isAccented)
        {
            if (data == null) { data = new List<double[]>(); };
            data.Add(new double[3] { durationCoefficient, Convert.ToDouble(
                isRest), Convert.ToDouble(isAccented) });
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Graphics.Scatter
{
    public abstract class Scatter2DBase
    {

        protected List<PointD> data = null;

        public double[,] GetData()
        {
            double[,] buf = null;
            if (data != null) {
                buf = new double[data.Count, 2];
                for (int i=0; i<=data.Count-1; i++)
                {
                    buf[i, 0] = data[i].X;
                    buf[i, 1] = data[i].Y;
                }
            };
            return buf;
        }

        public override string ToString()
        {
            StringBuilder str = new StringBuilder();
            foreach (PointD punto in data)
            {
                str.Append(punto.X);
                str.Append("\t");
                str.Append(punto.Y);
                str.AppendLine();
            }
            return str.ToString();
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Mercury.Domain.MusicalSymbol;
using Mercury.Calculations;
using Mercury.Domain.MusicalSymbol.Enumerations;
using Mercury.Calculations.Duration;

namespace Mercury.Calculations.Graphics.Scatter
{

    public class ScatterInconsistentChordsException : Exception {}

}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Mercury.Domain.MusicalSymbol;
using Mercury.Calculations;
using Mercury.Domain.MusicalSymbol.Enumerations;
using Mercury.Calculations.Duration;
using Mercury.Calculations.Tunings;

namespace Mercury.Calculations.Graphics.Scatter
{
    public static class ScatterFactory
    {
        public static DurationPitchScatter CreateDurationPitchScatter(List<
            MusicalSymbolBase> notes)
        {

            int normalNotes = 1;
            int actualNotes = 1;
            int duration = 1;
            double totalDuration = 0;
            double bufLigaduraValor = 0;

            //TO DO: comprobacion de que no existan acordes

            DurationPitchScatter data = new DurationPitchScatter();
            if (notes != null)
            {
                foreach (MusicalSymbolBase nota in notes)
                {
                    if (nota.GetType() == typeof(Note))
                    {
                        if (((Note)nota).NormalNotes == 0) { normalNotes = 1; }
                            else { normalNotes = ((Note)nota).NormalNotes; };
                        if (((Note)nota).ActualNotes == 0) { actualNotes = 1; }
                            else { actualNotes = ((Note)nota).ActualNotes; };
                        duration = (int)((Note)nota).Duration;
                        totalDuration = DurationCalculator.TotalDuration(((Note)
                            nota).NumberOfDots, duration, normalNotes,
                            actualNotes);

                        //Take into account the value tied notes...are
                            represented as one with the duration summatory of
                            all the tied notes.
```

```
                if ((((Note)nota).TieType == NoteTieType.Start)||(((Note
                    )nota).TieType == NoteTieType.StopAndStartAnother))
                {
                    bufLigaduraValor = bufLigaduraValor + totalDuration;
                }
                else if ((((Note)nota).TieType == NoteTieType.None) ||
                    (((Note)nota).TieType == NoteTieType.Stop))
                {
                    if (bufLigaduraValor != 0) { totalDuration =
                        totalDuration + bufLigaduraValor; }
                    data.AddDurationPitch(totalDuration, ((Note)nota).
                        MidiPitch);
                    bufLigaduraValor = 0;
                };

            }
            if (nota.GetType() == typeof(Rest))
            {
                if (((Rest)nota).NormalNotes == 0) { normalNotes = 1; }
                    else { normalNotes = ((Rest)nota).NormalNotes; };
                if (((Rest)nota).ActualNotes == 0) { actualNotes = 1; }
                    else { actualNotes = ((Rest)nota).ActualNotes; };
                duration = (int)((Rest)nota).Duration;
                totalDuration = DurationCalculator.TotalDuration(((Rest)
                    nota).NumberOfDots, duration, normalNotes,
                    actualNotes);
                data.AddDurationPitch(totalDuration, 0);
            }
        }
    }
    return data;
}

public static DurationPitchRestScatter CreateDurationPitchRestScatter(
    List<MusicalSymbolBase> notes)
{

    int normalNotes = 1;
    int actualNotes = 1;
    int duration = 1;
    double totalDuration = 0;
    double bufLigaduraValor = 0;
    MusicalSymbolBase nota = null;
    double averagePitch = 0;

    //TO DO: comprobacion de que no existan acordes
```

```
DurationPitchRestScatter data = new DurationPitchRestScatter();
if (notes != null)
{
    for (int i = 0; i <= notes.Count - 1; i++)
    {
        nota = notes[i];
        if (nota.GetType() == typeof(Note))
            {
                if (((Note)nota).NormalNotes == 0) { normalNotes = 1;
                     } else { normalNotes = ((Note)nota).NormalNotes
                    ; };
                if (((Note)nota).ActualNotes == 0) { actualNotes = 1;
                     } else { actualNotes = ((Note)nota).ActualNotes
                    ; };
                duration = (int)((Note)nota).Duration;
                totalDuration = DurationCalculator.TotalDuration(((
                    Note)nota).NumberOfDots, duration, normalNotes,
                    actualNotes);

                //Take into account the value tied notes...are
                    represented as one with the duration summatory
                    of all the tied notes.
                if ((((Note)nota).TieType == NoteTieType.Start) ||
                    (((Note)nota).TieType == NoteTieType.
                    StopAndStartAnother))
                {
                    bufLigaduraValor = bufLigaduraValor +
                        totalDuration;
                }
                else if ((((Note)nota).TieType == NoteTieType.None)
                    || (((Note)nota).TieType == NoteTieType.Stop))
                {
                    if (bufLigaduraValor != 0) { totalDuration =
                        totalDuration + bufLigaduraValor; }
                    data.AddDurationPitch(totalDuration, Convert.
                        ToDouble(((Note)nota).MidiPitch), false);
                    bufLigaduraValor = 0;
                };

            }
            if (nota.GetType() == typeof(Rest))
            {
                if (((Rest)nota).NormalNotes == 0) { normalNotes = 1;
                     } else { normalNotes = ((Rest)nota).NormalNotes
                    ; };
```

```
if (((Rest)nota).ActualNotes == 0) { actualNotes = 1;
     } else { actualNotes = ((Rest)nota).ActualNotes
    ; };
duration = (int)((Rest)nota).Duration;
totalDuration = DurationCalculator.TotalDuration(((
    Rest)nota).NumberOfDots, duration, normalNotes,
    actualNotes);

//Calculate the virtual pitch of the rest
averagePitch = 0;
if (i == 0)
{
    //Find the first note to the right that is not a
        rest, and store the midiPitch
    for (int j = 0; j <= notes.Count - 1; j++)
    {
        if (notes[j].GetType() == typeof(Note))
        {
            averagePitch = ((Note)notes[j]).MidiPitch;
            break;
        }
    }

}
else if (i == notes.Count - 1)
{
    //Find the first note to the left that is not a
        rest, and store the midiPitch
    for (int j = notes.Count - 1; j >= 0; j--)
    {
        if (notes[j].GetType() == typeof(Note))
        {
            averagePitch = ((Note)notes[j]).MidiPitch;
            break;
        }
    }

}
else
{
    //Find the first note to the right that is not a
        rest, and store the midiPitch
    for (int j = i; j <= notes.Count - 1; j++)
    {
        if (notes[j].GetType() == typeof(Note))
        {
```

```
                            averagePitch = ((Note)notes[j]).MidiPitch;
                            break;
                        }
                    }

                    //Find the first note to the left that is not a
                        rest, and store the midiPitch
                    for (int j = i; j >= 0; j--)
                    {
                        if (notes[j].GetType() == typeof(Note))
                        {
                            averagePitch = averagePitch + ((Note)notes
                                [j]).MidiPitch;
                            break;
                        }
                    }
                    //Calculate the average pitch of first to notes
                        close to the rest.
                    averagePitch = 0.5 * averagePitch;

                }

                data.AddDurationPitch(totalDuration, averagePitch,
                    true);
            }

        }

    }
    return data;
}

public static DurationPitchRestScatter
    CreateDurationMultiPitchRestScatter(List<MusicalSymbolBase> notes)
{

    int normalNotes = 1;
    int actualNotes = 1;
    int duration = 1;
    double totalDuration = 0;
    double bufLigaduraValor = 0;
    MusicalSymbolBase nota = null;
    double averagePitch = 0;
```

```
DurationMultiPitchRestScatter data = new
    DurationMultiPitchRestScatter();
if (notes != null)
{
    for (int i = 0; i <= notes.Count - 1; i++)
    {
        nota = notes[i];
        if (nota.GetType() == typeof(Note))
        {
            if (((Note)nota).NormalNotes == 0) { normalNotes = 1; }
                else { normalNotes = ((Note)nota).NormalNotes; };
            if (((Note)nota).ActualNotes == 0) { actualNotes = 1; }
                else { actualNotes = ((Note)nota).ActualNotes; };
            duration = (int)((Note)nota).Duration;
            totalDuration = DurationCalculator.TotalDuration(((Note)
                nota).NumberOfDots, duration, normalNotes,
                actualNotes);

            //Take into account the value tied notes...are
                represented as one with the duration summatory of
                all the tied notes.
            if ((((Note)nota).TieType == NoteTieType.Start) || (((
                Note)nota).TieType == NoteTieType.
                StopAndStartAnother))
            {
                bufLigaduraValor = bufLigaduraValor + totalDuration;
            }
            else if ((((Note)nota).TieType == NoteTieType.None) ||
                (((Note)nota).TieType == NoteTieType.Stop))
            {
                if (bufLigaduraValor != 0) { totalDuration =
                    totalDuration + bufLigaduraValor; }
                //data.AddDurationPitch(totalDuration, Convert.
                    ToDouble(((Note)nota).MidiPitch), false);
                bufLigaduraValor = 0;
            };

        }
        if (nota.GetType() == typeof(Rest))
        {
            if (((Rest)nota).NormalNotes == 0) { normalNotes = 1; }
                else { normalNotes = ((Rest)nota).NormalNotes; };
            if (((Rest)nota).ActualNotes == 0) { actualNotes = 1; }
                else { actualNotes = ((Rest)nota).ActualNotes; };
            duration = (int)((Rest)nota).Duration;
```

```
totalDuration = DurationCalculator.TotalDuration(((Rest)
    nota).NumberOfDots, duration, normalNotes,
    actualNotes);

//Calculate the virtual pitch of the rest
averagePitch = 0;
if (i == 0)
{
    //Find the first note to the right that is not a rest
        , and store the midiPitch
    for (int j = 0; j <= notes.Count - 1; j++)
    {
        if (notes[j].GetType() == typeof(Note))
        {
            averagePitch = ((Note)notes[j]).MidiPitch;
            break;
        }
    }

}
else if (i == notes.Count - 1)
{
    //Find the first note to the left that is not a rest,
         and store the midiPitch
    for (int j = notes.Count - 1; j >= 0; j--)
    {
        if (notes[j].GetType() == typeof(Note))
        {
            averagePitch = ((Note)notes[j]).MidiPitch;
            break;
        }
    }

}
else
{
    //Find the first note to the right that is not a rest
        , and store the midiPitch
    for (int j = i; j <= notes.Count - 1; j++)
    {
        if (notes[j].GetType() == typeof(Note))
        {
            averagePitch = ((Note)notes[j]).MidiPitch;
            break;
        }
    }
```

```
                    //Find the first note to the left that is not a rest,
                         and store the midiPitch
                    for (int j = i; j >= 0; j--)
                    {
                        if (notes[j].GetType() == typeof(Note))
                        {
                            averagePitch = averagePitch + ((Note)notes[j])
                                .MidiPitch;
                            break;
                        }
                    }
                    //Calculate the average pitch of first to notes close
                         to the rest.
                    averagePitch = 0.5 * averagePitch;

                }

                //data.AddDurationPitch(totalDuration, averagePitch,
                     true);
            }

        }

    }
    return null;
}

public static DurationPitchRestAccentScatter
    CreateDurationPitchRestsAccentScatter(List<MusicalSymbolBase>
    notes)
{

    int normalNotes = 1;
    int actualNotes = 1;
    int duration = 1;
    double totalDuration = 0;
    double bufLigaduraValor = 0;
    MusicalSymbolBase nota = null;
    double averagePitch = 0;
    bool accented = false;

    //TO DO: comprobacion de que no existan acordes

    DurationPitchRestAccentScatter data = new
        DurationPitchRestAccentScatter();
```

```
if (notes != null)
{
    for (int i = 0; i <= notes.Count - 1; i++)
    {
        nota = notes[i];
        if (nota.GetType() == typeof(Note))
        {
            if (((Note)nota).NormalNotes == 0) { normalNotes = 1; }
                else { normalNotes = ((Note)nota).NormalNotes; };
            if (((Note)nota).ActualNotes == 0) { actualNotes = 1; }
                else { actualNotes = ((Note)nota).ActualNotes; };
            duration = (int)((Note)nota).Duration;
            totalDuration = DurationCalculator.TotalDuration(((Note)
                nota).NumberOfDots, duration, normalNotes,
                actualNotes);


            //Take into account the articulation
            if ((((Note)nota).TieType == NoteTieType.Start) || (((
                Note)nota).TieType == NoteTieType.None))
            {
                accented = ((((Note)nota).Articulation ==
                    ArticulationType.Accent) | (((Note)nota).
                    Articulation == ArticulationType.StaccatoAccent)
                    );
            };

            //Take into account the value tied notes...are
                represented as one with the duration summatory of
                all the tied notes.
            if ((((Note)nota).TieType == NoteTieType.Start) || (((
                Note)nota).TieType == NoteTieType.
                StopAndStartAnother))
            {
                bufLigaduraValor = bufLigaduraValor + totalDuration;
            }
            else if ((((Note)nota).TieType == NoteTieType.None) ||
                (((Note)nota).TieType == NoteTieType.Stop))
            {
                if (bufLigaduraValor != 0) { totalDuration =
                    totalDuration + bufLigaduraValor; }
                data.AddDurationPitch(totalDuration, Convert.ToDouble
                    (((Note)nota).MidiPitch), false, accented);
                bufLigaduraValor = 0;
            };
```

```
}
if (nota.GetType() == typeof(Rest))
{
    if (((Rest)nota).NormalNotes == 0) { normalNotes = 1; }
        else { normalNotes = ((Rest)nota).NormalNotes; };
    if (((Rest)nota).ActualNotes == 0) { actualNotes = 1; }
        else { actualNotes = ((Rest)nota).ActualNotes; };
    duration = (int)((Rest)nota).Duration;
    totalDuration = DurationCalculator.TotalDuration(((Rest)
        nota).NumberOfDots, duration, normalNotes,
        actualNotes);

    //Calculate the virtual pitch of the rest
    averagePitch = 0;
    if (i == 0)
    {
        //Find the first note to the right that is not a rest
            , and store the midiPitch
        for (int j = 0; j <= notes.Count - 1; j++)
        {
            if (notes[j].GetType() == typeof(Note))
            {
                averagePitch = ((Note)notes[j]).MidiPitch;
                break;
            }
        }

    }
    else if (i == notes.Count - 1)
    {
        //Find the first note to the left that is not a rest,
             and store the midiPitch
        for (int j = notes.Count - 1; j >= 0; j--)
        {
            if (notes[j].GetType() == typeof(Note))
            {
                averagePitch = ((Note)notes[j]).MidiPitch;
                break;
            }
        }

    }
    else
    {
        //Find the first note to the right that is not a rest
            , and store the midiPitch
```

```
                    for (int j = i; j <= notes.Count - 1; j++)
                    {
                        if (notes[j].GetType() == typeof(Note))
                        {
                            averagePitch = ((Note)notes[j]).MidiPitch;
                            break;
                        }
                    }

                    //Find the first note to the left that is not a rest,
                         and store the midiPitch
                    for (int j = i; j >= 0; j--)
                    {
                        if (notes[j].GetType() == typeof(Note))
                        {
                            averagePitch = averagePitch + ((Note)notes[j])
                                .MidiPitch;
                            break;
                        }
                    }
                    //Calculate the average pitch of first to notes close
                         to the rest.
                    averagePitch = 0.5 * averagePitch;

                }

                data.AddDurationPitch(totalDuration, averagePitch, true,
                     false);
            }

        }

    }
    return data;
}

public static DurationPitchRestAccentStaccatoScatter
    CreateDurationPitchRestsAccentStacattoScatter(List<
    MusicalSymbolBase> notes)
{

    int normalNotes = 1;
    int actualNotes = 1;
    int duration = 1;
    double totalDuration = 0;
    double bufLigaduraValor = 0;
```

```
MusicalSymbolBase nota = null;
double averagePitch = 0;
bool accent = false;
bool staccato = false;

//TO DO: comprobacion de que no existan acordes

DurationPitchRestAccentStaccatoScatter data = new
    DurationPitchRestAccentStaccatoScatter();
if (notes != null)
{
    for (int i = 0; i <= notes.Count - 1; i++)
    {
        nota = notes[i];
        if (nota.GetType() == typeof(Note))
        {
            if (((Note)nota).NormalNotes == 0) { normalNotes = 1; }
                else { normalNotes = ((Note)nota).NormalNotes; };
            if (((Note)nota).ActualNotes == 0) { actualNotes = 1; }
                else { actualNotes = ((Note)nota).ActualNotes; };
            duration = (int)((Note)nota).Duration;
            totalDuration = DurationCalculator.TotalDuration(((Note)
                nota).NumberOfDots, duration, normalNotes,
                actualNotes);


