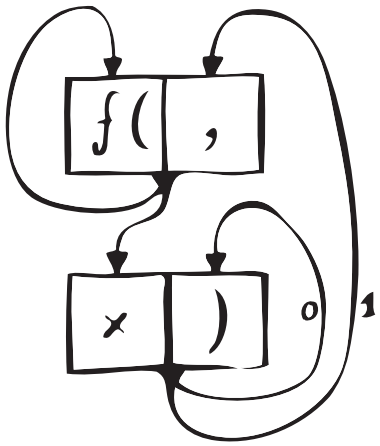
Universidad
Rey Juan Carlos

Curso 2012-13
Facultad de Ciencias del Turismo
Departamento de Ciencias de la Educación,
el Lenguaje, la Cultura y las Artes

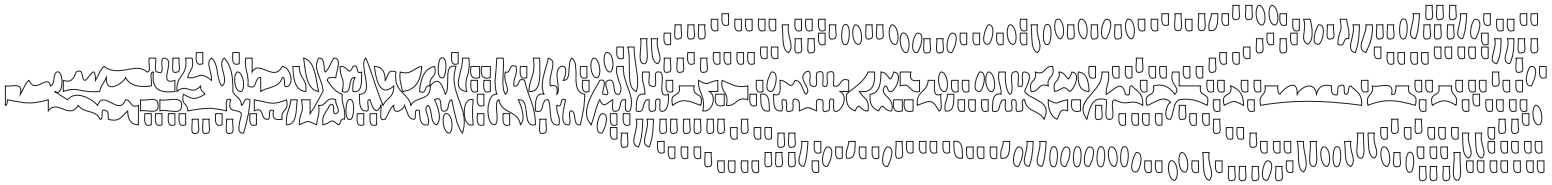
GenoMus

Prospección de técnicas de creatividad asistida por computadora mediante la metaprogramación de genotipos musicales



Realizado por
José López-Montes

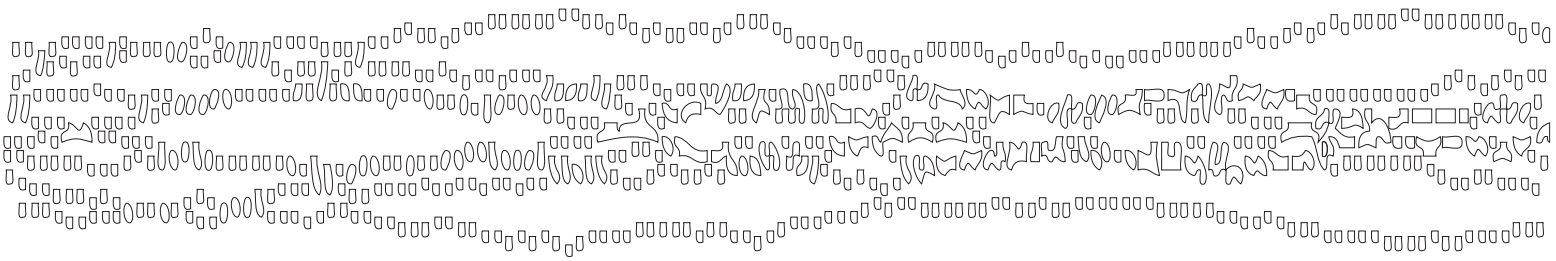
Tutor
Antonio Palmer Aparicio



La versión digital de este documento
junto a sus archivos generadores de LaTeX están disponibles en

http://www.lopezmontes.es/genomus.html

Agradecimientos



a Antonio Palmer*, tutor de esta investigación,*
quien comprendió desde el principio
la naturaleza de este proyecto
—a pesar de la vaguedad de los planteamientos iniciales—,
y aportó su amplitud de miras
así como su generosidad, lucidez y entusiasmo;

a los miembros de Taller Sonoro*,*
y especialmente a Javier Campaña*,* Ignacio Torner*,*
Camilo Irizo*,* Mery Coronado *y* Jesús Sánchez*,*
quienes materializaron
Threnody for Dimitris Christoulas
a pesar de sus casi utópicas exigencias técnicas;

a Trino Zurita *y* Óscar Martín*,*
quienes estrenaron con absoluta brillantez los
Ada + Babbage — Capricci;
Trino *supo leer como nadie el trasfondo de la partitura,*
Óscar *extrajo sonoridades ocultas*
que sorprendieron al propio compositor;

a Daniel Calandria*, cuyas agudas observaciones*
—cuando he alcanzado a entenderlas—
han sido fundamentales para un correcto enfoque de GenoMus
de cara a desarrollos futuros;

a la comunidad de LaTeX*,*
que ha hecho que la edición de este texto sea un placer;

y a Dori*,*
condición necesaria y suficiente
para que un trabajo como este viera la luz.

Resumen

GenoMus: prospección de técnicas de creatividad asistida por computadora mediante la metaprogramación de genotipos musicales

GenoMus explora un modelo de composición asistida por computadora en el que la máquina tiene gran autonomía creativa. La relación que el compositor establece con el sistema le permite sobrepasar sus propios prejuicios estéticos y ampliar su paleta expresiva y técnica mediante la exploración de los espacios musicales propuestos.

GenoMus está integrado por un metalenguaje musical procedimental basado en programación funcional, una librería de funciones que permiten la metaprogramación de abstracciones musicales denominadas *genotipos*, y un entorno para la experimentación en tiempo real.

Este trabajo expone la base conceptual y el funcionamiento de *GenoMus*, y muestra su aplicación en las composiciones *Threnody for Dimitris Christoulas*, para ensemble y electrónica, y *Ada + Babbage — Capricci*, para violonchelo y piano, desde la fase de experimentación previa hasta el estreno y la recepción de las obras.

Finalmente, se plantean los fundamentos de un programa de investigación más amplio centrado en la aplicación de técnicas de inteligencia artificial a la creación musical autoevolutiva basada en el paradigma de *GenoMus*.

Palabras clave: *GenoMus*, composición asistida, creatividad asistida, metalenguaje, metaprogramación, programación funcional, gramáticas generativas, algoritmos genéticos, genotipos musicales, autoevolución, autómatas, música generativa.

Índice

Figuras

1

Introducción

> *I have come to the conclusion that much can be learned about music by devoting oneself to the mushroom. For this purpose I have recently moved to the country. Much of my time is spent poring over "field companions" of fungi. These I obtain at half price in second-hand bookshops, which latter are in some rare next door to shops selling dog-eared sheets of music, such an occurrence being greeted by me as irrefutable evidence that I am on the right track.*[1]
>
> John Cage, *Music Lovers' Field Companion* [12]

La composición musical es un terreno abonado para los adictos a la formalización de lenguajes y a la abstracción de conceptos. Reemplazando setas por algoritmos, muchas veces me he identificado con este pasaje de Cage al ser consciente del tiempo que he dedicado a cuestiones que parecen ajenas a las tareas propias del músico. No obstante, considerando el importante número de investigadores ocupados en problemas similares, pienso que voy por un camino válido, o que al menos no me quedaré solo en el mar de los sargazos del código fuente.

Desde mis primeros trabajos compositivos con el ordenador he sentido una clara inclinación por los procesos de programación básica. Esto tal vez se deba a mi convicción de que cada *software* imprime una cierta pátina común en todos sus productos, mientras que el control manual de todos los procesos a bajo nivel mantiene una impronta mucho más personal en

[1] *He llegado a la conclusión de que podemos aprender mucho sobre la música si nos dedicamos a las setas. Para este propósito me he mudado recientemente al campo. Paso gran parte de mi tiempo estudiando minuciosamente "cuadernos de campo" sobre setas. Los consigo a mitad de precio en librerías de viejo, que en casos excepcionales están situadas al lado de tiendas que venden partituras manoseadas, un hecho que interpreto como evidencia irrefutable de que voy por buen camino.* (Trad. de Marina Pedraza [13])

cada obra, y además resulta más satisfactorio durante el proceso creativo. Al tiempo que han ido ampliándose mis recursos en la programación, mis intereses se han desplazado desde la síntesis de sonido y las manipulaciones algorítmicas directas hacia niveles crecientes de abstracción. Un repaso a las piezas compuestas en la última década [33] muestra la diversidad de tratamientos y técnicas empleadas en este camino.

Este estudio trata de consolidar de manera coherente las ideas, obsesiones e intuiciones que han nutrido mis composiciones previas y, ante todo, de establecer las bases para avanzar en una línea de investigación de mayor recorrido, que integre mis experiencias anteriores en un sistema de exploración musical compacto pero muy abierto. Este objetivo se aborda proponiendo un modelo particular de composición asistida por computadora denominado *GenoMus*, con el que se ha hecho una *prospección* para evaluar las potencialidades reales de la metaprogramación aplicada a la creatividad musical artificial.

GenoMus está articulado en tres componentes:

- un *metalenguaje musical procedimental* basado en programación funcional,
- un conjunto de funciones con los que manipular este metalenguaje,
- y una interfaz externa con la que gestionar el flujo de información.

La representación musical de *GenoMus* se basa en un metalenguaje procedimental porque éste no codifica eventos musicales concretos, sino los procedimientos subyacentes que los generan. Este metalenguaje tiene una característica particular: *está estructurado para facilitar la metaprogramación autónoma de sus propias expresiones.*

El sistema maneja los conceptos de *genotipo* y *fenotipo musical*, apelando a una analogía con la genética de amplia tradición en los estudios de composición asistida. En el metalenguaje propuesto, un genotipo es una expresión que codifica procesos musicales, y un fenotipo es el fragmento musical generado por un genotipo.

La composición musical mediante algoritmos bioinspirados está emergiendo como base de una gran cantidad de proyectos y, dada la rapidez con que se suceden las innovaciones, resultaría temerario y arrogante afirmar que este trabajo aporta novedades notables a los numerosos enfoques con que ya se está abordando la cuestión. Lo más significativo de los contenidos que siguen no son las técnicas concretas que se exponen, sino la esencia conceptual global, el paradigma hacia el que apunta. En ámbitos propios de la inteligencia superior, tales como el lenguaje o la creatividad artística y científica, los últimos avances —y los que ya se vislumbran— sugieren que los modelos de computación que conocemos dejarán paso a sistemas que paulatinamente exhibirán comportamientos más adaptativos y menos predecibles, más inteligentes y creativos. Condensando la filosofía de todo el trabajo en una frase:

es hora de afrontar en profundidad la composición asistida diseñando algoritmos creativos tan flexibles que sean capaces de aprender de sí mismos y del entorno, acercándose más a los mecanismos biológicos que a los de la ingeniería clásica. Estas técnicas también deben renovar otros dominios, como el análisis y la pedagogía musical. Es la gestión de la información, y no su acumulación, lo que debe orientar la actividad musical en el futuro próximo.

La investigación se completa con la presentación de dos obras en las que se han aplicado las técnicas descritas en dos momentos distintos de su desarrollo: *Threnody for Dimitris Christoulas*, para flauta, clarinete, violonchelo, piano y cinta, y *Ada + Babbage — Capricci*, para violonchelo y piano. El papel de *GenoMus* en estas composiciones ha consistido en funcionar como detonante de la creatividad mediante la generación automática de ideas musicales.

El concepto *creatividad asistida por computadora*, que figura en el título de este documento, alude al papel activo de la máquina en la proposición de ideas: el algoritmo tiene un margen de libertad que va más allá de las predicciones del programador; las ideas musicales que sugiere están libres de las preferencias y prejuicios estéticos del compositor. Se dispone así de una fuente de propuestas con las que modelar un material personal al que probablemente no se llegaría sin este mecanismo para *puentear la autocensura*. Lo que se pretende es expandir las posibilidades expresivas y computacionales propias para *componer lo que uno mismo no podría componer*.

La aplicación de *GenoMus* en ciclos de trabajo creativo completo, desde la concepción de una obra hasta su estreno y recepción, es una parte fundamental de este estudio, y trata de responder a cuestiones básicas: ¿tiene sentido automatizar la tarea de componer? ¿qué tipo de relaciones se establecen entren el compositor y el programa? ¿hasta qué punto es reducible una tarea que requiere un sentido casi lingüístico, una continua cosmovisión del trabajo, y en la que la individualidad y la emotividad son factores clave?

Modelizar procesos mentales nos confronta directamente con la naturaleza de la inteligencia. En el caso de la música, estamos además tratando con un producto destinado a la recreación estética, esto es, a una de las facetas más específicamente humanas y que más nos distancia de los impulsos biológicos básicos. La mera idea de que la inteligencia musical pueda ser emulada por ingenios mecánicos parece un atentado contra una de las actividades intelectuales que parece estar más al margen de las prosaicas tareas habituales de la cibernética. Los idiomas y estilos del lenguaje musical están repletos de sutilezas cuyo dominio parece estar sólo al alcance de músicos *inspirados*. Sin embargo, parece asimismo inevitable que una facción de compositores sienta una inevitable atracción hacia la formalización y la ampliación de su creatividad mediante las tecnologías disponibles.

La aún breve historia de la inteligencia artificial es ya suficiente para mostrar dos caras opuestas de un mismo fenómeno:

- La simulación del razonamiento parece poder acercarse a la captura y automatización de cualquier proceso mental.
- Una vez comprendido y formalizado un proceso mental, tenemos la fuerte impresión de que lo que denominamos inteligencia se encuentra más allá.

Extrapolando este pensamiento a la composición musical, podemos aventurar que, a pesar de la fértil y cada vez más profunda interacción entre hombre y máquina, ese delicado equilibrio entre forma y contenido que entendemos por *música* siempre parece empezar donde acaba lo ya asimilado y codificado. De alguna manera estamos condenados a avanzar continuamente para situarnos en el mismo punto a partir del cual es posible una experiencia musical significativa. Y si una de las esencias de lo humano es el autoconocimiento, se puede concluir que, en contra del tópico dominante, el desarrollo de la inteligencia artificial nos impulsa a hacernos más humanos y, en el terreno artístico, a llevarnos a situaciones creativas radicalmente novedosas en las que la comunicación musical puede tener lugar de maneras imprevistas.

En marco de estas ideas se sitúa la presente investigación.

2

Estado de la cuestión

> ***¿Gramáticas de la música?***
>
> *Tenemos luego la música. Se trata de un dominio del cual se puede suponer, en una primera consideración, que se prestaría admirablemente a ser codificado en una gramática* RTA,[2] *o en otro programa.* [...] *No hay referencias a cosas de "allí afuera" en los sonidos musicales; estos no son sino solamente una sintaxis.* [...]
>
> *Pero, un momento. Hay algo que no está bien en este análisis. ¿Por qué hay música que es mucho más profunda y más bella que otra? Porque la forma, en música, es expresiva, expresiva de algunas extrañas regiones subconscientes de nuestras mentes.* [...] *No, la gran música no surgirá de un formalismo tan sencillo como una gramática* RTA. [...]
>
> *La gramática habrá de definir no solamente estructuras musicales, sino las estructuras íntegras de la mente del espectador. La "gramática" será una gramática total del pensamiento y no, exclusivamente, una gramática de la música.*
>
> Douglas R. Hofstadter, *Gödel, Escher, Bach* [24]

Analizar el estado de la cuestión en cualquier investigación de composición asistida por computadora —en adelante CAC— se torna rápidamente inmanejable, dada la eclosión casi diaria de nuevos proyectos, artículos y monografías. Un sumario demasiado puntual de las líneas de trabajo actuales corre el riesgo de perder vigencia en muy poco tiempo, por lo que en este capítulo trataremos de clarificar con cierta perspectiva histórica cuáles

[2] RTA es la abreviatura de *redes de transición aumentada*, un concepto que expande el de RTR (*redes de transición recursiva*), relativo al conjunto de reglas estudiadas por la gramática generativa. Las RTA son redes integradas por RTR que se apelan unas a otras, por lo que el nivel de complejidad y autorreferencia de éstas puede llegar a ser enormemente complejo.

son los paradigmas principales de las últimas décadas, poniendo el foco en aquellos que se tocan directamente con el objeto de este trabajo.

En primer lugar, hay que observar que la CAC puede ubicarse en un espectro que va desde planteamientos meramente teóricos hasta la pura aplicación práctica. En la producción de cierto sector de los compositores actuales es imposible separar la obra musical del trabajo de investigación que la sustenta. En muchas ocasiones, pareciera que una única obra musical funciona como *prueba* de una conjetura creativa, lo que hace imposible discernir la potencialidad de una técnica de CAC, ya que contamos con muy pocas plasmaciones materiales de la misma, y la calidad de estas composiciones probablemente queda más condicionada por el talento del compositor que la usa, antes que por las bondades del sistema que se preconiza.[3] Meyer [38] distingue claramente estas dos facetas de la creación musical, la concerniente a la configuración del lenguaje y la relativa a su utilización efectiva:

> *La distinción entre reglas y estrategias ayuda, creo yo, a clarificar el concepto de originalidad, así como su correlato, la creatividad, porque sugiere que han de reconocerse dos clases de originalidad algo diferentes. La primera tiene que ver con la invención de reglas nuevas.* [...] *La segunda clase de originalidad, en el nivel de la estrategia, no implica cambiar las reglas sino discernir nuevas estrategias para realizar esas reglas.*

En la práctica de muchos compositores experimentales se funden la creación de sistemas y su aplicación sobre la partitura, lo que, como veremos más adelante, dificulta la evaluación y la crítica de estos productos musicales.

Cada nuevo sistema de CAC suele apuntalar sus planteamientos con argumentos de tipo *científico*. Es fácil que se dé cierta confusión entre la investigación con criterios científicos de cualquier aspecto de la práctica musical,[4] y la aplicación de herramientas propias de la física o las matemáticas al trabajo creativo. Sin embargo, no hay que perder de vista que el objeto último del método científico y el de los *métodos artísticos* es bien diferente: como expone Puckette [49], mientras la ciencia trata de hallar lo universal que unifica fenómenos aparentemente diferentes, el arte busca especímenes de valor único dentro de las diversas posibilidades de cada técnica o estética. En otras palabras: la ciencia versa sobre lo común que unifica lo diverso, y el arte sobre lo especial que destaca entre lo común. Este pensamiento aconseja considerar cada nueva técnica de CAC como parte del mundo creativo de

[3] La historia de la recepción y de la práctica del método serial de Schönberg es ilustrativo acerca de cómo los sistemas que se postulan como alternativas a los métodos habituales quedan ligados a las obras y compositores. La labor de *evangelización* que conlleva la defensa de cada nueva técnica crea tradición, y en cierto modo, una percepción sesgada y dirigida de sus posibilidades expresivas reales, lo que contradice las aspiraciones de universalidad de muchos de estos sistemas.

[4] Una obra enciclopédica que emplea el análisis lógico, topológico y algebraico a casi todos los aspectos de la música, incluidos los más intuitivos, como el fraseo o el rubato, puede hallarse en la producción de Mazzola [37].

cada compositor, y no como nuevos intentos de llegar a un sistema definitivo que deba ser abrazado por todos. En el dominio del arte, el conocimiento de la producción de otros artistas sirve en primera línea como estímulo a la creatividad personal, al margen de las herramientas e ideas útiles de las que uno pueda apropiarse.

2.1 ¿Creatividad artificial o artificio creativo?

Las artes están estrechamente ligadas a la tecnología de su época. Esta influencia es recíproca: a la par que las innovaciones hacen posibles nuevas vías de expresión, son también los artistas quienes estimulan la aparición y evolución de estas tecnologías aplicadas. Es evidente, por ejemplo, que el desarrollo del instrumentario de la orquesta moderna no podría entenderse sin las revoluciones industriales. Las demandas de los compositores y de los instrumentistas impulsaron un perfeccionamiento desconocido de los instrumentos tradicionales, y la aparición de otros nuevos. Con respecto a la interpretación, los autómatas musicales capaces de reproducir secuencias musicales existen desde antiguo. Más allá de su uso funcional, ritual o meramente lúdico, llegaron a ser tomados en serio por compositores como Haydn y Beethoven.[5] La música para pianola y otros ingenios mecánicos forma parte del catálogo de no pocos compositores. Finalmente la digitalización del sonido ha hecho ubicuos e imprescindibles los secuenciadores musicales.

La aplicación de la tecnología a la creatividad se sitúa en otro plano, y bordea el tabú que mitifica el genio musical como una actividad casi divina, que parece quedar fuera de la formalización matemática y computacional. La posibilidad de crear autómatas con un poder de razonamiento sobrehumano siempre ha ejercido una poderosa sugestión, ya que pone a prueba la extendida idea de que la inteligencia reside en una esfera que trasciende el mundo material.[6]

Los argumentos típicos que reaparecen una y otra vez en cualquier debate sobre el verdadero potencial de la inteligencia artificial —en adelante IA— se recrudecen cuando se focalizan en la *creatividad artificial*. El estudio serio de lo *artificial* suele conducir a contradicciones, y a la necesidad de redefinir los conceptos. La propia semántica de la palabra encierra esta paradoja: por

[5] Beethoven tuvo una estrecha relación con el inventor de ingenios mecánicos musicales J. N. Mälzel. El compositor de Bonn no sólo usó sistemáticamente el metrónomo de Mälzel, sino que frecuentaba su taller, e incluso compuso la primera versión de *Wellingtons Sieg*, Op. 91, para el panarmónico, un autómata capaz de tocar varios tipos de instrumentos y ejecutar marchas militares [43].

[6] Volviendo a la figura de Mälzel, encontramos una curiosa confluencia entre autómatas musicales e inteligencia artificial: Mälzel, antes de ser reconocido por el metrónomo, se hizo célebre por ofrecer espectáculos en los que un autómata jugaba partidas de ajedrez. Aunque este autómata resultó ser un fraude, no por ello deja de ser muy significativa la expectación que suscitó por sus implicaciones filosóficas, la cual Poe reflejó en su ensayo *Mälzel's Chess Player* [48].

un lado definimos como *artificial* lo que ha sido "hecho por mano o arte del hombre" [50], o "producido por el ingenio humano", es decir, que *lo artificial es creativo en sí mismo*; pero al mismo tiempo calificamos como artificial lo "no natural", lo "falso", o aún peor cuando se refiere a lo artístico, lo "artificioso". Esta contradicción emana de concepciones opuestas acerca de *lo humano*: lo que está en tela de juicio es la propia definición del hombre, bien como algo natural —con todas sus consecuencias—, o bien como una entidad separada de lo material en algún aspecto. En otras palabras, la IA, y aún más la creatividad artificial, apoya el argumento de que la inteligencia humana no está por encima del mundo físico. El progreso científico ha ido laminando todo aquello que la humanidad ha ido reservándose como privilegio sobre el resto del universo. La evidencia de que procesos de extraordinaria complejidad pueden desencadenarse a partir de mecanismos asombrosamente simples están atacando el último reducto de superioridad del hombre, reduciendo la propia inteligencia y la consciencia a un fenómeno natural más.

> *Yet in Wertern thought there is still a strong belief that there must be something fundamentally special about us. And nowadays the most common assumption is that it must have to do with the level of intelligence or complexity that we exhibit. But* [...] *the Principle of Computational Equivalence now makes the fairly dramatic statement that even in these ways there is nothing fundamentally special about us.*[7] [68]

Es por ello que sólo vislumbrar la posibilidad de la existencia de entes creativos artificiales produce recelo y fascinación a partes iguales.

En la actualidad los acercamientos a la IA suele estar concentrados en áreas específicas: desde la modelización de comportamientos triviales hasta la formulación automática de nuevos teoremas matemáticos. Hay también intentos de abordar la inteligencia en su conjunto —ámbito denominado como *inteligencia artificial general*—, aunque esto parece aún lejano, pues requiere un trabajo ingente de integración de competencias en muchos órdenes, y también la mejora de los órganos artificiales de percepción visual, sonora, táctil, etc.

Al mismo tiempo, la modelización del razonamiento humano está teniendo una enorme influencia en nuestra concepción de lo que realmente *es* el pensamiento y la consciencia. Los programas más recientes de investigación del funcionamiento cerebral [1] están más orientados hacia el análisis de problemas típicos de teoría de la información que a la fisiología: parece ser mucho más determinante comprender la emergencia de los procesos neuronales en red antes que los detalles concretos de su funcionamiento bioquímico.

[7] *En el pensamiento occidental aún persiste la fuerte creencia de que debe haber algo fundamentalmente especial acerca de nosotros. Y hoy en día la asunción más común es que esto debe tener que ver con el nivel de inteligencia y complejidad que exhibimos. Pero el Principio de Equivalencia Computacional hace ahora la genuinamente dramática afirmación de que no hay nada fundamentalmente especial en nosotros.* (Trad. del autor)

Esta convergencia entre cibernética y neurobiología es probablemente un síntoma de que nos acercamos al punto en que será evidente que las investigaciones en cualquier campo de la IA, incluida la creatividad artificial, no son sino indagaciones acerca de lo que nos hace ser lo que somos.

2.2 El rol de la máquina en la evolución del estilo en música

La idea de encontrar una receta infalible que componga automáticamente obras maestras únicas encierra una contradicción en su núcleo. El valor estético otorgado a una pieza musical no se explica únicamente por la partitura, sino que depende en gran medida del contexto de su creación y de su recepción. Por otra parte, hay composiciones geniales de las se infieren idénticas características estilísticas que de otras mediocres, por lo que tampoco parece ser de gran utilidad creativa la descripción formal exhaustiva de un idioma musical determinado. Es más, muchas obras clave de cada período histórico suelen situarse en los bordes de sus sistemas, a menudo arañando relaciones imprevistas y tensando las lógicas establecidas. A menudo, los tratados que mejor entienden y explican un estilo musical determinado aparecen una vez que el sistema en cuestión ha perdido su vigencia.[8]

Aceptando la idea de que lo relevante y especial del estilo de un gran compositor se suele encontrar en detalles que escapan al análisis descriptivo, ¿cómo puede asumir la CAC un papel verdaderamente creativo? Una perspectiva amplia de la evolución del estilo en los últimos siglos puede esclarecer esta cuestión. En términos muy generales podemos distinguir varias etapas:

- En las últimas décadas del siglo XVIII el sistema tonal está totamente cristalizado como un lenguaje internacional en Europa. Al margen de ciertas particularidades estilísticas, puede afirmarse que todos los compositores comparten un idioma común, que además suele mantenerse muy estable a lo largo de toda su producción.

- Durante el siglo XIX las individualidades tienden a ser cada vez más marcadas. Los compositores aspiran a que su estilo sea reconocible y, aunque hay transformaciones perceptibles en el lenguaje de cada compositor, los estilos individuales se mantienen relativamente identificables y siguen bien entroncados en la práctica común de la tonalidad.

- La crisis del sistema tonal a principios del siglo XX marca un importante punto de inflexión. En términos físicos podríamos decir que entramos

[8] En este sentido es paradigmático que los clásicos tratados pedagógicos de Schönberg, que tan profundo conocimiento muestran sobre la tonalidad, hayan sido escritos por el compositor después de abrir las puertas a la atonalidad radical y sistemática. Su famoso *Harmonielehre* [62] data de 1922; un año después hizo la primera presentación privada de su método serial.

en una fase de *turbulencia*. Surgen múltiples salidas alternativas a la tonalidad, y se suceden en rápida sucesión técnicas musicales diversas e incluso opuestas en sus planteamientos. En estas décadas comenzamos a encontrar compositores que hacen trayectorias artísticas que muestran grandes transformaciones en su lenguaje.[9] Al mismo tiempo los compositores se agrupan en corrientes estilísticas en continua controversia.

– Hacia 1950 la divergencia de estilos es ya tan extrema que puede afirmarse que, en muchos casos, cada creador puede representar virtualmente un estilo. Se empieza a generalizar la postulación de métodos personales.[10] Maconie [35] resalta que hay un interés por sondear y reinventar todos los mecanismos del lenguaje, tanto para el análisis como para la síntesis:

> *Una buena parte de la música progresista de mediados del siglo XX busca con gran deliberación eliminar el aspecto humano con el propósito de abrir la conciencia del público a las posibilidades de una organización musical—y mental—situadas más allá del alcance del reconocimiento y la memoria ordinarias.*

– En los últimos años el auge de la interactividad y el enorme impacto que ha supuesto la digitalización y la ubicuidad de la computadora está redefiniendo muchos conceptos acerca de lo que es la creación musical.[11]

Ha tenido lugar un cambio sustancial en la *consciencia de estilo*: si antes la gramática musical emergía de un lento proceso colectivo, modelado por la interacción entre músicos y público, a partir de principios del siglo XX encontramos que los nuevos sistemas musicales se formulan *a priori*, y las composiciones musicales se presentan como demostraciones de su validez.

Existe un interés creciente por la teorización del lenguaje, que llevado al extremo tiende al punto en que cada nueva obra es la enunciación de un estilo único; cada pieza musical se convierte en una nueva —y a veces efímera— definición de música. En este estado de cosas se comprende que exista un fuerte interés por entender los resortes internos de cada gramática musical. El trabajo de muchos compositores actuales tiene lugar en un nivel metamusical: ya no se trata tanto de situar las notas sobre el pentagrama

[9] El caso de Stravinsky encarna perfectamente la figura del compositor que se reinventa cada poco tiempo.

[10] Messiaen en Francia, Hindemith en Alemania o Barce en España, son ejemplos de compositores con una producción ligada a un método de composición personal formulado explícitamente.

[11] Di Scipio [19] ha realizado recientemente un amplio estudio de las consecuencias de las tecnologías de la información en las nuevas relaciones que se establecen entre creadores y espectadores.

como de precisar las reglas —o la ausencia de ellas— que organizarán esos sonidos. Esta focalización en los procesos conduce directamente al uso de las tecnologías de la información y al interés por la CAC.

Puede por tanto considerarse que la CAC es, consciente o inconscientemente, un trabajo de creación musical que reflexiona y actúa directamente sobre los resortes que configuran el estilo.

2.3 Breve panorámica de la CAC

Muchos musicólogos han afirmado que la estética y la técnica de las obras de un periodo histórico son un trasunto de las teorías científicas vigentes en cada época. En el siglo XX estos reflejos empiezan a tener lugar de una manera consciente. En cuando la computación automatizada empieza a estar disponible en los centros de investigación de las universidades, cierto sector de los compositores se lanza ávido hacia cada nuevo concepto matemático o físico para explorarlo como medio de producción de estructuras musicales o de señales acústicas.[12] La disponibilidad del ordenador personal aceleró exponencialmente esta tendencia.

En el dominio de la composición pueden rastrearse diferentes modas ligadas a técnicas como el análisis estocástico, las cadenas de Markov, la teoría de juegos, la geometría fractal, la creación de gramáticas, las diferentes vertientes de la teoría del caos, los autómatas celulares, las redes neuronales, las redes bayesianas, etc. El compositor contemporáneo dispone de un arsenal de herramientas que le permiten la organización del material sonoro. Muchas de estas técnicas implican grandes volúmenes de cálculo, por lo que la asistencia de la computadora se hace esencial. Es entonces cuando se acuña la expresión CAC y *música algorítmica*.[13]

Pueden apreciarse diversas líneas fundamentales en estas aplicaciones composicionales de la computación:

2.3.1 Exploración de procesos matemáticos abstractos

Aquí pueden diferenciarse claramente aquellas técnicas que transcriben la salida de los procesos como eventos musicales discretos —típicamente en el nivel de nota o proporciones temporales— de aquellos que realizan síntesis de sonido. La aplicación de estas técnicas puede ir desde la mera traducción mediante aplicaciones lineales y mapeos diversos, a su utilización como marcos conceptuales que den soporte a otras manipulaciones menos automatizadas y más simbólicas.

[12] En España es ejemplar el interesante espacio de fecundo intercambio entre artistas y científicos de diversas disciplinas que se creó en los años 60 y 70 alrededor del Centro de Cálculo de la Universidad de Madrid [65].

[13] Järveläinen [29] ha realizado un mapa conciso de los principales enfoques de la música algorítmica.

Los algoritmos preferidos suelen ser aquellos que exhiben complejidad y riqueza de subprocesos internos. La *musicalización* tiene lugar sobre todo en las decisiones respecto a cómo se establecen las correspondencias entre la salida del algoritmo y su traducción a partituras o señales de audio, por lo que en estas técnicas, "*mapping is everything*" [59]. El valor de estas músicas no se sigue de una supuesta musicalidad intrínseca de los algoritmos, sino que depende exclusivamente del talento del compositor para transformarlos en texturas sonoras significativas.

El uso de la expresión *música fractal* es muy significativo respecto al uso y abuso que en muchos casos se ha hecho de los procedimientos matemáticos. Es fácil encontrar ejemplos de música que se autojustifica con cierta sacralización de sus supuestas propiedades de autosimilaridad, olvidando que las reglas de la percepción auditiva son muy diferentes.

Mandelbrot, uno de los principales descubridores de la geometría fractal, considera que en las obras de arte hay implícitas ciertas desviaciones de los objetos geométricos estándar. Resulta significativo que en su monografía sobre la fractalidad del mundo natural incluya algunos ejemplos de figuras defectuosas [36]: debido a algún gazapo en la programación del algoritmo que generaba el gráfico, estas imágenes erróneas exhiben interesantes deformaciones que, puestas junto a las figuras geométricamente perfectas, dan inmediatamente la impresión de que existe una personalidad creativa que hace una interpretación alterada y única de la realidad.