            //Take into account the articulation
            if ((((Note)nota).TieType == NoteTieType.Start) || (((
                Note)nota).TieType == NoteTieType.None))
            {
                accent = ((((Note)nota).Articulation ==
                    ArticulationType.Accent) | (((Note)nota).
                    Articulation == ArticulationType.StaccatoAccent)
                    );
                staccato = ((((Note)nota).Articulation ==
                    ArticulationType.Staccato) | (((Note)nota).
                    Articulation == ArticulationType.StaccatoAccent)
                    );
            };

            //Take into account the value tied notes...are
                represented as one with the duration summatory of
                all the tied notes.
            if ((((Note)nota).TieType == NoteTieType.Start) || (((
                Note)nota).TieType == NoteTieType.
                StopAndStartAnother))
```

```
        {
            bufLigaduraValor = bufLigaduraValor + totalDuration;
        }
        else if ((((Note)nota).TieType == NoteTieType.None) ||
            (((Note)nota).TieType == NoteTieType.Stop))
        {
            if (bufLigaduraValor != 0) { totalDuration =
                totalDuration + bufLigaduraValor; }
            data.AddDurationPitch(totalDuration, Convert.ToDouble
                (((Note)nota).MidiPitch), false, accent,
                staccato);
            bufLigaduraValor = 0;
        };

    }
    if (nota.GetType() == typeof(Rest))
    {
        if (((Rest)nota).NormalNotes == 0) { normalNotes = 1; }
            else { normalNotes = ((Rest)nota).NormalNotes; };
        if (((Rest)nota).ActualNotes == 0) { actualNotes = 1; }
            else { actualNotes = ((Rest)nota).ActualNotes; };
        duration = (int)((Rest)nota).Duration;
        totalDuration = DurationCalculator.TotalDuration(((Rest)
            nota).NumberOfDots, duration, normalNotes,
            actualNotes);

        //Calculate the virtual pitch of the rest
        averagePitch = 0;
        if (i == 0)
        {
            //Find the first note to the right that is not a rest
                , and store the midiPitch
            for (int j = 0; j <= notes.Count - 1; j++)
            {
                if (notes[j].GetType() == typeof(Note))
                {
                    averagePitch = ((Note)notes[j]).MidiPitch;
                    break;
                }
            }

        }
        else if (i == notes.Count - 1)
        {
            //Find the first note to the left that is not a rest,
                 and store the midiPitch
```

```
                for (int j = notes.Count - 1; j >= 0; j--)
                {
                    if (notes[j].GetType() == typeof(Note))
                    {
                        averagePitch = ((Note)notes[j]).MidiPitch;
                        break;
                    }
                }

            }
            else
            {
                //Find the first note to the right that is not a rest
                    , and store the midiPitch
                for (int j = i; j <= notes.Count - 1; j++)
                {
                    if (notes[j].GetType() == typeof(Note))
                    {
                        averagePitch = ((Note)notes[j]).MidiPitch;
                        break;
                    }
                }

                //Find the first note to the left that is not a rest,
                     and store the midiPitch
                for (int j = i; j >= 0; j--)
                {
                    if (notes[j].GetType() == typeof(Note))
                    {
                        averagePitch = averagePitch + ((Note)notes[j])
                            .MidiPitch;
                        break;
                    }
                }
                //Calculate the average pitch of first to notes close
                     to the rest.
                averagePitch = 0.5 * averagePitch;

            }

            data.AddDurationPitch(totalDuration, averagePitch, true,
                 false, false);
        }

    }
```

```
    }
    return data;
}



public static CentMembershipScatter
    CreateCentMembershipScatter_Centroids(FuzzyTuningBase
    tunningSystem, double[,] frecuenciesHistogram)
{
    CentMembershipScatter data = new CentMembershipScatter();
    double[] notes = tunningSystem.Notes;
    double buf = 0;
    List<double> realNotes = new List<double>();

    //Remove the centroids with no one belonging note

    for (int j = 0; j <= notes.GetLength(0) - 1; j++)
    {
        buf = 0;
        for (int i = 0; i <= frecuenciesHistogram.GetLength(0) - 1; i++)
        {
            buf = buf + tunningSystem.CalculateMembership(notes[j],
                frecuenciesHistogram[i, 0]);
        }
        if (buf != 0) { realNotes.Add(notes[j]); }
    }

    for (int i = 0; i <= realNotes.Count - 1; i++)
    {
        data.AddCentMamberchip(realNotes[i], 1.0);
    }

    return data;
}

public static CentMembershipScatter CreateCentMembershipScatter_data(
    double[,] frecuenciesHistogram, FuzzyTuningBase tunningSystem)
{
    CentMembershipScatter data = new CentMembershipScatter();
    double[] notes = tunningSystem.Notes;
    double buf = 0;

    for (int i = 0; i<=frecuenciesHistogram.GetLength(0) - 1; i++)
    {
        for (int j = 0; j <= notes.GetLength(0) - 1; j++)
```

```
        {
            buf = tunningSystem.CalculateMembership(notes[j],
                frecuenciesHistogram[i, 0]);
            //Add only the notes with non cero membership
            if (buf > 0)
            {
                for (int k = 0; k <= frecuenciesHistogram[i, 1] - 1; k
                    ++)
                {
                    data.AddCentMamberchip(frecuenciesHistogram[i, 0],
                        buf);
                }
                break;
            }
        }

    }

    return data;
}

public static List<CentMembershipScatter>
    CreateCentMembershipScatter_FuzzyNotes(FuzzyTuningBase
    tunningSystem, double[,] frecuenciesHistogram)
{
    List<CentMembershipScatter> listFuzzyNotes = new List<
        CentMembershipScatter>();
    CentMembershipScatter data = new CentMembershipScatter();
    double[] notes = tunningSystem.Notes;
    double buf = 0;
    List<double> realNotes = new List<double>();

    //Remove the centroids with no one belonging note
    for (int j = 0; j <= notes.GetLength(0) - 1; j++)
    {
        buf = 0;
        for (int i = 0; i <= frecuenciesHistogram.GetLength(0) - 1; i++)
        {
            buf = buf + tunningSystem.CalculateMembership(notes[j],
                frecuenciesHistogram[i, 0]);
        }
        if (buf != 0) { realNotes.Add(notes[j]); }
    }


    for (int j = 0; j <= realNotes.Count - 1; j++)
```

```
    {
        data = new CentMembershipScatter();
        data.AddCentMamberchip(realNotes[j] - tunningSystem.Delta, 0);
        data.AddCentMamberchip(realNotes[j], 1);
        data.AddCentMamberchip(realNotes[j] + tunningSystem.Delta, 0);

        listFuzzyNotes.Add(data);
    }

    return listFuzzyNotes;
}



public static TuningScatter CreateTuningScatter(double[,] Data, int
    number)
{
    TuningScatter data = new TuningScatter();

    for(int i = 0; i<=Data.GetLength(0) - 1;i++)
    {
        data.addData(Data[i, 0], number);
    }
    return data;
}

public static SpectrumScatter CreateSpectrumScatter(double[,]
    spectrumData)
{
    SpectrumScatter data = new SpectrumScatter();
    data.ConvertDataToSpectrumRepresentation(spectrumData);
    return data;
}

public static WavScatter CreateWavScatter(double[] times, double[]
    samples)
{
    WavScatter data = new WavScatter();
    data.CreateData(times, samples);
    return data;
}

public static HarmonicScatter CreateHarmonicScatter(List<
    MusicalSymbolBase> notes)
{
```

```
List<double> bufPitches = new List<double>();
HarmonicScatter data = new HarmonicScatter();

if (notes != null)
{
    foreach (MusicalSymbolBase nota in notes)
    {
        if (nota.GetType() == typeof(Note))
        {

            if (((Note)nota).IsChordElement) //If note is part of
                the chord
            {

                //Acumulate the first pitch of the chord
                bufPitches.Add((double)(((Note)nota).MidiPitch));

            }
            else //First note of the chord (fundamental)
            {

                //If there are previous pitches, store the chord
                if (bufPitches.Count > 0)
                {
                    data.AddChord(bufPitches.ToArray()); //Store the
                        pitches of previous chord
                    bufPitches.Clear(); //Clear the variable
                }

                //Acumulate the first pitch of the chord
                bufPitches.Add((double)(((Note)nota).MidiPitch));

            };

        }
    }

    //Store the last one chord, if exists
    if (bufPitches.Count > 0)
    {
        data.AddChord(bufPitches.ToArray());
        bufPitches.Clear();
    }


    //Check that all the chords have the same number of notes
```

```
        if (!data.CheckEqualNumberOfDimensions()) { throw new
            ScatterInconsistentChordsException(); }


    }
    return data;
}

public static RhythmicalScatter CreateRhythmicalScatter(List<
    MusicalSymbolBase> notes)
{
    bool isAccented = false;
    int normalNotes = 1;
    int actualNotes = 1;
    int duration = 1;
    double totalDuration = 0;
    double bufLigaduraValor = 0;

    RhythmicalScatter data = new RhythmicalScatter();
    if (notes != null)
    {
        foreach (MusicalSymbolBase nota in notes)
        {
            if (nota.GetType() == typeof(Note))
            {
                //Get if the first note of any tied note is accented
                if ((((Note)nota).TieType == NoteTieType.None) || (((
                    Note)nota).TieType == NoteTieType.Start))
                {
                    if (((Note)nota).Articulation == ArticulationType.
                        Accent) { isAccented = true; } else { isAccented
                         = false; }
                }

                if (((Note)nota).NormalNotes == 0) { normalNotes = 1; }
                    else { normalNotes = ((Note)nota).NormalNotes; };
                if (((Note)nota).ActualNotes == 0) { actualNotes = 1; }
                    else { actualNotes = ((Note)nota).ActualNotes; };
                duration = (int)((Note)nota).Duration;
                totalDuration = DurationCalculator.TotalDuration(((Note)
                    nota).NumberOfDots, duration, normalNotes,
                    actualNotes);

                //Take into account the value tied notes...are
                    represented as one with the duration summatory of
                    all the tied notes.
```

```
                    if ((((Note)nota).TieType == NoteTieType.Start) || (((
                        Note)nota).TieType == NoteTieType.
                        StopAndStartAnother))
                    {
                        bufLigaduraValor = bufLigaduraValor + totalDuration;
                    }
                    else if ((((Note)nota).TieType == NoteTieType.None) ||
                        (((Note)nota).TieType == NoteTieType.Stop))
                    {
                        if (bufLigaduraValor != 0) { totalDuration =
                            totalDuration + bufLigaduraValor; }
                        data.AddRhythimcFigure(totalDuration, false,
                            isAccented);
                        bufLigaduraValor = 0;
                    };

                }
                if (nota.GetType() == typeof(Rest))
                {
                    if (((Rest)nota).NormalNotes == 0) { normalNotes = 1; }
                        else { normalNotes = ((Rest)nota).NormalNotes; };
                    if (((Rest)nota).ActualNotes == 0) { actualNotes = 1; }
                        else { actualNotes = ((Rest)nota).ActualNotes; };
                    duration = (int)((Rest)nota).Duration;
                    totalDuration = DurationCalculator.TotalDuration(((Rest)
                        nota).NumberOfDots, duration, normalNotes,
                        actualNotes);
                    data.AddRhythimcFigure(totalDuration, true, false);
                }
            }
        }
        return data;
    }
  }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Graphics.Scatter
{
    public abstract class ScatterNDBase
    {

        protected List<double[]> data = null; //points are stored into a list
            of n-dimensional array called "data"

        public double[,] GetData() //Convert the list of n-dimensional points
            into a matrix (number of points X number of dimensions)
        {
            int numberOfDimensions = data[0].GetLength(0);
            double[,] buf = null;
            if (data != null) {
                buf = new double[data.Count, numberOfDimensions];
                for (int i=0; i<=data.Count-1; i++)
                {
                    for (int d = 0; d <= numberOfDimensions - 1; d++)
                    {
                        buf[i, d] = data[i][d];
                    }
                }
            };
            return buf;
        }

        public bool CheckEqualNumberOfDimensions()
        {
            int dimensions = data[0].GetLength(0);
            foreach (double[] element in data)
            {
                if (dimensions != element.GetLength(0)) { return false; }
            }
            return true;
        }


        public override string ToString()
        {
            StringBuilder str = new StringBuilder();
```

```
            int numberOfDimensions = data[0].GetLength(0);
            foreach (double[] punto in data)
            {
                for (int d = 0; d <= numberOfDimensions - 1; d++)
                {
                    str.Append(punto[d]);
                    str.Append("\t");
                }
                str.AppendLine();
            }
            return str.ToString();
        }


    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Graphics.Scatter
{
    public class SpectrumScatter : Scatter2DBase
    {

        public SpectrumScatter() { }
        public void ConvertDataToSpectrumRepresentation(double[,] dataSpectrum)
        {
            data = new List<PointD>();;
            //Create a sparce matrix in order to represent a scatter chart of
                the spectrum with lines instead of splines
            for (int i = 0; i <= dataSpectrum.GetLength(0) - 1; i++)
            {
                data.Add(new PointD(0.99999 * dataSpectrum[i, 0], -140));
                data.Add(new PointD(dataSpectrum[i, 0], dataSpectrum[i, 1]));
                data.Add(new PointD(1.00001 * dataSpectrum[i, 0], -140));
            }

        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Graphics.Scatter
{
    public class TuningScatter : Scatter2DBase
    {
        public TuningScatter() { }