El uso de estas técnicas sólo ha sido verdaderamente productivo en manos de compositores para los que estos procedimientos han resultado inspiradores, y que al mismo tiempo no han perdido de vista las leyes que rigen la percepción musical humana.[14]

El auge de los autómatas celulares es algo posterior, y su *mística* propia también es diferente. Aunque el estudio de estos procesos dinámicos comenzó en los años 40 con von Neumann [66], su adecuación a la simulación por ordenador —dado que se trata de manipulaciones con variables discretas, fácilmente codificables en programas muy simples— produjo su popularización a partir de los años 70, con el famoso *Game of Life* de Conway. Los extensivos estudios posteriores de Wolfram [68] han dado pie a numerosas aplicaciones en la composición y la síntesis de sonido. Los experimentos de Miranda [11, 39] son algunos de los que han profundizado de manera más seria en las posibilidades de estos autómatas, tanto en la síntesis de sonido

[14] Las obras de Guerrero y Posadas son tal vez ejemplos de música que, aun basando sus estructuras en fractales, están configuradas sopensando cuidadosamente las facultades perceptivas reales, lo que hace que, en definitiva, más allá de la sonificación de complejas relaciones internas, sigan siendo esencialmente *música*. Al mismo tiempo, no hay que olvidar que, desde cierta óptica, toda la música tradicional tonal guarda evidentes autosimilaridades en sus estructuras formales y en sus diseños melódico–armónicos, sin que esto sea fruto de ninguna complicada maquinación, sino la consecuencia natural de la aplicación de unos mismos principios a varias escalas temporales. Desde este punto de vista, no es que exista música autosimilar, sino que lo difícil es encontrar música que no lo sea.

como en la de partituras.[15] Uno de los descubrimientos más importantes en este campo es la constatación de que autómatas celulares extremadamente simples pueden ser *máquinas de Turing universales* [15]. Esto implica que cualquier nivel de complejidad puede ser potencialmente alcanzado usando sólo estos mecanismos como base. En un plano musical esta idea es tremendamente sugerente, aunque llevarla a la práctica está aún en un horizonte lejano.

El interés de los músicos por estos procesos se centra en la fascinante capacidad de autoorganización que puede ser desplegada a partir de principios muy simples,[16] lo que se ha condensado con la etiqueta de *música generativa*. Si ubicamos la información percibida como musical en un área virtuosa situada entre el orden demasiado obvio y la excesiva complejidad, es comprensible el interés de los compositores por aquellos procesos que exhiben una mezcla equilibrada de azar y determinación, en la que pueden reconocerse subprocesos a varias escalas.

Al margen de las particularidades, existen lazos que conectan la geometría fractal, los sistemas caóticos y los autómatas celulares. Resulta significativa la observación de Voss y Clarke [67] de que el espectro de las señales musicales tiende a ser autosimilar en una relación $1/f$, siendo f la frecuencia analizada.[17] Esto conecta de manera sorprendente con el análisis que Ninagawa [40] hizo de un autómata celular muy simple equivalente a una máquina de Turing universal: el espectro del ruido generado por este autómata resultó corresponderse también con la relación $1/f$.

Hay que distinguir entre meras sonificaciones de estos procesos y trabajos auténticamente musicales. El papel del compositor se parece aquí más al de un *arreglista*, cuyo cometido es sacar el máximo partido expresivo de un material previo. El impacto musical del producto final es seguramente mucho más dependiente de la inteligencia de esas últimas elecciones que de las características de la *materia prima* —aunque aquí podría también argüirse que ¿acaso no fue siempre así?—; admitiendo que esta cuestión está llena de aristas estéticas, parece evidente que el uso de estos sistemas de CAC no exime al compositor de su responsabilidad última.

[15] Es notable que a partir de autómatas celulares simples o estructuras de simplicidad equivalente sea muy habitual que se obtengan estructuras fractales. Así, en cierto sentido las dos técnicas están ligadas, ya que la autosimilaridad puede incluirse como uno de los tipos de comportamiento que se generan con autómatas.

[16] En cierto sentido, el atractivo conceptual de estos principios conecta con la tradición, representada por Hanslick, de la búsqueda de la belleza formal mediante la derivación de todos los materiales a partir de pequeñas células temáticas.

[17] Mandelbrot [36] atribuye esta "propiedad escalante" de la música al hecho de que "las composiciones musicales son, como su propio nombre indica, compuestas". De nuevo, la fractalidad y la autorreferencia se producen por el trabajo simultáneo sobre varias escalas temporales, ya que "los maestros insisten en que hay que «componer» cada pieza musical hasta las menores subdivisiones significativas".

> *Mathematical chaos, when treated with insight, reveals astonishing beauty, a beauty whose regularity derives from the strictly deterministic techniques employed to give birth to it. Music, on the other hand, must deal not with number, but with real, sounding, materials. When treated with insight, these too reveal great beauty, rougher, less regular than fractal beauty, to be sure, but beauty within which the echoes of the real Chaos can clearly be heard.*[18] [6]

2.3.2 CAC a partir de gramáticas musicales

En este otro enfoque, la CAC parte de categorías más propias del análisis musical para manipularlas y obtener nuevos resultados. Al rebufo de la enorme influencia que tuvo la teoría de la gramática generativa enunciada por Chomsky no tardaron en aparecer intentos de aplicar estos mismos principios al lenguaje musical. Teóricos como Sloboda, Lerdhal o Jackendoff [31] han llegado muy lejos en la búsqueda de las leyes universales que condicionan la percepción musical.[19] El camino inverso al análisis ha sido la síntesis de música a partir de la formulación de gramáticas alternativas.[20]

La programación de reglas de inferencia gramaticales era ya posible con los primeros ordenadores, y tanto en la música electroacústica como en la CAC pronto se prospectaron estas técnicas. Roads [53] analizó en profundidad los diferentes enfoques con que puede tratarse la composición automatizada a partir de la definición de diferentes tipos de gramáticas.

Si en las técnicas del apartado anterior la clave reside en los procesos de conversión desde los datos en abstracto, en el caso de las gramáticas musicales lo que resulta más determinante es la potencialidad de los dominios que cada gramática crea, y las estrategias de búsqueda y selección con las que se exploran esos espacios. Basándose en la definición precisa de características musicales se han consolidado programas como *PWGL*[21] y *OpenMusic*[22], que basan parte de su poder en la capacidad de hacer búsquedas heurísticas a partir de los requerimientos del compositor.[23]

[18] *El caos matemático, cuando es tratado con perspicacia, revela una belleza deslumbrante, una belleza cuya regularidad deriva de las técnicas estrictamente deterministas empleadas para alumbrarlo. La música, por otra parte, se las tiene que ver no con los números, sino con los materiales sonoros reales. Cuando éstos se tratan con perspicacia también revelan una gran belleza, más áspera, con certeza menos regular que la belleza fractal, pero una belleza dentro de la cual los ecos del verdadero caos pueden ser claramente escuchados.* (Trad. del autor)

[19] El interés por la redefinición de gramáticas ha llegado incluso a métodos de enseñanza musical como el de Paynter [45], quien los usa como herramienta creativa y de reflexión sobre los mecanismos sintácticos que articulan el discurso musical a partir de actividades de iniciación muy sencillas.

[20] Para una exposición rigurosa del formalismo matemático de la gramática generativa, ver [3].

[21] `http://www2.siba.fi/PWGL`

[22] `http://repmus.ircam.fr/openmusic/home`

[23] Ambos descienden del paradigma de programación funcional del lenguaje *LISP*, apli-

Aunque estas técnicas pueden ser combinadas con las mencionadas anteriormente, en el caso de las gramáticas se han empleado con más asiduidad planteamientos cercanos a los parámetros tradicionales del análisis y la composición. En muchos casos se utilizan bases de datos, tablas de Markov o mecanismos similares para almacenar ciertos conocimientos previos que la máquina usa para derivar sus propias creaciones. El trabajo de Cope [16] ha sido especialmente exitoso en la imitación de estilos históricos.

Cada gramática pone el acento sobre ciertos aspectos de la textura musical, y se convierte de hecho en un sistema de representación.[24] Boulez [10] hace hincapié en la importancia de este hecho, y va aún más lejos considerando que las estructuras subyacentes de la gramática, más que herramienta de representación, *son* la propia música:

> *La música es un arte no significante; de ahí la importancia primordial de las estructuras propiamente lingüísticas ya que su vocabulario no podría asumir una simple función de transmisión.* [...] *Salta a la vista que el empleo de las palabras en un poema difiere fundamentalmente de la utilización corriente del vocabulario en una conversación. En música, por el contrario, la palabra es el pensamiento.*
>
> *¿Qué es entonces la música? A la vez un arte (medio de expresión), una ciencia y un artesanado* [...] *y el músico es a la vez un intelectual y un artesano.*

La configuración de una serie de reglas de inferencia está estrechamente ligada al sistema de representación con el que se *escriba* la música, y puede concebirse como un *metalenguaje musical*. Como ilustra la figura 2.1, desde la idea abstracta del compositor hasta la recepción íntima del oyente hay varios procesos sucesivos de traducción o *transducción*, cada uno con sus propias leyes y con su propia problemática. La gramática que articula cada uno de estos estratos configuran un *metalenguaje del nivel subsiguiente*.

Los sistemas de representación habituales de las diferentes técnicas de CAC pueden dividirse entre los que se centran en la partitura —que podemos llamar *gramáticas declarativas*, tales como *Csound*[25]— y los que modelan los procesos previos a la partitura —*gramáticas procedimentales*, como las de *PWGL* o *SuperCollider*[26]—. Ambos paradigmas pueden estar integrados: por ejemplo, en el caso de *Csound*, existen numerosas librerías y módulos que implementan una variedad de procedimientos que luego son convertidos a partituras.

Un punto primordial en los procesos de síntesis de partituras a partir de gramáticas es la gestión del flujo de información. El hecho de que una textura

cado sobre todo para la IA, que a pesar de su veteranía se ha revitalizado en los últimos años.

[24] Pueden hallarse interesantes reflexiones sobre los enfoques lingüísticos de la IA aplicada a la composición en [21].

[25] http://www.csounds.com

[26] http://supercollider.sourceforge.net

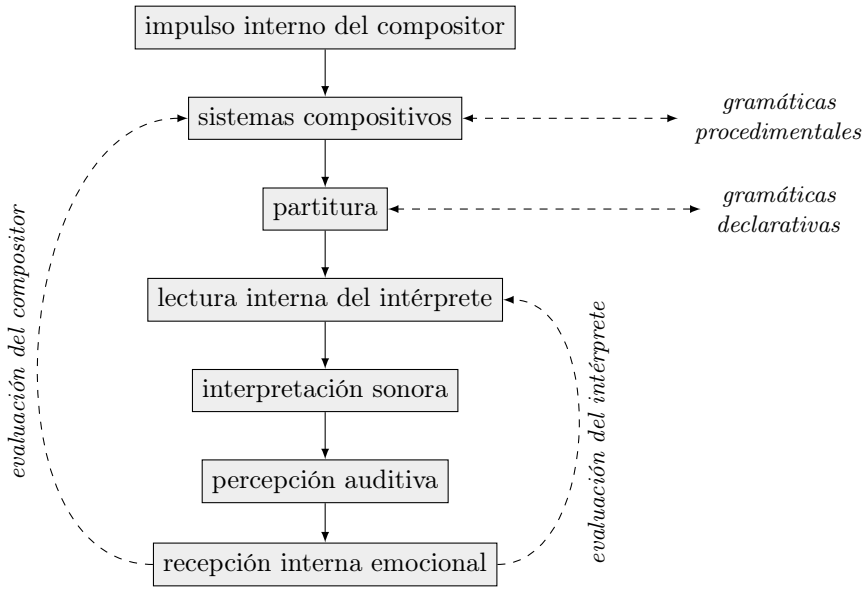


Figura 2.1: Procesos de *transducción* musical

musical esté perfectamente asentada sobre una lógica interna inferencial no implica que su recepción auditiva sea efectiva. La música serial fue ya un primer paso que en muchos casos sobrepasó ampliamente las posibilidades de la percepción de estructuras.[27] Los excesos del siglo XX en la complicación de la organización interna parece estar dejando paso a nuevas generaciones de compositores que ponderan mucho más las facultades cognitivas reales. En su teoría generativa de la música tonal, Lerdhal y Jackendoff [31] se refieren a esta cuestión:

> *Esta distinción que hemos hecho en la descripción de la música atonal y serial es igual o más pertinente todavía en el caso de los métodos de composición basados en la probabilidad, los métodos aleatorios, el serialismo extendido a la dimensión rítmica, u otros procedimientos que no requieren directamente la capacidad del oyente para organizar una superficie musical. En cada uno de estos casos, el abismo entre los principios compositivos y los perceptivos es extenso y profundo: en la medida en que no se cuenta con las capacidades del oyente, éste no puede deducir una organización rica, independientemente de cómo se haya compuesto la pieza o lo densa que sea su superficie musical. En este sentido es en el que una sonata de Mozart, sencilla en apariencia, es más compleja que muchas piezas del siglo XX que a primera vista parecen tremendamente complicadas.*

Finalmente, hay que distinguir nítidamente entre el uso del ordenador como mero asistente para el cálculo de estructuras diseñadas y modeladas por

[27] Muchos compositores seriales han admitido ser incapaces de distinguir música serial de la que no lo es.

un compositor externo, y los intentos de automatizar las decisiones estéticas —y en último término, de conseguir máquinas capaces de hacer música de calidad con autonomía—, lo que nos conduce al dominio de la IA.

2.4 Relaciones entre IA y CAC

La historia de la IA es irregular. Espectaculares avances se han seguido de etapas de cierto estancamiento, y las polémicas acerca de su verdadero significado y alcance persisten. Prácticamente todas las posibles técnicas de razonamiento automatizado han sido aplicadas a la música. Podemos distinguir tres grados en esta relación:

- Empleo del razonamiento automatizado para realizar *tareas especializadas*, para las que se programan algoritmos específicos.
- Uso de la máquina como herramienta de *creatividad asistida*, en la que se deja una amplia libertad a los algoritmos para explorar espacios musicales definidos conforme a ciertas reglas.[28]
- *Autonomía creativa completa*, mediante algoritmos que generan piezas musicales de manera autónoma.

No obstante, no se puede hablar propiamente de IA hasta que no existe una autoobservación de los procesos, es decir, hasta que no se establece un mecanismo de autoevaluación del sistema que permita una evolución autónoma de los productos de este razonamiento automatizado. Esta autoevaluación debe basarse en herramientas que analicen los resultados parciales que se van obteniendo. Es evidente que, más allá de ciertos parámetros superficiales de una composición, una evaluación estética real —esto es, experimentar placer en primera persona a través de cierto sentido de la belleza— implica cuestiones sociales, culturales, biológicas e incluso ideológicas que están fuera del alcance de las posibilidades actuales de las máquinas.

En IA existen dos posturas filosóficas contrapuestas: para los que defienden la IA *débil*, como Searle [63] la inteligencia puede ser imitada, pero la máquina nunca podrá experimentar una autoconsciencia real y, por consiguiente, nunca *sentirá* un verdadero placer estético; por contra, los defensores de la IA *fuerte*, encabezados por Turing, argumentan que nuestra propia consciencia no es sino el producto de una computación, y que a largo plazo la computación artificial conseguirá el mismo grado de percepción de sí misma que el ser humano. Llegados a este punto, serían las máquinas las que tomaran la iniciativa de crear sus propias producciones artísticas...

¿Cómo se recorrerá el camino hasta ese punto? La clave parece estar en las cada vez más abrumadoras pruebas que atestiguan que la complejidad

[28] La versión de *GenoMus* presentada en este trabajo se encuadra en este supuesto.

puede originarse desde la simplicidad extrema. El paradigma de la ingeniería clásica está cambiando hacia la idea de lograr dirigir estos procesos emergentes.[29] El punto clave sería pues articular algoritmos capaces de generar nuevos algoritmos más sofisticados. Los últimos avances se dirigen hacia la concepción de las llamadas máquinas de Gödel [61], que teóricamente podrían ser máquinas computacionales universales capaces de la resolución de cualquier tipo de problema mediante la reprogramación autónoma y la autorreferencia.[30] Al mismo tiempo, en los últimos años ha recobrado vigor la *teoría del todo* de Zuse [23], que postula una redefinición de las leyes físicas —y de todas las que emanan de ellas, incluidas las de la creatividad de la mente humana— en términos de pura computación.

2.4.1 Estrategias bioinspiradas

Los mecanismos de programación basados en la evolución y selección, representados principalmente por los algoritmos genéticos, llevan varias décadas experimentándose con resultados sorprendentes, sobre todo al aplicarlos a la búsqueda de soluciones a problemas bien definidos.[31] En la CAC existen numerosos experimentos que combinan sistemas de representación musical con algoritmos que buscan las mejores soluciones mediante cruce y mutación de generaciones sucesivas de los especímenes musicales generados[32]. Beyls [8] ha realizado interesantes combinaciones entre autómatas musicales y algoritmos evolutivos. El arte generativo se ha convertido casi en un género,[33] y en todas las disciplinas artísticas se están explorando las posibilidades creativas de las sociedades de individuos virtuales [32, 54]. Recientemente, el proyecto *Iamus* [5] ha llegado a un nivel de sofisticación en sus resultados que permite cuestionar si ya se ha superado un hipotético *test de Turing musical*. El uso de las llamadas redes neuronales se ha utilizado también para simular procesos de aprendizaje musical [42].

Ampliando la perspectiva histórica, puede considerarse que ya en los primeros postulados serialistas de Schönberg y Webern se barruntaba la idea de superar el tradicional *tema* y crear un *germen*. Más tarde, Stockhausen [64] acuñaría el término de *fórmula* como principio generador anterior a la *forma*:

[29] La investigación con células madre es un claro ejemplo de este cambio fundamental de la ingeniería aplicada a la medicina.

[30] Estos conceptos son significativamente similares a los que usa Hofstadter [25] para afirmar que la idea del yo y la autoconsciencia aparece cuando el pensamiento llega a ser capaz de la autorreferencia y el razonamiento recursivo.

[31] Son especialmente llamativos los experimentos relacionados con la locomoción animal, cuyos resultados coinciden notablemente con los desarrollados por la evolución biológica real.

[32] Existen también trabajos que, en la línea de *GenoMus*, aplican las técnicas evolutivas a gramáticas musicales [44].

[33] Existen incluso festivales centrados en estas investigaciones artísticas, tales como el *EvoMUSART*.

> *La fórmula es como el semen del hombre que fecunda a la mujer. Encuentro increíble que un elemento microscópico, invisible a simple vista, sea capaz de crear la vida de un ser humano dotado de infinitas características hereditarias. Es un misterio que afecta también a la genética musical.*
>
> *Componer sirviéndose de una fórmula surgida de un proceso mental, inteligente, intuitivo, ponderado e imaginado no es otra cosa que procreación. Esta descendencia musical y artística va acompañada de un modelo único y coherente que tiene un lugar en el orden general del universo.*

Schaathun [58] aclara este concepto de un modo menos alegórico:

> *What makes these immanent structures more audible, is that this approach to composition also takes care of the «higher levels», in the musical hierarchy: in addition to forming the basis of the microstructures, the individual note, the various rhythmic figures etc., it also takes into account the proportions of sections and, finally, the overall form of a composition.*[34]

Se han realizado numerosas aproximaciones para llevar a la práctica gramáticas que trabajen sobre estos modelos de germen musical. En muchos de estos trabajos se habla de *genotipos musicales* para designar una "fórmula" que, una vez procesada o decodificada, genera un *fenotipo musical* —normalmente una partitura o una señal de audio—. *GenoMus* usa también esta nomenclatura.

2.4.2 Metaprogramación y programación funcional

Una vez definida una gramática para la codificación de los genotipos musicales se abre un espacio de posibilidades a explorar. Para llevar a cabo esta prospección se hace necesario escribir los programas que posibiliten la escritura automática de genotipos —que no son sino otro tipo de programas—, y esto se consigue con la *metaprogramación*.

La metaprogramación, definida como la *escritura de programas que a su vez escriban otros programas*, es un campo en pleno desarrollo. Expertos como Rideau [52] la defienden como la programación del futuro, y Wolfram [68] afirma que en la capacidad de autoorganización de programas simples reside la verdadera esencia de la computación. Si el futuro de la IA pasa por la creación de algoritmos que puedan generar otros algoritmos, es imprescindible investigar cómo un fragmento de código puede llegar a escribir código más

[34] *Lo que hace estas estructuras más audibles es que este enfoque a la composición también atañe a los «niveles superiores» de la jerarquía musical: además de conformar la base de las microstructuras, la nota individual, las diversas figuras rítmicas, etc., también toma en consideración las proporciones de las secciones y, finalmente, la forma global de una composición.* (Trad. del autor)

sofisticado. Las analogías con la evolución biológica son obvios.[35] *GenoMus* está diseñado principalmente para explorar las posibilidades que ofrece el diseño de un metalenguaje que permita la metaprogramación de sus propias expresiones gramaticales. Uno de los paradigmas de programación que permite este código multinivel es la denominada *programación funcional*.

La programación funcional se basa en la utilización de funciones con un comportamiento interno independiente, es decir, que no afecta a variables externas ni es afectado por ellas. La idea principal es que cada procedimiento funcione de manera análoga a una función matemática estándar, y que por tanto esté perfectamente definida y su comportamiento sea idéntico y previsible en cualquier contexto del flujo de datos. La programación funcional es heredera directa de la notación empleada por el *cálculo lambda* (*λ–calculus*),[36] un formalismo ideado en los años 30 para la investigación del concepto de función, sus aplicaciones y la recursión. Muchos lenguajes —*LISP* o *Haskell* entre los más usados— pueden ser contemplados como aplicaciones del cálculo lambda, o pueden ser configurados para que en la práctica sean equivalentes.[37]

Aunque la programación funcional fue considerada durante mucho tiempo como algo más propio de los ámbitos académicos, alejada de la programación orientada a objetos usada en el desarrollo de aplicaciones comerciales, este concepto está cambiando en la última década.[38]

En la CAC se han creado bastantes aplicaciones basadas en este paradigma, como el lenguaje CANON de Dannenberg [17], o el sistema de la Universidad de Yale, basado en *Haskell* [26]. Las consecuencias musicales del uso de la programación funcional han sido analizadas por Hughes [27].

2.4.3 El problema de la evaluación estética automatizada

Ya se han considerado las herramientas necesarias para la escritura automatizada de música, pero una vez obtenido cualquier producto musical se necesita un mecanismo de autocrítica que permita la selección y evolución de los resultados. Una de las características esenciales de la inteligencia es la capacidad de evaluar sus propios razonamientos y acciones. La evaluación de

[35] Hofstadter [24] ilustra los mecanismos de metaprogramación con el funcionamiento de las enzimas: el código genético de una enzima, al activarse, realiza la síntesis de otros códigos genéticos que a su vez pueden formar otras enzimas, y así sucesivamente hasta llevar a cabo complejísimas operaciones dentro de la célula.

[36] Un tratado completo sobre el cálculo lamba y su aplicación en la programación funcional puede encontrarse en [51].

[37] La sección 6.2 expone cómo emplea *GenoMus* este tipo de estructura para construir su metalenguaje musical característico, mediante un uso de *JavaScript* basado en la programación funcional.

[38] El éxito de *software* basado en este tipo de programación, también llamada *simbólica*, como *Mathematica*, es un ejemplo claro de su potencialidad, especialmente en aplicaciones que requieren formalismos rigurosos.

procesos con fines bien definidos —como obtener la estructura de mayor resistencia, el sistema de locomoción más eficaz o la mejor jugada de ajedrez— se asienta en la medición de variables objetivas. En el caso de la CAC nos volvemos a topar con una pregunta fundamental problemática: *¿cuál es el objetivo último de la música? ¿cómo se mide la calidad de una composición, o el placer estético que es capaz de suscitar?*

La experiencia de todo músico corrobora que, en música, las categorías de corrección o coherencia no aportan casi nada a la valoración estética de un fragmento musical. Existe mucha música correcta y coherente —*bien formada* en el sentido gramatical— que es absolutamente prosaica, y viceversa. La historia de la codificación e implantación de la fuga escolástica en el ámbito académico da buena cuenta de lo estéril y absurdo de los planteamientos formalistas llevados al extremo. La didáctica del contrapunto ha sido objeto de controversias entre enfoques mecanicistas, que se reducen a la asimilación de reglas de formación correcta que bien pueden ser implementadas en un ordenador, y otros que tratan de tomar una perspectiva más amplia y menos dogmática para captar la esencia de la polifonía.[39]

Esta dicotomía en los paradigmas pedagógicos reproduce en cierto modo dos de las líneas básicas en que se ha desarrollado la IA:

- la vía tradicional, de pura transmisión de conocimiento previo, que equivaldría a la implementación de *sistemas expertos* —capaces de hacer muy bien una tarea, pero muy limitados para trascender sus resultados—,

- y una vía opuesta, propia de pedagogos más progresistas, basada en el estímulo de las capacidades cognitivas, equivalente en IA a los enfoques basados en la modelización de los procesos de aprendizaje y discernimiento —en los que no es tan importante *qué* se hace sino *cómo*—.

El proyecto *GenoMus* se encuadra en esta segunda línea: se trata de abrir espacios creativos y no de mecanizar técnicas compositivas concretas.

Volviendo al problema de cómo lograr que un algoritmo evalúe la calidad musical de sus propios resultados, hay que tener muy en cuenta lo que Meyer [38] afirma acerca de la percepción del valor de una obra musical:

> *Entendemos y apreciamos una obra de arte no sólo en términos de las posibilidades y probabilidades efectivamente realizadas, sino en términos de nuestra impresión de lo que podría haber ocurrido en un contexto compositivo específico. Esto tal vez será especialmente claro en la música.* [...]

[39] De la Motte [18] hace una aguda crítica a la enseñanza tradicional del contrapunto en especies, y propugna que se debe buscar la musicalidad desde el principio porque el mero ejercicio de una regla técnica no es significativo si está desconectada de una idea musical directora.

> *No sólo la comprensión depende del conocimiento estilístico; la evaluación también es dependiente, porque los modelos que surgen de las elecciones reales del compositor se juzgan y entienden desde el punto de vista de las opciones cuya disponibilidad se conocía dadas las constricciones del estilo que empleaba.*

Esto implica que hay que diferenciar bien entre la evaluación de un trabajo de imitación de estilos y otro en el que se busca desarrollar un estilo personal. En el primer caso es posible dotar a la máquina de muchos conocimientos (y valoraciones) previas, y después de cierto tiempo de exposición es factible desarrollar cierto sentido crítico automático. En el segundo, que es el aplicable cuando se trata de buscar la creatividad artificial y la emergencia de estilos individuales distinguibles, resulta enormemente escurridizo establecer criterios válidos de evaluación. Por otra parte, todo esto no difiere mucho de los problemas que enfrenta la crítica y la autocrítica musical humana.

El uso de nuevas técnicas compositivas, como las propias de la CAC, se sitúa por definición en una *terra incognita* musical donde es muy difícil ejercer una crítica sólida, pero al mismo tiempo están dotadas de un poder computacional enorme, que permite explorar un gran número de posibilidades en la búsqueda de estructuras musicales interesantes. Por el momento, en el extremo de cualquier proceso automatizado se encuentra un programador–compositor humano que, más o menos elusivamente, define criterios de selección. En el caso de músicas donde se abre tanto campo al azar, estos criterios *son* la composición, y serían los que finalmente definen el estilo y detentan la responsabilidad última del resultado estético:

> *Las elecciones humanas algo tienen que ver en la producción de música aleatoria.* [...] *Los compositores aleatorios realizan elecciones, pues, no en el nivel de las relaciones de sonido sucesivas dentro de las obras, sino en el nivel de las constricciones precompositivas* [...]. *Como resultado, parece razonable argüir que, aunque el estilo no juega ningún papel en la comprensión y experiencia de estas piezas por parte del oyente, el comportamiento del compositor tiene estilo y por este motivo puede ser evaluado.* [38]

2.4.4 Una teoría del placer estético

Algunos analistas han tratado de encontrar patrones comunes en la experiencia estética. Y, especialmente en el arte conceptual del siglo XX, parece que se valora la simplicidad y la concisión en la plasmación de una idea. El placer del científico al encontrar la fórmula más simple que explica un fenómeno tendría así un paralelo en las creaciones artísticas que son capaces de condensar un pensamiento, una emoción, en la mínima cantidad de información. Schmidhuber [60] ha formulado una teoría del placer estético centrada en la *compresión* de la información. Según ella, nos produce placer encontrar

la manera más concentrada y eficaz de expresar algo. Así, la mejor versión de un aforismo, un refrán o un chiste, es la que emplea menos información para hacer llegar el mensaje.

> *The principle of Occam's razor is not only relevant to science and mathematics, but to fine arts as well. Some artists consciously prefer "simple" art by claiming: "art is the art of omission". Furthermore, many famous works of art were either consciously or unconsciously designed to exhibit regularities that intuitively simplify them. For instance, every stylistic repetition and every symmetry in a painting allows one part of the painting to be described in terms of its other parts. Intuitively, redundancy of this kind helps to shorten the length of the description of the whole painting, thus making it simple in a certain sense.*[40]

Podemos así entender que el análisis musical busque extraer los elementos verdaderamente significativos de una estructura, y reducir una compleja superficie a unos pocos principios constructivos. Y en sentido inverso, debe producirnos placer la posibilidad de componer densas redes de relaciones musicales partiendo de principios muy simples. Muchos de los sistemas compositivos estructuralistas postonales pueden entenderse mejor con esta mirada.

[40] *El principio de la navaja de Occam no es sólo relevante para la ciencia y las matemáticas, sino también para las bellas artes. Algunos artistas prefieren conscientemente el arte "simple" con esta consigna: "el arte es el arte de la omisión". Además, muchas obras de arte famosas fueron consciente o inconscientemente diseñadas para exhibir regularidades que intuitivamente las simplifican. Por ejemplo, en un cuadro cada repetición estilística y cada simetría permiten que una parte de la pintura pueda ser descrita en función de sus otras partes. Intuitivamente, redundancias de este tipo ayudan a acortar la longitud de la descripción del cuadro completo, haciéndolo así simple en cierto sentido.* (Trad. del autor)

3

Objetivos

1. Fijar las bases conceptuales y prácticas de un metalenguaje musical basado en una librería de procesos composicionales abstractos.

2. Comprobar la flexibilidad expresiva de este metalenguaje usándolo en diferentes contextos instrumentales y estilísticos.

3. Completar un primer prototipo básico de *GenoMus* en *JavaScript* para visualizar la totalidad de mecanismos necesarios y así poder orientar una futura reimplementación del sistema en lenguajes más robustos.

4. Construir una interfaz para trabajar con este lenguaje, la cual permita su desarrollo y evaluación en tiempo real, y además facilite la manipulación de los fenotipos para su aplicación en la composición.

5. Ampliar la paleta expresiva propia mediante la creatividad asistida por computadora, usando *GenoMus* para evitar los propios prejuicios técnicos y estéticos mediante la exposición a muy numerosos y variados materiales propuestos por el sistema.

6. Componer y estrenar obras compuestas con genotipos producidos durante las diferentes etapas de evolución del proceso, para comparar la huella de las diferentes técnicas usadas.

7. Completar un ciclo completo de investigación y creación que guíe el diseño de las siguientes fases del proyecto.

4

Hipótesis

En el núcleo de esta investigación están los conceptos de genotipo y fenotipo musical, cuyo significado exacto en este contexto se delimita con detalle en la sección 6.2. Tomando prestada esta nomenclatura de la biogenética, se formula una hipótesis primaria:

> *Es posible crear un metalenguaje con el que representar procedimientos musicales que pueda ser lo suficientemente preciso, sencillo y modular como para permitir la metaprogramación de genotipos y la manipulación y evaluación de los fenotipos generados, todo ello en tiempo real.*
>
> *Además, más allá del formalismo teórico, es posible realizar composiciones viables usando este metalenguaje como herramienta principal.*

Hay que remarcar dos puntos que diferencian este planteamiento del de otros trabajos similares:

- Pone el énfasis en la creación de un metalenguaje que facilite su manejo autónomo, es decir, que esté orientado a la creación automática de expresiones sintácticamente bien formadas mediante la *metaprogramación*. Y si bien la formulación de gramáticas generativas y la creación de algoritmos para su manipulación se practica desde hace décadas, en el caso de *GenoMus* la gramática no se refiere directamente a eventos musicales, sino a los procesos que los organizan. En cierto sentido, un genotipo modeliza el pensamiento musical, mientras que su fenotipo correspondiente constituye una de los partituras posibles que ese procedimiento concreto puede originar.

- Considera parte fundamental de la investigación la aplicación del sistema propuesto en composiciones reales —no meramente ilustrativas del método— que cubran todo un ciclo creativo, incluyendo el estreno y recepción de las obras en un ambiente normalizado de difusión musical.

Por último, para no perder perspectiva es conveniente recordar aquí el sentido global de esta investigación, tal como se planteó en la introducción. La hipótesis enunciada es sólo un primer escalón que se inscribe en otra hipótesis de mucho más alcance que, aunque queda lejos de los resultados que aquí se presentan, constituye el horizonte al que dirigirse:

> *Es posible la composición musical artificial autónoma de nivel artístico superior, la cual evolucione mediante la comunicación con oyentes humanos o artificiales.*
>
> *Además, este conocimiento virtual, más allá de la mera imitación de modelos, puede fundamentarse en la comprensión de procesos compositivos y exhibir un comportamiento inequívocamente creativo.*

Abundando en estas últimas ideas, en el capítulo 8 se recogen algunos apuntes sobre las directrices que pueden guiar las siguientes fases de esta investigación para abordar esta última hipótesis.

5

Metodología

> *Computer programming is tremendous fun. Like music, it is a skill that derives from an unknown blend of innate talent and constant practice. Like drawing, it can be shaped to a variety of ends —commercial, artistic, and pure entertainment. Programmers have a well-deserved reputation for working long hours but are rarely credited with being driven by creative fevers. Programmers talk about software development on weekends, vacations, and over meals not because they lack imagination, but because their imagination reveals worlds that others cannot see.*[41]
>
> Larry O'Brien y Bruce Eckel [41]

5.1 Base de partida

5.1.1 El enfoque procedimental

El punto esencial que ha condicionado toda la metodología del proyecto es la decisión de representar la música desde un metanivel procedimental. En la base de esta investigación subyace la idea de modelizar la mente del compositor y no la partitura. Desde el inicio se puso el foco en la codificación de los procedimientos que generan las secuencias de eventos, y no en las secuencias mismas. Antes de acometer cualquier tarea de programación, se examinaron diferentes aproximaciones al problema de crear un metalenguaje

[41] *La programación informática es tremendamente divertida. Como la música, es una destreza que procede de una mezcla desconocida de talento innato y práctica constante. Como el dibujo, puede tomar forma con una variedad de fines —comercial, artístico o de puro entretenimiento. Los programadores tienen una bien merecida fama de trabajar durante largas horas, pero rara vez se les atribuye estar dirigidos por una fiebre creativa. Los programadores hablan sobre desarrollo de software en fines de semana, vacaciones o durante las comidas, no porque carezcan de imaginación, sino porque su imaginación revela mundos que otros no pueden ver.* (Trad. del autor)

musical que pudiera ser tan preciso como la partitura a la que se refiere, pero que al mismo tiempo reflejara los procesos que la originan. Estos primeros trabajos fueron decantándose rápidamente hacia el paradigma de la *programación funcional*. El siguiente paso fue elegir las herramientas informáticas adecuadas para hacer un primer prototipo de las ideas maduradas sobre el papel.

5.1.2 Programar es (meta)componer

La labor de diseñar algoritmos que generen música —o cualquier otro tipo de producto artístico— implica gran cantidad de decisiones que comportan una visión estética. Cada *software*, por muy abierto y flexible que sea, deja una huella en las salidas que produce. En la concepción de *GenoMus* ha estado muy presente la idea de que el diseño de una herramienta como esta es en primera línea un acto creativo más: al programar no se compone música, pero sí se establecen marcos y espacios en los que existe música potencial. En este sentido, *la programación se convierte en un acto de metacomposición.*

Uno de los objetivos principales del proyecto es conseguir un método de trabajo que eluda los prejuicios del compositor y expanda su paleta expresiva. La metodología empleada persigue una mayor independencia de la máquina en el momento de hacer propuestas, y la máxima apertura de los espacios musicales definibles; busca también facilitar las fases futuras del proyecto, centradas en la evaluación estética autónoma y autoevolutiva. Una buena arquitectura de representación musical es esencial para aplicar la potencia computacional a las gramáticas recursivas y abiertas sobre las que se asienta esta investigación.

5.2 La programación funcional como paradigma

La programación funcional está basada en la definición de funciones cuyo comportamiento es cerrado e independiente del programa en que están insertadas. El resultado de su evaluación no depende de estados previos del programa principal, y su salida tampoco crea efectos colaterales en el mismo. Un programa escrito con esta técnica puede considerarse en la práctica como un conjunto de funciones matemáticas abstractas que operan sobre un número y tipo determinado de variables, y que producen una salida bien definida. Una ventaja de este concepto es que, una vez definidas las funciones, es relativamente sencillo portarlas a cualquier lenguaje que soporte la programación funcional.

El elemento clave de toda la metodología de esta investigación gira en torno al formalismo de la programación funcional, cuyos principios teóricos y origen se han detallado en el punto 2.4.2. Este modelo de programación puede aplicarse de muy diferentes modos. En el caso de *GenoMus*, el concepto fundamental es que tanto los genotipos musicales como el código que

los genera y opera sobre ellos están basados en programación funcional, y en último término *se identifican procesos y datos*. Dicho de otro modo, las funciones pueden ser argumentos y viceversa.

5.2.1 Los genotipos como árboles de funciones

El metalenguaje de *GenoMus* parte de un modelo de genotipo musical que puede contemplarse como una estructura ramificada de funciones. Esta estructura es una cadena de caracteres que representa un árbol de procedimientos, y se sintetiza a partir de una librería de funciones que codifican manipulaciones musicales. En el tronco se encuentra una función principal cuyos argumentos son valores numéricos o funciones. Estas funciones pueden llamar a otras funciones, y así sucesivamente. El nivel de ramificación es indefinido y ha de limitarse. Cada genotipo es de hecho una nueva función de funciones, que se evalúa como código operativo en *JavaScript*. Un genotipo puede archivarse a su vez como una función más de la librería, que a su vez pueda ser llamada como argumento de otras funciones de nivel superior.

5.2.2 Código para operar con los genotipos

El código interno de *GenoMus* se articula como un conjunto de funciones independientes que pueden agruparse en familias, dependiendo de las tareas que realizan:

- *Librería de funciones musicales*, que son las piezas básicas que conforman los genotipos.
- *Operaciones sobre los genotipos*, tales como síntesis de expresiones bien formadas a partir de la librería, análisis, conversiones, formato, etc.
- *Funciones de comunicación con la interfaz externa*, que facilita la manipulación en tiempo real dentro de un *patch* de *MaxMSP*.
- *Funciones auxiliares*, que cubren múltiples necesidades prácticas del sistema.

5.3 Criterios para el prototipado del metalenguaje musical procedimental de *GenoMus*

Todo el sistema *GenoMus* gira alrededor del lenguaje de representación musical que conforman los genotipos. Como este lenguaje se refiere a procedimientos musicales y no a los eventos musicales generados, se ha dado en referirse a él como *metalenguaje musical procedimental*. Establecer principios adecuados para el prototipado de su gramática es el aspecto más importante de la metodología, ya que condiciona la flexibilidad, potencia expresiva,

facilidad de manejo y capacidad de evolución en la implementación práctica de este metalenguaje.

A continuación se detallan los criterios que se han tenido en cuenta para su diseño.

- **Conectividad, recursividad y estandarización**
  Puesto que la potencia de *GenoMus* radica en la posibilidad de que las funciones se llamen de manera autónoma unas a otras, es esencial fijar un prototipo de función que permita la máxima conectividad. Esto se consigue reduciendo los tipos de argumentos que una función puede requerir —o lo que es lo mismo, los tipos de salida que una función puede producir—, y homogeneizando los rangos de las variables. El prototipo de función básica se detalla en la sección 6.2. Sobre la base de este formalismo cualquier otro programador puede diseñar e incluir nuevas funciones en la librería.

- **Flexibilidad en el mapeo**
  Ya que el sistema comienza enlazando funciones de manera aleatoria, uno de los primeros problemas prácticos que surgen es la dificultad para que las variables se mantengan en rangos convenientes. Para permitir una alta tolerancia en estas combinaciones, algunas funciones incorporan mapeos internos que reubican los valores fuera de rango dentro de los límites requeridos. Cuando surgen problemas insalvables se aborta el proceso y se da paso a una nueva opción.

- **Límite a la recursividad**
  Para evitar el tamaño excesivo de los genotipos, al inicializar el proceso se impone un tope al nivel de ramificación que puede alcanzarse. Cuando se ha llegado a ese límite, el sistema fuerza terminar el crecimiento de esa rama usando una *función primitiva*, que no requiere ningún argumento.

- **Legibilidad para la operación manual**
  Los genotipos se usan como herramientas creativas que pueden ser manipuladas directamente por el compositor, por ello debe facilitarse la lectura y la interacción manual. Esto exige que la presentación del código de los genotipos, que puede llegar a ser enormemente intrincado, esté visualmente bien estructurada, y que la dependencia entre las funciones y sus argumentos pueda identificarse con facilidad. Además, en esta primera etapa de desarrollo, la legibilidad de los genotipos es fundamental para testar y depurar el código.

- **Isomorfismo entre la forma legible de los genotipos y la forma compacta en forma de matriz unidimensional**
  Un genotipo puede representarse de dos maneras:

  - como una cadena de texto, que se evalúa como una función estándar de *JavaScript*,
  - o encriptado como un *array* unidimensional (una secuencia de números).

  Existe un isomorfismo entre ambas formas de representación de genotipos, por lo que la conversión es unívoca y puede llevarse a cabo en cualquier sentido sin perder información. Como se verá en la sección 6.5.3, la posibilidad de expresar un genotipo en forma de secuencia de números es muy conveniente para el uso de procesos iterativos y para referenciar subprocesos que sean parte de estructuras mayores. Además, en futuros desarrollos del sistema, la reducción de los genotipos a *arrays* unidimensionales puede simplificar y acelerar todos los procesos internos previos a la salida en forma legible.

- **Inclusión e integración de cualquier técnica compositiva**
  GenoMus no impone ninguna estética o técnica concreta. No es más que una estructura modular que puede dar cabida a cualquier algoritmo que desee incluirse en la librería de funciones disponibles. Ya sean cadenas de Markov, autómatas celulares o reglas del contrapunto en especies, una vez incluidas en la librería siguiendo la plantilla de función estándar, pueden ser usadas por un genotipo. Las facilidades que la interfaz de *GenoMus* ofrece para trabajar en tiempo real con los fenotipos facilita la experimentación automática o manual con cualquier procedimiento compositivo.

5.4 Herramientas utilizadas

JavaScript

El núcleo de *GenoMus* está construido con funciones de *JavaScript*. Aunque otros lenguajes de programación, como *LISP* o *Haskell*, tienen mucha más tradición en la investigación en IA, para el desarrollo de *GenoMus* se optó por *JavaScript*. Esta elección se apoya en varias razones:

- *Se adapta fácilmente a la programación funcional.* A pesar de que *JavaScript* está orientado a un uso de alcance limitado, y está ideado sobre todo para la incrustación de pequeñas aplicaciones interactivas en páginas web, su flexibilidad permite adaptarlo muy bien a la programación funcional [9]. Los *argumentos* o parámetros de una función

de *JavaScript* pueden ser asimismo funciones, y esto permite la recursividad que una gramática generativa como la de *GenoMus* requiere.

- *Permite la metaprogramación*. La función `eval`[42], emparentada directamente con su homóloga en *LISP* permite evaluar una cadena de caracteres como código dentro del propio tiempo de ejecución del código principal. Esta capacidad del código de evaluarse y reprogramarse a sí mismo es fundamental para la evolución de los genotipos musicales.
- *Está integrado en MaxMSP*, gracias al objeto `js`. Esto agiliza la experimentación en tiempo real con los genotipos, ya que el código de *JavaScript* se puede actualizar al instante, al mismo tiempo que se dispone de todas las facilidades para trabajar con audio y MIDI, propias de *MaxMSP*.
- *El código es fácilmente legible y similar a C*. Esto facilitaría una posible reimplementación futura más potente y compacta en este último lenguaje.
- *Facilita el trabajo a un programador inexperto*. Algunas de estas características ventajosas son las variables de tipos dinámicos, los *arrays* de tamaño indefinido —las matrices unidimensionales que constituyen los fenotipos de *GenoMus*—, y la gestión automática de la memoria. Esto posibilita centrarse en el diseño de los algoritmos evitando problemas de optimización de código.

MaxMSP

El código de *JavaScript* se ha programado en el marco que ofrece el entorno *MaxMSP*. Estos son algunos de los motivos de esta elección:

- *Su modularidad posibilita conectar otros objetos a la entrada o salida de datos de GenoMus* y convertir los fenotipos obtenidos en otros formatos con rapidez (MIDI, audio, texto plano, archivos de texto, imagen fija o vídeo).
- *Facilita escuchar y visualizar en tiempo real los fenotipos generados*, lo que es indispensable para que la interacción creativa sea ágil y productiva.
- *MaxMSP se ha convertido en el estándar de facto para la creación de aplicaciones musicales interactivas*. La amplísima implantación de este *software* permite una fácil difusión y reutilización de *GenoMus* por parte de otros artistas.

[42] En *LISP* y *JavaScript*, la función `eval` evalúa (ejecuta) una cadena de texto como código de su propio lenguaje de programación. Este mecanismo es el que permite que un programa escriba código y lo ejecute de manera autónoma.

6

GenoMus

Quiero construir la música como
está construido un árbol.

Francisco Guerrero Marín[43]

El sistema de CAC denominado *GenoMus* es la principal aportación de esta investigación. Se trata de una herramienta original que aquí se presenta en una versión inicial aunque ya operativa. Este capítulo detalla cada uno de los aspectos de su funcionamiento.

Una de las características esenciales de *GenoMus* es su *énfasis en la creatividad asistida*. Su función central no es computar estructuras musicales dictadas por el compositor humano —aunque esto también es posible—, sino aportar una inagotable variedad de materiales sobre los que trabajar. La interacción entre hombre y máquina tiene lugar mediante la propuesta y el modelado en ambos sentidos: *GenoMus* puede proponer fragmentos musicales de manera totalmente autónoma, basarse en ciertos requerimientos determinados, o crear variantes a partir de ideas externas.

El otro aspecto central de esta herramienta es su *focalización en los procesos* que estructuran el discurso musical. En esta primera fase del proyecto se ha otorgado la máxima importancia al diseño de un lenguaje modular que represente los procedimientos que generan la partitura, y no la partitura misma. Todo el sistema gira alrededor de ese metalenguaje, por lo que un prototipado inteligente y flexible es la clave de una óptima implementación.

La versión actual de *GenoMus* en *JavaScript* puede considerarse aún un boceto. Hay que señalar que lo más importante de este capítulo es el *paradigma* que se propone, antes que los detalles del código.

[43] Frase atribuida a Guerrero en numerosas ocasiones por músicos y musicólogos cercanos al compositor, tales como Russomanno [55], entre otros.

6.1 Estructura de *GenoMus*

GenoMus está formado por tres componentes principales:

- Un *metalenguaje musical procedimental*, que es la gramática de representación musical que constituye el núcleo del sistema. Este metalenguaje se articula en *genotipos* y *fenotipos*. Los genotipos codifican los procedimientos, y sus fenotipos correspondientes son la realización de estos procesos en fragmentos musicales concretos. La sintaxis de este metalenguaje basado en programación funcional está documentada en la sección 6.2.

- Un conjunto de *funciones para operar con este metalenguaje*, codificadas en *JavaScript*, que podemos clasificar según su cometido:

  - *Librería de funciones disponibles con las que formar los genotipos.* Cada una de estas funciones realiza un procedimiento concreto, más o menos cercano a procesos musicales habituales. Una de estas funciones es, por ejemplo, `scoRep`, que repite un fragmento dado un número determinado de veces.
  - *Funciones para la creación y manipulación de genotipos.* Este grupo realiza todas las operaciones necesarias para la escritura automática de expresiones genotípicas bien formadas, y opera sobre los genotipos para hacerlos crecer, mutar, compactar, formatear, etc.
  - *Funciones que operan sobre los fenotipos.* Algunas de estas operaciones son la creación de fenotipos a partir de los genotipos, el análisis y filtrado según ciertos requerimientos, etc.
  - *Funciones auxiliares*, que realizan todo tipo de tareas necesarias para articular internamente los grupos anteriores, y para comunicarse con la interfaz del usuario.

- Una *interfaz* externa, ubicada en un patch de *MaxMSP*, que gestiona la entrada y salida de información.

La figura 6.1 representa la estructura básica del funcionamiento interno de *GenoMus*, y su integración en el entorno de otros programas. Es importante notar que se establece un flujo de datos que posibilita la retroalimentación del sistema: así, determinados fenotipos pueden integrarse en la estructura de los genotipos, y a su vez los genotipos seleccionados pueden llegar a formar parte de la librería de funciones disponibles.

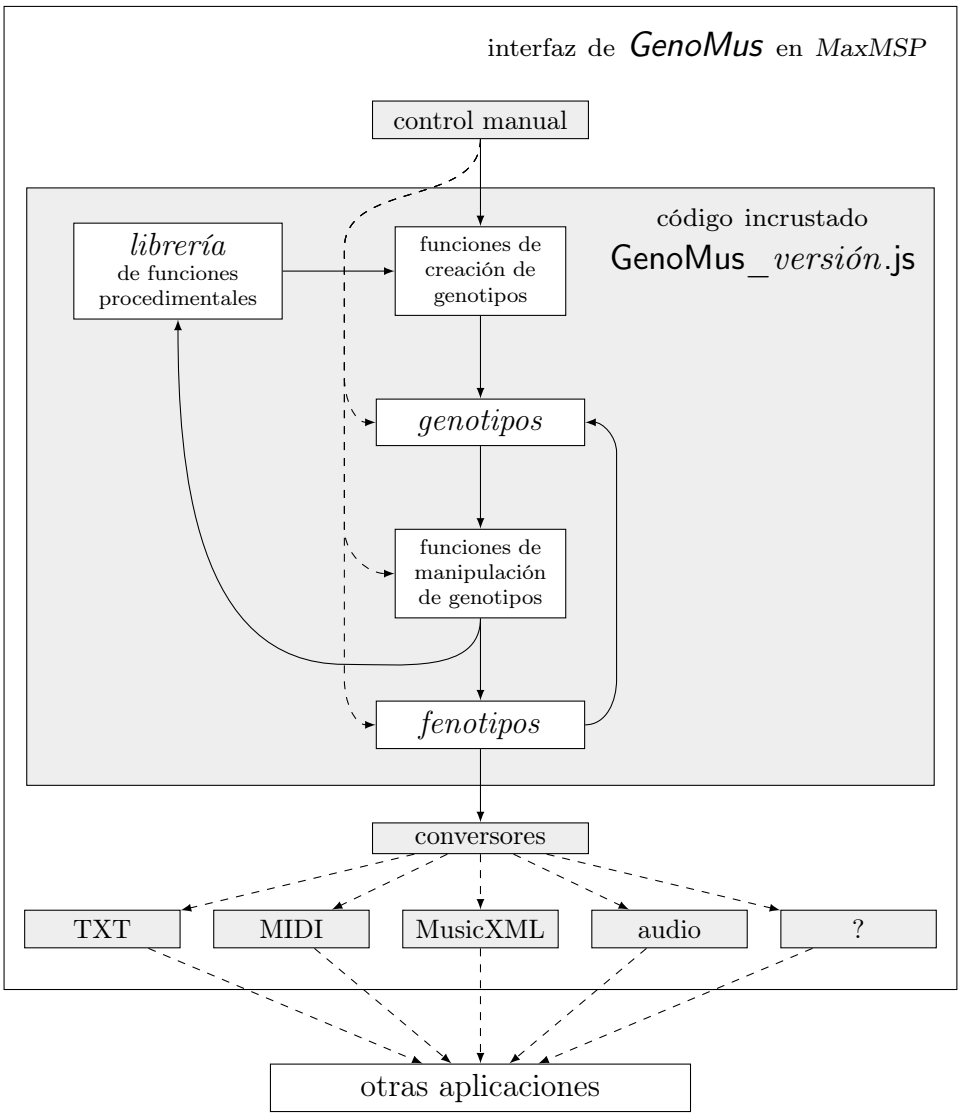


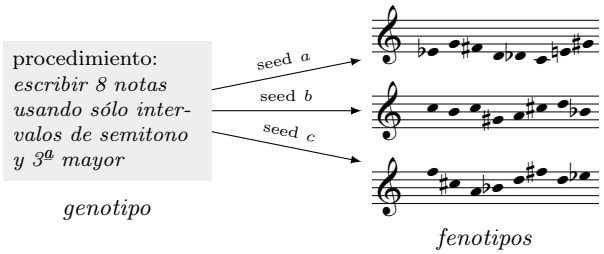
Figura 6.1: Esquema del funcionamiento de *GenoMus*.

6.2 Un prototipo de metalenguaje musical orientado a la metaprogramación

6.2.1 Genotipos y fenotipos musicales

En biología, un *genotipo* es el conjunto de los genes que existen en el núcleo celular de un organismo. El *fenotipo* es la realización visible del genotipo —es decir, el propio ser portador de los genes—, cuyos caracteres son el resultado de la interacción entre el genotipo y el medio. Esto implica que un mismo genotipo pueda dar lugar a fenotipos diferentes, como ocurre con los gemelos monocigóticos, quienes comparten un ADN idéntico pero evolucionan como individuos bien diferenciados.

El metalenguaje de *GenoMus* recurre a una analogía con estos conceptos. En el contexto de esta investigación, un *genotipo musical* es una expresión que codifica procedimientos musicales. Un *fenotipo musical* es el fragmento musical generado por un genotipo. Al igual que en biología, un genotipo musical puede dar lugar a muchos fenotipos diferentes, en función del medio. Los genotipos que incluyen funciones que implican algún grado de aleatoriedad pueden así generar fenotipos diferentes. Con el fin de hacer experimentos verificables, en *GenoMus* la aleatoriedad está controlada por un valor inicial *seed*[44] que desencadena el resto de valores aleatorios de manera repetible.

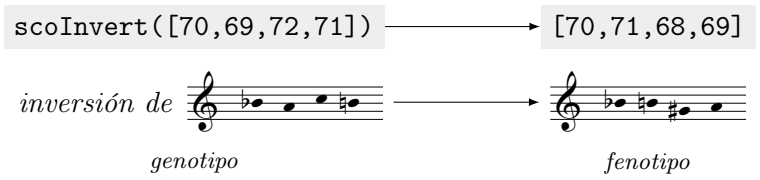


Por ejemplo, podemos definir la función `sqr` como aquella que devuelve el cuadrado de su entrada, es decir: $sqr(x) = x^2$. Por tanto, $sqr(4) = 16$. En notación de *GenoMus*:

La expresión `sqr(4)` se evalúa como código estándar de *JavaScript*, ya que los genotipos se formulan siguiendo la sintaxis de este lenguaje de programación.

[44] En programación, la variable *seed* (semilla) suele referirse al argumento de un generador aleatorio que determina la secuencia de números que este generador produce. Utilizando un mismo valor *seed* se puede repetir un proceso de computación con variables aleatorias en todos sus detalles. La sección 6.3.4 expone la estrategia de *GenoMus* para la gestión de la repetibilidad de los experimentos.

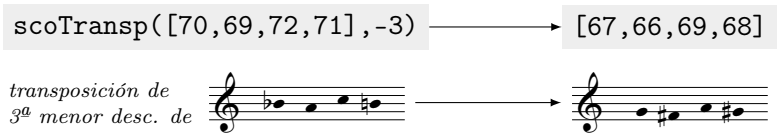
El argumento que una función toma puede ser una secuencia de números —una matriz unidimensional— y su salida puede ser también una secuencia. Podemos así definir `scoInvert` como la función que toma una secuencia de números y devuelve su inversión, tomando el primer valor como eje de simetría. Las secuencias son en realidad *arrays* de *JavaScript*, por lo que sus elementos se escriben entre corchetes y separados por comas. En la transcripción en partitura se han traducido los valores como alturas MIDI.[45]



Los genotipos se expresan en forma de *función*. Cada función realiza un proceso bien definido. La estructura sintáctica de una función es la usual en matemáticas:

$$nombreFuncion(argumento_1, argumento_2, ..., argumento_n)$$

Las funciones pueden requerir más de un argumento. La función `scoTransp` realiza la transposición de una secuencia. Los argumentos que requiere son dos y se escriben separados por una coma: la secuencia de entrada (un *array*), y un valor (una constante) que determina el intervalo de transposición.



6.2.2 Sintaxis en árbol de los genotipos

La potencia de la programación funcional radica en la posibilidad de que una función llame a otra función como argumento. Esto permite hacer composición de funciones de complejidad arbitraria.

La función `scoRetrog` devuelve la retrogradación de la secuencia de entrada —esto es, el *array* de entrada en orden inverso—. Ahora es posible, combinando esta función con las anteriores, hacer una transposición de la inversión retrogradada de una secuencia dada.

[45] En notación MIDI, el DO central recibe el valor 60, y cada entero ascendente (descendente) sube (baja) un semitono. Así, a la nota LA del diapasón le corresponde el número 69.

`scoTransp(scoInvert(scoRetrog([69,65,70,61])),14)` → `[75,66,71,67]`

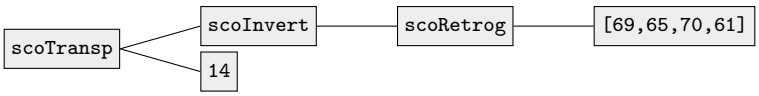
transposición de 9ª mayor asc. de la inversión de la retrogradación de

El mismo proceso paso a paso con la notación matemática para la composición de funciones:

$$(t \circ i \circ r)(K, 14) = t[i[r(K)], 14] =$$
$$= t[i[r(69\ 65\ 70\ 61)], 14] = t[i(61\ 70\ 65\ 69), 14] =$$
$$= t[(61\ 52\ 57\ 53), 14] = (75\ 66\ 71\ 67)$$

$$\begin{aligned} t(x,y) &= \texttt{scoTransp}([x],y) \\ i(x) &= \texttt{scoInvert}(x) \\ r(x) &= \texttt{scoRetrog}(x) \\ K &= (69\ 65\ 70\ 61) \end{aligned}$$

Otro modo de visualizar con mayor claridad la estructura de estas funciones compuestas es el diagrama de árbol, tan usual en todo tipo de gramáticas generativas como la propia de *GenoMus*. Este es el genotipo anterior representado como árbol de funciones:



Es evidente que en cuanto los genotipos empiezan a ramificarse el código se hace ilegible. Es por esto que para su manipulación manual los genotipos se formatean de manera que esta estructura en árbol quede reflejada en la expresión de texto. Así pueden visualizarse con rapidez las dependencias entre funciones y argumentos. Este es el código del mismo genotipo anterior, tras aplicarle el formateo automático:

```
scoTransp(
    scoInvert(
        scoRetrog(
            [69,65,70,61])),
    14)
```

Con el tabulado es sencillo identificar los elementos alineados verticalmente como argumentos de la función inmediatamente superior a la izquierda. En el ejemplo anterior, ahora se visualiza rápidamente que `scoInvert` y `14` son los argumentos de `scoTransp`.

En definitiva, un genotipo no es más que una función de funciones, y el fenotipo es el resultado de su evaluación.

6.2.3 Tipos de entrada y salida de las funciones

No todas las funciones son combinables entre sí. Cada función requiere como entrada (*input*) uno o varios argumentos con una tipología bien determinada. La función `sqr` requiere un número no negativo, mientras que `scoTransp` necesita una secuencia y un número. Igualmente, cada función da como producto un tipo concreto de salida (*output*): `scoInvert` devuelve una secuencia, `sqr` un sólo valor.

Dado que *GenoMus* usa como recurso esencial la recombinación aleatoria de funciones, se imponen simplificar y armonizar al máximo los tipos de entrada y salida que aceptan y devuelven las funciones de la librería base. Encontrar el equilibrio correcto entre flexibilidad y especificidad de los tipos empleados es sin duda una de las decisiones que más condiciona el desarrollo posterior del sistema.

En la versión actual de *GenoMus*, estos son los tipos de datos que una función puede pedir o producir:

- Un *número* (`value`); puede especificarse que sea entero (`integ`) o decimal (`float`).
- Una *secuencia* numérica (`score`), que es un *array* de *JavaScript*.
- Una *expresión genotípica* entre comillas (`recur`), típicamente para manipulaciones recursivas internas de una función.
- Otros *tipos específicos* requeridos por ciertas funciones, como por ejemplo *pitch class set* (`pcset`), que suelen ser matrices de dimensiones y características bien definidas.

Las etiquetas de cinco caracteres entre paréntesis se usan para la identificación automática de los tipos de entradas y salidas de cada función. Por ejemplo, en la declaración de la función `scoTransp` los nombres de los argumentos determinan las funciones a las que pueden llamar:

```
function scoTransp(score, value) {
```

A su vez, en este caso sería necesario saber qué funciones producen *outputs* de tipo `score` y `value` para hacer conexiones correctas en el genotipo. Esto se consigue recurriendo a un listado automático que recoge esta información, creado mediante la función auxiliar `addFunctionsTypeItem`:

```
addFunctionsTypeItem("scoInvert", "score");
```

Mediante esta línea de código se ha informado previamente de que la función `scoInvert` da una salida de tipo `score`, por lo que `scoTransp` puede llamarla como primer argumento. Este mecanismo se estudia con más profundidad en la sección 6.3.6.

6.2.4 Independencia de los procedimientos respecto de la escala

Un hecho notable de las funciones de *GenoMus* es su independencia respecto de la escala —ya sea temporal, de alturas, o de cualquier otro parámetro vertical u horizontal—. Procedimientos como la repetición o la realización de una curva melódica afectan a células, motivos, frases, o grandes bloques temporales. Pueden aplicarse incluso en la escala microtemporal, en el caso de la electroacústica. Este planteamiento busca evitar cualquier estratificación jerárquica cerrada basada en estructuras sintácticas típicas, y propiciar el crecimiento orgánico de los niveles gramaticales percibidos. No se adaptan los procedimientos a esquemas preestablecidos de frase, período o sección, sino que se persigue que esas estructuras emerjan con naturalidad de los propios procesos constructivos. Esta idea entronca con uno de los principios fundamentales de la tradición schenkeriana:

> [...]*varios aspectos de la estructura a gran escala aparecen con frecuencia reflejados en la pequeña escala, y* [...] *esas aparentemente pequeñas configuraciones pueden resultar más significativas de lo que al principio parecen ser.* [20]

No obstante, en el ámbito de la música experimental retomamos esta idea desde una perspectiva mucho más abierta y dinámica que en su contexto original, muy ligado a la tonalidad y a la retórica musical clásica. En las músicas contemporáneas, y muy especialmente en las que se califican como *generativas*, la conexión entre las diferentes escalas no sólo se encuentra en el producto musical acabado, sino sobre todo en los propios procesos creativos.

6.3 Mecanismos de metaprogramación

La sintaxis del metalenguaje expuesta arriba no aporta en sí misma ninguna novedad a los numerosos lenguajes de programación funcional o simbólica que existen. Su única característica relevante para esta investigación es que se ha construido buscando la simplificación máxima de una gramática que al mismo tiempo sea capaz de representar cualquier fragmento musical. Esta simplicidad es un requisito esencial para articular los mecanismos de metaprogramación que se exponen en esta sección, y que constituyen el núcleo de *GenoMus*.

6.3.1 Autómata básico de generación de genotipos

El metalenguaje de *GenoMus* puede definirse como una gramática recursiva. Esta gramática está integrada por un conjunto cerrado de signos y una serie de reglas que los combinan. Esta sintaxis tan limitada permite diseñar

con facilidad un algoritmo que escriba expresiones *bien formadas*[46] en este metalenguaje. Los signos que forman cualquier expresión genotípica pueden agruparse en cuatro elementos:

- *Nombres de funciones* (que son elementos del conjunto de los nombres de funciones disponibles), a los que nos referiremos simplemente como f, dando a entender que es uno cualquiera de estos nombres.
- Signos de *paréntesis*, para enmarcar los argumentos de una función.
- Signo de *coma*, para separar argumentos.
- *Argumentos numéricos* de las funciones (que pueden ser un sólo número, o una secuencia de números separados por comas y acotada por corchetes). Por ahora nos referiremos genéricamente a estos argumentos numéricos como x_n, denotando así el argumento n de su función correspondiente.

El algoritmo de escritura automática de funciones puede contemplarse como un autómata finito no determinista[47] que va pasando por cuatro estados, y con funciones de transición que pueden representarse gráficamente como muestra la figura 6.2. En este autómata el símbolo f y el paréntesis izquierdo se unen como un único estado, ya que siempre aparecen unidos, y de este modo se simplifica el número de estados y sus transiciones.

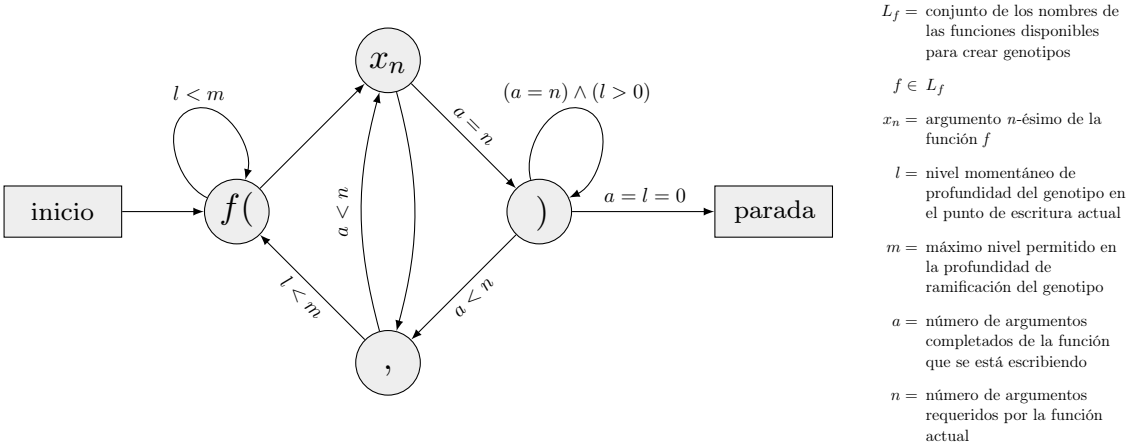


Figura 6.2: Autómata finito generador de genotipos

[46] En lógica matemática y en gramática generativa, el concepto de expresión *bien formada* (*well-formed*) denota una cadena de símbolos construida correctamente según las reglas de inferencia del sistema. Que una cadena esté bien formada no implica que sea cierta, sino sólo que tiene sentido en ese lenguaje concreto.

[47] Una descripción formal precisa de este tipo de autómatas puede encontrarse en [3].

Las expresiones genotípicas siempre empiezan con el nombre de una función que es el tronco principal del que se irá ramificando toda la expresión. Para evitar un crecimiento y ramificación indefinida de los genotipos, se introduce la variable entera m, que limita la profundidad de las ramificaciones. Así, si $m = 3$, se permite usar un máximo de tres niveles de paréntesis, como en la siguiente expresión:

$$f(f(x_1, f(x_1, x_2)))$$

El estado de parada del autómata se alcanza cuando se han terminado de escribir todos los argumentos requeridos por todas las funciones llamadas. Dependiendo del anidamiento de la estructura —es decir, de cuantas funciones hayan llamado a su vez a otras funciones como argumentos—, este autómata puede producir expresiones de longitud muy variable. La figura 6.3 es una versión compactada del autómata anterior, más cercana a la implementación real de la función `createExpression` de *GenoMus*. En este caso se han prescindido de los nodos de inicio y parada, y los cuatro nodos principales de han agrupado en dos nodos dobles.