        public void addData(double y, int number)
        {
            if (data == null) { data = new List<PointD>(); };
            this.data.Add(new PointD(number, y));
        }
    }
}
```

A.8 Mercury.Calculations.Notes

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Mercury.Domain.MusicalSymbol;
using Mercury.Calculations.Duration;
using Mercury.Domain.MusicalSymbol.Enumerations;

namespace Mercury.Calculations.Notes
{

    public static class NotesCalculator
    {

        public static List<MusicalSymbolBase> GetNotes(SortedDictionary<double,
             List<RhythmicFigure>> PossibleDurations, List<double>
            PossiblePitchs, double[,] Data, int tempo)
        {
            List<MusicalSymbolBase> notes = new List<MusicalSymbolBase>();
            int pitchBuf = 0;
            double totalDuration = 0;

            for (int i = 0; i <= Data.GetLength(0) - 1; i++)
            {
                //Calculate the midiPitch
                pitchBuf = (int)GetCloserPitch(Data[i, 1], PossiblePitchs);
                //Calculate the rhythmic value
                List<RhythmicFigure> DurationsListBuf = null;
                DurationsListBuf = RhythmicFigureCalculator.GetCloserFigure(Data
                    [i,0], PossibleDurations);

                //Calculate the type of tie
                NoteTieType tieType = NoteTieType.None;
                if (DurationsListBuf.Count == 1)
                { tieType = NoteTieType.None; }
                else
                { tieType = NoteTieType.Start; }

                for (int j = 0; j <= DurationsListBuf.Count-1;j++)
                {
                    if ((DurationsListBuf.Count > 1) && (j > 0) && (j <
                        DurationsListBuf.Count - 1))
                    { tieType = NoteTieType.StopAndStartAnother; }
```

```
            else if ((DurationsListBuf.Count > 1) && (j ==
                DurationsListBuf.Count - 1))
            { tieType = NoteTieType.Stop; }

            Note note = new Note("C", 0, 0, (MusicalSymbolDuration)
                DurationsListBuf[j].duration, NoteStemDirection.Up,
                tieType, new List<NoteBeamType> {NoteBeamType.Single});
            note.MidiPitch = pitchBuf;
            note.NumberOfDots = (int)DurationsListBuf[j].dots;
            note.ActualNotes = (int)DurationsListBuf[j].actualNotes;
            note.NormalNotes = (int)DurationsListBuf[j].normalNotes;
            note.CurrentTempo = tempo;
            notes.Add(note);

            totalDuration = totalDuration + DurationCalculator.
                TotalDuration(note.NumberOfDots, (int)note.Duration,
                note.NormalNotes, note.ActualNotes);

        }
    }

    //Add final measure bar
    notes.Add(new Barline());

    //Add key, clef and time signature
    uint numberOfQuarters = (uint)(Math.Truncate(totalDuration / 0.25)
        + Math.Ceiling(totalDuration % 0.25));
    List<MusicalSymbolBase> buf = new List<MusicalSymbolBase>();
    buf.Add(new Key(0));
    buf.Add(new TimeSignature(TimeSignatureType.Numbers,
        numberOfQuarters, 4));
    buf.Add(new Clef(ClefType.GClef, 2));
    notes.InsertRange(0, buf);

    return notes;
}

public static List<MusicalSymbolBase> GetNotesRests(SortedDictionary<
    double, List<RhythmicFigure>> PossibleDurations, List<double>
    PossiblePitchs, double[,] Data, int tempo, double silenceThreshold
    )
{
    List<MusicalSymbolBase> notes = new List<MusicalSymbolBase>();
    int pitchBuf = 0;
    double totalDuration = 0;
```

```
for (int i = 0; i <= Data.GetLength(0) - 1; i++)
{
    //Calculate the midiPitch
    pitchBuf = (int)GetCloserPitch(Data[i, 1], PossiblePitchs);
    //Calculate the rhythmic value
    List<RhythmicFigure> DurationsListBuf = null;
    DurationsListBuf = RhythmicFigureCalculator.GetCloserFigure(Data
        [i, 0], PossibleDurations);

    //Calculate the type of tie
    NoteTieType tieType = NoteTieType.None;
    if (DurationsListBuf.Count == 1)
    { tieType = NoteTieType.None; }
    else
    { tieType = NoteTieType.Start; }

    for (int j = 0; j <= DurationsListBuf.Count - 1; j++)
    {
        if ((DurationsListBuf.Count > 1) && (j > 0) && (j <
            DurationsListBuf.Count - 1))
        { tieType = NoteTieType.StopAndStartAnother; }
        else if ((DurationsListBuf.Count > 1) && (j ==
            DurationsListBuf.Count - 1))
        { tieType = NoteTieType.Stop; }

        //if ((Data[i, 2] > 0.0) && (Math.Log10(Data[i, 2]) >
            silenceThreshold))
        if (Data[i, 2] > 0.5)
        {
            //REST
            Rest rest = new Rest((MusicalSymbolDuration)
                DurationsListBuf[j].duration);
            rest.NumberOfDots = (int)DurationsListBuf[j].dots;
            rest.ActualNotes = (int)DurationsListBuf[j].actualNotes;
            rest.NormalNotes = (int)DurationsListBuf[j].normalNotes;
            rest.CurrentTempo = tempo;
            notes.Add(rest);
            totalDuration = totalDuration + DurationCalculator.
                TotalDuration(rest.NumberOfDots, (int)rest.Duration,
                 rest.NormalNotes, rest.ActualNotes);
        }
        else
        {
            //NOTE
            Note note = new Note("C", 0, 0, (MusicalSymbolDuration)
                DurationsListBuf[j].duration, NoteStemDirection.Up,
```

```
                    tieType, new List<NoteBeamType> { NoteBeamType.
                    Single });
                note.MidiPitch = pitchBuf;
                note.NumberOfDots = (int)DurationsListBuf[j].dots;
                note.ActualNotes = (int)DurationsListBuf[j].actualNotes;
                note.NormalNotes = (int)DurationsListBuf[j].normalNotes;
                note.CurrentTempo = tempo;
                notes.Add(note);
                totalDuration = totalDuration + DurationCalculator.
                    TotalDuration(note.NumberOfDots, (int)note.Duration,
                     note.NormalNotes, note.ActualNotes);
            }


        }
    }

    //Add final measure bar
    notes.Add(new Barline());

    //Add key, clef and time signature
    uint numberOfQuarters = (uint)(Math.Truncate(totalDuration / 0.25)
        + Math.Ceiling(totalDuration % 0.25));
    List<MusicalSymbolBase> buf = new List<MusicalSymbolBase>();
    buf.Add(new Key(0));
    buf.Add(new TimeSignature(TimeSignatureType.Numbers,
        numberOfQuarters, 4));
    buf.Add(new Clef(ClefType.GClef, 2));
    notes.InsertRange(0, buf);

    return notes;
}

public static List<MusicalSymbolBase> GetNotesRestsAccents(
    SortedDictionary<double, List<RhythmicFigure>> PossibleDurations,
    List<double> PossiblePitchs, double[,] Data, int tempo)
{
    List<MusicalSymbolBase> notes = new List<MusicalSymbolBase>();
    int pitchBuf = 0;
    double totalDuration = 0;
    bool accented = false;

    for (int i = 0; i <= Data.GetLength(0) - 1; i++)
    {
        //Calculate the midiPitch
        pitchBuf = (int)GetCloserPitch(Data[i, 1], PossiblePitchs);
```

```
//Calculate the rhythmic value
List<RhythmicFigure> DurationsListBuf = null;
DurationsListBuf = RhythmicFigureCalculator.GetCloserFigure(Data
    [i, 0], PossibleDurations);

//Calculate the type of tie
NoteTieType tieType = NoteTieType.None;
if (DurationsListBuf.Count == 1)
{ tieType = NoteTieType.None; }
else
{ tieType = NoteTieType.Start; }

for (int j = 0; j <= DurationsListBuf.Count - 1; j++)
{
    if ((DurationsListBuf.Count > 1) && (j > 0) && (j <
        DurationsListBuf.Count - 1))
    { tieType = NoteTieType.StopAndStartAnother; }
    else if ((DurationsListBuf.Count > 1) && (j ==
        DurationsListBuf.Count - 1))
    { tieType = NoteTieType.Stop; }

    //Get if a note has to be accented or not
    accented = (Data[i, 3] > 0.5);

    if (Data[i, 2] > 0.5)
    {
        //REST
        Rest rest = new Rest((MusicalSymbolDuration)
            DurationsListBuf[j].duration);
        rest.NumberOfDots = (int)DurationsListBuf[j].dots;
        rest.ActualNotes = (int)DurationsListBuf[j].actualNotes;
        rest.NormalNotes = (int)DurationsListBuf[j].normalNotes;
        rest.CurrentTempo = tempo;
        notes.Add(rest);
        totalDuration = totalDuration + DurationCalculator.
            TotalDuration(rest.NumberOfDots, (int)rest.Duration,
             rest.NormalNotes, rest.ActualNotes);
    }
    else
    {
        //NOTE
        Note note = new Note("C", 0, 0, (MusicalSymbolDuration)
            DurationsListBuf[j].duration, NoteStemDirection.Up,
            tieType, new List<NoteBeamType> { NoteBeamType.
            Single });
        note.MidiPitch = pitchBuf;
```

```
                note.NumberOfDots = (int)DurationsListBuf[j].dots;
                note.ActualNotes = (int)DurationsListBuf[j].actualNotes;
                note.NormalNotes = (int)DurationsListBuf[j].normalNotes;
                note.CurrentTempo = tempo;
                if (accented) { note.Articulation = ArticulationType.
                    Accent; }
                notes.Add(note);
                totalDuration = totalDuration + DurationCalculator.
                    TotalDuration(note.NumberOfDots, (int)note.Duration,
                     note.NormalNotes, note.ActualNotes);
            }


        }
    }

    //Add final measure bar
    notes.Add(new Barline());

    //Add key, clef and time signature
    uint numberOfQuarters = (uint)(Math.Truncate(totalDuration / 0.25)
        + Math.Ceiling(totalDuration % 0.25));
    List<MusicalSymbolBase> buf = new List<MusicalSymbolBase>();
    buf.Add(new Key(0));
    buf.Add(new TimeSignature(TimeSignatureType.Numbers,
        numberOfQuarters, 4));
    buf.Add(new Clef(ClefType.GClef, 2));
    notes.InsertRange(0, buf);

    return notes;
}

public static List<MusicalSymbolBase> GetNotesRestsAccentsStaccato(
    SortedDictionary<double, List<RhythmicFigure>> PossibleDurations,
    List<double> PossiblePitchs, double[,] Data, int tempo)
{
    List<MusicalSymbolBase> notes = new List<MusicalSymbolBase>();
    int pitchBuf = 0;
    double totalDuration = 0;
    bool accented = false;
    bool staccato = false;

    try
    {

        for (int i = 0; i <= Data.GetLength(0) - 1; i++)
```

```
{
    //Calculate the midiPitch
    pitchBuf = (int)GetCloserPitch(Data[i, 1], PossiblePitchs);
    //Calculate the rhythmic value
    List<RhythmicFigure> DurationsListBuf = null;
    DurationsListBuf = RhythmicFigureCalculator.GetCloserFigure(
        Data[i, 0], PossibleDurations);

    //Calculate the type of tie
    NoteTieType tieType = NoteTieType.None;
    if (DurationsListBuf.Count == 1)
    { tieType = NoteTieType.None; }
    else
    { tieType = NoteTieType.Start; }

    for (int j = 0; j <= DurationsListBuf.Count - 1; j++)
    {
        if ((DurationsListBuf.Count > 1) && (j > 0) && (j <
            DurationsListBuf.Count - 1))
        { tieType = NoteTieType.StopAndStartAnother; }
        else if ((DurationsListBuf.Count > 1) && (j ==
            DurationsListBuf.Count - 1))
        { tieType = NoteTieType.Stop; }

        //Get if a note has to be accented or not
        accented = (Data[i, 3] > 0.5);

        //Get if a note has to be staccato or not
        staccato = (Data[i, 4] > 0.5);

        if (Data[i, 2] > 0.5)
        {
            //REST
            Rest rest = new Rest((MusicalSymbolDuration)
                DurationsListBuf[j].duration);
            rest.NumberOfDots = (int)DurationsListBuf[j].dots;
            rest.ActualNotes = (int)DurationsListBuf[j].
                actualNotes;
            rest.NormalNotes = (int)DurationsListBuf[j].
                normalNotes;
            rest.CurrentTempo = tempo;
            notes.Add(rest);
            totalDuration = totalDuration + DurationCalculator.
                TotalDuration(rest.NumberOfDots, (int)rest.
                Duration, rest.NormalNotes, rest.ActualNotes);
        }
```

```
            else
            {
                //NOTE
                Note note = new Note("C", 0, 0, (
                    MusicalSymbolDuration)DurationsListBuf[j].
                    duration, NoteStemDirection.Up, tieType, new
                    List<NoteBeamType> { NoteBeamType.Single });
                note.MidiPitch = pitchBuf;
                note.NumberOfDots = (int)DurationsListBuf[j].dots;
                note.ActualNotes = (int)DurationsListBuf[j].
                    actualNotes;
                note.NormalNotes = (int)DurationsListBuf[j].
                    normalNotes;
                note.CurrentTempo = tempo;

                if (accented & staccato) { note.Articulation =
                    ArticulationType.StaccatoAccent; }
                else if (!accented & staccato) { note.Articulation =
                    ArticulationType.Staccato; }
                else if (accented & !staccato) { note.Articulation =
                    ArticulationType.Accent; }
                else { note.Articulation = ArticulationType.None; }

                notes.Add(note);
                totalDuration = totalDuration + DurationCalculator.
                    TotalDuration(note.NumberOfDots, (int)note.
                    Duration, note.NormalNotes, note.ActualNotes);
            }


        }
    }

    //Add final measure bar
    notes.Add(new Barline());

    //Add key, clef and time signature
    uint numberOfQuarters = (uint)(Math.Truncate(totalDuration /
        0.25) + Math.Ceiling(totalDuration % 0.25));
    List<MusicalSymbolBase> buf = new List<MusicalSymbolBase>();
    buf.Add(new Key(0));
    buf.Add(new TimeSignature(TimeSignatureType.Numbers,
        numberOfQuarters, 4));
    buf.Add(new Clef(ClefType.GClef, 2));
    notes.InsertRange(0, buf);
```

```
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }

    return notes;
}


public static List<MusicalSymbolBase> GetChords(List<double>
    PossiblePitchs, double[,] Data, int tempo)
{
    List<MusicalSymbolBase> notes = new List<MusicalSymbolBase>();

    int pitchBuf = 0;

    for (int i = 0; i <= Data.GetLength(0) - 1; i++)
    {

        for (int d = 0; d <= Data.GetLength(1) - 1; d++) //Each note of
            the chord
        {

            //Calculate the midiPitch
            pitchBuf = (int)GetCloserPitch(Data[i, d], PossiblePitchs);

            Note note = new Note("C", 0, 0, MusicalSymbolDuration.Whole,
                 NoteStemDirection.Up, NoteTieType.None, new List<
                NoteBeamType> { NoteBeamType.Single });
            note.MidiPitch = pitchBuf;
            note.NumberOfDots = 0;
            note.ActualNotes = 1;
            note.NormalNotes = 1;
            note.CurrentTempo = tempo;
            if (d == 0) { note.IsChordElement = false; } else { note.
                IsChordElement = true; };
            notes.Add(note);

        }
        notes.Add(new Barline());
    }


    //Add key, clef and time signature
    List<MusicalSymbolBase> buf = new List<MusicalSymbolBase>();
```

```
    buf.Add(new Key(0));
    buf.Add(new TimeSignature(TimeSignatureType.Numbers, 4, 4));
    buf.Add(new Clef(ClefType.GClef, 2));
    notes.InsertRange(0, buf);

    return notes;
}

public static List<MusicalSymbolBase> GetChordsWithoutMeasureBars(List<
    double> PossiblePitchs, double[,] Data, int tempo,
    MusicalSymbolDuration Duration, int dots)
{
    List<MusicalSymbolBase> notes = new List<MusicalSymbolBase>();

    int pitchBuf = 0;
    double stepDuration = 0.5;
    double totalDuration = 0;
    switch(Duration)
    {
        case MusicalSymbolDuration.Whole:
            stepDuration = 1;
            break;
        case MusicalSymbolDuration.Half:
            stepDuration = 0.5;
            break;
        case MusicalSymbolDuration.Quarter:
            stepDuration = 0.25;
            break;
        case MusicalSymbolDuration.Eighth:
            stepDuration = 0.125;
            break;
        case MusicalSymbolDuration.Sixteenth:
            stepDuration = 0.0625;
            break;
        case MusicalSymbolDuration.d32nd:
            stepDuration = 0.03125;
            break;
   }



    for (int i = 0; i <= Data.GetLength(0) - 1; i++)
    {
        totalDuration = totalDuration + stepDuration;
        for (int d = 0; d <= Data.GetLength(1) - 1; d++) //Each note of
            the chord
```

```
        {

            //Calculate the midiPitch
            pitchBuf = (int)GetCloserPitch(Data[i, d], PossiblePitchs);

            Note note = new Note("C", 0, 0, Duration, NoteStemDirection.
                Up, NoteTieType.None, new List<NoteBeamType> {
                NoteBeamType.Single });
            note.MidiPitch = pitchBuf;
            note.NumberOfDots = dots;
            note.ActualNotes = 1;
            note.NormalNotes = 1;
            note.CurrentTempo = tempo;
            if (d == 0) { note.IsChordElement = false; } else { note.
                IsChordElement = true; };
            notes.Add(note);

        }

    }



    //Add final measure bar
    notes.Add(new Barline());

    //Add key, clef and time signature
    uint numberOfQuarters = (uint)(Math.Truncate(totalDuration / 0.25)
        + Math.Ceiling(totalDuration % 0.25));
    List<MusicalSymbolBase> buf = new List<MusicalSymbolBase>();
    buf.Add(new Key(0));
    buf.Add(new TimeSignature(TimeSignatureType.Numbers,
        numberOfQuarters, 4));
    buf.Add(new Clef(ClefType.FClef, 4));
    notes.InsertRange(0, buf);

    return notes;
}


public static List<MusicalSymbolBase> GetRhythms(SortedDictionary<
    double, List<RhythmicFigure>> PossibleDurations, double[,] Data,
    int tempo)
{
    List<MusicalSymbolBase> notes = new List<MusicalSymbolBase>();
    bool isAccented = false;
```

```
bool isRest = false;
NoteTieType tieType = NoteTieType.None;
double totalDuration = 0;

for (int i = 0; i <= Data.GetLength(0) - 1; i++)
{
    //Calculate if the note is rest
    if (Data[i, 1] >= 0.5) { isRest = true; } else { isRest = false;
         }

    //Calculate if the note is accented
    if (Data[i, 2] >= 0.5) { isAccented = true; } else { isAccented
        = false; }

    //Calculate the rhythmic value
    List<RhythmicFigure> DurationsListBuf = null;
    DurationsListBuf = RhythmicFigureCalculator.GetCloserFigure(Data
        [i, 0], PossibleDurations);

    if (!isRest)
    {
        //NOTE
        //Calculate the type of tie
        tieType = NoteTieType.None;
        if (DurationsListBuf.Count == 1)
        { tieType = NoteTieType.None; }
        else
        { tieType = NoteTieType.Start; }

        for (int j = 0; j <= DurationsListBuf.Count - 1; j++)
        {
            if ((DurationsListBuf.Count > 1) && (j > 0) && (j <
                DurationsListBuf.Count - 1))
            { tieType = NoteTieType.StopAndStartAnother; }
            else if ((DurationsListBuf.Count > 1) && (j ==
                DurationsListBuf.Count - 1))
            { tieType = NoteTieType.Stop; }

            Note note = new Note("C", 0, 0, (MusicalSymbolDuration)
                DurationsListBuf[j].duration, NoteStemDirection.Up,
                tieType, new List<NoteBeamType> { NoteBeamType.
                Single });
            note.MidiPitch = 60;
            note.NumberOfDots = (int)DurationsListBuf[j].dots;
            note.ActualNotes = (int)DurationsListBuf[j].actualNotes;
            note.NormalNotes = (int)DurationsListBuf[j].normalNotes;
```

```
            note.CurrentTempo = tempo;

            //If the note is the first, and it sshould be accented,
                we will mark as accented the firts one of all ties
                notes
            if ((j == 0) && (isAccented)) { note.Articulation =
                ArticulationType.Accent; }

            notes.Add(note);
            totalDuration = totalDuration + DurationCalculator.
                TotalDuration(note.NumberOfDots, (int)note.Duration,
                 note.NormalNotes, note.ActualNotes);
        }
    }
    else
    {
        //REST
        for (int j = 0; j <= DurationsListBuf.Count - 1; j++)
        {
            Rest rest = new Rest( (MusicalSymbolDuration)
                DurationsListBuf[j].duration);
            rest.NumberOfDots = (int)DurationsListBuf[j].dots;
            rest.ActualNotes = (int)DurationsListBuf[j].actualNotes;
            rest.NormalNotes = (int)DurationsListBuf[j].normalNotes;
            rest.CurrentTempo = tempo;
            notes.Add(rest);
            totalDuration = totalDuration + DurationCalculator.
                TotalDuration(rest.NumberOfDots, (int)rest.Duration,
                 rest.NormalNotes, rest.ActualNotes);
        }
    }

}
//Add final measure bar
notes.Add(new Barline());

//Add key, clef and time signature
uint numberOfQuarters = (uint)(Math.Truncate(totalDuration / 0.25)
    + Math.Ceiling(totalDuration % 0.25));
List<MusicalSymbolBase> buf = new List<MusicalSymbolBase>();
buf.Add(new Key(0));
buf.Add(new TimeSignature(TimeSignatureType.Numbers,
    numberOfQuarters, 4));
buf.Add(new Clef(ClefType.GClef, 2));
notes.InsertRange(0, buf);
```

```
            return notes;
        }

        public static double GetCloserPitch(double pitch, List<double>
            PossiblePitchs)
        {
            int i = 0;
            double[] pitchs = PossiblePitchs.ToArray();
            double buf = Math.Abs(pitch - pitchs[0]);
            while ((i <= pitchs.Length-1) && (buf >= Math.Abs(pitch - pitchs[i
                ])))
            {
                buf = Math.Abs(pitch - pitchs[i]);
                i = i + 1;
            }
            return pitchs[i-1];
        }

    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Mercury.Domain.MusicalSymbol;
using Mercury.Calculations.Duration;
using Mercury.Domain.MusicalSymbol.Enumerations;

namespace Mercury.Calculations.Notes
{

    public static class PossibleMIDIPitchs
    {
        private static List<double> _possibleMidiPitchs = null;
        public static List<double> GetPossibleMidiPitchs()
        {
            if (_possibleMidiPitchs == null)
            {
                _possibleMidiPitchs = new List<double>();
                for (int i = 0; i <= 127; i++) { _possibleMidiPitchs.Add((double
                    )i); }
            }
            return _possibleMidiPitchs;
        }
    }
}
```