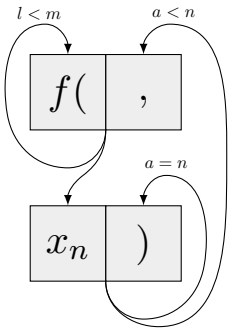


Figura 6.3: Equivalente mínimo del autómata finito generador de genotipos

La figura 6.4 muestra el funcionamiento de este autómata a través de un árbol de posibilidades de formación de expresiones bien formadas. Este ejemplo incluye todas los recorridos posibles del autómata usando sólo funciones con dos argumentos, y un máximo de tres capas de profundidad (tres niveles de paréntesis).

El análisis de todas las expresiones válidas obtenidas en función del número de estados que el autómata realiza antes de llegar al estado de parada —es decir, el número de elementos que se escriben en cada genotipo—, muestra la emergencia de una estructura autosimilar (figura 6.5).

6.3.2 La función `createExpression`

La escritura de los genotipos iniciales se realiza mediante una llamada a la función `createExpression`. Los genotipos que se producen se construyen ajustándose a estas variables:

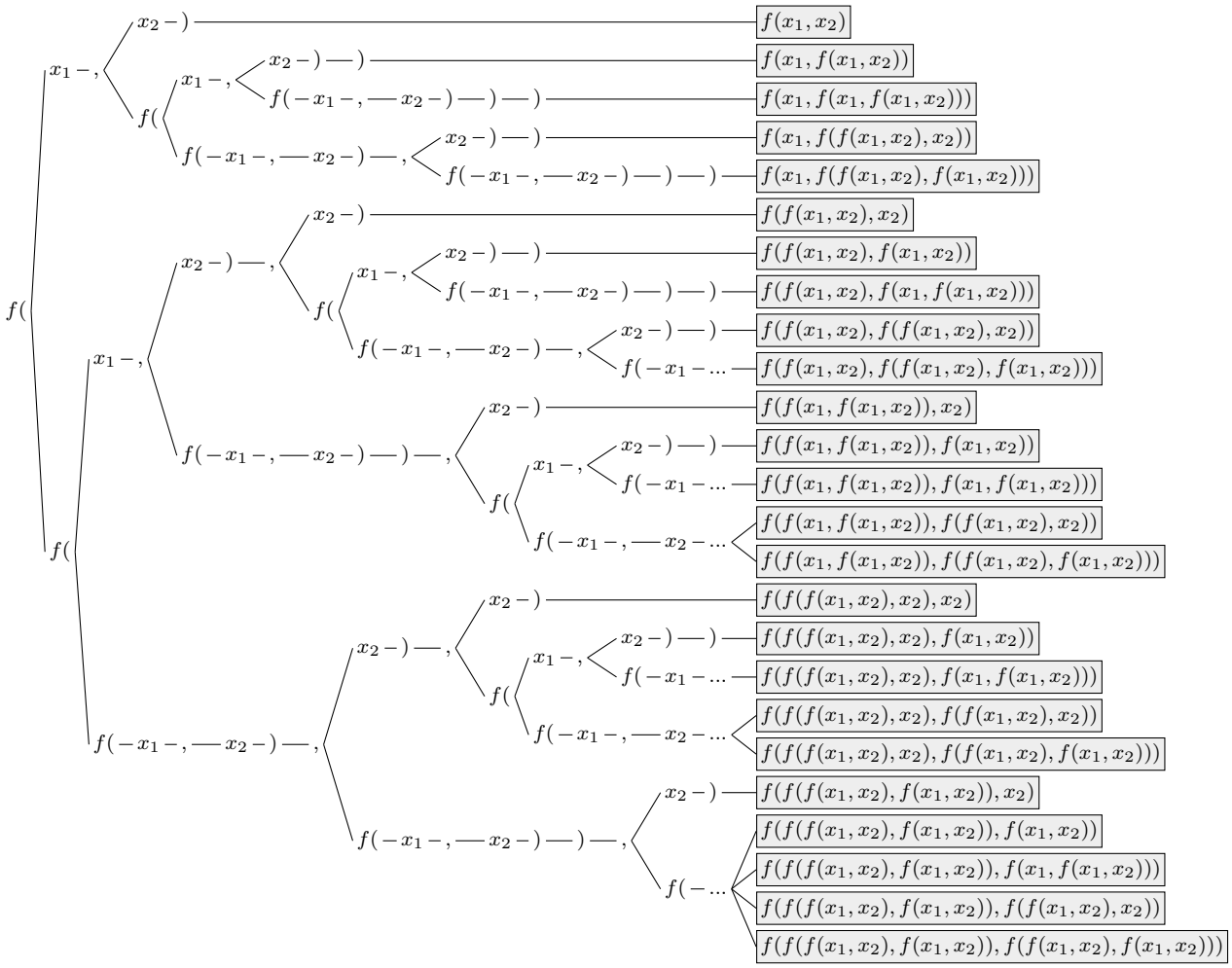


Figura 6.4: Árbol de posibilidades de expresiones bien formadas

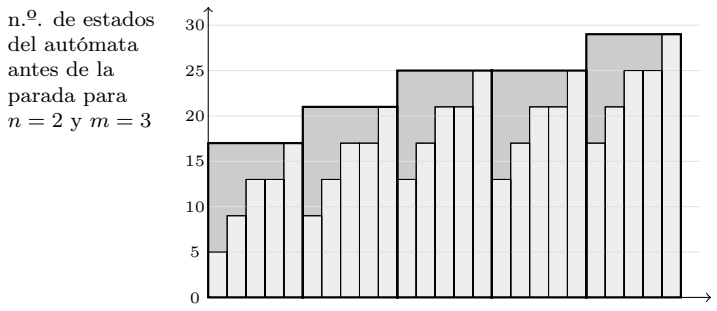


Figure 6.5: Autosimilaridad del árbol de expresiones bien formadas

- `maxLevels` determina el máximo nivel de paréntesis anidados permitido. Un número demasiado alto puede motivar procesos de búsqueda demasiado largos; uno muy bajo limita en exceso las posibilidades. En los experimentos documentados se ha utilizado típicamente un valor entre 5 y 7, y hasta 20 niveles como máximo.

- `maxNumberFuncions` limita el número de funciones que un genotipo puede incluir, en caso de que sea conveniente controlarlo. Si este valor es superado, el genotipo de desecha y se inicia la escritura de uno nuevo.

- `maxSearchIterarions` establece el máximo número de genotipos que se construyen en cada llamada a `createExpression`. En ocasiones los criterios de búsqueda pueden hacer difícil o imposible encontrar un genotipo que los satisfaga. Este limitador detiene el ciclo si no se encuentra una solución válida.

- `minScoreSize` y `maxScoreSize` establecen respectivamente el número mínimo y máximo de eventos que debe contener el fenotipo generado.

La figura 6.6 muestra el funcionamiento de la función `createExpression`. Una vez completada la escritura de un genotipo mediante el autómata generador de genotipos (fig. 6.3), la expresión se evalúa para generar un fenotipo. En la versión actual el fenotipo sólo debe pasar las condiciones de tamaño requerido, aunque está previsto que puedan añadirse muchos más condicionantes de interés propiamente musical. Si el número de eventos del fenotipo está dentro del rango correcto, la función da como salida válida el par genotipo–fenotipo y termina el proceso de búsqueda. Si se agota el número de intentos determinado por `maxSearchIterarions`, el ciclo se detiene y un mensaje informa de la conclusión infructuosa de la búsqueda. La figura 6.7 muestra un par genotipo–fenotipo obtenido tras 27 iteraciones con la función `createExpression`, usando estos condicionantes:

- `maxLevels` = 6
- `maxNumberFuncions` = 200
- `maxSearchIterarions` = 100
- `minScoreSize` = 30
- `maxScoreSize` = 200

En la figura 6.8, la representación en árbol del genotipo obtenido muestra con claridad que efectivamente ninguna rama ha traspasado el sexto nivel de ramificación.[48]

[48] El nivel 0, correspondiente a la función `scoRender` en el tronco, no cuenta como nivel de ramificación.

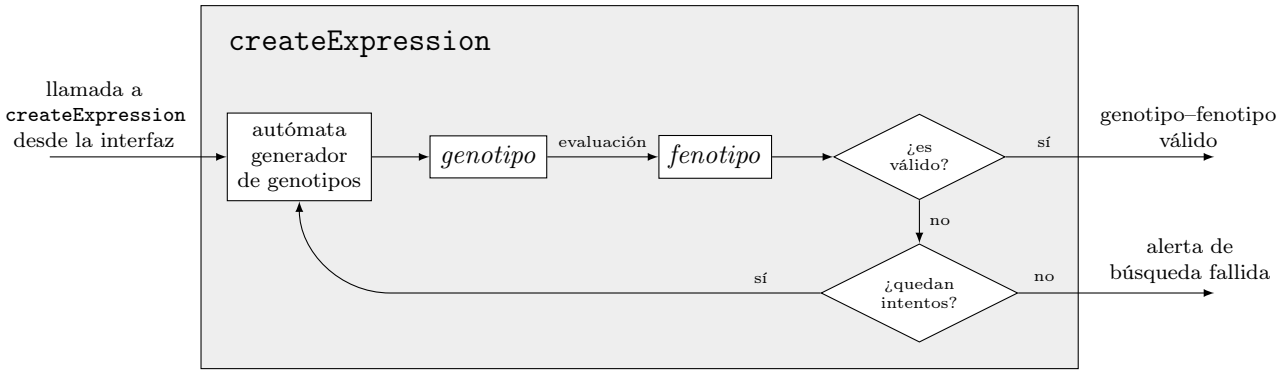


Figura 6.6: Funcionamiento de `createExpression`

6.3.3 Legibilidad de los genotipos

Un genotipo con tantas ramificaciones no permite su lectura ni su manipulación manual. La función `expandExpre` toma como argumento la propia expresión genotípica y la devuelve tras insertar saltos de línea y tabulaciones de manera que se alineen los argumentos que corresponden al mismo nivel. La figura 6.9 reproduce la expresión del genotipo anterior tras ser procesada por `expandExpre`. Su función inversa es `compressExpre`, que compacta una función expandida.

genotipo

```
scoRender(scoRand(rInt(mod(pyth(floPrim(), intPrim())
    , ratio(intPrim(), log(intPrim(), floPrim()))),
    intPrim()), rInt(absDif(rInt(prod(floPrim(),
    intPrim()), randInt(floPrim(), intPrim())), sum(
    intPrim(), intPrim())), sqr(randFlo(randInt(
    floPrim(), floPrim()), floPrim()))), randInt(
    floPrim(), floPrim())))
```

fenotipo

```
[78,72,89,81,92,89,69,84,82,91,71,69,78,73,87,88,86,
    93,78,90,73,80,82,88,89,78,81,90,83,89,86,87,86,
    74,90,87,76,78,88,89,78,76,85,87,76,71,84,95,71,
    86,87,69,92,87,91,72,94,78,88,69,85,76,74,71,69,
    94,78,84,87]
```

Figura 6.7: Ejemplo de par genotipo–fenotipo

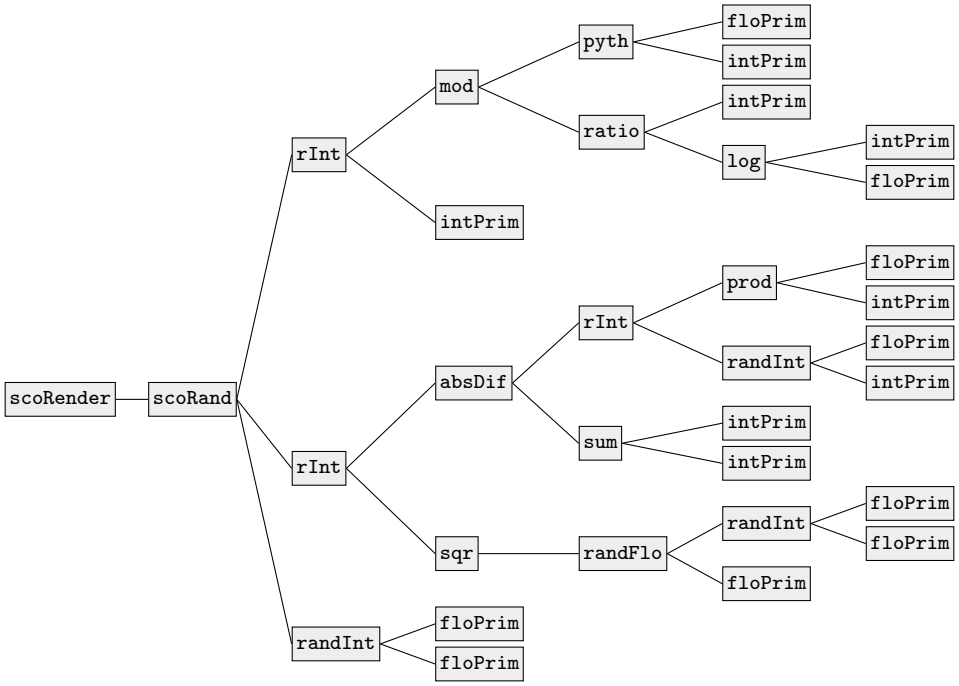


Figura 6.8: Representación en árbol de un genotipo con nivel 6 de ramificación

6.3.4 El rol de la aleatoriedad

Los experimentos realizados con *GenoMus* se han diseñado para ser repetibles; es por tanto un sistema determinista. Esto implica que para todos los procesos que implican azar hay que proveer un mecanismo que pueda reproducir las mismas elecciones aleatorias siempre que sea preciso.

Al contrario que muchos otros lenguajes de programación, *JavaScript* no dispone de un generador aleatorio nativo que acepte un valor *seed*, por lo que se ha diseñado un generador aleatorio propio de *GenoMus*, para poder reproducir un mismo fenotipo en el caso de genotipos con salidas variables. Esto es imprescindible en el trabajo de composición, ya que en muchos casos un mismo genotipo puede producir fenotipos de interés musical enormemente dispar. Poder recuperar un resultado concreto de interés sólo es posible si se cuenta con una aleatoriedad determinista.

6.3.5 Implementación de `randomSeed`

El generador de números aleatorios de *GenoMus* es la función `randomSeed`, que para cada llamada devuelve un número comprendido en el intervalo $[0, 1)$:

```
scoRender(
    scoRand(
        rInt(
            mod(
                pyth(
                    floPrim(),
                    intPrim()),
                ratio(
                    intPrim(),
                    log(
                        intPrim(),
                        floPrim()))),
            intPrim()),
        rInt(
            absDif(
                rInt(
                    prod(
                        floPrim(),
                        intPrim()),
                    randInt(
                        floPrim(),
                        intPrim())),
                sum(
                    intPrim(),
                    intPrim())),
            sqr(
                randFlo(
                    randInt(
                        floPrim(),
                        floPrim()),
                    floPrim()))),
        randInt(
            floPrim(),
            floPrim())))
```

Figura 6.9: Expresión genotípica formateada por `expandExpre`

```
function randomSeed() {
    seed = (seed*dispersionAlea*(1-seed)) % 1;
    return seed;
}
```

La primera vez que se aplica esta función se utiliza el valor fijo predeterminado para la variable global `seed`. Esta variable se actualiza tras cada llamada a `randomSeed`, para funcionar a su vez como *seed* de la próxima iteración. Cuando se reinicializa la función `createFunction` —por ejemplo, para la búsqueda de un nuevo genotipo después de un intento fallido— la variable `seed` se restaura a su valor inicial, que había sido almacenado en la constante `initSeed`.

La generación de valores aleatorios se hace usando una ecuación recursiva clásica, la *ecuación logística*:

$$x_{n+1} = r x_n (1 - x_n), \qquad r > 0 \tag{6.1}$$

Partiendo de un valor inicial x_0 se calcula un nuevo valor x_1, que a su vez se usa para calcular x_2, y así sucesivamente. Sin entrar en detalles[49], puede decirse que con esta ecuación es posible generar en función de la constante r resultados que pueden pasar gradualmente desde la estabilidad al caos, pasando por fases intermedias de autoorganización cada vez más compleja. Como muestra la figura 6.10, cuando r está muy próxima a 4 el comportamiento de la ecuación se parece mucho al de un generador aleatorio al uso, con una distribución casi uniforme de los valores obtenidos sobre el intervalo $[0, 1)$.

Cuando $r > 4$ la ecuación logística deja de ser estable y los términos de la serie generada tienden rápidamente a infinito. La función `randomSeed` emplea esta ecuación con una modificación que permite usar cualquier valor para r. Se elimina la parte entera de los resultados mediante la operación módulo (`%1`),[50] con lo que los valores que se obtienen siempre quedan en el intervalo $[0, 1)$.

$$seed_{n+1} = \Big(dispersionAlea \cdot seed_n \cdot (1 - seed_n)\Big) \bmod 1 \tag{6.2}$$

La variable `dispersionAlea` se corresponde con r y determina la distribución de probabilidad de `randomSeed`. Cuando `dispersionAlea` está en el

[49] Existen numerosos trabajos de diferente extensión y profundidad sobre esta interesante ecuación, que es uno de los primeros iteradores caóticos analizados. Un estudio exhaustivo puede encontrarse en la monografía de Peitgen [47].

[50] La operación *módulo* (mod) calcula el resto de la división de dos números. Así, el módulo 5 de 8 es 3, lo que se expresa con la notación: $8 \bmod 5 = 3$. Aunque la operación módulo se usa principalmente con números enteros, en programación el uso del módulo 1 es una manera rápida de extraer la parte decimal de un número real positivo; por ej.: $8.645 \bmod 1 = 0.645$.

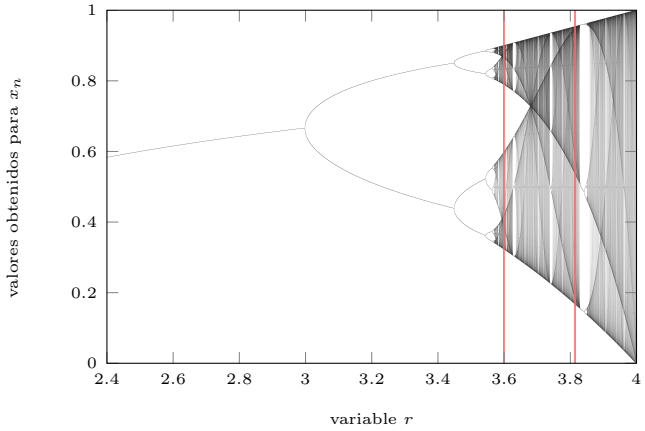


Figura 6.10: Mapa de bifurcaciones de la ecuación logística

intervalo $(0, 4]$ el comportamiento de `randomSeed` es idéntico a la de la ecuación logística, como puede cotejarse con los histogramas de la figura 6.11. Estos ejemplos y los siguientes muestran los resultados de generar enteros entre 0 y 99 usando `randomSeed` (simplemente multiplicando por 100 el número generado y extrayendo la parte entera). Puede comprobarse que cuando `dispersionAlea` ≤ 4 la frecuencia de aparición de cada número se corresponde con la prevista por el mapa de bifurcaciones de la figura 6.10, en donde se han marcado con líneas verticales los valores que se han usado para `dispersionAlea`. Si `dispersionAlea` ≥ 4 pueden obtenerse todos los enteros del intervalo $[0, 99]$, aunque con una probabilidad mucho más elevada para los valores más próximos a los extremos del intervalo.

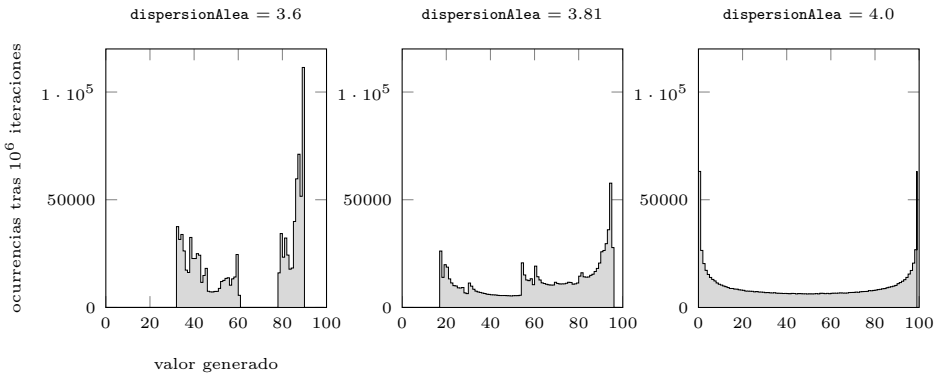


Figura 6.11: Histogramas de `randomSeed` con `dispersionAlea` ≤ 4

Para valores de `dispersionAlea` apenas superiores a 4 se siguen obteniendo todos los enteros del intervalo, pero la distribución de posibilidades

comienza a desestabilizarse rápidamente —los primeros experimentos apuntan a un comportamiento análogo al mostrado en la figura 6.10, pero una mejor compresión requeriría de un estudio más extenso—. No obstante, esto puede ser un elemento susceptible de ser usado artísticamente. La riqueza de *espectros* combinatorios que pueden conseguirse con muy pequeñas alteraciones de `dispersionAlea` se presta a su aplicación a diferentes parámetros musicales, tanto horizontales (tiempo) como verticales (altura, timbre...), y por supuesto a la mecánica generativa de *GenoMus*.

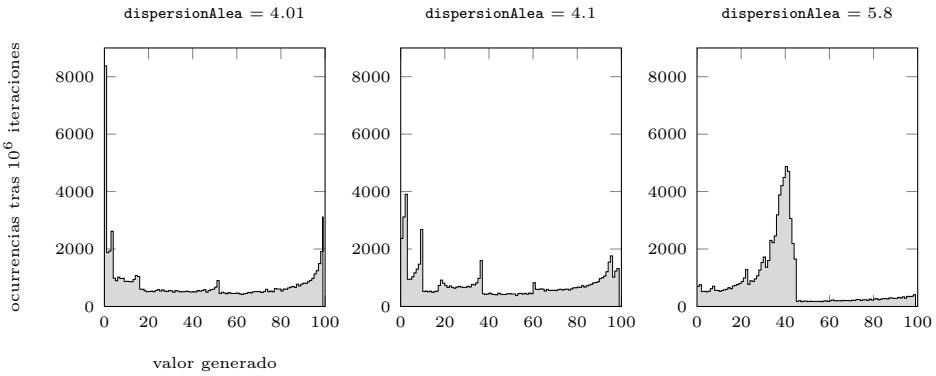


Figura 6.12: Histogramas de `randomSeed` con `dispersionAlea` ligeramente por encima de 4

Para conseguir una distribución uniforme estándar de `randomSeed` hay que usar valores elevados de `dispersionAlea`. La figura 6.13 muestra las ocurrencias de cada valor del intervalo $[0, 99]$ para `dispersionAlea` $= 10^5$. Aunque, como es habitual, la distribución va siendo más uniforme a medida que aumentan las iteraciones, es notable el hecho de que el espectro de frecuencias tiende a mantenerse. En este ejemplo, con veinte mil iteraciones hay un pico de frecuencia para el valor 21. Con medio millón de iteraciones ese pico aún es apreciable. Las observaciones realizadas sugieren que a cada valor de `dispersionAlea` le corresponde un perfil de probabilidad estable.

6.3.6 Reconocimiento automático de los argumentos de una función

En la sección 6.2.3 se introdujo la estructura básica de una función, y los tipos de entrada/salida que puede requerir. Los tipos de datos que actualmente se manejan de forma automatizada son:

- *Valores numéricos* (constantes):
  - `integ` (números enteros). Ej.: `4`
  - `float` (números decimales). Ej.: `20.347687`

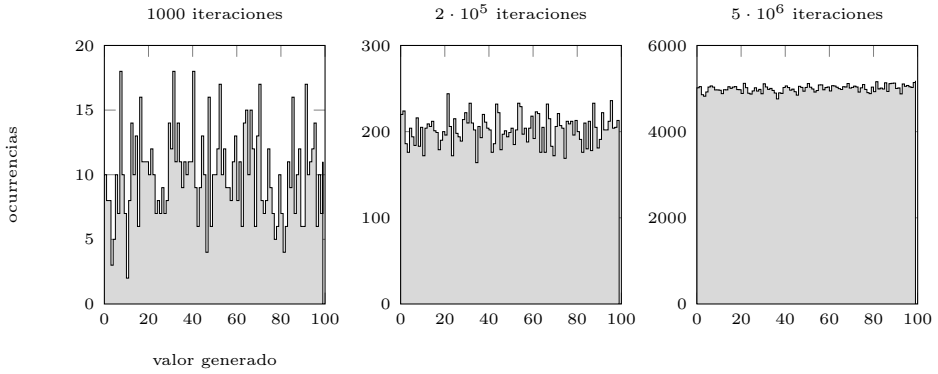


Figura 6.13: Histogramas de `randomSeed` con `dispersionAlea` $= 10^5$

  - `value` (número entero o decimal)
- *Matrices unidimensionales* (*arrays*):
  - `score` (secuencia de eventos musicales). Ej.: `[72,74,77,76]`
  - `pcset` (*pitch class set*, con formato predefinido). Ej.: `[0,2,3,6,9]`
  - `insco` (referencia interna al propio genotipo). Ej.: `[0,0,1,0,2,1]`
- *Expresiones genotípicas*
  - `recur` (estructura recursiva interna). Ej.: `‘‘randFlo(intrv(s,4), pyth(intrv(s,1),abs(tan(intrv(s,2)))))’’`[51]

El significado y formato de cada tipo se detallan más adelante. Por ahora lo importante es advertir que cada tipo de dato tiene una etiqueta de cinco letras que es un identificador que dirige el proceso de síntesis automática de expresiones genotípicas.

Una vez seleccionada una función para ser incluida en el genotipo, entra en juego la función auxiliar `getArgType`, que reconoce el tipo de dato que una función espera para cada argumento, según esta especificación:

`getArgType(`*nombre de función*`,` *ordinal del argumento*`)`

Supongamos que se llama a la función `scoExcerpt`, cuya implementación empieza así:

```
function scoExcerpt(score, value_1, value_2) {
```

[51] Es importante notar aquí el uso de las comillas, que son necesarias porque esta expresión genotípica *no es* parte del genotipo, sino el argumento de una función. La función que usa este argumento ejecutará entonces una evaluación de este genotipo como un proceso interno a la propia función. En la sección 6.5.2 se dan más detalles sobre la problemática y potencialidades de la incrustación de genotipos dentro de genotipos de nivel superior.

Cuando el autómata de generación de genotipos llega al punto en que hay que escribir los argumentos para esta función, obtiene la información sobre el tipo de dato esperado mediante estas expresiones:

```
getArgType(scoExcerpt,0) ⟶ score
getArgType(scoExcerpt,1) ⟶ value
getArgType(scoExcerpt,2) ⟶ value
```

Con esta información ahora es posible dirigirse con precisión al subconjunto de funciones que sí pueden satisfacer el tipo de dato requerido. El proceso se repite a su vez con cada llamada a estas nuevas funciones. El proceso no puede prolongarse de manera indefinida y debe terminar con llamadas a *funciones primitivas*, que son funciones que no requieren ningún argumento y que se sitúan como *hojas* en los extremos de cada rama del árbol genotípico.

6.4 La librería de funciones

Las expresiones genotípicas se forman combinando funciones disponibles en una librería. Antes de avanzar con las técnicas de generación y transformación de genotipos es conveniente conocer la tipología de las funciones que un genotipo puede conectar. Esta sección aborda los conceptos básicos que estructuran la librería de funciones, pero no hace un repaso de todas las funciones que la integran. El código completo de la librería con indicaciones sobre las tareas que realiza cada función puede consultarse en el anexo A.2.

La presente versión de *GenoMus* usa una librería muy limitada de funciones básicas. El propósito de esta fase de desarrollo es visualizar el funcionamiento del sistema en su conjunto, por ello sólo se han implementado un conjunto de procedimientos básicos que representen las distintas combinaciones de argumentos requeridos y tipos de salida que pueden configurar una función.

GenoMus está orientado a la metaprogramación, es decir, a que sea el propio sistema el que aprenda de sí mismo y nutra de manera automática su librería a partir de combinaciones de un número relativamente pequeño de funciones básicas. El objetivo primario de la librería no es ser un compendio enciclopédico de *sistemas expertos* musicales, sino disponer una amplia panoplia de funciones que hagan tareas básicas y precisas diseñadas para sacar partido del poder de la recombinación.

6.4.1 Estructura de la librería

La librería está alojada en el archivo genomus_library_*versión*.js. Este archivo está separado del código principal, y en él están todas las funciones que pueden ser llamadas por el generador de genotipos. También incluye algunas

funciones auxiliares más, que cumplen funciones logísticas en la gestión de las expresiones y que normalmente no son llamadas por un genotipo. Las funciones están agrupadas en listas según el tipo de salida que producen, lo que permite conocer cuáles son las opciones disponibles para satisfacer el tipo de argumento requerido por una función. Estas listas se crean en forma de *arrays* de nombres de funciones:

```
var integ_functions = ["rInt", "randInt", "intPrim"];
var integ_primitive = ["intPrim"];
var float_functions = ["rFlo", "floPrim", "sqr", "sum
    ", "dif"];
var float_primitive = ["floPrim"];
var value_functions = [];
value_functions = value_functions.concat(
    integ_functions, float_functions);
var value_primitive = [];
value_primitive = value_primitive.concat(
    integ_primitive, float_primitive);
```

En la primera línea se declara la variable `integ_functions`, que enumera las funciones disponibles que producen enteros, mientras que la tercera línea hace lo propio con las funciones que devuelven decimales. En la sexta línea se unen estas dos listas bajo el nombre de `value_functions`, formando así una lista de todas las funciones disponibles que dan un valor numérico único como salida. Los nombres de estas listas siguen también un formato predeterminado, diseñado para facilitar las llamadas automáticas a estos *arrays*: el nombre se forma con la misma etiqueta de cinco caracteres que identifica el tipo de salida de las funciones que lista (`integ`, `float`, `value`, `score`, etc.), al que se añade la extension `_functions`. La extensión `_primitive` se usa para acotar el grupo de funciones primitivas del tipo correspondiente.

6.4.2 Funciones primitivas y fenotipos múltiples

Antes de sintetizar un genotipo se fijan, entre otras variables, el máximo nivel de profundidad (o ramificación) que las expresiones genotípicas pueden alcanzar como máximo. Cuando se alcanza este nivel (visible por los niveles de paréntesis) es evidente que la función que se encuentra al final de la rama no puede llamar a otras funciones como argumentos. En estos extremos de la estructura se utilizan las *funciones primitivas*.

Una función primitiva genera su salida de manera autónoma, sin requerir ningún argumento previo. Por ejemplo, la función `intPrim` devuelve un entero, pero no requiere de ningún argumento para generarlo, por lo que no contiene nada entre paréntesis:

Es evidente que este entero debe estar dentro de un rango prefijado, y esto aconseja que los valores que manejen las funciones no primitivas estén hasta cierto punto normalizados. Esto es básico para crear estructuras musicalizables con parámetros que estén dentro de los rangos convenientes.

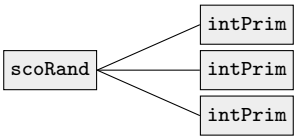
El empleo de las funciones primitivas puede ser facultativo (ya que están incluidas en el catálogo general de su tipo y pueden ser llamadas en cualquier momento) u obligatorio (cuando se alcanza el nivel máximo de ramificación permitido). En el genotipo de las figuras 6.8 y 6.9 puede observarse que en todos los extremos se encuentran funciones primitivas (reconocibles por el sufijo `Prim`).

Tal como se explicó en la sección 6.3.4, la variable `seed` permite la repetibilidad en la generación de los fenotipos. En el caso de genotipos con cierto grado de aleatoriedad, usando un mismo valor de `seed` siempre se obtiene el mismo resultado. Por tanto, para estos genotipos es posible obtener múltiples fenotipos cambiando el valor de `seed` antes de su evaluación, mediante la función `rewriteSco`. Esta función registra el valor `seed` correspondiente a cada nueva versión de un fenotipo, lo que permite seleccionar y recuperar los mejores fenotipos después de un proceso de búsqueda, sin tener que almacenarlos todos.

Veamos un ejemplo de fenotipos múltiples generados con `scoRand`, función que genera una secuencia aleatoria de enteros conforme a tres argumentos:

`scoRand(`*n.º de elementos*`,` *valor mínimo*`,` *valor máximo*`)`

El genotipo mínimo con esta función requiere tres funciones primitivas que den contenido a los argumentos requeridos.



Una vez formado este genotipo, podemos generar fenotipos diferentes con `rewriteSco`, cada uno de los cuales será reproducido de manera idéntica siempre que usemos su valor `seed` correspondiente. La figura 6.14 muestra tres fenotipos posibles, con `minScoreSize` = 5 y `maxScoreSize` = 15.

En ocasiones resulta conveniente fijar los argumentos de un genotipo sustituyendo las funciones primitivas —que implican algún grado de azar— por un valor numérico fijo. La función `renderAllPrimitives` identifica las funciones primitivas y las reescribe como variables numéricas. Además, cada vez que vuelve a llamarse actúa como `rewriteSco`, y ofrece una nueva versión del genotipo con variables numéricas diferentes. La función que realiza la operación inversa es `unrenderAllPrimitives`, que vuelve a convertir los valores numéricos fijos en funciones primitivas. La figura 6.15 visualiza el resultado de aplicar estas transformaciones a la expresión genotípica anterior.