A.9 Mercury.Calculations.Tunings

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Tunings
{
    [Serializable]
    public class EqualTemperament : FuzzyTuningBase
    {
        public EqualTemperament() { }
        public EqualTemperament(string Name) : base(Name) { }
        protected override double[] GetNotes()
        {
            double[] buf = new double[13] {0, 100, 200, 300, 400, 500, 600,
                700, 800, 900, 1000, 1100, 1200};
            return buf;
        }

        public override FuzzyTuningBase Clone()
        {
            EqualTemperament buf = new EqualTemperament();
            return buf;
        }
    }
}
```

```
using System;
using System.Xml.Serialization;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Tunings
{
    [XmlInclude(typeof(EqualTemperament))]
    [XmlInclude(typeof(JustIntonation))]
    [XmlInclude(typeof(KellnerBach))]
    [XmlInclude(typeof(Mesotonic14))]
    [XmlInclude(typeof(Pythagorian))]
    [Serializable]
    public abstract class FuzzyTuningBase
    {
        public double Delta = 50;
        public double[] Notes
        {
            get { return GetNotes(); }
        }
        public string Name;
        public FuzzyTuningBase() { }
        public FuzzyTuningBase(string Name) { this.Name = Name; }
        protected abstract double[] GetNotes();
        public override string ToString()
        {
            return this.Name;
        }
        public double CalculateMembership(double Note, double x)
        {
            double buf = 0;
            if (Math.Abs(Note - x) < Delta)
            {
                buf = 1 - Math.Abs(Note - x) / Delta;
            }
            else
            {
                buf = 0;
            }
            return buf;
        }
        public abstract FuzzyTuningBase Clone();
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Tunings
{
    [Serializable]
    public class JustIntonation : FuzzyTuningBase
    {
        public JustIntonation() { }
        public JustIntonation(string Name) : base(Name) { }
        protected override double[] GetNotes()
        {
            double[] buf = new double[13] {0, 71, 204, 316, 386, 498, 590, 702,
                 773, 884, 1018, 1088, 1200};
            return buf;
        }

        public override FuzzyTuningBase Clone()
        {
            JustIntonation buf = new JustIntonation();
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Tunings
{
    [Serializable]
    public class KellnerBach : FuzzyTuningBase
    {
        public KellnerBach() { }
        public KellnerBach(string Name) : base(Name) { }
        protected override double[] GetNotes()
        {
            double[] buf = new double[13] {0, 90, 195, 294, 389, 498, 588, 697,
                 792, 892, 996, 1091, 1200};
            return buf;
        }

        public override FuzzyTuningBase Clone()
        {
            KellnerBach buf = new KellnerBach();
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Tunings
{
    [Serializable]
    public class Mesotonic14 : FuzzyTuningBase
    {
        public Mesotonic14() { }
        public Mesotonic14(string Name) : base(Name) { }
        protected override double[] GetNotes()
        {
            double[] buf = new double[13] {0, 76, 193, 310, 386, 503, 579, 697,
                 773, 890, 1007, 1083, 1200};
            return buf;
        }

        public override FuzzyTuningBase Clone()
        {
            Mesotonic14 buf = new Mesotonic14();
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Tunings
{
    [Serializable]
    public class Pythagorian : FuzzyTuningBase
    {
        public Pythagorian() { }
        public Pythagorian(string Name) : base(Name) { }
        protected override double[] GetNotes()
        {
            double[] buf = new double[13] {0, 114, 204, 294, 408, 498, 612,
                702, 816, 906, 996, 1110, 1200};
            return buf;
        }

        public override FuzzyTuningBase Clone()
        {
            Pythagorian buf = new Pythagorian();
            return buf;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Mercury.Calculations.Tunings
{
    [Serializable]
    public class Zarlino : FuzzyTuningBase
    {
        public Zarlino() { }
        public Zarlino(string Name) : base(Name) { }
        protected override double[] GetNotes()
        {
            double[] buf = new double[19] {0, 70.67, 133.24, 203.94, 274.58,
                315.64, 386.31, 456.99, 498.04, 568.72, 631.28, 701.96, 772.63,
                 813.69, 884.36, 955.03, 1017.60, 1088.27, 1158.94};
            return buf;
        }

        public override FuzzyTuningBase Clone()
        {
            Zarlino buf = new Zarlino();
            return buf;
        }
    }
}
```

A.10 Mercury.Calculations.FundamentalFrequency

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Mercury.Calculations.FFT;

namespace Mercury.Calculations.FundamentalFrequency
{
    public static class BiggerDerivative
    {

        public static double Calculate(double[] spectrum, double[] Hz, byte
            numberOfOvertones, double[] HPS)
        {
            double fundamentalFrequency = 0;
            byte R = numberOfOvertones;
            HPS = null;
            List<double[]> spectrums = new List<double[]>();
            spectrums.Add(LeaveOnlyMaxPoints(Hz, RemoveLowFrequencies(spectrum,
                Hz)));

            for (byte i = 2; i <= R; i++)
            {
                spectrums.Add(Scale(i, spectrums[0]));
            }

            HPS = new double[spectrums[0].Length];
            for (int i = 0; i <= HPS.Length - 1; i++)
            {
                HPS[i] = spectrums[0][i];
            }


            for (int i = 0; i <= HPS.Length - 1; i++)
            {
                for (int j = 0; j <= spectrums.Count - 1; j++)
                {
                    HPS[i] = HPS[i] * spectrums[j][i];
                }
            }


            for (int i = 0; i <= HPS.Length - 1; i++)
```

```
    {
        if (HPS[i] > 0)
        {
            HPS[i] = Math.Log10(HPS[i]);
        }
        else
        {
            HPS[i] = 0;
        }
    }

    for (int i = 0; i <= HPS.Length - 1; i++)
    {
        if (HPS[i] > 0)
        {
            fundamentalFrequency=Hz[i];
            break;
        }
    }


    return fundamentalFrequency;
}

private static double[] Scale(byte factor, double[] spectrum)
{
    double[] buf = new double[spectrum.Length];
    int resto = spectrum.Length % (int)factor;
    double max = 0;
    int k = 0;

    //Initialize with zero
    for (int i = 0; i <= buf.Length - 1; i++)
    {
        buf[i] = 0;
    }


    k = 0;
    for (int i = 0; i <= buf.Length - resto - 1; i = i + factor)
    {
        //Calculate the maximum of the values to redux in the scaling
            process
        for (int j = 0; j <= factor - 1; j++)
        {
            if((j==0)|(spectrum[i+j] > max))
```

```
            {
                max = spectrum[i+j];
            }
        }
        buf[k] = max;
        k++;
    }

    //Calculate the rest in case they exist
    for (int i = buf.Length - resto; i<=buf.Length - 1; i++)
    {
            if ((i == buf.Length - resto) | (spectrum[i] > max))
            {
                max = spectrum[i];
            }

        buf[k] = max;
    }

    return buf;
}

private static double[] RemoveLowFrequencies(double[] spectrum, double
    [] Hz)
{
    double[] buf = new double[spectrum.Length];
    double threesold = 60;

    if (Hz.Length != spectrum.Length) { throw new Exception("
        Inconsistent arrays"); }
    for (int i = 0; i <= Hz.Length - 1; i++)
    {
        if (Hz[i] <= threesold)
        {
            buf[i] = 0;
        }
        else
        {
            buf[i] = spectrum[i];
        }
    }
    return buf;
}

private static double[] LeaveOnlyMaxPoints(double[] x, double[] y)
{
```

```
double[] Max = null;
double[] dYdX = null;

if (x.Length != y.Length) { throw new Exception("Inconsistent
    arrays to calculate the differenciation"); }
if (x.Length < 2) { throw new Exception("Array too small for
    differenciate"); }

 Max = new double[y.Length];
 dYdX = Mercury.Calculations.Analysis.Derivative.Calculate(x, y);
 if (x.Length != dYdX.Length) { throw new Exception("Inconsistent
     arrays to calculate the differenciation"); }

//Initialize to zero
for (int i = 0; i <= y.Length - 1; i++)
{
    Max[i] = 0;
}

//Calculate the average diff
double averageDiff = 0;
for (int i = 0; i <= dYdX.Length - 1; i++)
{
    averageDiff = averageDiff + Math.Abs(dYdX[i]);
}
averageDiff = averageDiff / dYdX.Length;

for (int i = 0; i <= dYdX.Length - 1; i++)
{
    //Si la derivada cambia de positiva a negativa, es un máximo
    if ((i < dYdX.Length - 2) && (i > 1))
    {
        if ((Math.Sign(dYdX[i]) == 1) && (Math.Sign(dYdX[i + 1]) ==
            -1) && (Math.Abs(dYdX[i]) + Math.Abs(dYdX[i + 1]) >= 0.2
             * averageDiff))
        {
            if(i!=0)
            {
                Max[i + 1] = y[i + 1];

                ////Lets keep the values in a sourronding of position
                     i
                //for (int k = 1; k <= 2 * entorno; k++)
                //{
                //   if ((i - entorno + k <= y.Length - 1) && (i -
                    entorno + k >= 0))
```

```
                        //    {
                        //        Max[i - entorno + k] = y[i - entorno + k];
                        //    }
                        //}
                    }
                }
            }
            else
            {
                if (i == x.Length - 1)
                {
                    if ((Math.Sign(dYdX[i - 1]) == 1) && (Math.Sign(dYdX[i])
                        == -1))
                    {
                        Max[i] = y[i];
                    }
                }
            }
        }
        return Max;
    }
  }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Mercury.Calculations.FFT;

namespace Mercury.Calculations.FundamentalFrequency
{
    public static class HarmonicProductSpectrum
    {
        public static double Calculate(double[] spectrum, double[] Hz, byte
            numberOfOvertones)
        {
            double[] buf = new double[0];
            return Calculate(spectrum, Hz, numberOfOvertones, ref buf);

        }
        public static double Calculate(double[] spectrum, double[] Hz, byte
            numberOfOvertones, ref double[] HPS)
        {
            byte R = numberOfOvertones;
            HPS = null;
            List<double[]> spectrums = new List<double[]>();
            //spectrums.Add(LeaveOnlyMaxPoints(Hz, RemoveLowFrequencies(
                spectrum, Hz)));
            spectrums.Add(RemoveLowFrequencies(spectrum, Hz));

            for (byte i = 2; i <= R; i++)
            {
                spectrums.Add(Scale(i, spectrums[0]));
            }

            HPS = new double[spectrums[0].Length];
            for (int i = 0; i <= HPS.Length - 1; i++)
            {
                HPS[i] = spectrums[0][i];
            }

            for (int i = 0; i <= HPS.Length - 1; i++)
            {
                for (int j = 0; j <= spectrums.Count - 1; j++)
                {
                    HPS[i] = HPS[i] * spectrums[j][i];
                }
            }
```

```
    //Get the maximun value index
    int maxValueIndex = 0;
    double maxBuf = HPS[0];
    for (int i = 0; i <= HPS.Length - 1; i++)
    {
        if (HPS[i] > maxBuf)
        {
            maxBuf = HPS[i];
            maxValueIndex = i;
        }
    }

    //Show the results in log form
    for (int i = 0; i <= HPS.Length - 1; i++)
    {
        if (HPS[i] > 0)
        {
            HPS[i] = Math.Log10(HPS[i]);
        }
        else
        {
            HPS[i] = 0;
        }
    }
    return Hz[maxValueIndex];
}

private static double[] Scale(byte factor, double[] spectrum)
{
    double[] buf = new double[spectrum.Length];
    int resto = spectrum.Length % (int)factor;
    double max = 0;
    int k = 0;

    //Initialize with zero
    for (int i = 0; i <= buf.Length - 1; i++)
    {
        buf[i] = 0;
    }

    k = 0;
    for (int i = 0; i <= buf.Length - resto - 1; i = i + factor)
    {
        //Calculate the maximum of the values to redux in the scaling
            process
```

```
            for (int j = 0; j <= factor - 1; j++)
            {
                if((j==0)|(spectrum[i+j] > max))
                {
                    max = spectrum[i+j];
                }
            }
            buf[k] = max;
            k++;
        }

        //Calculate the rest in case they exist
        for (int i = buf.Length - resto; i<=buf.Length - 1; i++)
        {
               if ((i == buf.Length - resto) | (spectrum[i] > max))
               {
                   max = spectrum[i];
               }

            buf[k] = max;
        }
        return buf;
    }
    private static double[] RemoveLowFrequencies(double[] spectrum, double
         [] Hz)
    {
        double[] buf = new double[spectrum.Length];
        double threesold = 60;
        if (Hz.Length != spectrum.Length) { throw new Exception("
            Inconsistent arrays"); }
        for (int i = 0; i <= Hz.Length - 1; i++)
        {
            if (Hz[i] <= threesold)
            {
                buf[i] = 0;
            }
            else
            {
                buf[i] = spectrum[i];
            }
        }
        return buf;
    }
  }
}
```

Apéndice B

Transiciones difusas

B.1 Proceso compositivo

La obra transiciones difusas, para cuarteto de cuerdas, se divide en tres movimientos. El material utilizado para la composición de la obra procede de varias transiciones completas realizadas mediante MERCURY. En cada uno de los tres movimientos se ha utilizado un tipo de transición distinta: en el primer movimiento se han utilizado transiciones melódicas, en el segundo movimiento transiciones armónicas y rítmicas, y en el tercer movimiento transiciones exclusivamente rítmicas. A continuación describiremos el proceso mediante el cual se ha generado el material musical.

El material musical básico procede de la siguiente serie de doce notas, que expresadas como un conjunto de *pitch class* queda de la siguiente manera $\{3, 5, 2, 7, 1, 9, 10, 8, 11, 6, 0, 4\}$. En la siguiente figura podemos ver su representación musical

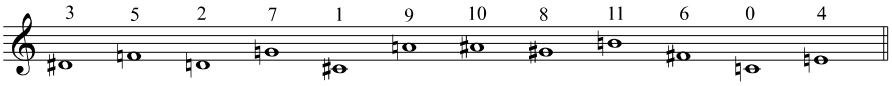


Figura B.1: Serie de doce notas principal.

B.1.1 Primer movimiento

Para generar el tema melódico principal, construiremos la siguiente serie de doce valores rítmicos $\{6, 3, 8, 2, 10, 11, 9, 12, 7, 1, 5, 4\}$, procedente directamente de una permutación de la serie anterior. La unidad mínima 1 es equivalente a una semicorchea.

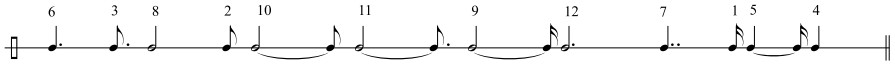


Figura B.2: Serie de doce valores rítmicos.

De la combinación de la series de doce notas y doce valores rítmicos surgirá el tema principal B.

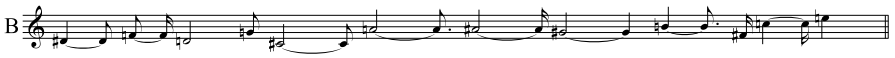


Figura B.3: Tema B.

Por otra parte, el tema A se construirá mediante el intervalo de quinta disminuida o cuarta aumentada, sobre las notas *La* - *Re*♯, y con una figuración rítmica sencilla y libre.

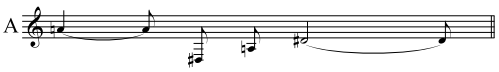


Figura B.4: Tema A.