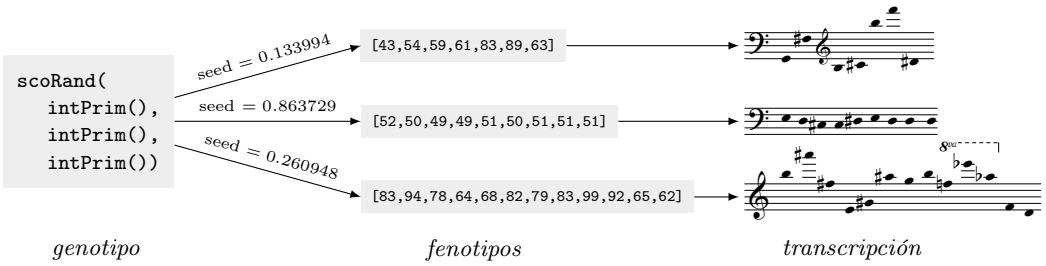


Figura 6.14: Fenotipos múltiples con varios valores para `seed`

Una vez hallada una versión que se desea conservar, se puede llamar a la función `updateRenderedCurrentExpression`, que fija los valores numéricos actuales en el genotipo para que éstos no se vean afectados por ulteriores transformaciones.

Este sistema es aún un primer apunte de cómo poder dirigir desde la interfaz el modelado de los pares genotipo–fenotipo. En las próximas versiones estas herramientas deberán ser mucho más flexibles y selectivas respecto a qué partes del genotipo se quieren variar, y de qué modo (nueva generación de valores, mutación de mayor o menor alcance de los valores actuales, etc.).

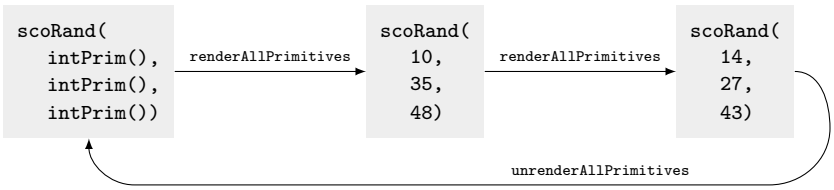


Figura 6.15: Conversiones entre funciones primitivas y constantes

6.4.3 Manipulación y crecimiento del genotipo

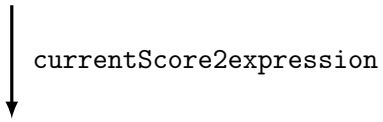
Partiendo de una expresión genotípica válida, es posible aplicar diferentes transformaciones. Estas técnicas de derivación son similares hasta cierto punto a la construcción de fragmentos musicales de entidad mayor a partir de motivos o células iniciales, propios de la composición tradicional.

Se han diseñado algunos mecanismos básicos para explorar posibles tácticas de manipulación de los genotipos, y así detectar los problemas prácticos que hay que solventar. En la presente versión de *GenoMus* existen tres modos básicos de derivación de genotipos:

– *Aplanamiento del genotipo.* En algunos casos, y muy especialmente en el diseño de pequeños motivos en los que la información del fenotipo

es mucho menor que la del genotipo que la ha producido, puede ser conveniente y más eficiente compactar un genotipo en una sola función. Esta manipulación la realiza la función `currentScore2expression`, y consiste en integrar el fenotipo dentro de la propia expresión genotípica como argumento de la función `scoRender`, que se limita a alojar la secuencia de eventos y darle salida. En la figura 6.16 puede comprobarse el resultado de aplicar este procedimiento sobre el genotipo anterior.[52]

```
scoTranspMatrix(
    scoEnumInt(
        randInt(
            randFlo(
                intPrim(),
                ratio(
                    intPrim(),
                    floPrim())),
            intPrim()),
        randInt(
            intPrim(),
            intPrim())),
    scoPitchClass(
        scoRandPrim(),
        scoPCGenPrim()))
```




```
scoRender
    ([158,157,156,194,193,192,297,296,295,321,320,319,
        381,380,379])
```

Figura 6.16: Aplanamiento de funciones con `currentScore2expression`

– *Transformación sin aumento del tamaño del fenotipo.* Se trata de modificaciones directas de un fenotipo —tales como la transposición, la mutación de alturas o la permutación de eventos— que no implican mayor complejidad de la estructura musical. La figura 6.17 muestra posibles transformaciones del mismo genotipo con la función `transformExpr`.

– *Inserción del genotipo en estructuras formales mayores*, con aumento de los niveles sintácticos del fenotipo. En este caso el fenotipo original

[52] Se podría comparar este aplanamiento o compresión de datos con la utilización que un improvisador o compositor hace de acordes, fórmulas, motivos cadenciales, etc., que no hay que recalcular o reescribir cada vez, sino que se recuperan desde la memoria.

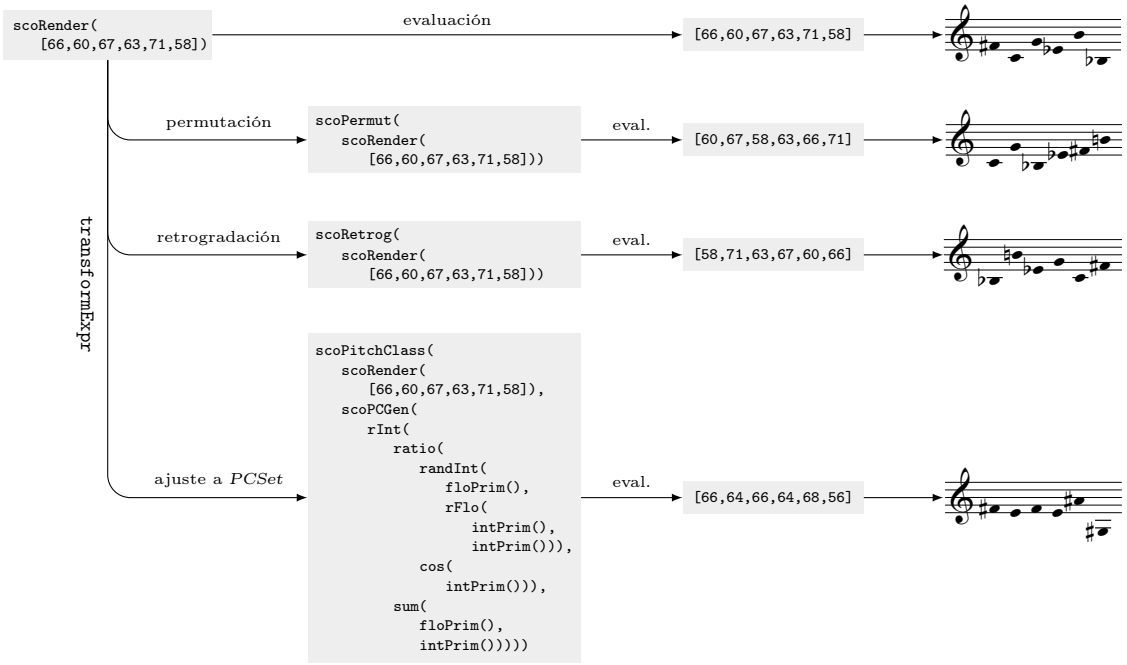


Figura 6.17: Transformación de un genotipo sin ampliación de la estructura

se ve incrustado como una rama de una estructura musical de más estratos. Algunos ejemplos de manipulaciones simples con este criterio son la mera repetición de una estructura, la concatenación de varios genotipos o la secuenciación de variaciones de un elemento. La función `growExpr` reelabora la expresión genotípica tal como ilustra la figura 6.18.

Las funciones que pueden usarse en estas manipulaciones están agrupadas en la librería como listas de procedimientos disponibles, del mismo modo que se agrupan funciones por tipologías de salida, o con cualquier otro criterio:

```
var grow_evolvFunctions = ["scoConcat", "scoRep", "
    scoTranspMatrix", "scoExpandIter", "scoPermutIter"
    , "scoExcerptMultiTransp"];
```

Es fácil crear nuevas funciones específicas de manipulación que se remitan a sus listas correspondientes de procedimientos.

Durante el proceso de modelado de un genotipo es muy útil contar con la función `updateRenderedCurrentExpression` para ir fijando las variables numéricas y así consolidar los fenotipos seleccionados antes de cada nueva manipulación.

Es importante recordar que el diseño de estas estrategias está orientada a una futura *autoevolución de los genotipos*. Estas funciones, que por el momento sólo se usan de manera manual y con resultados muy limitados, serán embebidas en rutinas más amplias que permitan ciclos de transformación de mucho más alcance. Al igual que en el trabajo del compositor humano, para conseguir esta automatización es imprescindible articular una retroalimentación entre la creación–manipulación de fragmentos musicales y la autocrítica continua de los mismos. La figura 6.19 da una idea del enfoque de este mecanismo autoevolutivo. Sólo un buen *equilibrio entre la amplitud de los espacios de búsqueda musical y la agudeza de las herramientas de análisis y evaluación* puede conducir hacia el objetivo de esta investigación. Este texto se centra en lo primero, por lo que la *autoobservación* del proceso está fuera de su alcance, y será sin duda la piedra angular de las siguientes fases de trabajo.

6.5 Recursión y autorreferencia

En casi cualquier proyecto de IA la recursión y la autorreferencia son dos conceptos clave, que reaparecen una y otra vez en múltiples contextos. Como muchos otros procesos biológicos y psicológicos, la composición y la improvisación musical depende de una percepción continua que guía los procesos creativos. La figura 6.19 es un clarísimo ejemplo de algoritmo recursivo. Por su parte, la autorreferencia está emparentada con la recursión, y puede decirse que aparece en virtualmente cualquier composición musical: la

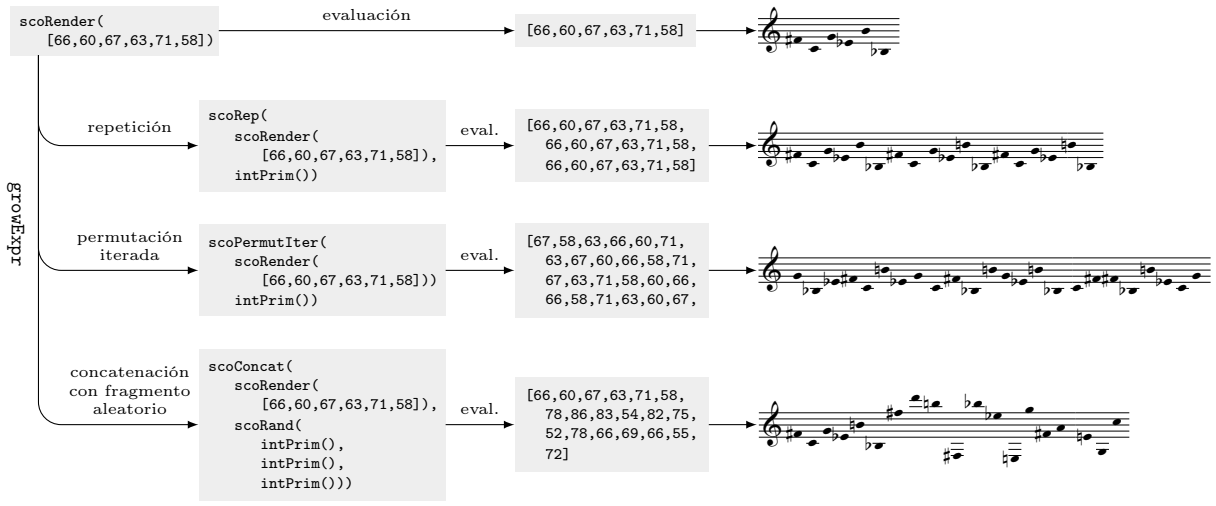


Figura 6.18: Crecimiento de un genotipo con ampliación de la estructura

reutilización de cualquier célula, motivo o tema, ya sea como repetición o variación, es un caso de autorreferencia.

6.5.1 Manejo de la autorreferencia en los genotipos

Si, como acabamos de afirmar, la autorreferencia es un recurso esencial de la forma musical, se hace imprescindible implementar un mecanismo que permita a un genotipo referirse a partes de sí mismo. Una vez que una subexpresión de un genotipo ha formulado un elemento musical cualquiera —un motivo melódico, una figura rítmica, una serie, un *pitch class set*, etc.— es evidente que cada vez que se quiera recurrir a éste no tiene sentido volver a codificar todo el proceso que lo produce. Es mucho más eficaz apuntar al punto de la expresión que contiene el elemento que se quiere reutilizar. En la práctica, esto se consigue haciendo una referencia a la dirección única que corresponde a cada elemento de una expresión genotípica.

La autorreferencia en un genotipo se logra con la función `scoInter`, cuyo único argumento es un *array* con la dirección de un nodo del propio genotipo. La expresión `scoInter[0,3,1]` remite al segundo argumento del cuarto argumento del primer argumento de la función troncal.[53] Con la representación en árbol de un genotipo es mucho más sencillo visualizar cómo funcionan estas coordenadas. La figura 6.20 muestra un genotipo con las coordenadas de cada nodo. Al evaluar el genotipo, los nodos en los que hay una función `scoInter` son sustituidos por el contenido de las ramas hacia las que éstos

[53] Los argumentos de una función de *JavaScript* se numeran comenzando por el cero.

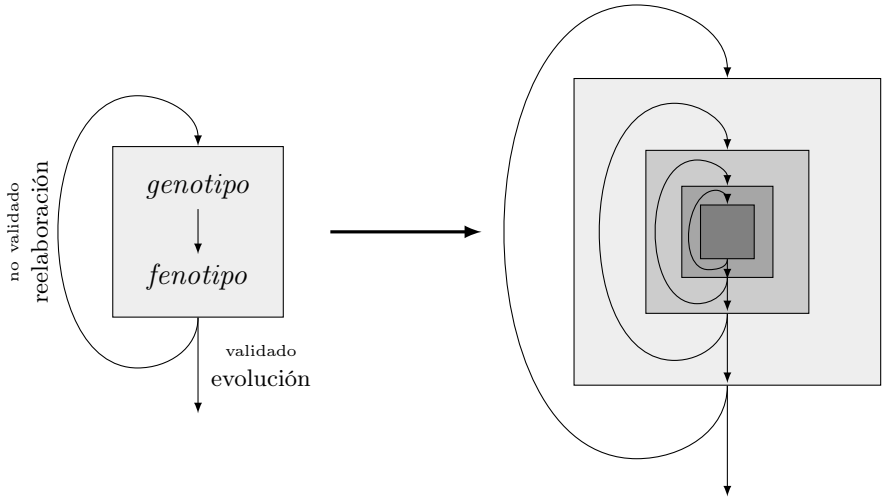


Figura 6.19: Evaluación y evolución de los pares genotipo–fenotipo

apuntan. Hay que tener un cuenta que un nodo no puede referenciar una rama que lo contenga a él mismo, porque esto resultaría en un *loop* infinito. Por ello, el nodo inicial o función troncal no puede ser referenciado por ningún nodo.

Veamos a continuación un ejemplo operativo simple. Sea el siguiente genotipo, cuyo fenotipo son tres enteros del intervalo $[1, 12]$:

```
scoRand(
   3,
   1,
   12)
```
→ `[7,4,8]`

En el siguiente paso se usa `scoConcat` —que concatena dos secuencias de eventos— para unir el genotipo anterior con una autorreferencia a esa misma expresión. Al evaluar el genotipo resultante, el nodo donde está `scoInter` da como salida una nueva evaluación de la subexpresión referenciada, lo que en la práctica equivale a sustituir este nodo por la propia subexpresión. Es notable que, al tratarse de una nueva evaluación de `scoRand`, los elementos aleatorios dan dos resultados diferentes, siendo los últimos tres enteros diferentes de los tres primeros.

```
scoConcat(
   scoRand(
      3,
      1,
      12)
   scoInter(
      [0,0]))
```
→ equivalencia →
```
scoConcat(
   scoRand(
      3,
      1,
      12)
   scoRand(
      3,
      1,
      12)
```
→ evaluación → `[7,4,8,2,11,9]`

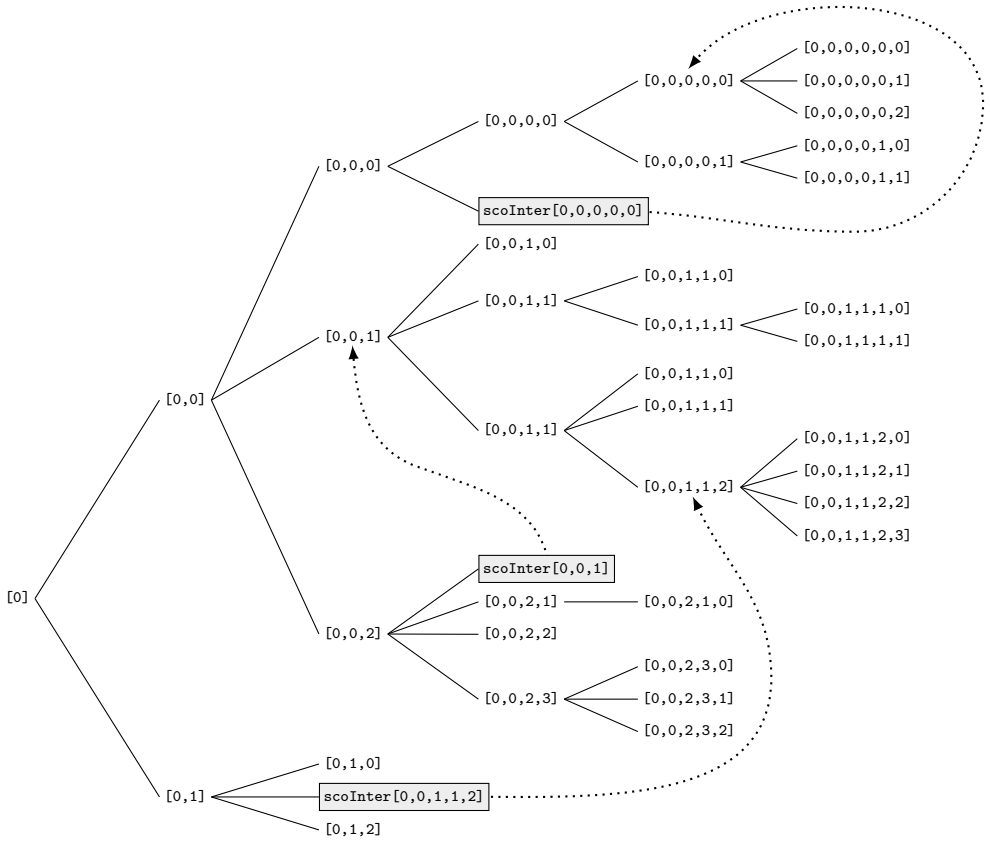
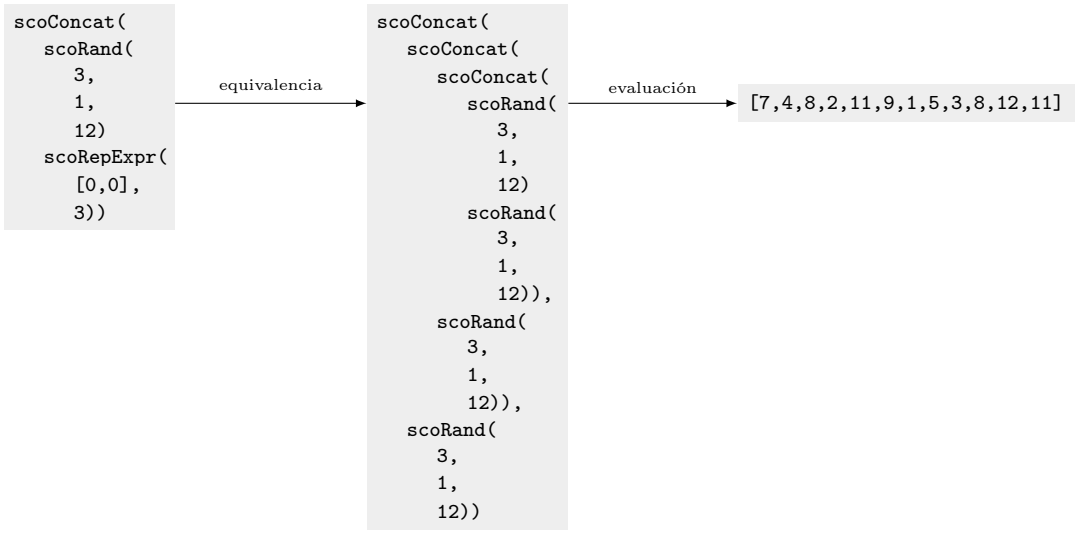


Figura 6.20: Autorreferencias en un genotipo

Partiendo de `scoInter` pueden construirse otros procedimientos basados en la autorreferencia. Uno de ellos es `scoRepExpr`, que repite la subexpresión referenciada un número de veces determinado por el segundo argumento, obteniendo así diferentes resultados en cada iteración:



No todos los nodos son aptos para ser referenciados, por lo que `scoInter` debe disponer de información acerca de qué ramas del genotipo representan secuencias de eventos que puedan ser reutilizadas. Esta información la facilita la función `mapSubScores`, que devuelve las direcciones de las subexpresiones que producen secuencias de eventos. La localización de estos puntos se ha facilitado mediante la convención de que todos los nombres de funciones con salida de tipo `score` comiencen con el prefijo `-sco`, con lo cual el problema se reduce a una mera búsqueda de este prefijo dentro de la expresión genotípica. La figura 6.21 muestra la representación en árbol de un genotipo y el resultado de aplicarle la función `mapSubScores`.

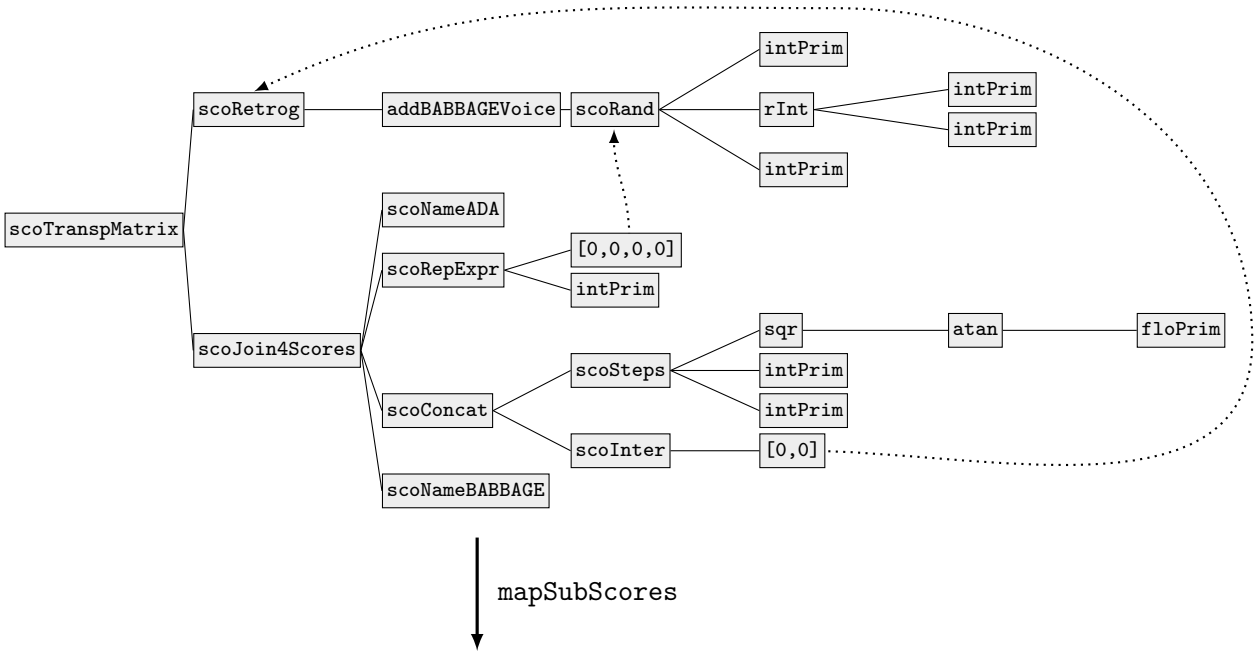
6.5.2 Funciones recursivas dentro de un fenotipo

Un ejemplo de proceso recursivo —muy conocido y usado por los músicos— es la serie de Fibonacci,[54] que puede obtenerse con la ecuación

$$x_n = x_{n-2} + x_{n-1}. \tag{6.3}$$

El valor de x_n se calcula usando los dos valores previos obtenidos por la propia ecuación en las iteraciones precedentes. Para empezar la serie se necesitan dos valores dados, llamadas *condiciones iniciales*. A partir de ahí la

[54] Otra ecuación recursiva es la ecuación logística que se ha abordado en la sección 6.3.5.




```
[0],[0,0],[0,0,0,0],[0,1],[0,1,0],[0,1,1],[0,1,2],
[0,1,2,0],[0,1,2,1],[0,1,3]
```

Figura 6.21: Extracción de subexpresiones válidas con `mapSubScores`

ecuación va calculando cada nuevo valor usando los valores calculados por ella misma.

El empleo de funciones recursivas es de sumo interés, dado la enorme variedad de comportamientos que pueden exhibir, y de hecho este tipo de manipulaciones constituye el núcleo del método compositivo utilizado en la composición de *Threnody for Dimitris Christoulas*, documentada en el anexo B. Sin embargo, la inclusión de procesos recursivos dentro de un genotipo reviste cierta dificultad, derivada de la necesidad de *incrustar expresiones genotípicas como argumento de una función*. Los problemas y las soluciones que se han aplicado pueden ilustrarse mejor con un ejemplo:

Sea la función `scoRecursio`, que devuelve una secuencia basada en una ecuación recursiva y en una secuencia de eventos como condición inicial, definida según esta sintaxis:

`scoRecursio(`*condición inicial*`,` *ec. recursiva*`,` *n.º de eventos generados*`)`

Con ella podemos generar los primeros diez términos de la serie de Fibonacci `[1,1,2,3,5,8,13,21,34,55]` mediante este genotipo:

```
scoRecursio(
    [1,1],
    encodeExpr(
        "sum(
            nthLastNote(
                s,
                2),
            nthLastNote(
                s,
                1))"),
    10)
```

En este genotipo la ecuación recursiva aparece incrustada dentro de la función `encodeExpr` —tratada en la sección siguiente—, y en ella se hace una suma de las dos últimas notas (función `nthLastNote`) de la secuencia dada como condiciones iniciales, `[1,1]`, y referenciada por convención por la letra `s`. Es importante notar las comillas que enmarcan la función recursiva, que son imprescindibles para que esta expresión no sea evaluada como parte del genotipo principal, sino pasada como argumento para su evaluación dentro de la función `scoRecursio`.

Las expresiones recursivas se generan con la función `createRecursio`, que es una variante del autómata generador de fenotipos mostrado en la figura 6.2, la cual toma como único argumento el número de eventos de la secuencia que actúa como condiciones iniciales de la recursión. Así, la expresión `createRecursio(3)` puede devolver como salida una expresión como esta:

```
ratio(
    invers(
        sum(
            intrv(
                s,
                3),
            intrv(
                s,
                1))),
    cos(
        intrv(
            s,
            2)))
```

En este caso no se toman en cuenta las notas de las condiciones iniciales sino el intervalo que forman: `interv(s,2)` devuelve el penúltimo intervalo de la secuencia que se evalúa. Así, si las condiciones iniciales vienen dadas por el vector (n_1, n_2, n_3, n_4), esta recursión puede expresarse así en notación estándar:

$$n_0 = \frac{1}{(n_4 - n_3) + (n_2 - n_1)} : \cos(n_3 - n_2) \tag{6.4}$$

En *JavaScript*, el uso de expresiones genotípicas como argumento de una función exige el uso de comillas. Surge entonces un problema de sintaxis cuando se desea anillar recursiones dentro de otras recursiones: no podemos crear niveles sucesivos de comillas como sí es posible hacer con los paréntesis. El siguiente texto no se entiende como dos niveles de comillas anidados:

`"nivel1 "nivel2" continúa nivel1"`

Sino que el compilador lo interpreta así:

`"nivel1 "nivel2" continúa nivel1"`

Para solventar esta dificultad, *GenoMus* codifica las expresiones genotípicas en forma de *arrays*, con la función `encodeExpr`. Su funcionamiento se describe a continuación.

6.5.3 Funciones de conversión entre genotipo y fenotipo

La posibilidad de convertir las complejas expresiones alfanuméricas en formatos más simples, puramente numéricos, conlleva varias ventajas. La información se comprime enormemente, lo que redunda en una mayor eficiencia de las manipulaciones de síntesis y análisis que ocurren dentro del programa. Puede afirmarse que lo ideal es que los genotipos y fenotipos adquieran forma legible sólo cuando va a hacerse una lectura o escritura manual. Para el resto de procesos no es necesaria la legibilidad, sino la mayor compresión de los datos.

GenoMus incorpora la función `encodeExpr` para la conversión de genotipos en *arrays*; la función `decodeExpr` realiza la operación inversa. La implementación de estas funciones atendió en primer lugar la necesidad de articular el anidamiento de expresiones recursivas, tal como se expuso en la sección anterior, pero al margen de esta utilidad práctica, es evidente que futuros desarrollos de la herramienta exigirán un trabajo de compresión de los genotipos y fenotipos mucho mayor, para una gestión más eficiente de todos los procesos.

La función `encodeExpr` recibe como argumento una expresión genotípica entre comillas, y devuelve un *array* o matriz unidimensional como salida, lo que *de facto* convierte un genotipo en un fenotipo. Veamos un ejemplo de encriptación de un genotipo que usa la función `scoExcerpt`, la cual extrae un fragmento de una secuencia dada según esta sintaxis:

`scoExcerpt(`*secuencia*`,` *evento inicial*`,` *n.º de eventos extraídos*`)`[55]

Sea el siguiente genotipo:

```
scoExcerpt(
    [2,6,4,8,7,3,5,2,15,9],
    intPrim(),
    6)
```

Esta expresión devuelve una secuencia de 6 eventos que comienza en el elemento determinado al azar por la función primitiva `intPrim`. Si `intPrim` toma el valor 3, el fenotipo que se consigue es `[8,7,3,5,2,15]`.[56] Si esta función se pasa (entre comillas) como argumento de la función codificadora, de este modo:

```
encodeExpr(
    "scoExcerpt(
        [2,6,4,8,7,3,5,2,15,9],
        intPrim(),
        6)")
```

se obtiene como salida una secuencia de números que es indistinguible de un fenotipo:

```
[1,36,5,4,10,2,6,4,8,7,3,5,2,15,9,2,1,3,5,0,2,3,6,0]
```

La codificación se hace usando un identificador numérico para cada uno de los elementos que pueden formar parte de una expresión genotípica, seguido de los valores o funciones que se encriptan siguiendo estas reglas de conversión:

[55] Si el número de eventos extraídos es negativo, se extraen los elementos en sentido retrogradado.

[56] El primer elemento de un *array* se numera con el 0.

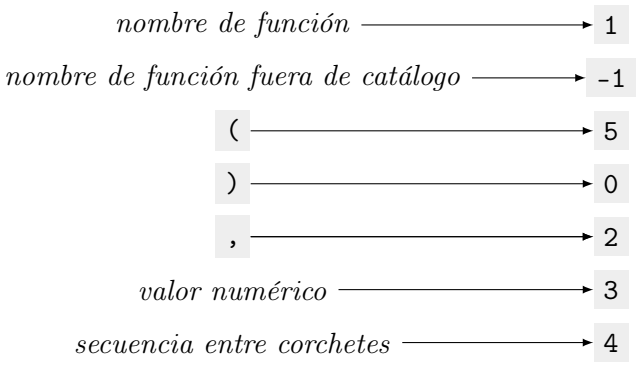


Figura 6.22: Identificadores numéricos de los elementos de los genotipos

Ahora podemos seguir paso a paso la conversión del genotipo del ejemplo:

`scoExcerpt(` → `1,36,5`

Al encontrar un nombre de función se escribe un `1`, seguido por el código `36`, que se corresponde con la función `scoExcerpt` según un catálogo generado previamente de manera automática por la librería de funciones. Por último, el paréntesis izquierdo se codifica con el `5`.

En el caso de funciones que no están catalogadas, se realiza una conversión carácter a carácter según el estándar ASCII, lo que resulta en una codificación mucho menos comprimida. Si `scoExcerpt` estuviera fuera del catálogo su codificación comenzaría con el identificador `-1` seguido por el número de caracteres que se codifican (`10`), y por el valor ASCII de cada letra, con lo que la expresión resultante es mucho más larga, lo que en genotipos complejos supondría multiplicar su extensión e ineficiencia:

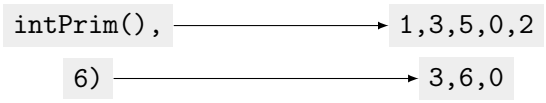
`scoExcerpt(` → `-1,10,115,99,111,69,120,99,101,114,112,116,5`

La segunda línea del genotipo es una secuencia numérica entre corchetes, codificada así:

`[2,6,4,8,7,3,5,2,15,9],` → `4,10,2,6,4,8,7,3,5,2,15,9,2`

Una vez encontrado un corchete izquierdo, se escribe un `4`, al que sigue otro valor que indica el número de elementos contenidos en el *array*, y los propios valores de la secuencia. Por último, el `2` identifica la coma separadora de argumentos.

Sabiendo que el identificador de `intPrim` es `3`, resulta fácilmente comprensible la codificación de las últimas dos líneas:



La función de decodificación `decodeExpr` realiza la operación opuesta: tomando un *array* como argumento, se construye una expresión genotípica usando las mismas conversiones en sentido inverso. Para reconstruir el genotipo del ejemplo, basta invocar `decodeExpr` de este modo:

```
decodeExpr(
    [1,36,5,4,10,2,6,4,8,7,3,5,2,15,9,2,1,3,5,
       0,2,3,6,0])
```

Volviendo al problema de las funciones recursivas incrustadas en los genotipos, la codificación de la recursión en forma de *array* facilita su manejo. Ahora es posible reescribir el genotipo que ejemplificaba la sección anterior de este modo:

```
scoRecursio(
    [1,1],
    encodeExpr(
        "sum(
            nthLastNote(
                s,
                2),
            nthLastNote(
                s,
                1))"),
    10)
```

```
scoRecursio(
    [1,1],
    [1,6,5,1,78,5,-1,1,115,2,3,2,0,2,1,78,5,-1,1,115,
         2,3,1,0,0]
    10)
```

Aún mas importante, al eliminar las comillas y poder referirnos a una expresión genotípica como argumento bajo la forma de un *array* convencional, se hace posible establecer recursiones que incluyan otras recursiones sin ninguna traba sintáctica. Consideremos este genotipo:

```
scoRecursio(
    [1,1],
    encodeExpr(
        "sum(
            nthLastNote(
                s,
                2),
            scoRecursio(
                [3,5],
                encodeExpr(
                    "sum(
                        nthLastNote(
                            s,
                            2);
                        nthLastNote(
                            s,
                            1))"),
                1))"),
    10)
```

Esta expresión con una función recursiva que incluye otra función recursiva interna, es inviable por el conflicto que plantea el uso de las comillas.[57] Sin embargo, al encriptar las expresiones entre comillas este problema desaparece. Veámoslo en dos tiempos:

```
scoRecursio(
    [1,1],
    encodeExpr(
        "sum(
            nthLastNote(
                s,
                2),
            scoRecursio(
                [3,5],
                [1,6,5,1,78,5,-1,1,115,2,3,2,0,2,1,78,
                    5,-1,1,115,2,3,1,0,0]
                1))"),
    10)
```

Podemos ahora encriptar la función recursiva principal, ya que la recursión secundaria se ha convertido en un *array* convencional y es codificado como tal:

```
scoRecursio(
    [1,1],
    [1,6,5,1,78,5,-1,1,115,2,3,2,0,2,1,79,5,4,2,3,5,2,
     4,25,1,6,5,1,78,5,-1,1,115,2,3,2,0,2,1,78,5,
     -1,1,115,2,3,1,0,0,3,1,0,0],
    10)
```

[57] Puede comprobarse cómo el propio intérprete gramatical de *JavaScript* que colorea automáticamente el listado de código interpreta "erróneamente" este anidamiento.

Y como se vio anteriormente, aún es posible una última encriptación que codifique el genotipo como un único *array*, con lo que llegamos de nuevo al punto en que genotipos y fenotipos se presentan bajo el mismo formato:

```
[1,79,5,4,2,1,1,2,4,53,1,6,5,1,78,5,-1,1,115,2,3,2,
    0,2,1,79,5,4,2,3,5,2,4,25,1,6,5,1,78,5,-1,1,115,
    2,3,2,0,2,1,78,5,-1,1,115,2,3,1,0,0,3,1,0,0,2,3,
    10,0]
```

6.5.4 Procesos = datos

Una consecuencia importante derivada de los isomorfismos mostrados en la sección anterior es que *cualquier proceso puede ser un fenotipo*, mientras que, bajo ciertas claves de conversión, *algunos fenotipos pueden ser genotipos*. Esta correspondencia guarda semejanzas con la que se crea con la *numeración de Gödel*, en la que a cada teorema se le asigna un número compuesto aplicando una serie de conversiones numéricas a los símbolos que lo formulan.[58] Esto nos lleva a la atractiva idea de que cualquier proceso musical imaginable está escondido en los espacios vectoriales de las matrices unidimensionales, y a la vez es similar a la situación creativa cotidiana del compositor, que se ve envuelto de continuo en la toma de decisiones estéticas: al tiempo que cada elección puede seguirse de un método, el método se ve transformado en cada plasmación, originándose así un mecanismo retroalimentado de evolución estilística.

La conversión de un genotipo en un fenotipo, o dicho de otro modo, la posibilidad de que *los propios procesos musicales puedan codificarse en forma de "melodía" numérica*, sugiere inmediatamente la extraña idea de *hacer sonar los procesos en sí mismos*. Aunque *a priori* esto parece un sinsentido, sólo hay que pararse a pensarlo dos veces para advertir que no hay tanta distancia entre este aparente absurdo y los procesos musicales convencionales, puesto que en definitiva de lo que se trata es de la organización de sonido conforme a reglas de inferencia. El que la fluidez de la información consiga el punto virtuoso de comunicabilidad y expresión que denominamos *música* seguirá siendo el eterno problema del compositor. Las mejores soluciones probablemente no dependen tanto de las relaciones internas de los sonidos como del talento para su proyección—en el sentido de *arreglo* musical—, que optimice la mejor percepción posible de las características que los articulan y les dan coherencia. Por lo pronto, la versión preliminar de *GenoMus* presentada en este texto se limita a disponer una plataforma de experimentación adecuada a la exploración musical basada en la metaprogramación; las decisiones últimas siguen siendo responsabilidad del compositor humano.

[58] Una definición rigurosa del funcionamiento de la numeración de Gödel con un análisis de sus consecuencias puede hallarse en [3], mientras que una explicación menos técnica y un análisis mucho más inspirador y revelador de sus conexiones con los ámbitos de la música, las artes plásticas, la psicología y la filosofía se encuentra en el clásico de Hofstadter [24].

6.6 El entorno de experimentación

Tanto el desarrollo del código de *GenoMus* como los genotipos y fenotipos usados en las composiciones presentadas se han llevado a la práctica en el marco de un *patch* de *MaxMSP*, denominado GenoMus_interface. Este entorno de trabajo se reduce a lo esencial para permitir la comunicación entre los algoritmos de *JavaScript* y el resto de facilidades que ofrece *MaxMSP*. El foco del trabajo está en las manipulaciones del metalenguaje interno, el cual está diseñado para una utilización independiente de los diferentes contextos posibles de *software*, plataformas o dispositivos. La interfaz que se presenta en esta sección busca la mayor simplicidad para que el trabajo con los genotipos y fenotipos pueda estar bien encapsulado y conectarse con facilidad a otras aplicaciones musicales, ya sean de audio, MIDI, notación, multimedia, etc. Más que los detalles de su implementación, lo primordial es visualizar qué elementos son necesarios para que el trabajo del compositor con la herramienta sea productivo y le permita concentrarse en lo esencial del trabajo creativo.

6.6.1 Elementos de la interfaz de *MaxMSP*

La interfaz[59] (figuras 6.23 y 6.24) consta de varios módulos:

- GenoMus_interface es la ventana principal que controla la generación y el modelado de genotipos y fenotipos. Aquí está alojado el código *JavaScript* de *GenoMus*.
- genotype despliega las expresiones genotípicas y permite la edición manual de estas expresiones.
- phenotype muestra los *arrays* numéricos que forman los fenotipos.
- auxiliary se usa para mostrar información complementaria, normalmente usada durante el desarrollo y depuración de nuevo código.
- laboratory es el espacio para la experimentación y la creación de nuevos elementos de comunicación entre la interfaz y el código interno de *GenoMus*.
- roll hace una representación gráfica de la conversión del fenotipo a notas MIDI.
- outputs es el espacio en que se realizan las conversiones de los fenotipos a diferentes formatos de salida, como MIDI o texto, y desde donde puede establecerse la comunicación con otros *patches* de *MaxMSP* o con aplicaciones externas.

[59] La última versión operativa del código fuente de *GenoMus* y de la interfaz de *MaxMSP* puede descargarse en http://www.lopezmontes.es/genomus.html.

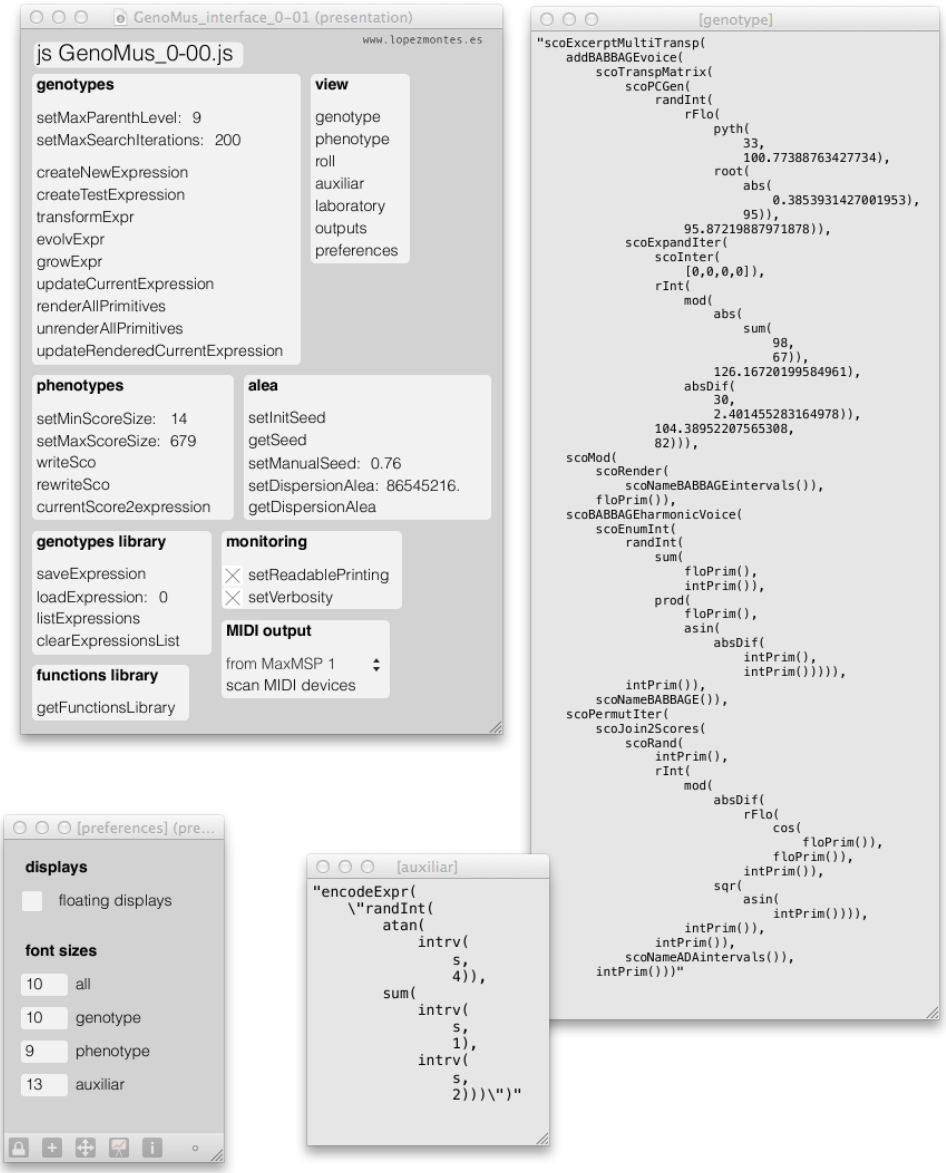


Figura 6.23: Interfaz de *GenoMus* (control de genotipos)

– preferences controla varios aspectos de la interfaz, tales como el tamaño de la fuente de las ventanas de texto, y el tipo de las ventanas de monitoreo.

– La ventana Max, propia de *MaxMSP*, imprime los mensajes generados durante el funcionamiento de *GenoMus*.

6.6.2 Operaciones desde la interfaz

Desde la ventana principal se activan todas las operaciones sobre genotipos y fenotipos. Muchos de estos botones son en realidad llamadas a las funciones de idéntico nombre del código interno de *GenoMus*. Podemos conocer, por ejemplo, los procesos que activa el botón rewriteSco consultando el código de la función `rewriteSco` del archivo GenoMus_*versión*.js, al que se accede haciendo doble clic sobre el objeto js.

Las operaciones de la interfaz se agrupan por paneles:

Panel genotypes

– setMaxParenthLevel determina la profundidad máxima de niveles de paréntesis de las expresiones genotípicas.

– setMaxSearchIterations fija el máximo número de intentos de la rutina de creación de genotipos.

– createNewExpression genera un genotipo cuya expresión se despliega en la ventana genotype.

– createTestExpression genera un genotipo forzando el uso de una o varias funciones específicas, ya sea para su depuración o como parte del trabajo compositivo. Estas funciones se enumeran en el *array* `tests_functions` del archivo genomus_library_*versión*.js.

– transformExpr hace una variación del genotipo actual sin aumentar los niveles jerárquicos del fenotipo.

– evolvExpr inserta el genotipo actual como parte de una expresión mayor.

– growExpr inserta el genotipo actual en una expresión mayor que incluya autorreferencias al material previo.

– updateCurrentExpression fija las transformaciones de las funciones anteriores como nuevo genotipo de referencia.

– renderAllPrimitives convierte las funciones primitivas en valores numéricos visibles. Cada vez que se repite esta operación se asignan valores diferentes a esas variables.

– unrenderAllPrimitives revierte el efecto de la función anterior, volviendo a la expresión genotípica base.

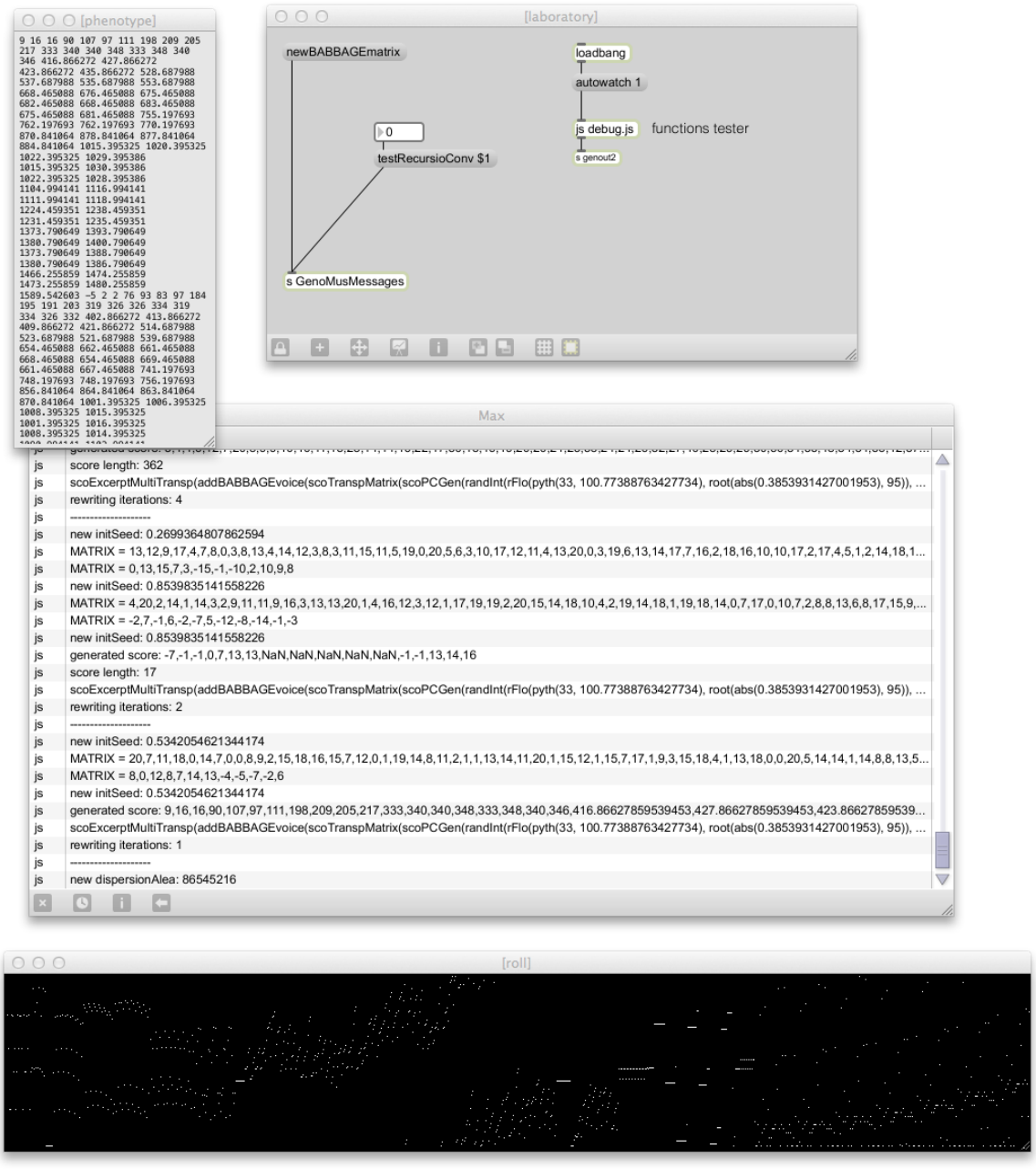


Figura 6.24: Interfaz de *GenoMus* (monitorización de fenotipos)

– updateRenderedCurrentExpression fija en un nuevo genotipo de referencia los valores asignados por renderAllPrimitives.

Panel phenotypes

– setMinScoreSize y setMaxScoreSize fijan respectivamente el mínimo y máximo número de eventos que puede contener un fenotipo.
– writeSco escribe un fenotipo a partir del genotipo actual.
– rewriteSco escribe un nuevo fenotipo usando un valor `seed` diferente.
– currentScore2expression aplana el fenotipo, convirtiendo el genotipo actual en parte del genotipo, reduciéndolo a una función declarativa de eventos.

Panel genotypes library

– saveExpression guarda el genotipo actual con un número de identificación.
– loadExpression carga el genotipo guardado con el número de identificación correspondiente.
– listExpressions imprime en la ventana Max la lista de expresiones genotípicas guardadas junto a sus números identificativos.
– clearExpressionsList borra la lista de expresiones genotípicas guardadas.

Panel functions library

– getFunctionsLibrary imprime en la ventana Max la lista de funciones disponibles para la formación de genotipos, junto a su número identificativo.

Panel view

– Cada botón abre la ventana correspondiente.

Panel alea

– setInitSeed establece un nuevo valor aleatorio para `seed`.
– getSeed imprime en la ventana Max el valor actual de `seed`.
– setManualSeed usa el número introducido por el usuario como nuevo valor de `seed`.
– setDispersionAlea usa el número introducido por el usuario como nuevo valor de `dispersionAlea`.
– getDispersionAlea imprime en la ventana Max el valor actual de `dispersionAlea`.

Panel monitoring

- setReadablePrinting conmuta entre la visualización de los genotipos con o sin saltos de línea y tabulado automático.
- setVerbosity establece si la ventana Max imprime todos los mensajes generados por *GenoMus* o una versión reducida de esta información.

Panel MIDI output

- El primer botón es un desplegable para seleccionar el dispositivo de salida MIDI al que se dirigirá la salida del fenotipo.
- scan MIDI devices busca los dispositivos MIDI disponibles.

6.6.3 Manipulación manual de los genotipos

Las ventanas genotype y phenotype, además de mostrar las expresiones de los genotipos y fenotipos, funcionan en la práctica como una consola de *JavaScript* que se comunica directamente con el código de *GenoMus*. Esto posibilita la edición manual directa de genotipos, ya sea desde cero, o modificando expresiones previas, y además facilita el desarrollo y depuración de la nuevas funciones. También es posible copiar y pegar texto, con lo que resulta muy sencillo guardar y reutilizar genotipos.

Cuando la opción setReadablePrinting está activada, el formateo de los genotipos con saltos de línea y tabulado automático se realiza inmediatamente después de ser introducido el texto, lo que permite detectar rápidamente cualquier error.

6.6.4 Tipos de salida

Un fenotipo es una secuencia ordenada de números. El *subpatch* outputs es el lugar donde estos datos se manipulan para convertirlos al formato deseado. En el caso de las composiciones documentadas en los anexos B y C, durante las tareas de composición se combinaron salidas de texto plano, conversión a archivos MIDI, síntesis de audio en tiempo real, envíos como flujo directo de datos MIDI a editores de partituras, secuenciadores e instrumentos virtuales, y traducción a salidas gráficas.

Se ha experimentado igualmente con *scripts* de conversión al prometedor formato *MusicXML*.[60] Este formato pugna por convertirse en el ansiado estándar de intercambio de archivos musicales que integre la información relativa a la notación musical y la propia de la ejecución secuencial. Una implementación de *GenoMus* en una plataforma *web* debería contar con la conversión de los fenotipos a este formato.

[60] http://www.musicxml.com

7

Conclusiones

Este trabajo se presenta como un primer alto en un camino de más alcance. Tal y como se expuso en el capítulo 1, en este punto se trata de sentar bases sólidas para las fases posteriores de investigación. Esto explica que las conclusiones aquí enumeradas sean necesariamente parciales, y tengan dos caras: una que mira a lo ya realizado y otra que considera cuáles son las proyecciones de las mismas de cara al diseño de nuevos experimentos.

1. *La programación funcional tiene capacidad expresiva para representar materiales musicales muy diversos.*

   La programación funcional puede describir técnicas musicales propias de estilos diferentes. Posibilita la integración entre procesos musicales tradicionales, métodos propios de la música contemporánea y, lo que la hace más interesante, procedimientos compositivos que requieren gran volumen de computación y que sólo se pueden explorar ágilmente con ayuda del ordenador. La librería actual de funciones con la que se ha trabajado es aún muy limitada. Para que su potencia de representación sea virtualmente universal —esto es, sea capaz de computar cualquier manipulación musical posible—, se requieren varias condiciones, algunas expuestas en la sección 8.1: el manejo de eventos de varias dimensiones, la abstracción automática de nuevas funciones a partir de las preexistentes, o el manejo de variables globales que afecten a fragmentos completos.

2. *Para conseguir una orientación óptima del metalenguaje de GenoMus hacia la metaprogramación los tipos de entrada y salida de las funciones que lo integran deben reducirse y homogeneizarse al máximo.*

   El principio de modularidad y compatibilidad entre funciones se ha plasmado reduciendo al mínimo los tipos de datos que se intercambian e introduciendo conversiones internas que puedan extraer datos útiles de casi cualquier entrada. Esto multiplica las posibilidades de crear expresiones bien formadas y potencia la exploración de relaciones imprevistas. La contrapartida puede ser la pérdida de legibilidad de los genotipos —ya que los mapeos internos pueden ser bastante crípticos— y la aparición de ramificaciones de complicación injustificada. Es importante optimizar estos dos aspectos para el trabajo futuro con genotipos más complejos, especialmente al manejar eventos musicales de varios parámetros.

3. *La librería de funciones puede integrar procesos primarios con sistemas expertos.*

   Las funciones llamadas por un genotipo pueden ser operaciones primitivas, operaciones compuestas —más o menos relacionadas con procesos musicales— o incluso sistemas expertos que realicen tareas especializadas y bien definidas. El desarrollo de GenoMus hacia su propia autonomía irá probablemente dirigido hacia las primeras. No obstante, en estas etapas iniciales en que se precisa un control manual de los genotipos, ha sido muy útil definir funciones con manipulaciones específicas del material, para ver su interacción con las operaciones básicas. La evolución del sistema quizá permita la formulación automática de sistemas expertos.

4. *La recursión a partir de funciones sencillas es capaz de generar comportamientos musicales coherentes y, en algunos casos, más interesantes que los producidos a partir de funciones que modelan procesos más explícitamente musicales.*

   Los primeros ensayos con funciones aritméticas básicas mostraron una considerable variedad de resultados. Aunque inicialmente fueron realizados sólo para testar los algoritmos centrales de síntesis de genotipos, los resultados de estos experimentos previos constituyeron una base suficiente para sustentar obras de cierta envergadura, tal como documenta el anexo B. Usada como recurso compositivo, la recursión aplicada a esta aritmética básica garantiza en gran medida la coherencia

interna de las texturas, al tiempo que puede exhibir comportamientos complejos. De hecho, algunos fragmentos generados con la recursión mostraron más sutileza que los procedentes de funciones que modelan procedimientos propiamente musicales. Este hecho sugiere dirigir los próximos pasos hacia la exploración de interacciones entre operaciones simples, en detrimento de la implementación de técnicas convencionales de composición. La combinatoria de funciones primitivas es el espacio natural de la metaprogramación, por tanto esta investigación debe centrarse en mejorar las herramientas de análisis y evolución de genotipos, y no tanto en incrementar el repertorio de funciones complejas preprogramadas.

5. *La interactividad en tiempo real es imprescindible para el desarrollo y uso de GenoMus.*

   La formulación de espacios musicales y su evaluación en tiempo real con ayuda de la computadora permiten crear estructuras de interés musical imposibles de realizar manualmente. En ocasiones, las regularidades de estas estructuras sólo emergen tras numerosas operaciones y sutiles modificaciones de las condiciones iniciales, por lo que el cálculo acelerado es imprescindible para trabajar con los genotipos. Dado que en música el interés suele encontrarse en un área situada entre lo trivial y lo excesivamente complicado, la sensibilidad de muchos de estos procesos a las condiciones iniciales —lo que implica que un mismo genotipo pueda desencadenar fenotipos enormemente diferentes en función de pequeñas variaciones en las variables que lo conforman— hace imprescindible la disposición de una interfaz que permita evaluar y refinar estos procesos en tiempo real.

6. *El progreso simultáneo de las herramientas informáticas y de las composiciones que las utilizan es más productivo.*

   El avance de todos los frentes en paralelo, desde el prototipado del lenguaje a la composición real, parece ser más productivo y realista que el desarrollo de herramientas en abstracto, sin contacto directo con sus aplicaciones artísticas inmediatas. El enfoque de *GenoMus* hacia la creatividad asistida se centra en la facultad del sistema para proponer posibilidades no previstas por el programador. Parece pues conveniente evitar funciones demasiado condicionadas por idiomas musicales llenos de apriorismos, así como abrir el campo de acción a la generalización de procesos más abiertos, pero confrontados con la práctica artística continua.

7. *La programación de herramientas informáticas creativas implica tantas decisiones artísticas como su aplicación directa en la composición.*

   GenoMus intenta dar un paso hacia la CAC en un nivel de abstracción mayor. La ubicuidad de las computadoras conlleva el interés de un sector de los compositores por las manipulaciones creativas de la gramática interna al propio lenguaje musical. La comprensión y codificación concreta de esas abstracciones lógicas implican no pocas decisiones arbitrarias, las cuales conforman un nuevo mundo personal de cada autor/programador. En muchos casos hay un límite difuso entre el diseño de una herramienta y su aplicación, por ello la programación forma parte de la acción creativa y al mismo tiempo la redefine.

8. *La modelización de una memoria musical virtual es necesaria para crear identidad artística.*

   Un estilo concreto se caracteriza por el uso específico que se hace de un conjunto de procedimientos. En desarrollos posteriores de *GenoMus*, la creación de una memoria a largo plazo de los genotipos seleccionados puede permitir una relación personal y proyectada en el tiempo entre un compositor y el programa. Sería posible definir perfiles estilísticos bien diferenciados, almacenados en una librería específica que detalle las preferencias de un usuario (real o virtual) individual. Un mecanismo de modelización de la memoria es esencial para permitir el autoaprendizaje del sistema y la acumulación de conocimiento.

9. *Las siguientes etapas de investigación precisan del uso de herramientas eficaces para la autoevaluación de genotipos y fenotipos.*

   El algoritmo actual es como un compositor sin apenas filtros de autocrítica. Tiene ya bastantes medios de expresión, pero trabaja a ciegas; depende de la evaluación humana para el filtrado de resultados y para la derivación de materiales de interés creciente. Llevar a cabo procesos exitosos de autoevolución requerirá el uso de una panoplia de funciones de análisis capaces de clasificar los fenotipos según multitud de parámetros y de dirigir las búsquedas subsiguientes de genotipos derivados. La sección 8.2 trata de anticipar un enfoque adecuado para la integración de estas herramientas.

10. *Los procesos son datos, y los datos pueden ser procesos.*

    En analogía a lo que sucede en la mente de un compositor, en *GenoMus* determinados procesos se compactan como datos precalculados y son almacenados en la memoria como secuencias numéricas. En la sección 6.5.3 se constató la conveniencia de poder codificar una expresión en forma de matriz unidimensional: así se posibilitó que un fragmento de genotipo se use como variable de otro genotipo. Al margen de sus ventajas prácticas para la computación, el hecho de que un proceso pueda presentarse en el mismo formato que los datos que representa tiene implicaciones de importancia: se pueden anidar múltiples funciones recursivas. Un importante consecuencia conceptual es que cualquier secuencia de datos puede verse como un proceso en potencia, y la propia codificación de un proceso podría entenderse y manipularse como una musicalización más. La autorreferencia supuso un problema insalvable para la lógica clásica pero, en cambio, constituye un interesante e inesperado campo de exploración musical.

8

Líneas de investigación abiertas

> *Tout le monde vous dira que je ne suis pas un musicien. C'est juste. Dès le début de ma carrière, je me suis, de suite, classé parmi les phonométrographes. Mes travaux sont de la pure phonométrique. Que l'on prenne le «Fils des Etoiles» ou les «Morceaux en forme de poire», «En habit de Cheval» ou les «Sarabandes», on perçoit qu'aucune idée musicale n'a présidé à la création de ces œuvres. C'est la pensée scientifique qui domine.*
>
> *Du reste, j'ai plus de plaisir à mesurer un son que je n'en ai à l'entendre. Le phonomètre à la main, je travaille joyeusement et sûrement.*[61]
>
> Erik Satie, *Mémoires d'un amnésique* [56]

El proyecto *GenoMus*, aún en un estado preliminar, se sitúa dentro de los estudios de IA que se focalizan en modelizar los mecanismos de adquisición de conocimiento y de recombinación en nuevos productos. Si la creatividad es la capacidad para crear soluciones nuevas a partir de conocimiento previo, *GenoMus* trata de articular un mecanismo análogo, articulando una estructura gramatical lo suficientemente modularizada como para posibilitar procesos de análisis, aprendizaje y síntesis de materiales musicales. En esta fase únicamente se ha desarrollado un modelo preliminar de gramática modularizada junto a los mecanismos básicos que la gestionan.

[61] *Todo el mundo les dirá que no soy músico. Es verdad, desde el principio de mi carrera me clasifiqué enseguida entre los fonometrógrafos. Mis trabajos son pura fonométrica. Si se cogen «Fils des Etoiles», o «Morceaux en forme de poire», «En habit de Cheval» o «Sarabandes», se verá que ninguna idea musical ha guiado la creación de estas obras. La reflexión científica es lo que domina.*

Por lo demás, me lo paso mejor midiendo un sonido que escuchándolo. Con el fonómetro trabajo alegre y seguro. (Trad. de Loreto Casado [57])

Otra característica esencial de *GenoMus* es que la semántica de esta gramática no se refiere a los eventos musicales mismos, sino a los procedimientos que los generan. En ese sentido, se alinea con otros metalenguajes musicales y al mismo tiempo con las técnicas de metaprogramación. Esto es una apuesta conceptual por la potencialidad de la metaprogramación combinada con la recursividad. Aún en una escala limitadísima, la idea clave es la de conseguir una cadena de autorreferencias que pueda llegar a automatizar el proceso de aprendizaje, creación y evaluación en órbitas de acción cada vez mayores.

GenoMus trata de atender simultáneamente a dos impulsos conectados pero aún muy distantes: por un lado está concebido como una herramienta práctica que se ha utilizado desde sus primeros prototipos para la composición de música real, y al mismo tiempo debe servir como un modelo a pequeña escala que ilustre un planteamiento de mucho más alcance para el desarrollo futuro de la creatividad artificial.

Las etapas intermedias hasta la emergencia de procesos genuinamente creativos son aún muchas e implican grandes desafíos técnicos y teóricos, pero al mismo tiempo son tan fascinantes en sí mismas que el camino promete ser gratificante en cada peldaño. Un experto en la materia como Goertzel [22] reconoce el aún largo trecho por recorrer, y remarca que una de las condiciones necesarias para que surjan comportamientos que puedan ser identificados como netamente creativos es la existencia de fuertes individualidades reconocibles entre otras muchas:

> *We were able to see, in specific cases, how the genetic algorithm creates complex forms* [...]. *With the study of creativity, however, we have reached a point where concrete simulation or calculation becomes very difficult. To get a self system to emerge from the dual network, one would need considerably larger computer systems than we have at present.* [...] *And in order to truly simulate creativity, one would have to make this system complex enough to produce, not only "artificial selfhood", dissociated selves – creative selves, capable of flexible reorganization and large-scale emergent pattern generation.*[62]

Llegado el punto en que esto llegase a ser posible, una de las preguntas más fascinantes que podrían ser resueltas sería: ¿cómo será la música en la que estas individualidades artificiales encuentren la belleza?