Ejemplo B.1.1 *Transición entre el tema A y el tema B, con los parámetros de* fuzzy coefficient $\lambda = 2$*, función de vecindad gaussiana y métrica euclídea.*

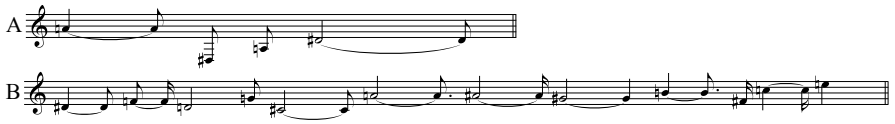


Figura B.5: Transición inicial entre el Tema A y B.

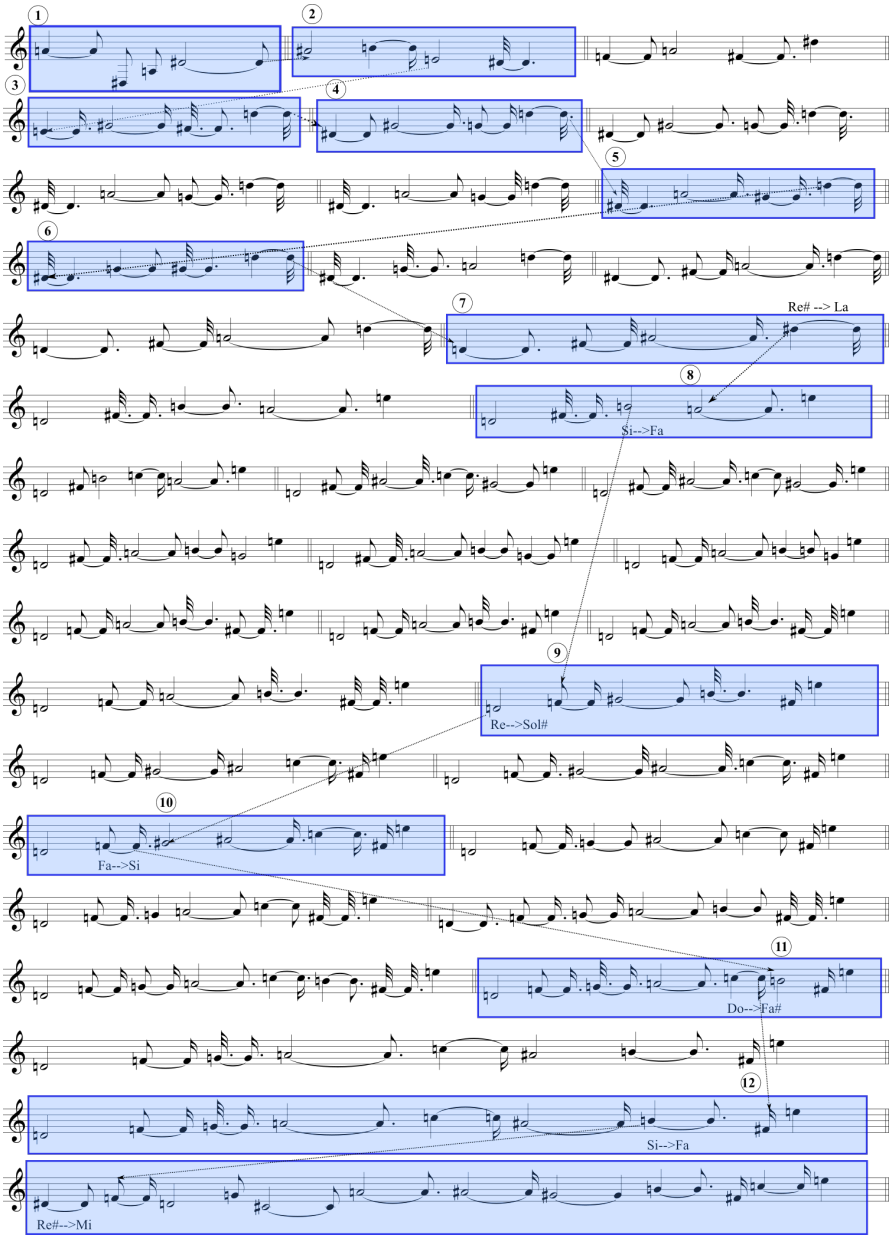


Figura B.6: Material utilizado de la ransición inicial entre el Tema A y B.

Ejemplo B.1.2 *Transición entre el tema A y el tema B, ambos retrogradados previamente, con los parámetros de* fuzzy coefficient $\lambda = 2$*, función de vecindad gaussiana y métrica euclídea.*

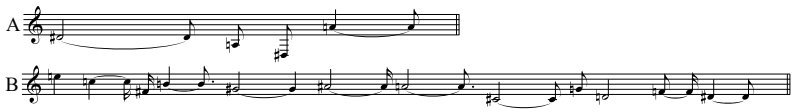


Figura B.7: Temas A y B retrogradados.

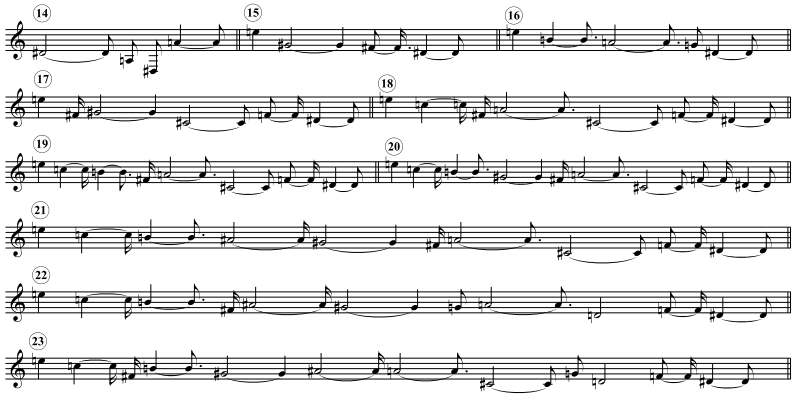


Figura B.8: Resultados de la transición entre los temas A y B retrogradados.

El movimiento se construye como una pequeña forma ternaria A_1BA_2. En la sección A_1 se utilizará una selección del material musical generado por la transición, de forma polifónica, de tal manera que las melodías estén relacionadas siempre por el intervalo del tritono. En la sección A_2 se concatenarán las melodías generadas mediante la segunda transición, pero dispuestas en orden inverso, hasta llegar nuevamente al tema A. La sección B expone la siguiente secuencia armónica, con acordes procedentes de la verticalización de la serie de notas principal.

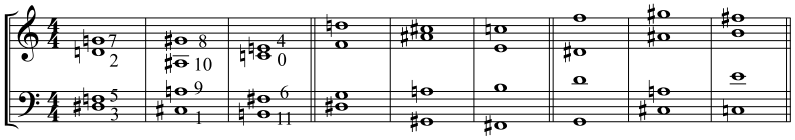


Figura B.9: Acordes utilizados en la sección B.

B.1.2 Segundo movimiento

El segundo movimiento constituye un coral en el que la armonía utilizada ha sido generada mediante una transición entre la armonía A y B. La armonía B consiste en una serie de doce acordes de cuatro notas construidos mediante cuatro tipos de agrupaciones de notas consecutivas de la serie de notas principales, tal y como se muestra en la siguiente figura:

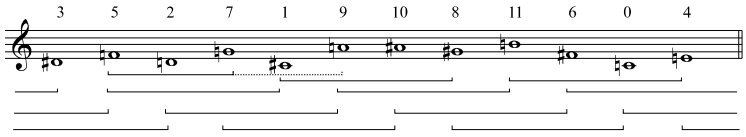


Figura B.10: Distintos agrupamientos de cuatríadas sobre la serie.

Por motivos estrictamente musicales decidimos que el primer acorde no se corresponda con las primeras cuatro notas de la serie, sino que se forme mediante las notas número dos, tres, cuatro y seis, constituyendo así una excepción a la construcción generalizada del resto de acordes, que sí se han formado utilizando cuatro notas consecutivas de la serie inicial. Los acordes resultantes son los siguientes:

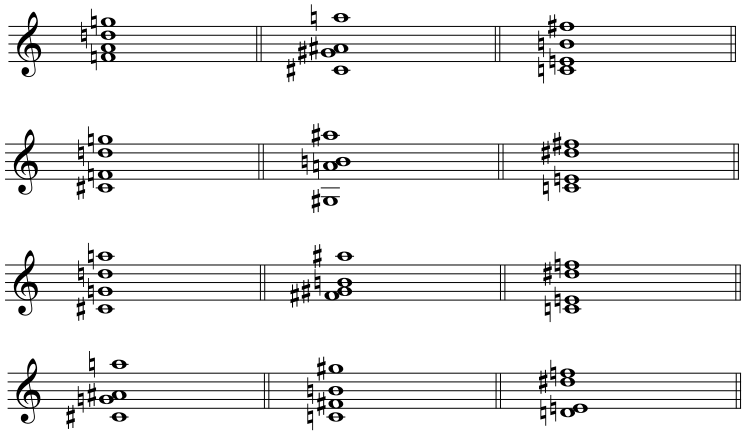


Figura B.11: Doce acordes resultantes.

La disposición de las voces en los doce acordes ha sido calculada de tal manera que se minimiza la distancia en la que cada voz se mueve. Por otra parte, los acordes de tipo A se construyen mediante el intervalo de tritono entre las notas *La* - *Re*♯, utilizando diferentes transposiciones.

Ejemplo B.1.3 *Transición entre el los acordes iniciales A y secuencia armónica final B, con los parámetros de* fuzzy coefficient $\lambda = 6$*, función de vecindad gaussiana y métrica* chord*.*

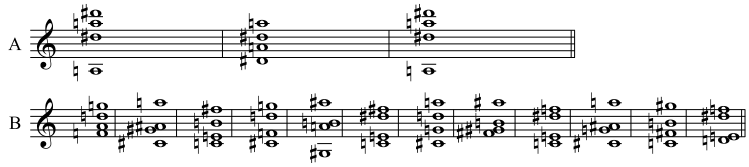


Figura B.12: Armonías A y B.

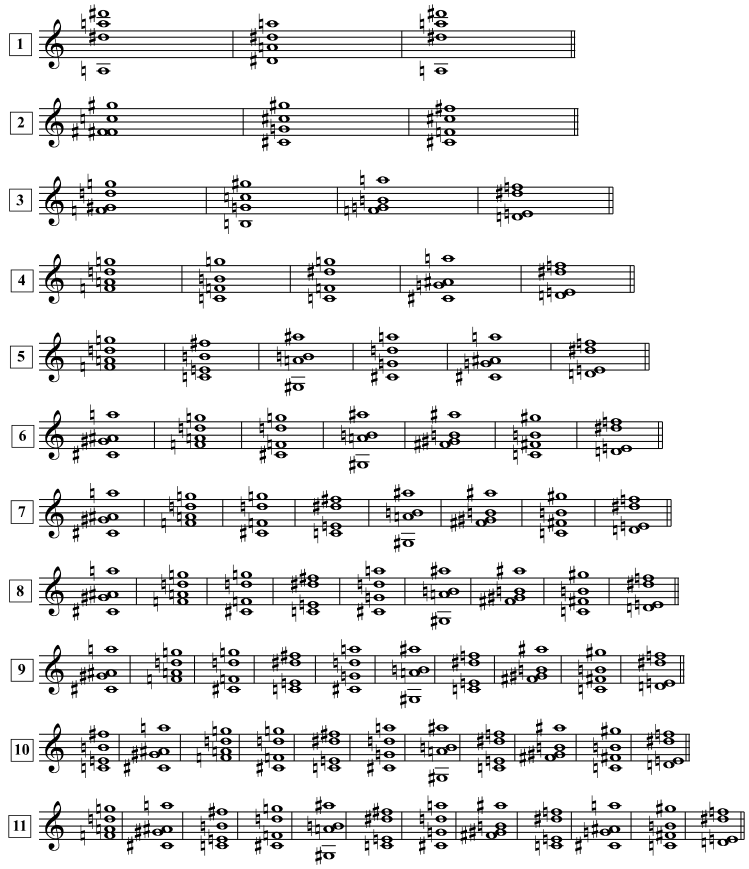


Figura B.13: Resultado de la transición entre las armonías A y B.

Además, para la figuración rítmica del coral se ha utilizado una transición ríntimca entre dos ritmos procedentes de los decîtalas hindÃºes, tal y como se muestra a continuación.

Ejemplo B.1.4 *Transición entre el los ritmos inicial A y final B (*simhavikrîdita*), con los parámetros de* fuzzy coefficient $\lambda = 5$*, función de vecindad gaussiana y métrica* chord*, utilizados para la figuración rítmica del segundo movimiento.*

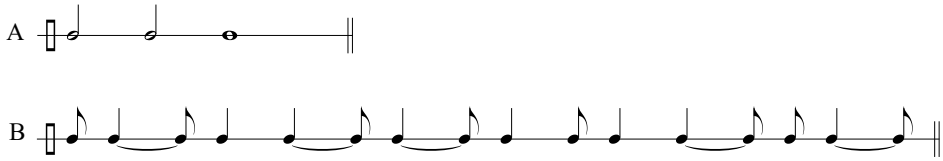


Figura B.14: Ritmos A y B (*simhavikrîdita*).

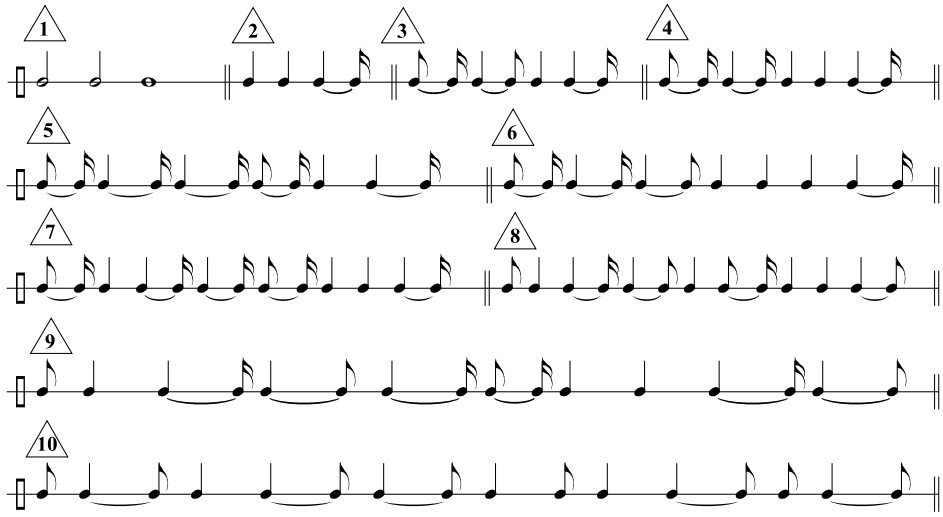


Figura B.15: Resultado de la transición entre los ritmos A y B.

B.1.3 Tercer movimiento

Ejemplo B.1.5 *Transición entre el los ritmos inicial A (*miçra varna*) y final B (*caccarî*), con los parámetros de* fuzzy coefficient $\lambda = 7$*, función de vecindad gaussiana y métrica* chord*, utilizados para la figuración rítmica del tercer movimiento.*

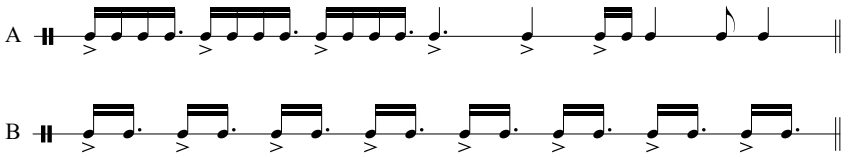


Figura B.16: Ritmos A (*miçra varna*) y B (*caccarî*).

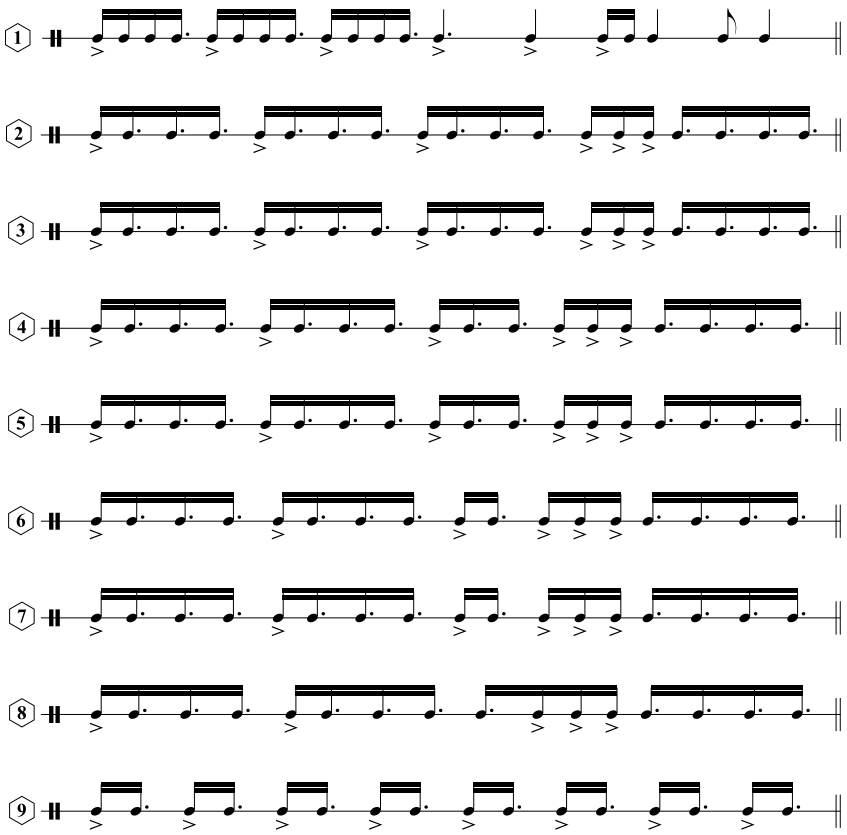


Figura B.17: Resultados de la transición entre los ritmos A y B.

B.2 Enlaces de escucha online

La obra *Transiciones difusas*, para cuarteto de cuerdas, puede escucharse *online* en los siguientes enlaces:

- Primer movimiento. *Andante*

  `https://soundcloud.com/briancomposer/transiciones-difusas-i`

- Segundo movimiento. *Corale*

  `https://soundcloud.com/briancomposer/transiciones-difusas-ii`

- Tercer movimiento. *Presto con fuoco*

  `https://soundcloud.com/briancomposer/transiciones-difusas-iii`

B.3 Partitura de la obra

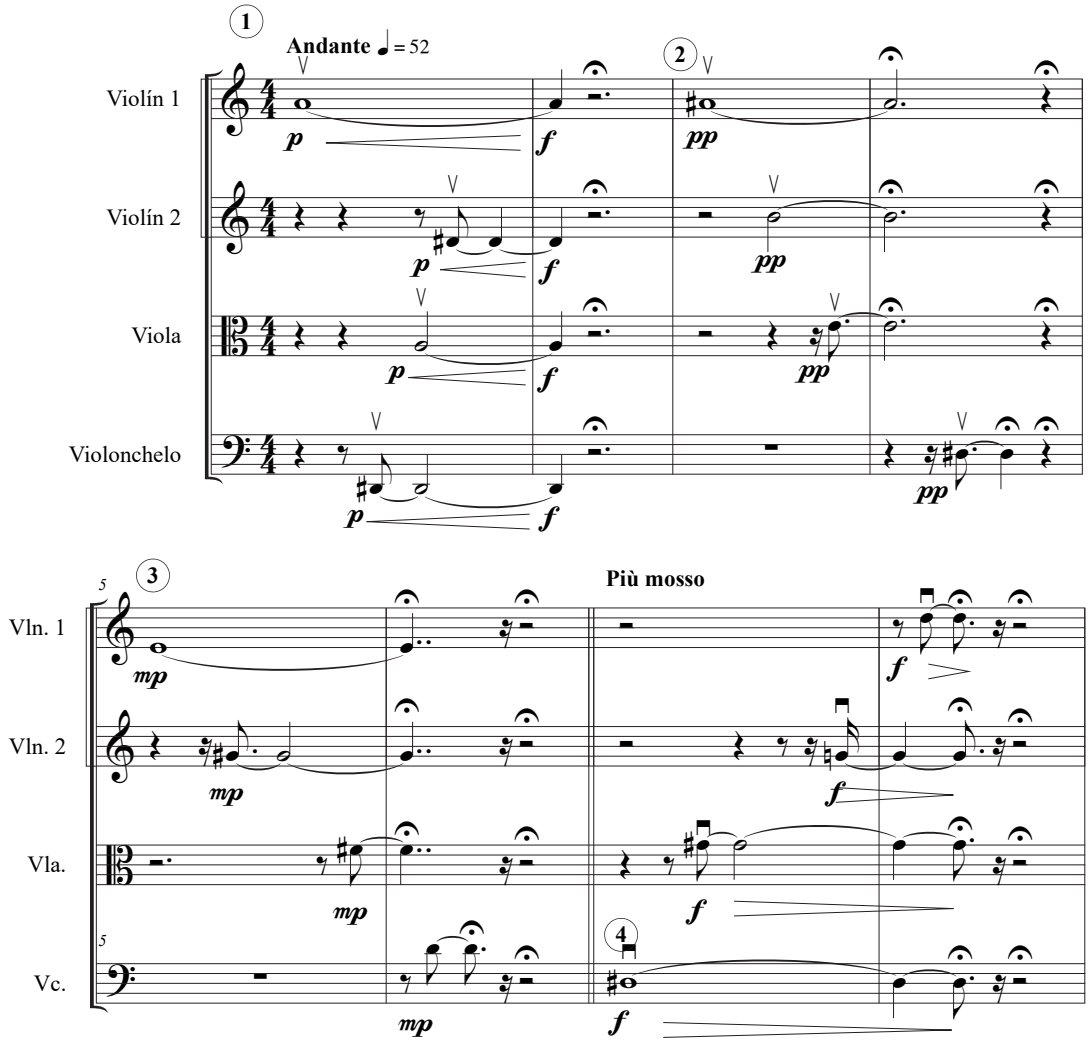
Transiciones difusas

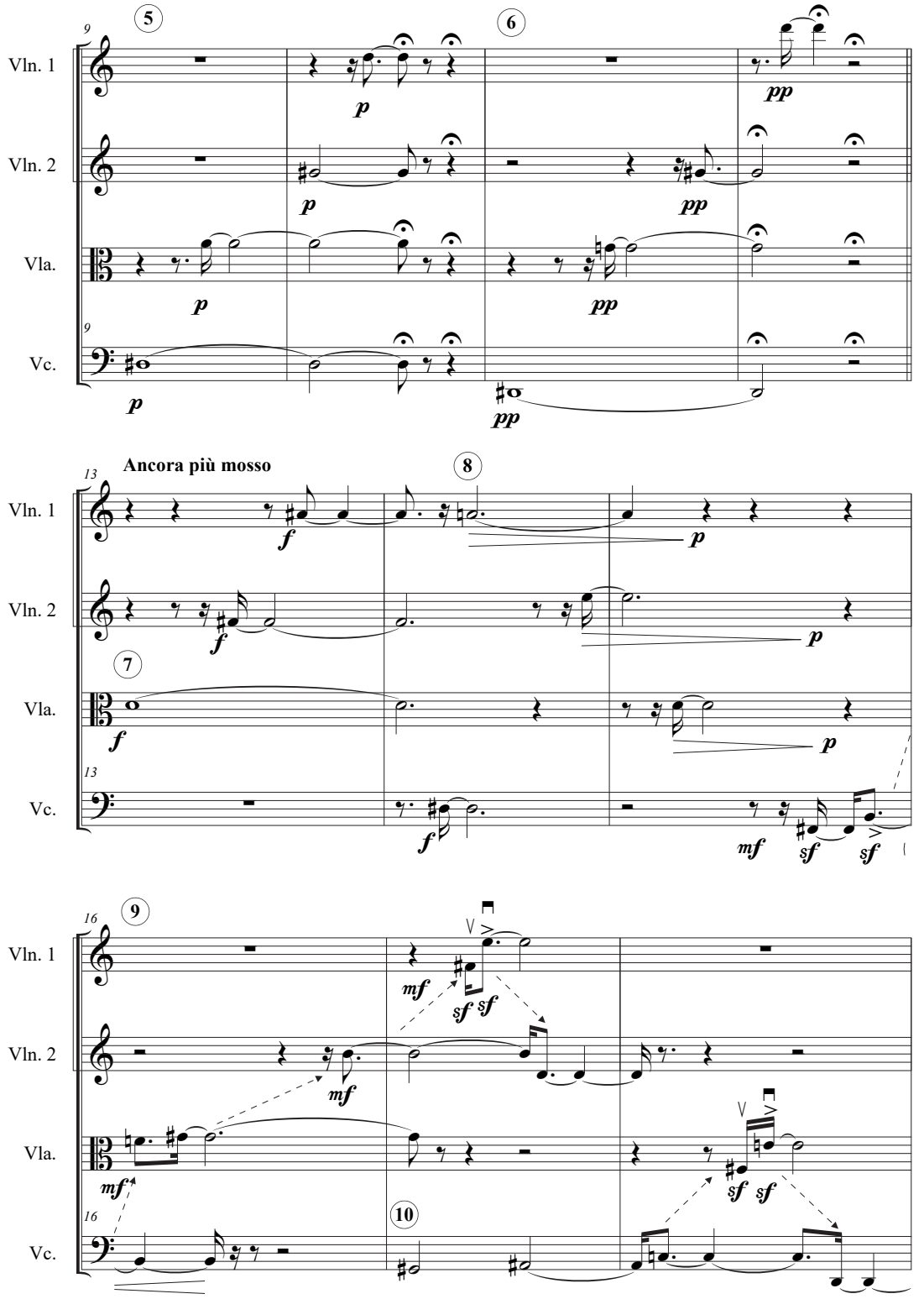
para cuarteto de cuerdas

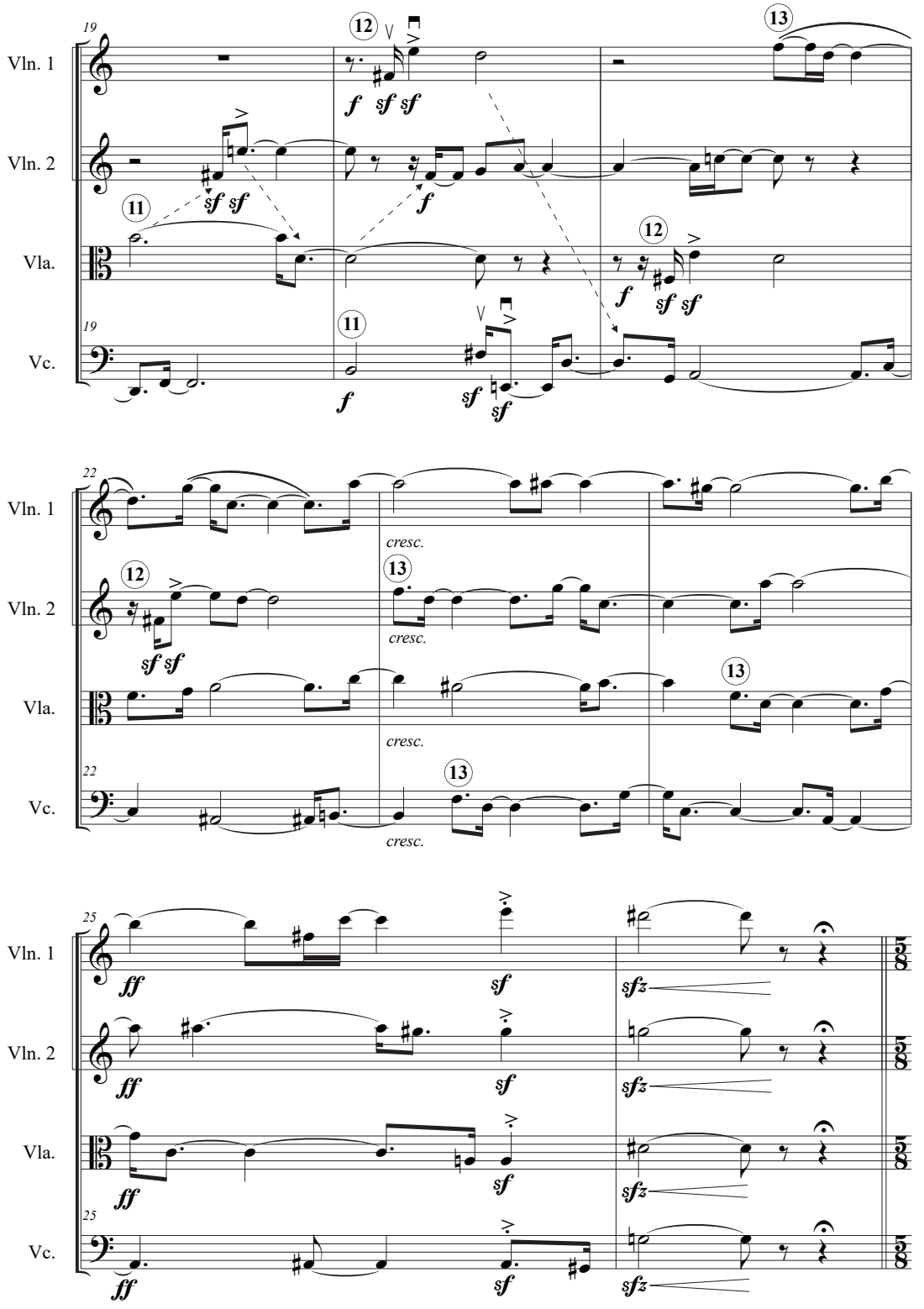
dedicada a Vicente Liern Carrión