[62] *Hemos podido observar, en casos específicos, cómo el algoritmo genético crea formas complejas. Con el estudio de la creatividad, sin embargo, hemos llegado a un punto en el que la simulación concreta o el cálculo se hace muy difícil. Para conseguir que un sistema propio emerja de la red dual, se necesitarían sistemas de computadoras considerablemente mayores que los actuales. Y de cara a una verdadera simulación de la creatividad, el sistema se tendría que hacer lo suficientemente complejo para producir no sólo "individualidad artificial" sino identidades disociadas – identidades creativas, capaces de reorganización flexible y de generar la emergencia de patrones a gran escala.* (Trad. del autor)

8.1 Próximos pasos en el desarrollo de *GenoMus*

La ampliación de un sistema como *GenoMus* puede degenerar rápidamente en un grado de complicación inmanejable. La concepción inicial del proyecto consiste en crear un algoritmo capaz de representar procesos musicales complejos, dotado de la máxima libertad en el manejo autónomo de sus conocimientos musicales internos. El objetivo fundamental siguiente consiste en aumentar la potencia expresiva del metalenguaje y su flexibilidad combinatoria, al mismo tiempo que se dirigen las búsquedas adecuadamente para obtener resultados de valor. En esa dirección, estos son algunos de los desarrollos inmediatos a los que apunta la continuación lógica del proyecto:

8.1.1 Fenotipos multidimensionales

La versión de *GenoMus* presentada aquí genera fenotipos unidimensionales que operan sobre un solo parámetro para cada evento sonoro. Por simplicidad, en el caso de las composiciones compuestas para esta investigación los fenotipos codifican sólo el parámetro altura. Trabajar con flexibilidad sobre cualquier número y tipo de parámetros entraña cambios importantes en la estructuración tanto de los genotipos como de los fenotipos:

- Es necesario un método flexible para definir el número y tipo de parámetros que representarán los fenotipos.

- Hay que reprogramar la mayoría de funciones para adaptarlas a la gestión de parámetros múltiples.

- La representación de la polifonía debe admitir varios enfoques en función de si ésta se concibe como coexistencia de líneas independientes o como texturas más compactas y granulares.

- Los fenotipos deben contar con un bloque de metadatos que defina el número y tipo de parámetros que caracterizan la lista de eventos.

- Para manejar los fenotipos probablemente es conveniente contar con dos formas de representación: como matriz bidimensional, para visualizar fácilmente la lista de eventos —y poder usarla en con otras aplicaciones o lenguajes de programación, como *Csound*— y como *array* numérico unidimensional, de manera análoga a las conversiones discutidas en la sección 6.5.3.

- Es conveniente etiquetar ciertos parámetros habituales, tales como altura, duración, tiempo de inicio, dinámica o timbre, de manera que sea más directo referirse a ellos en las funciones de los genotipos, y diseñar funciones que los afecten selectivamente.

– La interfaz externa de *GenoMus* debe flexibilizarse para poder tratar ágilmente con fenotipos muy diversos.

8.1.2 Variables globales

En la creación de un proceso musical hay parámetros que dependen de la forma global y que afectan a grupos de eventos, cuando no a la totalidad. Es imprescindible incluir un mecanismo para modelizar y manejar este tipo de variables, entre las que se encuentran el tiempo transcurrido global, la evolución de la dinámica general, el nivel de disonancia, el uso de la tesitura, los planos armónicos activos o la evolución de la densidad rítmica. Estas variables deben considerar su estado actual y su historial, a fin de influir en las decisiones automáticas con un sentido de estructura unitaria.

La importancia de las líneas globales en la construcción de una composición y en la determinación de los niveles sintácticos sucesivos que pueden distinguirse ha sido analizada en profundidad por Berry [7], quien afirma:

> [...]*the "goal" of a directed textural progression within a phrase may thus be a point within a broader progression (viewed from a greater "distance") into which its effect is subsumed: or the aim toward which a phrase-level progression is directed may, viewed more broadly, be seen as well the goal of directed element-progressions within the scope of an entire structure. Textural structure is thus manifest at various levels, individual shapes falling into a hierarchic order of relatively subordinate relation to functional higher-level successions, with that of the overall structure encompassing and emerging as the fulfillment of all. And a comprehensive, "deep" view of musical structure requires here as elsewhere that observation and analysis be made within formal-structural dimensions of various breadths and depths—at diverse hierarchic levels from that of the smallest formal unit to those of broadly conceived contexts.*[63]

Y como aglutinante de todos los parámetros que afectan a grupos de eventos y a la globalidad de una composición, será fundamental introducir el factor *tiempo* para ponderar las propiedades del flujo de información de un fenotipo. Una correcta implementación de esta variable global —quizá la

[63] *La "meta" de una progresión textual dirigida dentro de una frase puede ser así un punto dentro de una progresión más amplia (vista desde una mayor "distancia") en la cual queda subsumido su efecto: o el objetivo hacia el cual se dirige una progresión al nivel de frase puede, visto más ampliamente, ser contemplado también como el objetivo de progresiones de elementos dirigidos dentro del ámbito de la estructura global. La estructura textural se manifiesta así en varios niveles, formas individuales formando parte de un orden jerárquico con una relación relativamente subordinada a sucesiones funcionales de más alto nivel, de las que emerge la estructura total como culminación de todo. Y una comprensiva y "profunda" visión de la estructura musical requiere, aquí como en otros lugares, que la observación y el análisis sea realizado usando dimensiones morfo-estructurales de varias escalas y profundidades—en diversos niveles jerárquicos desde la más pequeña unidad formal hasta los contextos más amplios concebibles.* (Trad. del autor)

más importante en música, ya que el resto de parámetros pueden reducirse en última instancia a proporciones y frecuencias temporales— es imprescindible. Como Maconie [35] asevera, "el compositor investiga el tiempo".

8.1.3 Abstracción automática de funciones

Uno de los resortes básicos para la evolución del sistema será la posibilidad de abstraer nuevas funciones de manera autónoma. Una vez seleccionada una expresión dentro de un genotipo, es sencillo compactarla en una única función y archivarla en una librería. Esta nueva función tendrá la forma de una función única con un número de argumentos derivado de la expresión inicial. La figura 8.1 muestra el esbozo de un posible mecanismo de abstracción automática de una nueva función a partir de una expresión compuesta.

8.1.4 Memoria a largo plazo y referencias externas a librerías de genotipos

La autorreferencia, tal como se ha trabajado en la sección 6.5.1, sirve únicamente para representar la memoria a corto plazo dentro de un genotipo —lo que equivaldría a la memoria de los últimos elementos usados en un breve discurso musical—. Este tipo de memoria es útil para la reutilización de elementos previos dentro de un fragmento, y se ha implementado haciendo referencias internas a fragmentos del propio genotipo. Extendiendo este concepto, el siguiente paso será posibilitar las referencias a genotipos y fenotipos externos. Esta técnica es análoga a la reelaboración que un compositor hace de elementos propios o ajenos usados con anterioridad, y que se han revelado como exitosos. Las referencias externas deben poder referirse a genotipos —procedimientos compositivos— o a fenotipos —melodías, armonías, etc., ya generadas y archivadas—. Para posibilitar este modelo de memorias multinivel que puedan corresponderse con perfiles estilísticos diferenciados se deberá estructurar la gestión automática de un sistema de archivos que organice este conocimiento virtual.[64] Será fascinante comprobar si el trabajo continuado de cada unidad de composición hombre–máquina —o incluso de cada máquina en forma autónoma— puede dar lugar a personalidades artísticas identificables.

8.2 La autoevaluación en la siguiente fase

Como ya se expuso en la sección 6.4.3, articular procedimientos para la autoevaluación autónoma de los resultados de *GenoMus* es la clave de un avance significativo del proyecto. Esta autoobservación del sistema probablemente debe sustentarse en la integración de muchas herramientas de análisis que

[64] Pueden ser de utilidad las conclusiones de Laske [30] en su estudio de la modelización de la memoria musical.

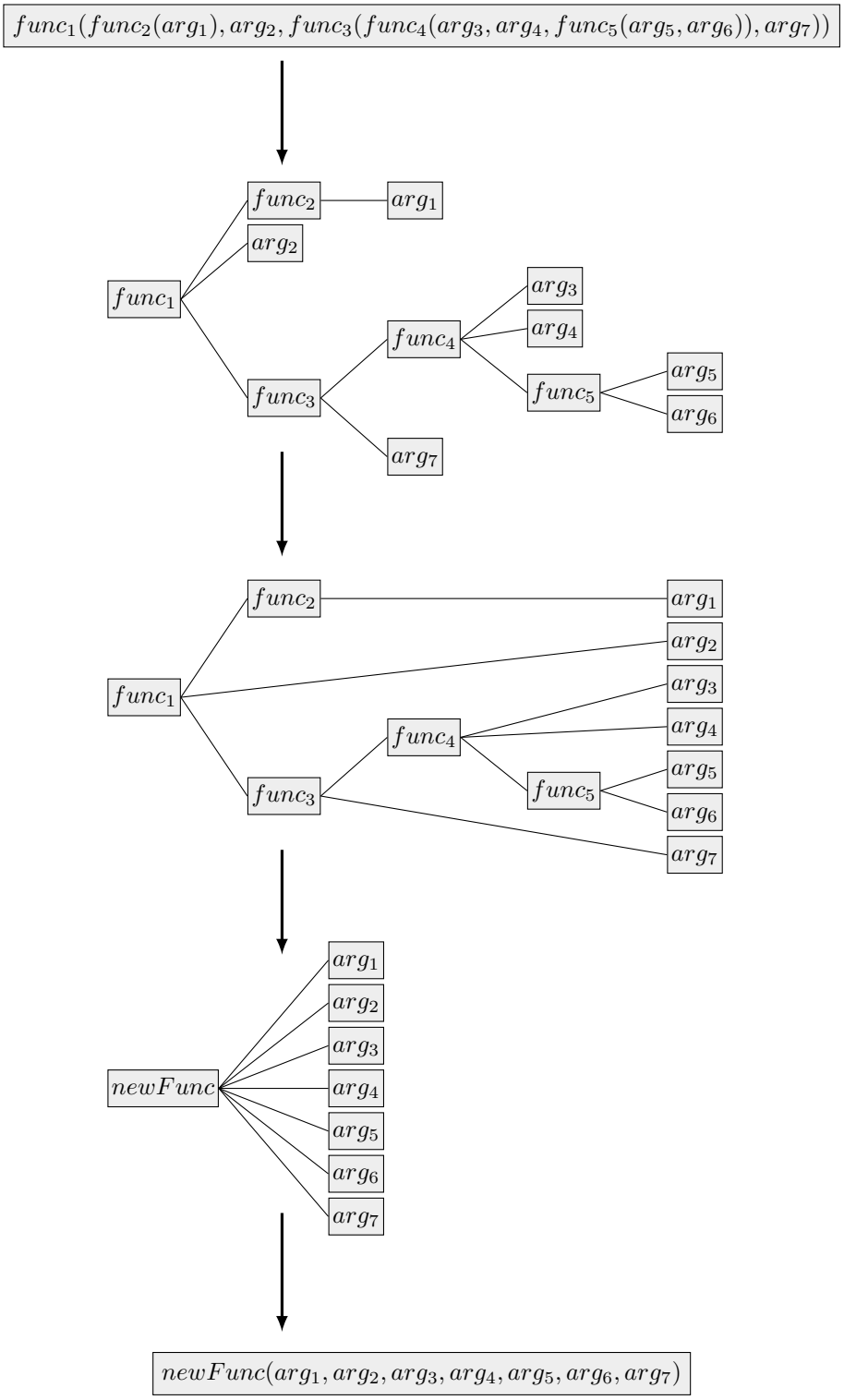


Figura 8.1: Abstracción de nuevas funciones a partir de los genotipos

trabajen en paralelo. Aún considerando la subjetividad intrínseca de la crítica musical, parece concebible imaginar la programación de algoritmos que puedan clasificar fragmentos musicales con arreglo a criterios tales como:

- El equilibrio en la cantidad de información por unidad de tiempo.
- La variedad en el uso de texturas
- La cantidad de relaciones internas que cohesionan el discurso.
- La estimación de lo común que es un fragmento en relación a la media de los fragmentos musicales que constituyan la memoria musical virtual del sistema.
- La eficiencia del par genotipo–fenotipo, que podría medirse considerando la razón entre los bits necesarios para la codificación de la expresión genotípica y la riqueza textural del fenotipo.
- La memoria sobre las valoraciones externas —humanas o algorítmicas— de fragmentos anteriores, que permitan hacer predicciones sobre el *éxito* de cada nuevo resultado.

Un sistema de evaluación bien calibrado es en la práctica un método heurístico de composición: *componer es seleccionar*.

8.3 Posibles desarrollos y aplicaciones de *GenoMus*

Una vez completado el ciclo propuesto en la figura 6.1, se hace posible abrir el campo de acción de *GenoMus* a aplicaciones más allá de la CAC:

- En el ámbito del *análisis*, sería interesante aplicar el concepto genotipo–fenotipo mediante *ingeniería inversa*: dado el fenotipo de una composición, el sistema debería encontrar los genotipos más concisos que expliquen la organización interna de la obra estudiada. Este tipo de análisis requerirá el manejo de una ingente cantidad de datos, pero es un enfoque que puede renovar el análisis musical tradicional.
- *GenoMus* puede tener igualmente su interés en la *didáctica musical*. Los genotipos tienen una sintaxis fácil de aprender y podrían usarse como un tipo de escritura musical procedimental que ayude a entender estos planteamientos conociéndolos de primera mano.
- La *improvisación asistida* es similar en muchos aspectos a la CAC. Pueden imaginarse situaciones en que la máquina vaya aprendiendo en tiempo real a iteractuar con otros improvisadores, infiriendo gramáticas ajenas y complementándolas con la suya propia. Los desafíos

técnicos de algo así son enormes, aunque ya existen progresos interesantes en experiencias como *OMax*[65] y en estudios sobre la anticipación musical [14].

– Es igualmente factible aplicar *GenoMus* al *arte multimedia*. El lenguaje de la animación y la visualización, así como cualquier otro elemento en juego, puede codificarse en gramáticas afines que permitan escribir genotipos generadores de eventos multimedia, que articulen procesos integrados de música, imagen, iluminación, movimiento, etc.

8.4 Importancia de la equivalencia entre procesos y datos

La visión de los procedimientos como datos y su concepción inversa, estudiada en la sección 6.5.4, tiene grandes implicaciones teóricas y prácticas. Al margen de las resonancias gödelianas de este isomorfismo, aplicaciones como las descritas en la sección anterior requerirán de una *compresión máxima de los datos*. Para el avance real de *GenoMus* es imprescindible la codificación de los procedimientos en *arrays* binarios, para que todos los procesos internos de creación, manipulación y evaluación tengan lugar al más bajo nivel. La traducción a salidas alfanuméricas de los genotipos y fenotipos deberá tener lugar sólo en el momento en que sea necesaria la lectura humana.

8.5 Música sin meta

La versión de *GenoMus* presentada en este trabajo tiene vocación de prototipo, no de programa acabado. El objetivo principal del trabajo presentado en esta investigación ha sido visualizar con claridad las estrategias de metaprogramación posibles y los escollos a superar, antes de dar un paso adelante en el nuevo nivel de abstracción que será necesario de aquí en adelante.

La metaprogramación no es sino programación en un plano superior de abstracción; la metamúsica producida con *GenoMus* no es sino música. La imperiosa necesidad de usar el prefijo *meta* al que este texto ha recurrido una y otra vez es un claro síntoma de que estamos pasando a otro nivel para referirnos a la creación musical. Este nivel engloba las técnicas tradicionales de composición —incluyendo las tradiciones del siglo XX— pero sobre todo abren un espectro potencial de relaciones nuevas e insospechadas.

La potencia computacional junto a la comprensión de los resortes de la inteligencia nos acerca a un nuevo escenario donde la interactividad, la im-

[65] *OMax* es un programa que escucha y aprende en tiempo real de las improvisaciones de otros músicos, para a su vez tomar parte activa integrándose en el discurso. Su plataforma *web* se encuentra en `http://repmus.ircam.fr/omax/home`.

provisación y el surgimiento de nuevos instrumentos y de nuevos modos de emplear los antiguos van a redefinir los roles del músico y del oyente.

La mente humana no cesa de crear y expandir espacios de conocimiento científico y artístico. Desde esa perspectiva, toda esta investigación se ha sostenido desde el convencimiento de que la IA no viene a suplantar la labor del creador humano sino que, en último término, es el espejo en el que nos reflejamos nosotros mismos. El estudio de la creatividad asistida nos confronta con la esencia de lo humano, caracterizada precisamente por la ausencia de una programación biológica cerrada.

La máquina no viene a sustituir al artista: muy al contrario, lo situará ante nuevas herramientas y formas de expresión que responderán no sólo a sus manos o a sus oídos, sino directamente a su pensamiento.

Y, finalmente, seguiremos siendo nosotros los que estamos ahí.

Anexo A

Código fuente de *GenoMus*

Archivos necesarios para ejecutar *GenoMus*:

- GenoMus_0-00.js es el archivo principal donde se encuentran las funciones que constituyen el núcleo del sistema. [Anexo A.1]
- genomus_library_00.js contiene las funciones con las que se forman los genotipos. [Anexo A.2]
- GenoMus_interface_0-01.maxpat es el *patch* con la interfaz que ejecuta *GenoMus* y permite interactuar con el programa en tiempo real.

  Estos tres archivos deben ubicarse en el mismo directorio.

- La función auxiliar include_lh.js permite que el archivo principal llame a las funciones de la librería, y debe instalarse en la carpeta jsextensions de *MaxMSP*. [Anexo A.3]

El código fuente de *GenoMus* y la interfaz de usuario se han desarrollado con la versión 5 de *MaxMSP*,[66] pero funciona perfectamente también en la versión 6.

Los comentarios insertados a lo largo de los *scripts* deben ser suficientemente autoexplicativos acerca de los procedimientos que realiza cada función.

La última actualización de estos archivos puede descargarse junto a otros materiales adicionales en http://www.lopezmontes.es/genomus.html.

[66] http://www.cycling74.com

A.1 Archivo principal GenoMus_0-00.js

```
////////// GenoMus_0-00.js
////////// -------------------------
////////// by Jose Lopez-Montes
////////// http://www.lopezmontes.es/genomus.html

// IMPORTANT: this script requires "include_lh.js", by Luke Hall, for including external
.js files
// extracted from: http://hallluke.wordpress.com/2010/10/31/including-extra-javascript-files/
// "include_lh.js" must be placed in the "jsextensions" Max' directory

outlets = 5;

//////////////////////
// global variables
var maxLevels = 5; // maximal number of parenthesis levels
var isBranch; // indentifies whether or not the actual expression is a subexpression
var maxNumberFuncions = 200; // maximal number of functions per expresssion
var openParenth = 0; // current level of parenthesis
var argumentStatus = []; // 0 = writing first argument, 1 = writing second argument, etc
var currentFunctionName; // current function being written
var newExpression; // new expression or branch created by core functions createExpression
var growingExpression; // provisional grown function from currentExpression;
var provisoryExpression = [];
var usedFunctionsArray = []; // array of used functions
var currentExpression; // the active expression
var currentExpressionRendered = []; // current expression with primitives rendered
var savedExpressions = []; // array of manually saved expressions
var generatedScoreLength;
var goodScore = 1; // controls whether or not the expression is well-formed
var minScoreSize = 1; // min. number of events in the generated scores
var maxScoreSize = 500; // max. number of events in the generated scores
var goodScoreSize = 0; // boolean for testing score size
var maxSearchIterarions = 100; // max. number allowed of iterations, searching a score
var readablePrinting = 1; // boolean for activate a formatted print of expr., best for
human reading
var dispersionAlea = 97940987898; // (0, 4) -> like logistic equation. For uniform-like
distribution, uses very long numbers
var chromaticQuantiz = 12; // number of divisions per octave (12 for usual chromatic division)
var alwaysAutoreference; // always include autoreferences in new branchs
var verboseOutput = 0;


////////////////////////////
// imported library of functions used to construct the expression
include(this,"/genomus_library_00.js");
////////////////////////////

//////////////////////
// set constants
function setMaxParenthLevel(x) {
    maxLevels = x;
}

function setMaxScoreSize(x) {
    maxScoreSize = x;
}

function setMinScoreSize(x) {
    minScoreSize = x;
```

```
}

function setMaxSearchIterations(x) {
    maxSearchIterarions = x;
}

function setReadablePrinting(x) {
    readablePrinting = x;
    printExpression(currentExpression);
}

function setVerbosity(x) {
    if ( x != 0 ) { verboseOutput = 1 }
    else { verboseOutput = 0 }
}

//// randomSeed generator ( float (0, 1) ), to do repeatable evaluations of functions
function setDispersionAlea(x) {
    dispersionAlea = x;
    if (verboseOutput == 1) { getDispersionAlea() };
}

function getDispersionAlea() {
    post("new dispersionAlea: " + dispersionAlea + "\n");
}

var initSeed = 0.93459283; // seed ( 0 < seed < 1 )
var seed = initSeed; // this value changes every time randomSeed function is called

function setInitSeed() {
    initSeed = Math.random();
    seed = initSeed;
    if (verboseOutput == 1) { getSeed() };
}

function setManualSeed(x) {
    initSeed = x;
    seed = initSeed;
    if (verboseOutput == 1) { getSeed() };
}

function getSeed() {
    post("new initSeed: " + initSeed + "\n");
}

function randomSeed() {
    seed = (seed*dispersionAlea*(1-seed)) % 1;
    return seed;
}

//////////////////////
// primitive funcions for expression manipulations

// adds a right parenthesis to expression
function addRightParenth(expression) {
    openParenth--;
    return expression + ")";
}

// adds a comma
function addComma(expression) {
    argumentStatus[openParenth]--;
```

```
    return expression + ", ";
}

// returns an integer constant from an interval
function rInt(value_1, value_2) {
    var minimum = Math.min(value_1, value_2);
    var maximum = Math.max(value_1, value_2) + 1;
    return Math.floor(randomSeed() * (maximum-minimum) + minimum);
}

// returns an integer constant
function intPrim() {
    return rInt(0, 127);
}

// returns a float constant from an interval
function rFlo(value_1, value_2) {
    var minimum = Math.min(value_1, value_2);
    var maximum = Math.max(value_1, value_2);
    return randomSeed() * (maximum-minimum) + minimum;
}

// returns a float constant
function floPrim() {
    return rFlo(0, 127);
}

// adds an integer constant to expression
function addIntConstant(expression, limit1, limit2) {
    return expression + rInt(limit1, limit2);
}

// adds a float constant to expression
function addFloatConstant(expression, limit1, limit2) {
    return expression + rFlo(limit1, limit2);
}

// adds a functionName to expression
function addFunctionName(expression, functionType, primitiveFlag) {
    if (primitiveFlag == 0) {
        var functions_set = eval(functionType + "_functions");
    }
    else {
        var functions_set = eval(functionType + "_primitive");
    }
    openParenth++;
    currentFunctionName = functions_set[rInt(0, functions_set.length - 1)];
    usedFunctionsArray[openParenth - 1] = currentFunctionName;
    var currentArgument = getArgs(eval(String(usedFunctionsArray[openParenth-1]))).length;
    argumentStatus[openParenth] = eval(currentFunctionName).length - 1;
    return expression + currentFunctionName + "(";
}

// gets the arguments type of a function (to determine how call to new functions)
// get all arguments name as array
function getArgs(func) {
    var str = func.toString().match(/\([\w*\s*\,*]*\)/).toString();
    return  str.substring(1, str.length - 1).split(",");
}

// gets the first five characters of an argument (to compare and delete index of several
arguments and retain only type)
```

```
function getArgType(functionName, argumentNum) {
    if (argumentNum == 0) {
        return getArgs(functionName)[argumentNum].slice(0,5);
    }
    else
    {
        return getArgs(functionName)[argumentNum].slice(1,6);
    }
}

// compress an expanded expression
function compressExpr(expandedFormExpr) {
    var temporaryExpr = "";
    for ( charIndx = 0; charIndx < expandedFormExpr.length; charIndx++) {
        if ( expandedFormExpr.charAt(charIndx) != " " && expandedFormExpr.charAt(charIndx)
!= "\n") {
            temporaryExpr = temporaryExpr + expandedFormExpr.charAt(charIndx);
        }
    }
    temporaryExpr = temporaryExpr.replace(/,/g,", ");
    expandedFormExpr = temporaryExpr;
    return expandedFormExpr;
}

// expands a compressed expression in a readable form
function expandExpr(compressedFormExpr) {
    compressExpr(compressedFormExpr);
    var expandedExpression = "";
    compressedFormExpr = compressedFormExpr.replace(/\(/g,"(\n");
    compressedFormExpr = compressedFormExpr.replace(/, /g,",\n");
    compressedFormExpr = compressedFormExpr.replace(/\n\)/g,")");
    var parenthCount = 0;
    for ( charIndx = 0; charIndx < compressedFormExpr.length; charIndx++) {
        expandedExpression = expandedExpression + compressedFormExpr.charAt(charIndx);
        if ( compressedFormExpr.charAt(charIndx) == "(" ) { parenthCount++ }
        if ( compressedFormExpr.charAt(charIndx) == ")" ) { parenthCount-- }
        if ( compressedFormExpr.charAt(charIndx) == "\n" ) {
            var tabulation = "    ";
            for ( n = 0; n < parenthCount; n++ ) {
                expandedExpression = expandedExpression + tabulation;
            }
        }
    }
    // rewrites expandedExpr mantaining matrices in a single line
    var matrixCompactExpr = "";
    var matrixOpen = 0;
    for ( charIndx = 0; charIndx < expandedExpression.length; charIndx++) {
        if ( expandedExpression.charAt(charIndx) == "[" ) { matrixOpen++ };
        if ( expandedExpression.charAt(charIndx) == "]" ) { matrixOpen-- };
        if ( matrixOpen > 0 ) {
            if ( expandedExpression.charAt(charIndx) != "\n" && expandedExpression.charAt(charIndx)
!= " " ) {
            matrixCompactExpr = matrixCompactExpr + expandedExpression.charAt(charIndx);
            }
        }
        else {
            matrixCompactExpr = matrixCompactExpr + expandedExpression.charAt(charIndx);
        }
    }
    compressedFormExpr = matrixCompactExpr;
    return compressedFormExpr;;
}
```

```
// prints currentExpression
function printExpression(expr) {
    if ( readablePrinting == 1 ) {
        outlet( 1, expandExpr(expr) );
        if ( verboseOutput == 1 ) {
            post( compressExpr(expr) + "\n" );
        }
    }
    else {
        outlet( 1, compressExpr(expr) );
        if ( verboseOutput == 1 ) {
            post( expr + "\n" );
        }
    }
}

function writeScore(newWrittenExpression ) {
    seed = initSeed;
    var generatedScore = "";
    generatedScore = eval(newWrittenExpression);
    generatedScoreLength = generatedScore.length
    if (goodScore == 1 && generatedScoreLength >= minScoreSize && generatedScoreLength
<= maxScoreSize ) {
        if ( isBranch == 0 && verboseOutput == 1 ) {
            post("generated score: " + generatedScore + "\n");
            post("score length: " + generatedScoreLength  + "\n");
        }
        outlet(0, "clear");
        outlet(2, generatedScore);
        for(i=0; i< generatedScoreLength; i++) {
            outlet(0, generatedScore[i]);
            outlet(0, "cr");
        }
        if ( isBranch == 0 ) { printExpression( newWrittenExpression ) };
        goodScoreSize = 1;
    }
}

// looks for a variation of the same expression with different seed
function rewriteScore(newWrittenExpression) {
    goodScoreSize = 0;
    numIter = 0;
    var newGeneratedScore = "";
    do {
        numIter++;
        setInitSeed();
        newGeneratedScore = eval( newWrittenExpression );
    } while ( (newGeneratedScore.length < minScoreSize || newGeneratedScore.length > maxScoreSize
) && numIter < maxSearchIterarions );
    if ( verboseOutput == 1 ) {
        getSeed();
        post("generated score: " + newGeneratedScore + "\n");
        post("score length: " + newGeneratedScore.length  + "\n");
    }
    outlet(0, "clear");
    outlet(2, newGeneratedScore);
    for(i=0; i< newGeneratedScore.length; i++) {
        outlet(0, newGeneratedScore[i]);
        outlet(0, "cr");
    }
    printExpression( newWrittenExpression );
```

```
    goodScoreSize = 1;
    if ( verboseOutput == 1 ) {
        post("rewriting iterations: " + numIter + "\n");
    }
}

////////////// CORE FUNCTION
// creates functions from scratch from library of functions
function createExpression(expressionType) {
    var createExpressionIteration = 0;
    var internalExpression, numFunctions;
    newExpression = "";
    do {
        createExpressionIteration++;
        // init goodScore
        goodScore = 1;
        goodScoreSize = 0;
        // init new function
        var createdExpression = "";
        usedFunctionsArray = [];
        var activeArgument;
        var functionTypeAsked;
        openParenth = 0;
        var writingAutomataStatus = 0;
        // write the first functionName
        createdExpression = addFunctionName(createdExpression,expressionType,0);
        // recursive writing process
        numFunctions = 0;
                while (openParenth > 0) {
            // after writing a functionName
            switch (writingAutomataStatus) {
                // last written: functionName or comma
                case 0:
                    activeArgument =  (getArgs(eval(String(usedFunctionsArray[openParenth-1]))).length
- argumentStatus[openParenth] -1) ;
                    functionTypeAsked = getArgType(eval(String(usedFunctionsArray[openParenth-1])),
activeArgument);2
                    if (Math.random() < .5 && openParenth < maxLevels) {
                        createdExpression = addFunctionName(createdExpression, functionTypeAsked,
0);
                        numFunctions++;
                        // number of functions used is limited to constant maxNumberFuncions,
to avoid too big expressions
                        if ( numFunctions > maxNumberFuncions ) {
                            //post("error: too many functions\n");
                            break;
                        }
                    }
                    else {
                        createdExpression = addFunctionName(createdExpression, functionTypeAsked,
1);
                        writingAutomataStatus = 1;
                        numFunctions++;
                    }
                    break;
                // last written: constant or right parenthesis
                case 1:
                    if ( argumentStatus[openParenth] > 0 ) {
                        createdExpression = addComma(createdExpression);
                        writingAutomataStatus = 0;
                    }
                    else {
```

```
                        createdExpression = addRightParenth(createdExpression);
                        writingAutomataStatus = 1;
                    }
                    break;
                }
            }
            // writes autoreferences, if needed
            if ( createdExpression.search("preScoInter()") != -1 ) {
                internalExpression = "scoInter([0," + eval(insco_primitive[rInt(0, insco_primitive.length
- 1)]) + "])";
                createdExpression = createdExpression.replace(/preScoInter\(\)/g, internalExpression);
            }
            if ( createdExpression.search("preScoRepExpr()") != -1 ) {
                internalExpression = "scoRepExpr([0," + eval(insco_primitive[rInt(0, insco_primitive.length
- 1)]) + "], intPrim())";
                createdExpression = createdExpression.replace(/preScoRepExpr\(\)/g, internalExpression);
            }
            if ( createdExpression.search("preEncRecursio()") != -1 ) {
                internalExpression = "[" + encodeExpr( createRecursio(4) ) + "]";
                createdExpression = createdExpression.replace(/preEncRecursio\(\)/g, internalExpression);
            }
            if ( createdExpression.search("preEncRecursioBABBAGE()") != -1 ) {
                internalExpression = "[" + encodeExpr( createRecursio(6) ) + "]";
                createdExpression = createdExpression.replace(/preEncRecursioBABBAGE\(\)/g,
internalExpression);
            }
            newExpression = createdExpression;
            writeScore( newExpression );
            if ( isBranch == 1 ) { goodScoreSize = 1 };
            // posts("Branch=" + isBranch + " goodSco=" + goodScore + " goodScoreSize=" +
goodScoreSize + "\n");
        } while ( createExpressionIteration < maxSearchIterarions && (goodScore == 0 || goodScoreSize
== 0) );
        if ( isBranch == 0 ) { post("expr. creation iterations: " + createExpressionIteration
+ "\n") };
        return newExpression;
}

//// automatic evolution of currentExpression, using only functions of functionSet as
the basis function of the evolved new expression
function evolveCurrentExpression(functionSet) {
        var evolIter = 0;
        var newRightBranch, evolvMainFunction, evolvMainFunctionArgType, numberNewBranches,
memoMinScoreSize;
        provisoryExpression = reindexInternalAddresses(currentExpression, "0,");
        isBranch = 1;
        do {
            newRightBranch = [];
            evolvMainFunction = eval(functionSet)[rInt(0, eval(functionSet).length - 1)];
// chooses a function from the evolv_functions set
            numberNewBranches = getArgs(eval(evolvMainFunction)).length - 1; // looks for
the number of new branches required
            growingExpression = evolvMainFunction + "(" + provisoryExpression; // + ", " +
")"; // composes a provisional but correct expression to avoid syntax errors
            for (newBranch = 1; newBranch <= numberNewBranches ; newBranch++) {
                evolvMainFunctionArgType = getArgType(eval(evolvMainFunction), newBranch);
                newRightBranch = createExpression( evolvMainFunctionArgType );
                // controls if autoreference in new branchs is compulsory
                if ( alwaysAutoreference == 1 && newRightBranch.search("scoInter") == -1 &&
newRightBranch.search("scoRepExpr") == -1 ) {
                    goodScore = 0;
                }
```

```
            growingExpression = growingExpression + ", " + newRightBranch;
        }
        growingExpression = growingExpression + ")";
        memoMinScoreSize = minScoreSize;  // suspends the evaluation of scoreLength while
constructing new branchs
        maxScoreSize = maxScoreSize * 10;
        writeScore( growingExpression );
        maxScoreSize = maxScoreSize / 10;
        minScoreSize = memoMinScoreSize;
        evolIter++;
        if ( generatedScoreLength < minScoreSize || generatedScoreLength > maxScoreSize
) { goodScoreSize = 0 };
    } while ( (goodScore != 1 || goodScoreSize != 1) && evolIter < 20 );  // allow only
20 iterations for evolv
    if ( alwaysAutoreference == 1 ) { alwaysAutoreference = 0 };
    if ( goodScoreSize == 1 ) {
        printExpression( growingExpression )
        if ( verboseOutput == 1 ) {
            post("score length: " + generatedScoreLength  + "\n");
        }
        post("evolv. iterations: " + evolIter + "\n");
    }
    else {
        post("solution not found" + "\n");
    }
    separator();
}

// types of evolution
function evolvExpr() {
    evolveCurrentExpression("all_evolvFunctions");
}

function growExpr() {
    alwaysAutoreference = 1;
    evolveCurrentExpression("grow_evolvFunctions");
}

function transformExpr() {
    evolveCurrentExpression("transform_evolvFunctions");
}

// converts primitives in numeric values
function renderPrimitiveFunctions( expressionToConvert ) {
    var regExpress, replaceExpr;
    for (priFunctNum = 0; priFunctNum < reducible_score_functions.length; priFunctNum++)
{
            functionToReduce = reducible_score_functions[priFunctNum];
            regExpress = "/" + functionToReduce + "\\(\\)/g";
            putKlammers = "[" + functionToReduce + "()]";
            expressionToConvert = expressionToConvert.replace( eval(regExpress), putKlammers
);
    }
    for (priFunctNum2 = 0; priFunctNum2 < all_reducible_functions.length; priFunctNum2++)
{
        do {
            functionToReduce = all_reducible_functions[priFunctNum2];
            regExpress = "/" + functionToReduce + "\\(\\)/";
            replaceExpr = eval(functionToReduce);
            expressionToConvert = expressionToConvert.replace( eval(regExpress), replaceExpr);
        } while ( expressionToConvert.search(functionToReduce) != -1 );
    }
```

```
    return expressionToConvert;
}

function testRecursioConv(x) {
    updateCurrentExpression();
    recursioTestExp = createRecursiveFunction( currentExpression, x );
    printExpression( recursioTestExp );
}

function renderAllPrimitives() {
    updateCurrentExpression();
    currentExpressionRendered = renderPrimitiveFunctions( currentExpression );
    rewriteScore( currentExpressionRendered );
}

function unrenderAllPrimitives() {
    text (currentExpression);
}

function updateRenderedCurrentExpression() {
    text (currentExpressionRendered);
}

////// activates functions from Max interface
function createNewExpression() {
    isBranch = 0;
    insco_primitive = [];
    currentExpression = createExpression("available");
    growingExpression = currentExpression;
    mapSubScores( currentExpression );
    separator();
}

function createTestExpression() {
    isBranch = 0;
    insco_primitive = [];
    currentExpression = createExpression("tests");
    growingExpression = currentExpression;
    mapSubScores( currentExpression );
    separator();
}

function writeSco() {
    writeScore( currentExpression );
    separator();
}

function rewriteSco() {
    rewriteScore( currentExpression );
    separator();
}

function separator() {
    if ( verboseOutput == 1 ) {
        post("--------------------\n");
    }
}

////////////////////////// functions to manipulate, store and recall current expressions

// receives direct text input from interface genotype window
function text(externalExp) {
```

```
    //currentExpression,
    growingExpression = compressExpr(externalExp);
    updateCurrentExpression();
    seed = initSeed;
    var generatScore = eval(currentExpression);
    if ( verboseOutput == 1 ) {
        post("activating external expression: " + currentExpression + "\n");
        post("generatedScore = " + generatScore + "\n");
        post("score length = " + generatScore.length  + "\n");
    }
    outlet(0, "clear");
    outlet(2, generatScore);
    // send score to text object in Max interface
    for(i=0; i< generatScore.length; i++) {
        outlet(0, generatScore[i]);
        outlet(0, "cr");
    }
    printExpression( currentExpression );
}

// saves the current expression
function saveExpression() {
    savedExpressions[savedExpressions.length] = currentExpression;
    post("total saved expressions: " + savedExpressions.length + "\n");
}

// print all saved expressions
function listExpressions() {
    for (i=0; i<savedExpressions.length; i++)
    {
        post("expression " + i + ": " + savedExpressions[i] + "\n");
    }
}

// loads a saved expression and activates ir
function loadExpression(expression_number) {
    currentExpression = savedExpressions[expression_number];
    printExpression( currentExpression );
}

function clearExpressionsList() {
    savedExpressions = [];
    post("total saved expressions: " + savedExpressions.length + "\n");
}

////////////// functions to evolve an expression

// functions to repeat the current expression
function repeatCurrentExpression() {
    currentExpression = reindexInternalAddresses(currentExpression, "0,");
    growingExpression = "scoRep(" + currentExpression + ", " + "intPrim())";
    updateCurrentExpression();
}

// uses the approved growingExpression as the new updated currentExpression
function updateCurrentExpression() {
    currentExpression = growingExpression;
    printExpression( currentExpression );
    mapSubScores( currentExpression );
    outlet(3, generatedScoreLength);
}
```

```
// converts a expression to a score, to use like data into another expression
function currentScore2expression() {
    seed = initSeed;
    var convertedScore = [];
    convertedScore = "scoRender([" + (eval(growingExpression)).join(", ") + "])";
    growingExpression = convertedScore;
    updateCurrentExpression();
}

// compares 2 arrays (auxiliary function for subExpr), returns 1 (0) if both arrays are
equal (diff.),
function compareArrays(addr1, addr2) {
    var equality = true;
    var positionEvalued = 0;
    do {
        if ( addr1[positionEvalued] != addr2[positionEvalued] ) {
            equality = false;
        }
        positionEvalued++;
    } while ( positionEvalued < Math.max(addr1.length, addr2.length) && equality == true
);
    return equality;
}

// extracts a subExpression corresponding to the address position (string tree form) from
expression
function extractSubExpression(expression, address) {
    address = eval(address); // converts address from string to array
    var addressPosition = [0]; // momentan address in expression
    var level = 0; // matrix position (level of the branch)
    var matchAddresses = false; // boolean to compare addresses
    // looks for the position of the address
    for ( charIndx = 0; charIndx < expression.length && compareArrays(address, addressPosition)
== false; charIndx++) {
        if (expression.charAt(charIndx) == "(" && expression.charAt(charIndx+1) != ")"
) {
            level++;
            if ( addressPosition[level] == undefined ) { addressPosition[level] = 0 }
            else {
                addressPosition[level] = addressPosition[level] + 1;
            }
        }
        if (expression.charAt(charIndx) == "," ) {
            addressPosition[level] = addressPosition[level] + 1;
        }
        if (expression.charAt(charIndx) == ")" && expression.charAt(charIndx-1) != "("
) {
            addressPosition.pop();
            level--;
        }
    }
    // copies and returns the selected substring
    if ( charIndx == expression.length ) { post("error: substring not found") }
    else {
        var subExpr = "";
        level = 0;
        copyChar = charIndx;
        do {
            subExpr = subExpr + expression.charAt(copyChar);
            if ( expression.charAt(copyChar) == "(" ) { level++ };
            copyChar++;
        } while ( level == 0 && copyChar <= expression.length );
```

```
        do {
            subExpr = subExpr + expression.charAt(copyChar);
            if ( expression.charAt(copyChar) == "(" ) { level++ } ;
            if ( expression.charAt(copyChar) == ")" ) { level-- } ;
            copyChar++;
        } while ( level != 0 && copyChar <= expression.length );
        return subExpr;
    }
}

// returns a map of the subScores from a given expression,
// and stores it in an array used like variables for certain functions
function mapSubScores(expression) {
    insco_primitive = [];
    var addressPosit = [0]; // momentan address in expression
    var levl = 0; // matrix position (level of the branch)
    var numSubSco = 0;
    // look for the position of the address
    for ( charIndx = 0; charIndx < expression.length; charIndx++) {
        if (expression.charAt(charIndx) == "(" && expression.charAt(charIndx+1) != ")"
) {
            levl++;
            if ( addressPosit[levl] == undefined ) { addressPosit[levl] = 0 }
            else {
                addressPosit[levl] = addressPosit[levl] + 1;
            }
        }
        if (expression.charAt(charIndx) == "," ) {
            addressPosit[levl] = addressPosit[levl] + 1;
        }
        if (expression.charAt(charIndx) == ")" && expression.charAt(charIndx-1) != "("
) {
            addressPosit.pop();
            levl--;
        }
        if (expression.substr(charIndx,3) == "sco") {
            insco_primitive[numSubSco] = "[" + addressPosit.toString() + "]";
            numSubSco++;
        }
    }
    if ( verboseOutput == 1 ) {
        post("internal scores addresses: " + insco_primitive + "\n");
    }
}

function createRecursiveFunction( expressionToConvert, numRecursiveVariables ) {
    var regExpress, replaceExpr, randVal;
    for (priFunctNum = 0; priFunctNum < reducible_score_functions.length; priFunctNum++)
{
            functionToReduce = reducible_score_functions[priFunctNum];
            regExpress = "/" + functionToReduce + "\\(\\)/g";
            putKlammers = "[" + functionToReduce + "()]";
            expressionToConvert = expressionToConvert.replace( eval(regExpress), putKlammers
);
    }
    for (priFunctNum2 = 0; priFunctNum2 < all_reducible_functions.length; priFunctNum2++)
{
        do {
            functionToReduce = all_reducible_functions[priFunctNum2];
            regExpress = "/" + functionToReduce + "\\(\\)/";
            // substitutions types (using last intervals, last notes, or constants)
            var randVal = 0;//randomSeed();
```

```
            if ( randVal <= randomSeed() ) {
                replaceExpr = "intrv(s, " + rInt(1, numRecursiveVariables) + ")";
            }
            else {
                replaceExpr = eval(functionToReduce);
            }
            expressionToConvert = expressionToConvert.replace( eval(regExpress), replaceExpr);
        } while ( expressionToConvert.search(functionToReduce) != -1 );
    }
    return expressionToConvert;
}

function createRecursio(intervalsIntoAccount) {
    isBranch = 1;
    //var retainMaxNotes = maxScoreSize;
    //maxScoreSize = 50000;
    var recursiveExpression = createRecursiveFunction( createExpression("value"), intervalsIntoAccount);
    post(recursiveExpression + "\n");
    //maxScoreSize = retainMaxNotes;
    return recursiveExpression;
}

//////// encoding / decoding expressions

function encodeExpr (expr) {
    // first compress expr
    expr = compressExpr(expr);
    var encodedExpr = [];
    var asciiVal, subStrRead, arrayRead;
    for( charIndex = 0; charIndex < expr.length; charIndex++ ) {
        // if found a function, writes "1" followed by the index function name
        // first, function name is read, from the first character to "(", or a non name
character ir found
        asciiVal = expr.charCodeAt(charIndex);
        if ( (asciiVal > 64 && asciiVal < 91) || (asciiVal > 96 && asciiVal < 123) ) {
            subStrRead = "";
            while ( (asciiVal > 64 && asciiVal < 91) || (asciiVal > 94 && asciiVal < 123)
|| (asciiVal > 47 && asciiVal < 58) ) {
                subStrRead = subStrRead + expr[charIndex];
                charIndex++;
                asciiVal = expr.charCodeAt(charIndex);
            }
            charIndex--;
            //charIndex--;
            // if function name is indexed, flag is 1 followed by the function index
            // else, flag is -1, followed by the length of this substring, and the substring
itself coded as ascii numbers
            if ( (function2index[subStrRead]) == undefined ) {
                encodedExpr.push(-1); // 1 is the flag for a non indexed string
                encodedExpr.push(subStrRead.length); // 1 is the flag for a non indexed
string
                // 1 is the flag for a non indexed string
                for ( strch = 0; strch < subStrRead.length; strch++) {
                    encodedExpr.push(subStrRead.charCodeAt(strch));
                }
            }
            else {
                encodedExpr.push(1); // 1 is the flag for a function name
                encodedExpr.push(function2index[subStrRead]);
            }
        }
        // if "(" is found, adds "5"
```

```
        else if ( asciiVal == 40 ) {
            encodedExpr.push(5); // 0 is the flag for a left parenth
        }
        // if ")" is found, adds "0"
        else if ( asciiVal == 41 ) {
            encodedExpr.push(0); // 0 is the flag for a right parenth
        }
        // if a single comma is found (not as part of an array), adds flag "2"
        else if ( asciiVal == 44 ) {
            encodedExpr.push(2);
        }
        // if a single number is found, is preceded by flag 3
        else if ( (asciiVal > 44 && asciiVal < 47) || (asciiVal > 47 && asciiVal < 58)
) {
            subStrRead = "";
            // read number as string
            while ( (asciiVal > 44 && asciiVal < 47) || (asciiVal > 47 && asciiVal < 58)
) {
                asciiVal = expr.charCodeAt(charIndex);
                subStrRead = subStrRead + expr[charIndex];
                charIndex++;
            }
            // convert substring into actual number
            encodedExpr.push(3);
            encodedExpr.push(parseFloat(subStrRead));
            charIndex--;
            charIndex--;
        }
        // if "[" is found, adds the flag 4, the lenght of the array, and parses the expression
as actual numbers into de encoded array, until "]" is found
        // example: "[3, 45.2, 0, 1.03]" would be encoded as 3, 4, 3, 45.2, 0, 1.03
        else if ( asciiVal == 91 ) {
            encodedExpr.push(4); // 4 is the flag for "["
            charIndex++;
            arrayRead = [];
            do {
                subStrRead = "";
                // read numbers as strings
                do {
                    asciiVal = expr.charCodeAt(charIndex);
                    if ( (asciiVal > 44 && asciiVal < 47) || (asciiVal > 47 && asciiVal
< 58) ) {
                        subStrRead = subStrRead + expr[charIndex];
                    }
                    charIndex++;
                } while ( expr[charIndex] != "," && expr[charIndex] != "]" );
                // convert substring into actual number
                arrayRead.push(parseFloat(subStrRead));
            } while ( expr[charIndex] != "]" );
            encodedExpr.push(arrayRead.length); // arrayRead.length tells how long is
the numeric array
            encodedExpr = encodedExpr.concat(arrayRead);
        }
    }
    return encodedExpr;
}
addFunctionsTypeItem("encodeExpr", "score");

function decodeExpr (score) {
    var arrayLength;
    var decodedExpr = "";
    for ( scoElem = 0; scoElem < score.length; scoElem++ ) {
```

```
        switch (score[scoElem]) {
            // right parenth
            case 0:
                decodedExpr = decodedExpr + ")";
                break;
            // function name
            case 1:
                scoElem++;
                decodedExpr = decodedExpr + index2function[score[scoElem]];
                break;
            // comma
            case 2:
                decodedExpr = decodedExpr + ", ";
                break;
            // single value
            case 3:
                scoElem++;
                decodedExpr = decodedExpr + score[scoElem];
                break;
            // numeric array
            case 4:
                scoElem++;
                decodedExpr = decodedExpr + "[";
                arrayLength = score[scoElem];
                scoElem++;
                for( arrEl=0; arrEl < arrayLength - 1; arrEl++ ) {
                    decodedExpr = decodedExpr + score[scoElem] + ", ";
                    scoElem++;
                }
                decodedExpr = decodedExpr + score[scoElem] + "]";
                break;
            // non indexed substring
            case 5:
            // left parenth
                decodedExpr = decodedExpr + "(";
                break;
            case -1:
                scoElem++;
                arrayLength = score[scoElem];
                for( el=0; el < arrayLength; el++ ) {
                    scoElem++;
                    decodedExpr = decodedExpr + String.fromCharCode(score[scoElem])
                }
                break;
            // bad expression
            default:
                post("error: probably bad encoded expression" + "\n");
                goodScore = 0;
                return [-1];
        }
    }
    return decodedExpr;
}
addFunctionsTypeItem("decodeExpr", "expre");

// EOF
```