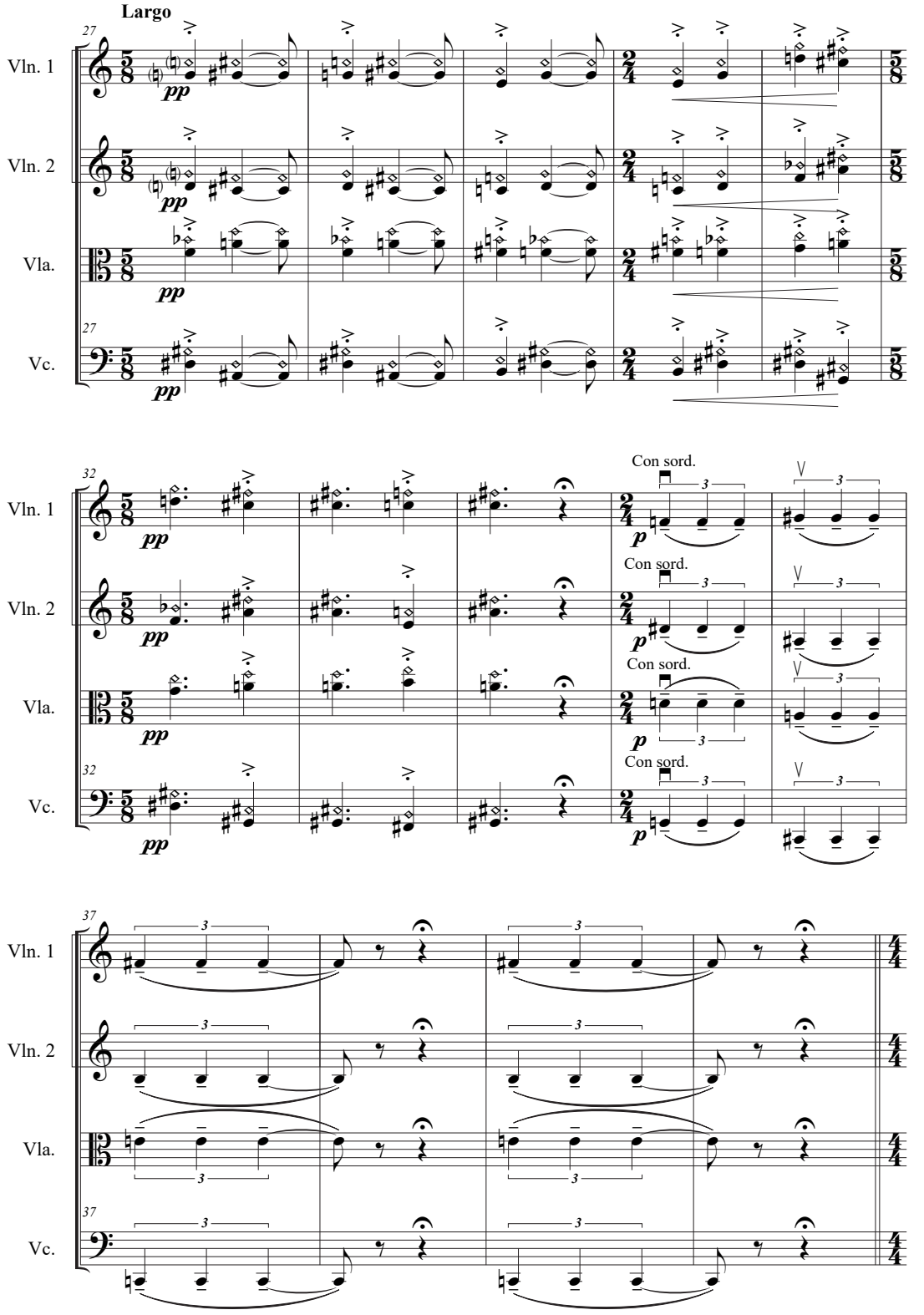
Brian Martínez
Murcia, Abril de 2019

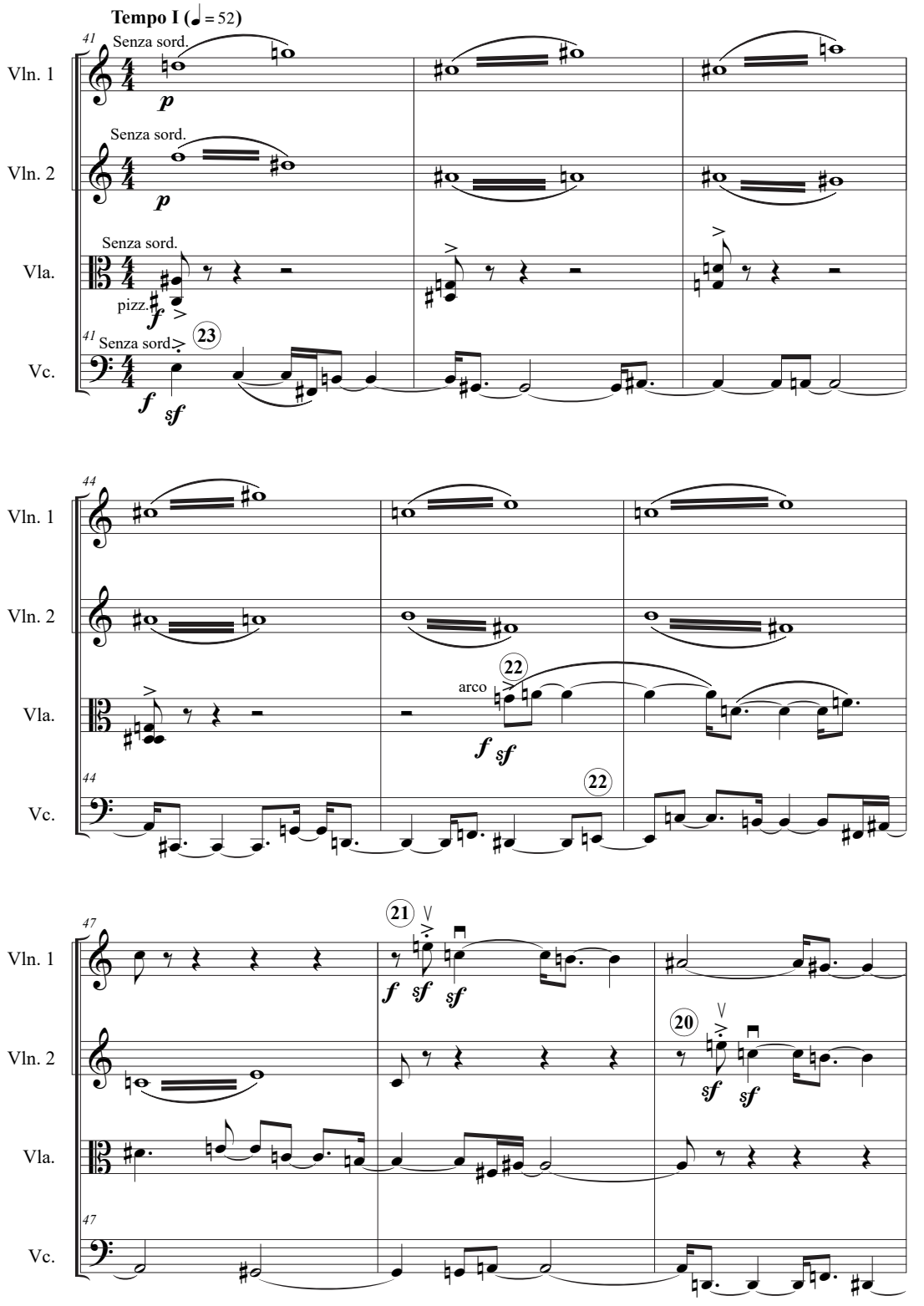
I. Andante

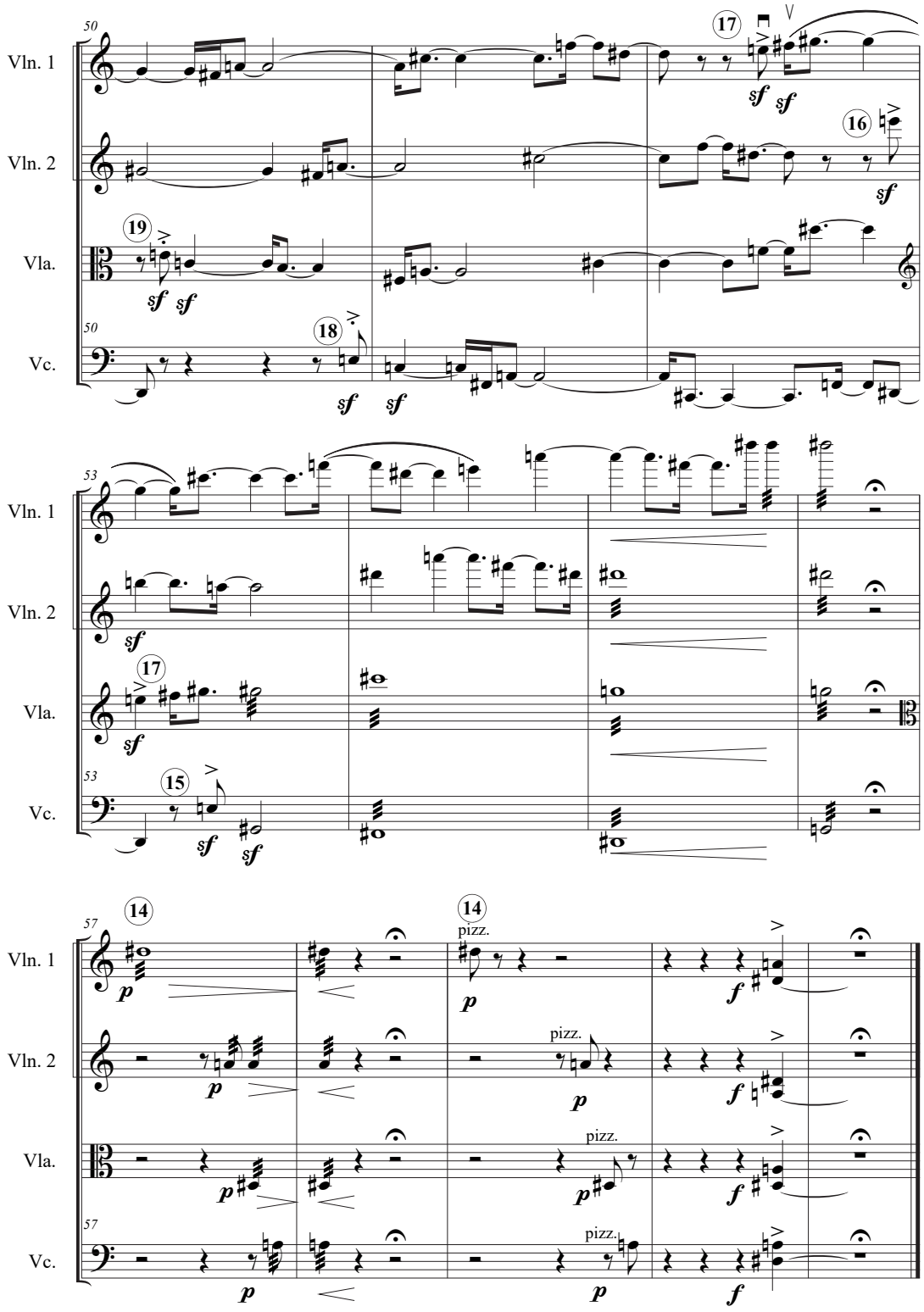



9
5
6
Vln. 1
Vln. 2
Vla.
Vc.
p
pp
Ancora più mosso
13
8
7
f
mf
sf
16
9
10

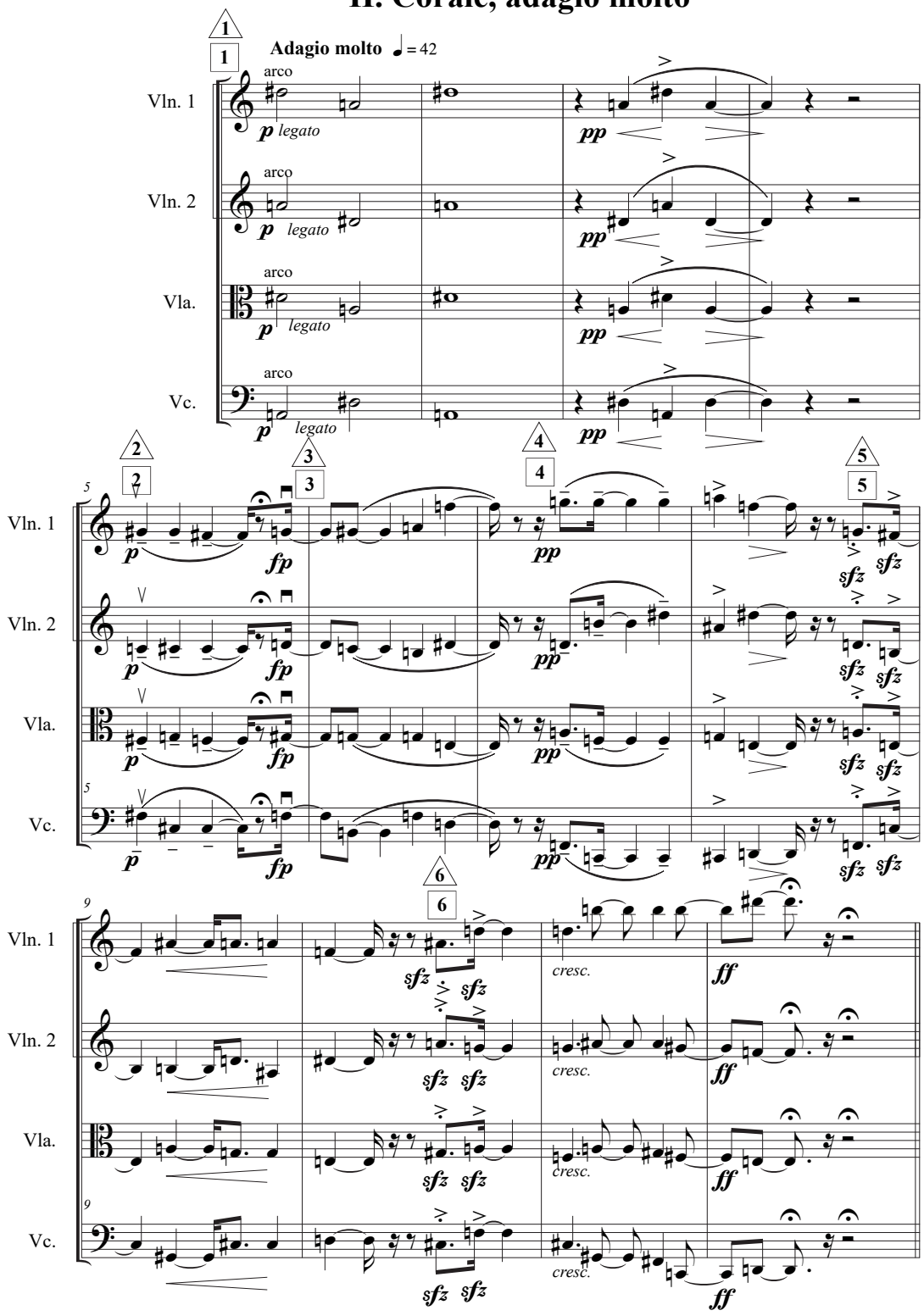

19
Vln. 1
Vln. 2
Vla.
Vc.
11
12
13
f
sf
22
cresc.
25
ff
sfz

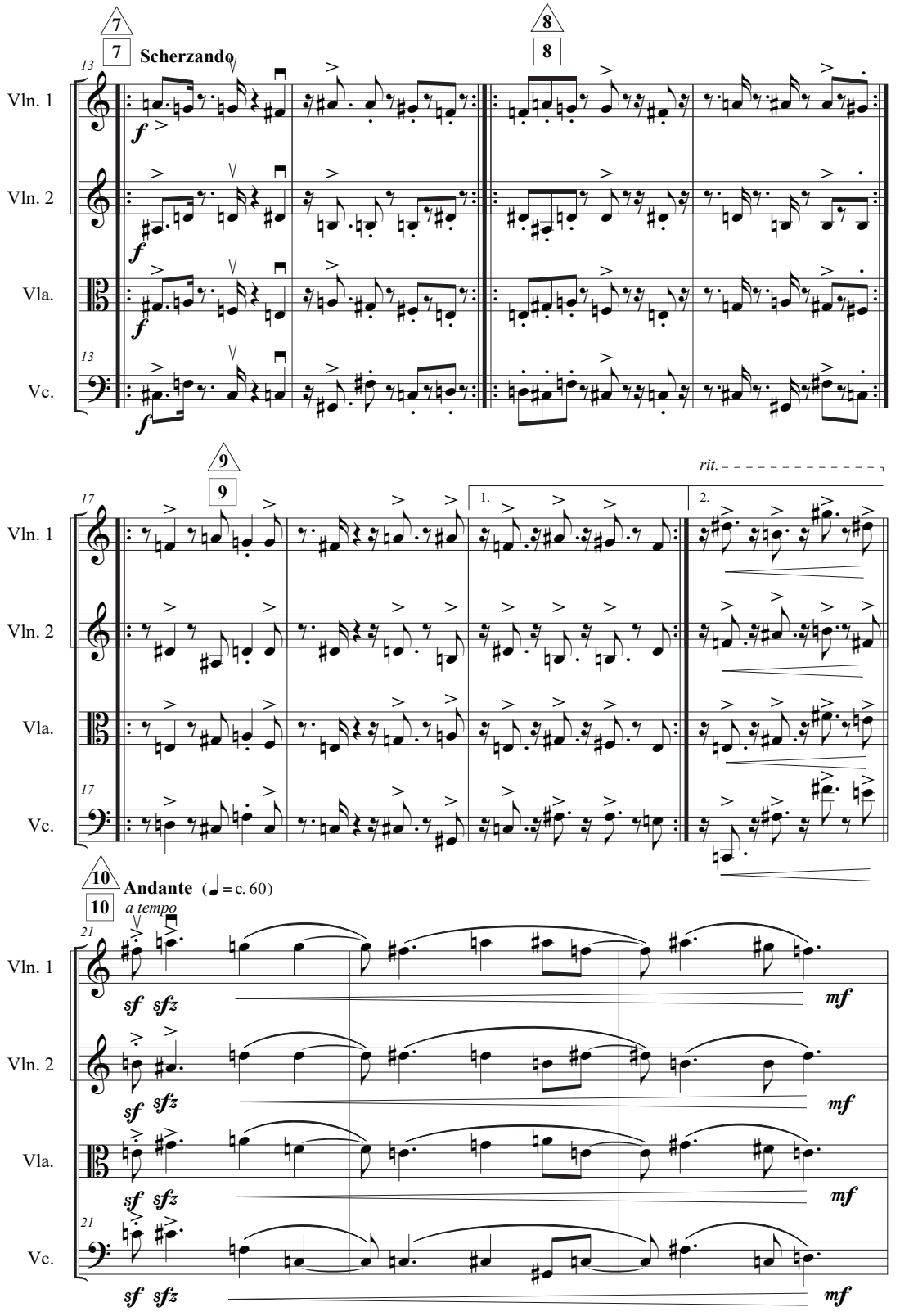

Largo
27
Vln. 1
Vln. 2
Vla.
Vc.
pp
32
Con sord.
p
37

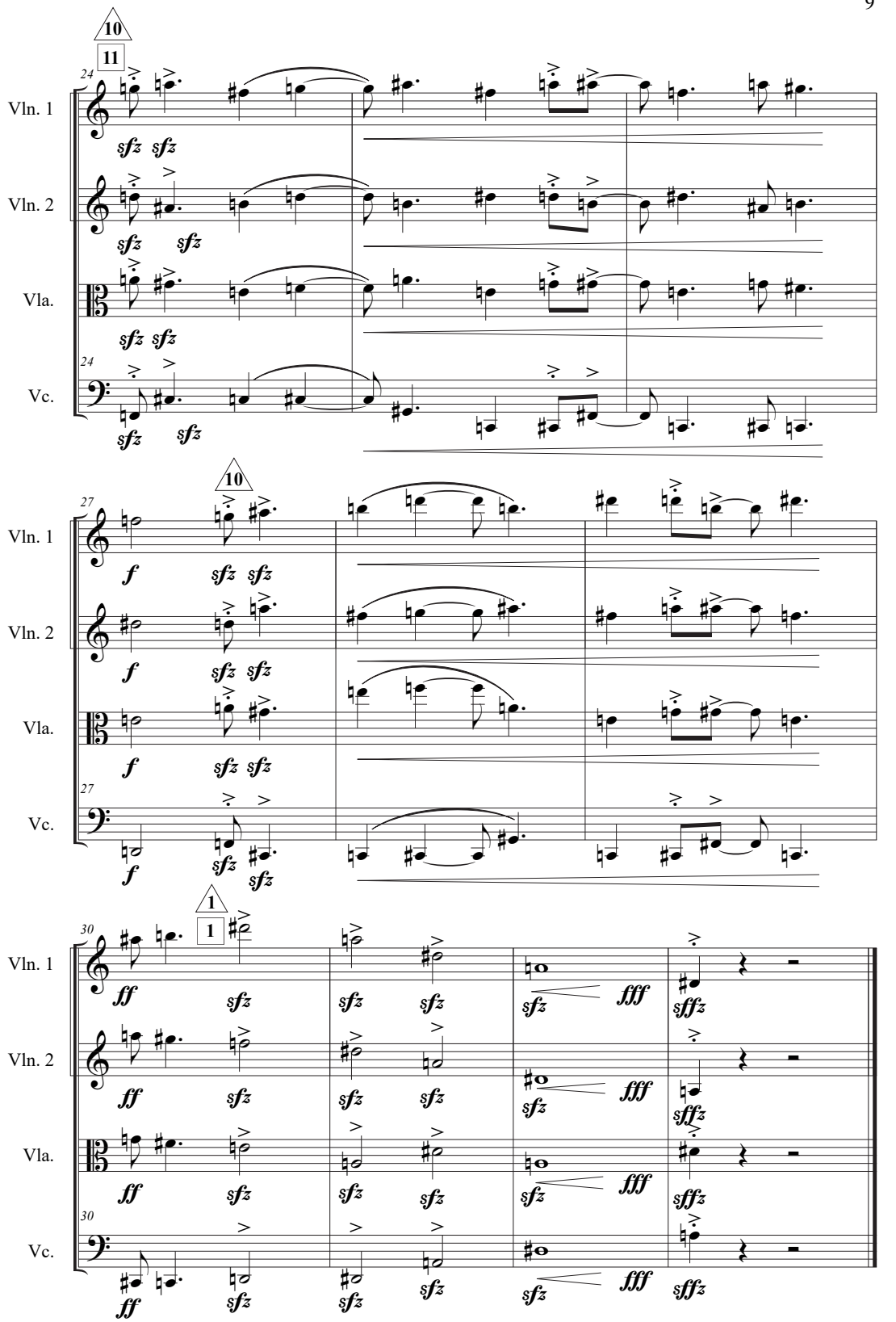

Tempo I (♩ = 52)
41
Senza sord.
Vln. 1
p
Senza sord.
Vln. 2
p
Senza sord.
Vla.
pizz.
f
41
Senza sord.
23
Vc.
f
sf
44
Vln. 1
Vln. 2
Vla.
arco
22
f
sf
44
22
Vc.
47
21
Vln. 1
f
sf
sf
20
Vln. 2
sf
sf
Vla.
47
Vc.


50
Vln. 1
Vln. 2
Vla.
Vc.
17
16
19
18
sf
53
15
pizz.
57
14
p
f

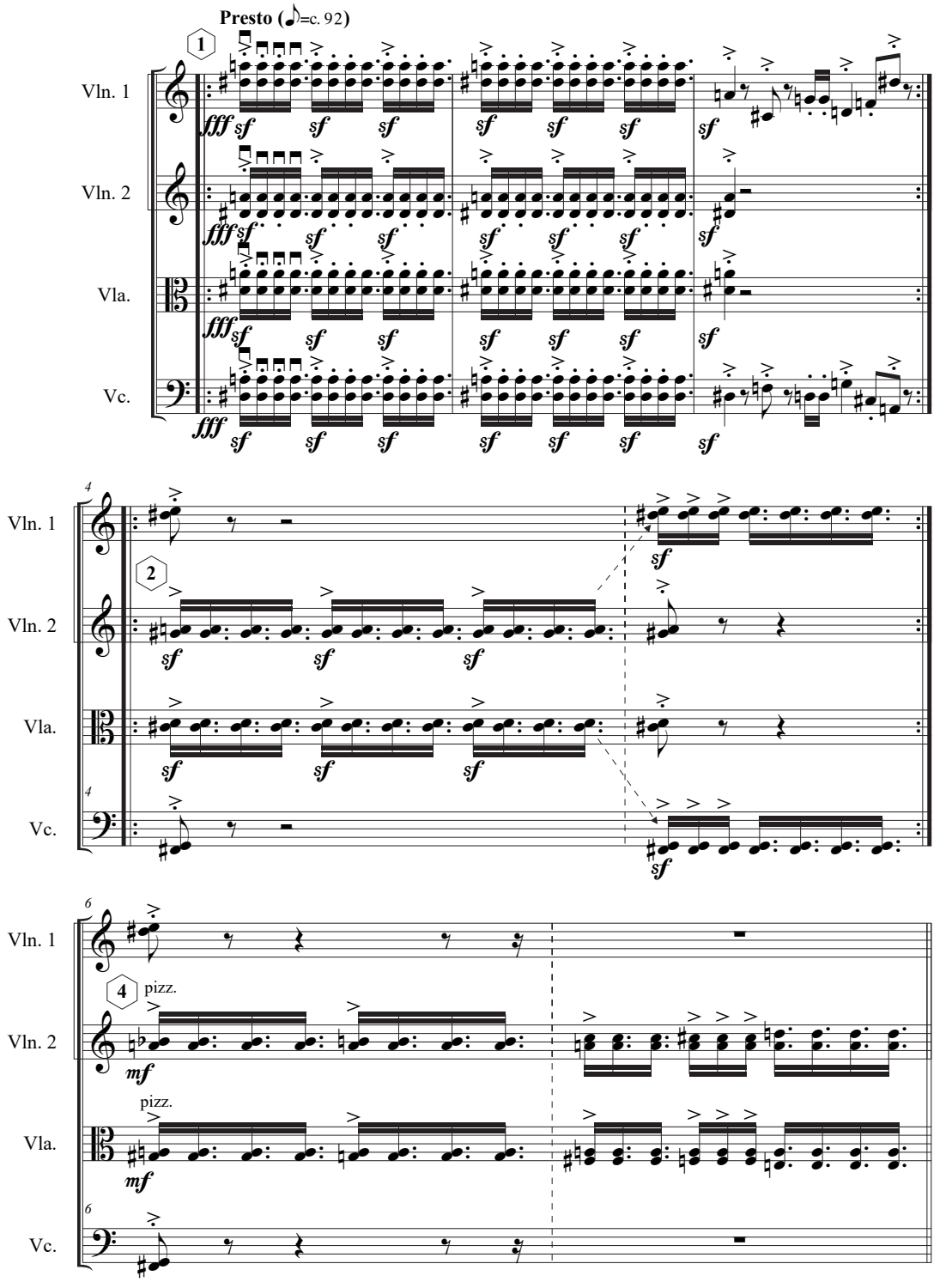
II. Corale, adagio molto

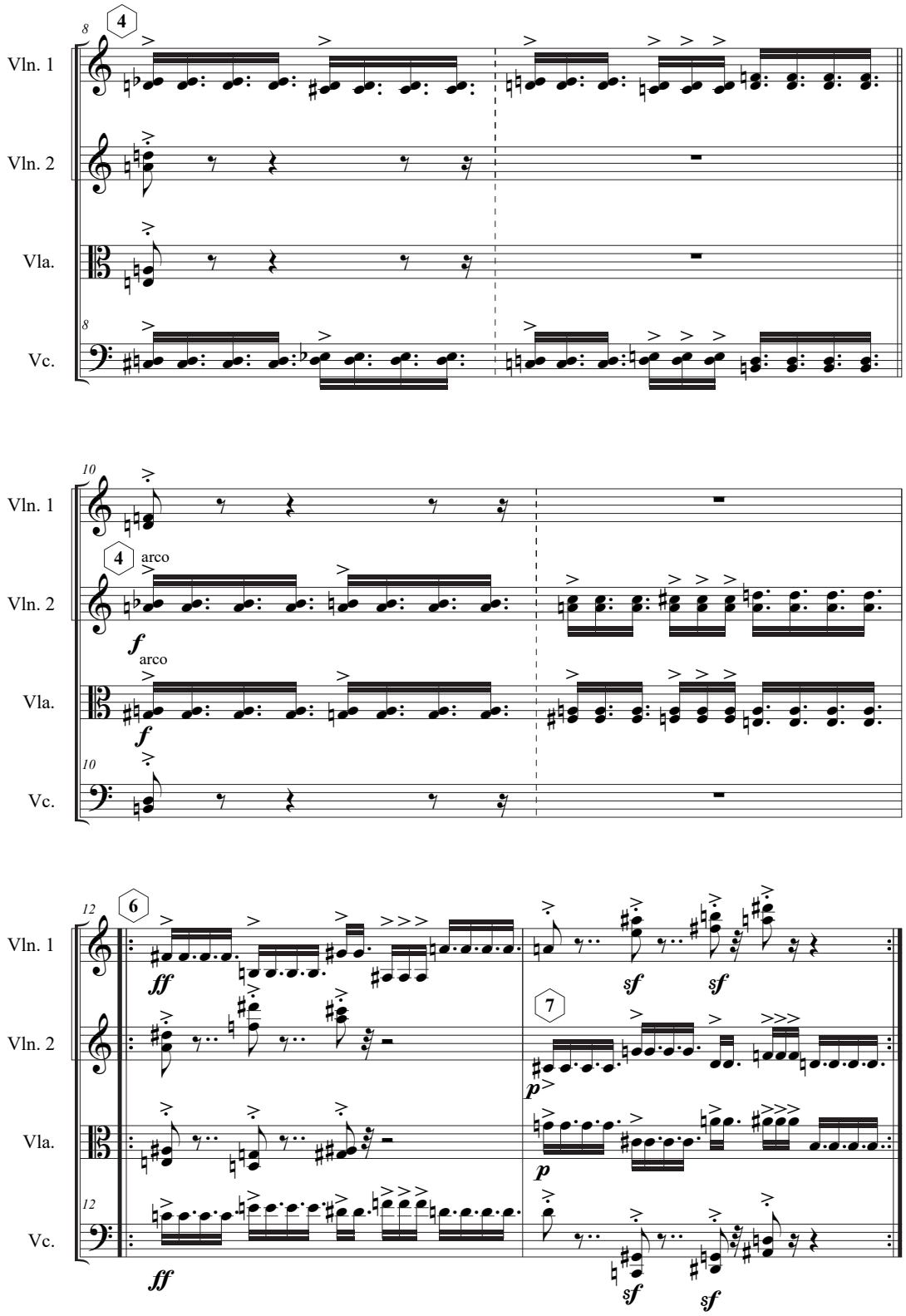


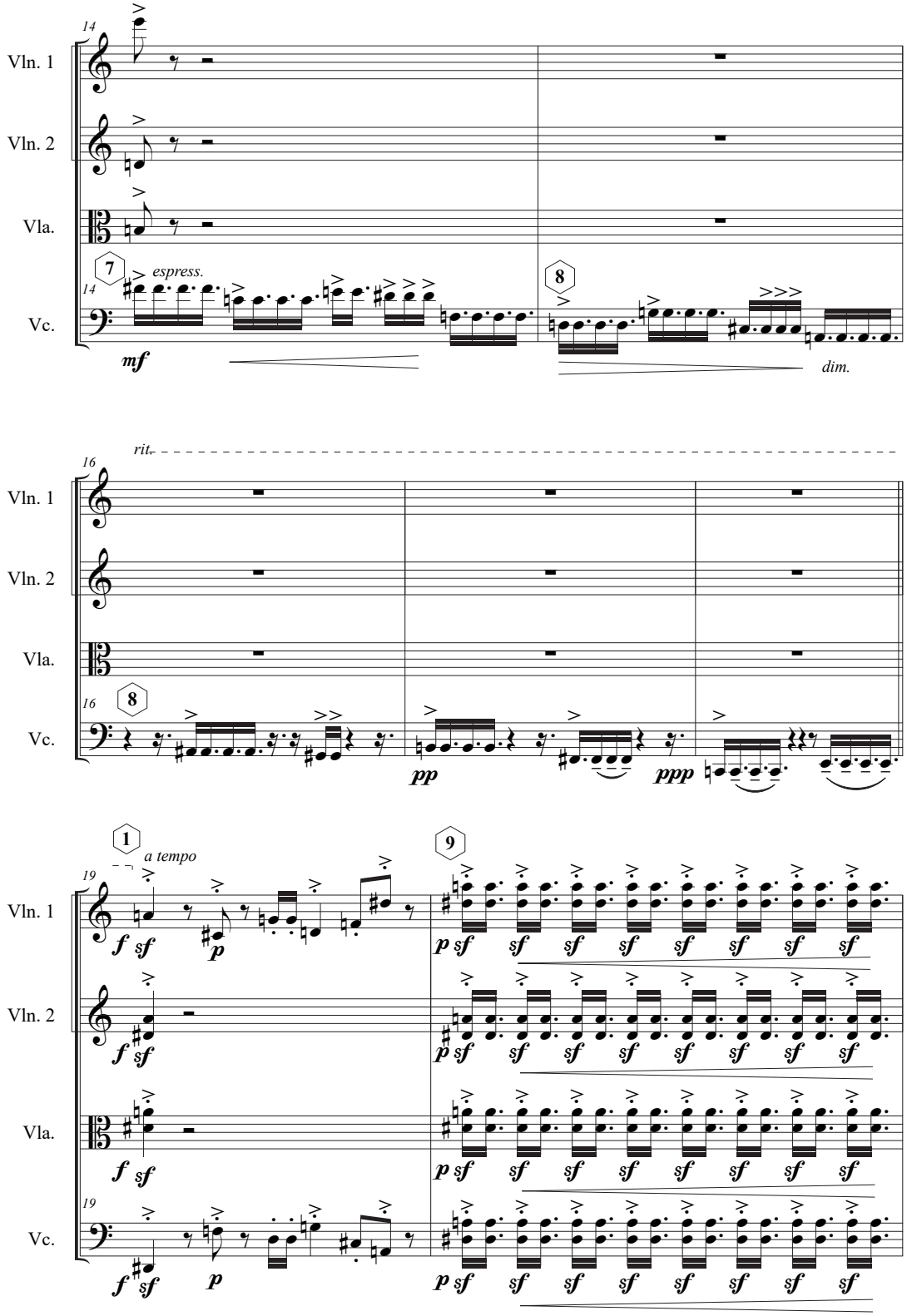

7
7
Scherzando
8
8
13
Vln. 1
Vln. 2
Vla.
Vc.
f
9
9
rit.
17
1.
2.
10
10
Andante (♩ = c. 60)
a tempo
21
sf sfz
mf

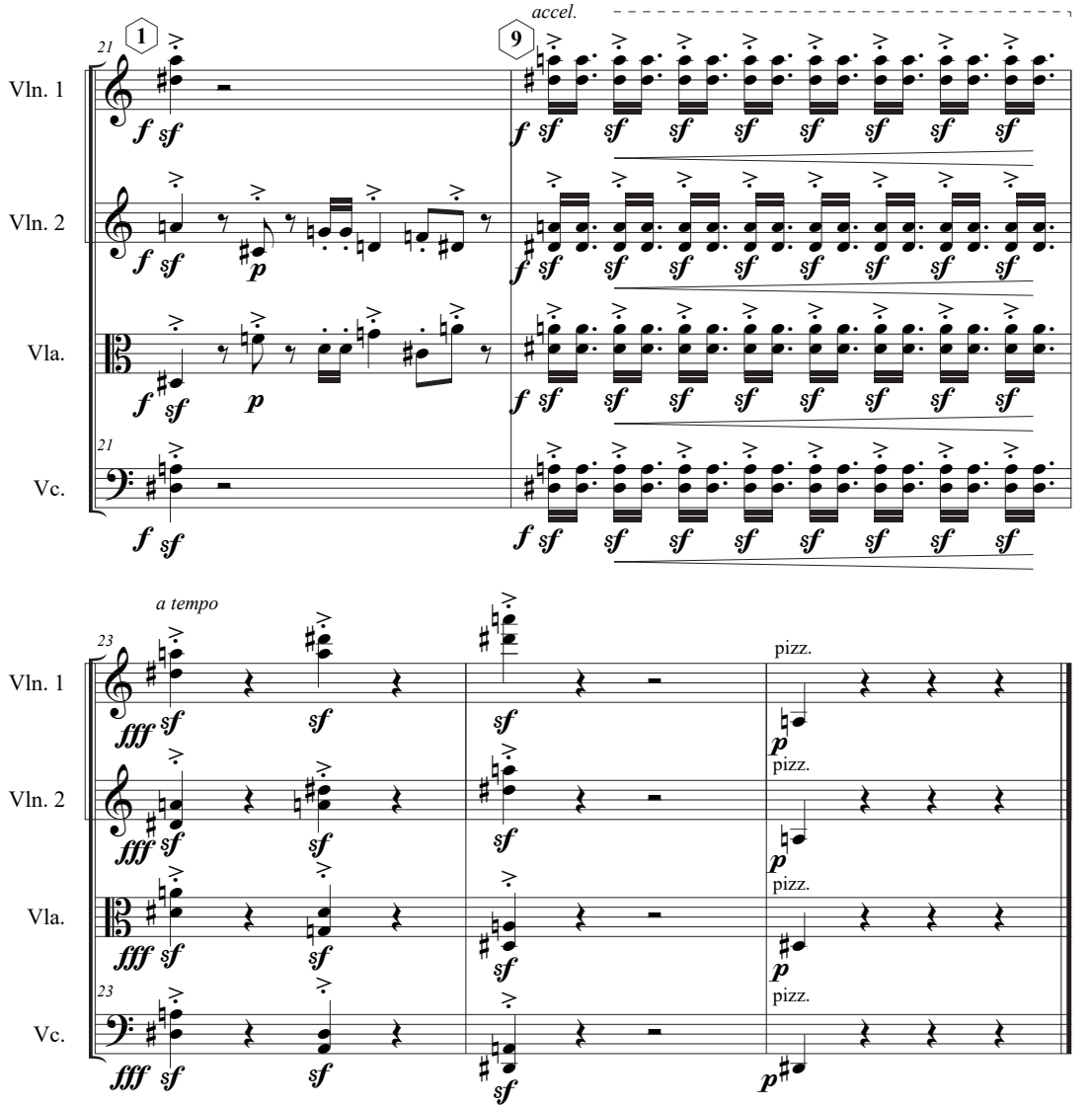

10
11
24
Vln. 1
Vln. 2
Vla.
Vc.
sfz sfz
10
27
f
1
1
30
ff
fff
sffz

III. Presto con fuoco




8
4
Vln. 1
Vln. 2
Vla.
8
Vc.
10
4
arco
f
arco
f
10
12
6
ff
7
sf
sf
p
p
12
ff
sf
sf


14
Vln. 1
Vln. 2
Vla.
Vc.
7
espress.
mf
8
dim.
rit.
16
8
pp
ppp
1
a tempo
19
f sf
p
9
sf


accel.
1
9
21
Vln. 1
Vln. 2
Vla.
Vc.
f sf
p
a tempo
23
fff sf
pizz.
p