A.2 Librería de funciones genomus_library_00.js

```
//////////////////// library of functions for creation of genotypes

////////// array of all functions with its associated output. At the same time create
the links functionName->number, to expression encoding

// link a function name with a number in both directions
var function2index = [];
var index2function = [];
var functionsType = [];
function addFunctionsTypeItem(functionName, outputType) {
    functionsType[functionName] = outputType;
    function2index[functionName] = index2function.length;
    index2function[index2function.length] = functionName;
}

var intervalMatrixGlobal = [3,6,4,7,8,4,1,4,7,4,2,1]; // global vertical intervals matrix
function setIntervalMatrixGlobal (score) {
    intervalMatrixGlobal = score;
}
addFunctionsTypeItem("intervalMatrixGlobal", "score");

function initIntervalMatrixGlobal (score) {
    intervalMatrixGlobal = matrixUnidim(12, 1, 12);
    post("intervalMatrixGlobal = " + intervalMatrixGlobal + "\n");
}

// index primitive functions of main script
addFunctionsTypeItem("rInt", "integ");
addFunctionsTypeItem("rFlo", "float");
addFunctionsTypeItem("intPrim", "integ");
addFunctionsTypeItem("floPrim", "float");

// arithmetic functions

function sqr(value) {
    return value*value;
}
addFunctionsTypeItem("sqr", "float"); // index this function with its output type

function sum(value_1, value_2) {
    return value_1 + value_2;
}
addFunctionsTypeItem("sum", "float");

function dif(value_1, value_2) {
    return value_1 - value_2;
}
addFunctionsTypeItem("dif", "float");

function absDif(value_1, value_2) {
    return Math.abs(value_1 - value_2);
}
addFunctionsTypeItem("absDif", "float");

function prod(value_1, value_2) {
    return value_1 * value_2;
}
addFunctionsTypeItem("prod", "float");
```

```
function ratio(value_1, value_2) {
    if ( value_2 == 0 ) {
        goodScore = 0;
        return [-1];
    }
    return value_1 / value_2;
}
addFunctionsTypeItem("ratio", "float");

function mod(value_1, value_2) {
    return value_1 % value_2;
}
addFunctionsTypeItem("mod", "float");

function invers(value) {
    if ( value == 0 ) {
        goodScore = 0;
        return [-1];
    }
    return 1 / value;
}
addFunctionsTypeItem("invers", "float");

// returns value_1th root of value_2
function root(value_1, value_2) {
    result = Math.pow(Math.abs(value_2), 1/value_1);
    if (Math.abs(result) > 1000000000 ) {
        post("too big number");
        return [-1];
    }
    return result;
}
addFunctionsTypeItem("root", "float");

function pyth(value_1, value_2) {
    return root(sum(sqr(value_1),sqr(value_2)),2);
}
addFunctionsTypeItem("pyth", "float");

function log(value_1, value_2) {
    if ( value_1 < 0 || value_2 <= 0 || value_2 == 1 ) {
        goodScore = 0;
        return [-1];
    }
    return Math.log(value_1) / Math.log(value_2);
}
addFunctionsTypeItem("log", "float");

function sin(value) {
    return Math.sin(value);
}
addFunctionsTypeItem("sin", "float");

function scoSin (score, value) {
    for (n=0; n<score.length; n++) {
        score[n] = Math.abs(Math.sin(score[n]) * value);
    }
    return score;
}
addFunctionsTypeItem("scoSin", "score");

function cos(value) {
```

```
    return Math.cos(value);
}
addFunctionsTypeItem("cos", "float");

function scoCos (score, value) {
    for (n=0; n<score.length; n++) {
        score[n] = Math.abs(Math.cos(score[n]) * value);
    }
    return score;
}
addFunctionsTypeItem("scoCos", "score");

function asin(value) {
    return Math.asin(value % 1);
}
addFunctionsTypeItem("asin", "float");

function acos(value) {
    return Math.acos(value % 1);
}
addFunctionsTypeItem("acos", "float");

function tan(value) {
    return Math.tan(value % 1.57079); // avoids too long results
}
addFunctionsTypeItem("tan", "float");

function atan(value) {
    return Math.atan(value);
}
addFunctionsTypeItem("atan", "float");

function abs(value) {
    return Math.abs(value);
}
addFunctionsTypeItem("abs", "float");

// returns a random float from the interval [value_1,value_2)
function randFlo(value_1, value_2) {
    return randomSeed() * (value_2 - value_1) + value_1;
}
addFunctionsTypeItem("randFlo", "float");

// returns a random int from the interval [value_1,value_2] (both included)
function randInt(value_1, value_2) {
    return Math.round(randomSeed() * (Math.round(value_2) - Math.round(value_1)) + Math.round(value_1));
}
addFunctionsTypeItem("randInt", "integ");

////////////// functions with musical matrix
/// musical matrix (or score) is normalized according to a score type previously defined

// returns score % value
function scoMod(score, value) {
    if ( value == 0 ) {
        goodScore = 0;
        return [-1];
    }
    for (a=0; a<score.length; a++) {
        score[a] = score[a] % value;
    }
    return score;
```

```
}
addFunctionsTypeItem("scoMod", "score");

// transposes a score value interval
function scoTransp(score, value) {
    for (i=0; i<score.length; i++) {
        score[i] = score[i] + value;
    }
    return score;
}
addFunctionsTypeItem("scoTransp", "score");

// composes a score transposing score_1 to each score_2 value
function scoTranspMatrix(score_1, score_2) {
    if ( score_1.length * score_2.length > maxScoreSize ) {
        goodScore = 0;
        return [-1];
    }
    var composedScore = [];
    for ( a=0; a<score_2.length; a++ ) {
        composedScore = composedScore.concat(scoTransp(score_1, score_2[a]));
    }
    return composedScore;
}
addFunctionsTypeItem("scoTranspMatrix", "score");

// repeats a musical matrix integ times
function scoRep(score, integ) {
    if (score.length * integ <= maxScoreSize) {
        var tempScore = [];
        for (i=0; i<integ; i++) {
            tempScore = tempScore.concat(score);
        }
        return tempScore;
    }
    else {
        goodScore = 0;
        return [-1];
    }
}
addFunctionsTypeItem("scoRep", "score");

// concatenates two musical matrix
function scoConcat(score_1, score_2) {
    if (score_1.length + score_2.length <= maxScoreSize) {
        return score_1.concat(score_2);
    }
    else {
        goodScore = 0;
        return [-1];
    }
}
addFunctionsTypeItem("scoConcat", "score");

// returns retrogradation of score
function scoRetrog(score) {
    return score.reverse();
}
addFunctionsTypeItem("scoRetrog", "score");

// returns invertion of score
function scoInvert(score) {
```

```
    var scoreCopy = score.slice(0); // copies array without loops
    for( ele=1; ele<score.length; ele++ ) {
        score[ele] = score[ele-1] - (scoreCopy[ele] - scoreCopy[ele-1]);
    }
    return score;
}
addFunctionsTypeItem("scoInvert", "score");

// return score expanded according to factor value_1, using first as axis
function scoExpand(score, value_1) {
    var expandedScore = score.slice(0);
    for( ele=1; ele<score.length; ele++ ) {
        expandedScore[ele] = score[0] + (expandedScore[ele] - score[0]) * value_1;
    }
    return expandedScore;
}
addFunctionsTypeItem("scoExpand", "score");

// iterated version of scoExpandIter, expanding score value_3 times, from factor value_1
to value_2
function scoExpandIter(score, value_1, value_2, value_3) {
    if ( score.length * Math.abs(value_3) > maxScoreSize ) {
        goodScore = 0;
        return [-1];
    }
    var expansionIndex = scoSteps(value_1, value_2, value_3);
    var expandedScore = [];
    for ( a=0; a<Math.round(Math.abs(value_3)); a++ ) {
        expandedScore = expandedScore.concat(scoExpand(score, expansionIndex[a]));
    }
    return expandedScore;
}
addFunctionsTypeItem("scoExpandIter", "score");

// returns a new score extract with value_2 notes starting in value_1 note. If value_2
is negative, excerpt is returned retrogradated
function scoExcerpt(score, value_1, value_2) {
    value_1 = Math.round(Math.abs(value_1)) % score.length;
    if ( value_2 == 0 ) { // if value_2 is 0, excerpt have only 1 value
        value_2 = 1;
    }
    var step = 1;
    if ( value_2 < 0 ) { step = -1 };
    var excerpt = [];
    var newEle = 0;
    for ( ele = value_1; ele != value_1 + value_2 && ele < score.length && ele >= 0; ele
= ele + step ) {
        excerpt[newEle] = score[ele];
        newEle++;
    }
    return excerpt;
}
addFunctionsTypeItem("scoExcerpt", "score");

// returns a series of excerpts from score_1, from initial note to score_2 notes
function scoExcerptMulti(score_1, score_2) {
    if ( absSumArray(score_2) > maxScoreSize ) {
        goodScore = 0;
        return [-1];
    }
    var composedScore = [];
    var sco1length = score_1.length;
```

```
    for(it=0; it<score_2.length; it++) {
        composedScore = composedScore.concat(scoExcerpt(score_1, 0, score_2[it] % sco1length));
    }
    return composedScore;
}
addFunctionsTypeItem("scoExcerptMulti", "score");

// returns a series of excerpts from score_1, from score_2 notes to score_3 notes
function scoExcerptMulti2(score_1, score_2, score_3) {
    if ( absSumArray(score_2) > maxScoreSize || absSumArray(score_3) > maxScoreSize )
{
        goodScore = 0;
        return [-1];
    }
    var composedScore = [];
    var sco1length = score_1.length;
    for(it=0; it<Math.min(score_2.length, score_3.length); it++) {
        composedScore = composedScore.concat(scoExcerpt(score_1, score_2[it] % sco1length,
score_3[it] % sco1length));
    }
    return composedScore;
}
addFunctionsTypeItem("scoExcerptMulti2", "score");

// returns a series of excerpts from score_1, from score_2 notes to score_3 notes, transposed
score_4 intervals
function scoExcerptMultiTransp(score_1, score_2, score_3, score_4) {
    if ( absSumArray(score_2) > maxScoreSize || absSumArray(score_3) > maxScoreSize )
{
        goodScore = 0;
        return [-1];
    }
    var composedScore = [];
    var sco1length = score_1.length;
    for(it=0; it<Math.min(score_2.length, score_3.length, score_4.length); it++) {
        composedScore = composedScore.concat(scoTransp(scoExcerpt(score_1, score_2[it]
% sco1length, score_3[it] % sco1length),score_4[it]));
    }
    return composedScore;
}
addFunctionsTypeItem("scoExcerptMultiTransp", "score");

// returns a new score with the same notes in a new random order
function scoPermut(score) {
    var scoreCopy = score.slice(0);
    var permutScore = [];
    var selectedElem;
    do {
        selectedElem = parseInt( scoreCopy.length * randomSeed() );
        permutScore.push( scoreCopy[selectedElem] );
        scoreCopy.splice(selectedElem, 1);
    } while ( scoreCopy.length > 0 );
    return permutScore;
}
addFunctionsTypeItem("scoPermut", "score");

// iterated version of scoPermut: return a new score with the same notes in a new random
order, value_1 times
function scoPermutIter(score, value_1) {
    if ( score.length * Math.abs(value_1) > maxScoreSize ) {
        goodScore = 0;
        return [-1];
```

```
    }
    var scoreCopy;
    var permutScore = [];
    var selectedElem;
    for ( a=0; a<Math.round(Math.abs(value_1)); a++ ) {
        scoreCopy = score.slice(0);
        do {
            selectedElem = parseInt( scoreCopy.length * randomSeed() );
            permutScore.push( scoreCopy[selectedElem] );
            scoreCopy.splice(selectedElem, 1);
        } while ( scoreCopy.length > 0 );
    }
    return permutScore;
}
addFunctionsTypeItem("scoPermutIter", "score");

// returns array from value_1 to value_2 in value_3 steps
function scoSteps(value_1, value_2, value_3) {
    value_3 = Math.round(Math.abs(value_3));
    if ( value_3 < 2 && value_3 > -2 ) {
        goodScore = 0;
        return [-1];
    }
    var stepsArray = [];
    var step = (value_2 - value_1) / (value_3 - 1);
    for ( a=0; a<value_3; a++ ) {
        stepsArray[a] = step * a + value_1;
    }
    return stepsArray;
}
addFunctionsTypeItem("scoSteps", "score");

// creates a random integer score, with integ_1 items, between boundaries integ_2 and
integ_3)
function scoRand(integ_1, integ_2, integ_3) {
    if (integ_1 > maxScoreSize) {
        goodScore = 0;
        return [-1];
    }
    var tempScore = [];
    for (i=0; i<integ_1; i++) {
        tempScore[i] = rInt(integ_2, integ_3);
    }
    return tempScore;
}
addFunctionsTypeItem("scoRand", "score");

// creates a random array (without limitations os scoreSize), with integ_1 items, between
boundaries integ_2 and integ_3)
function matrixUnidim(integ_1, integ_2, integ_3) {
    var tempScore = [];
    for (i=0; i<integ_1; i++) {
        tempScore[i] = rInt(integ_2, integ_3);
    }
    return tempScore;
}
addFunctionsTypeItem("matrixUnidim", "score");

// creates a random float score, with integ_1 items, between boundaries float_1 and float_2)
function scoRandFlo(integ_1, float_1, float_2) {
    if (integ_1 > maxScoreSize) {
        goodScore = 0;
```

```
        return [-1];
    }
    var tempScore = [];
    for (i=0; i<integ_1; i++) {
        tempScore[i] = rFlo(float_1, float_2);
    }
    return tempScore;
}
addFunctionsTypeItem("scoRandFlo", "score");

// creates a primitive score from scratch
function scoRandPrim() {
    return eval(scoRand(rInt(3, 15), 21, 108));
}
addFunctionsTypeItem("scoRandPrim", "score");

// returns an array of integers from integ_1 to integ_2
function scoEnumInt(integ_1, integ_2) {
    if (Math.abs(integ_1 - integ_2) > maxScoreSize) {
         goodScore = 0;
        return [-1];
    }
    var tempArray = [];
    if (integ_1>integ_2) { var step = -1 }
    else { var step = 1 }
    for (i=integ_1; i!=integ_2+step; i=i+step) {
        tempArray[tempArray.length] = i;
    }
    return tempArray;
}
addFunctionsTypeItem("scoEnumInt", "score");

// returns the score with some differences according to float_1 (0=no mutation, 0.999=max
mutation), and value_2 (range of mutation)
function scoMutate(score, float_1, value_2) {
    float_1 = float_1 % 1;
    for(i=0; i<score.length; i++) {
        if (randomSeed() < float_1) {
            score[i] = score[i] + rFlo(value_2*(-1), value_2);
        }
    }
    return score;
}
addFunctionsTypeItem("scoMutate", "score");

function scoRender(score) {
    return score;
}
addFunctionsTypeItem("scoRender", "score");

// generates a random pitch class set
function scoPCGen(integ) {
    // number of elements of PCS from 1 to chromaticQuantiz
    var numberPitchClasses = (Math.abs(integ) - 1) % chromaticQuantiz + 1;
    var pitchClassSet = [];
    var newElem;
    do {
        newElem = rInt(0,chromaticQuantiz-1);
        if ( pitchClassSet.indexOf(newElem) == -1 ) {
            pitchClassSet.push(newElem);
        }
    } while ( pitchClassSet.length < numberPitchClasses );
```

```
    return pitchClassSet.sort(function(a,b){return a-b});
}
addFunctionsTypeItem("scoPCGen", "score");

function scoPCGenPrim() {
    return scoPCGen(rInt(1,11));
}
addFunctionsTypeItem("scoPCGenPrim", "score");

// adjusts the notes of a score to a pitch class set
function scoPitchClass(score, pcset) {
    //var pcset = pcsFromScore(score_2);
    var octaveNotes;
    var tempScore = [];
    // this loop look for the more accurate conversion to a note from the pcset
    for (b=0; b<score.length; b++) {
        var c = 0;
        do {
            c++;
            octaveNotes = Math.floor(score[b]/chromaticQuantiz)*chromaticQuantiz;
            tempScore[b] = pcset[c-1] + octaveNotes;
        } while ( ( Math.abs(score[b] - (octaveNotes + pcset[c-1]) ) > Math.abs(score[b]
- (octaveNotes + pcset[c]) ) ) && c < pcset.length )
    }
    return tempScore;
}
addFunctionsTypeItem("scoPitchClass", "score");

/////// ADA+BABBAGE-Capricci functions

function scoNameBABBAGE () {
    return [58,57,58,58,57,55,52];
}
addFunctionsTypeItem("scoNameBABBAGE", "score");

function scoNameBABBAGEintervals () {
    return [-1,1,0,-1,-2,-3];
}
addFunctionsTypeItem("scoNameBABBAGEintervals", "score");

function scoNameADA () {
    return [57,50,57];
}
addFunctionsTypeItem("scoNameADA", "score");

function scoNameADAintervals () {
    return [-7,7];
}
addFunctionsTypeItem("scoNameADAintervals", "score");

// returns score with value new notes, calculated according to a 2-dim. matrix of the
last two intervals
function scoADA(score, value) {
    var newScore = score.slice(0);
    // // 7x7 table to calculate new interval (interval from -3 to 3)
    var biggestDescendingInterval = -7;
    var biggestAscendingInterval = 7;
    var rangeOfIntervalsInMatrix = 15;
    var lastInterval, prelastInterval;
    var nextIntervalMatrix = matrixUnidim(225, biggestDescendingInterval, biggestAscendingInterval);
// table to calculate new interval
    var lastInterval =  Math.round((newScore[newScore.length - 1] - newScore[newScore.length
```

```
- 2]) % 7);
    var prelastInterval = Math.round((newScore[newScore.length - 2] - newScore[newScore.length
- 3]) % 7);
    for ( newEl = 0; newEl < Math.abs(value); newEl++ ) {
        lastInterval = Math.round((newScore[newScore.length - 1] - newScore[newScore.length
- 2]) % 7);
        prelastInterval = Math.round((newScore[newScore.length - 2] - newScore[newScore.length
- 3]) % 7);
        newInterval = nextIntervalMatrix[(prelastInterval+7)*15 + lastInterval+7];
        if ( newInterval == undefined ) { newInterval = 0 };
        newScore.push( newScore[newScore.length - 1] + newInterval );
    }
    return newScore;
}
addFunctionsTypeItem("scoADA", "score");

// returns a second voice, with another matrix with vertical intervals
function scoADAnewVoice(score) {
    var newInterval;
    var lowScore = score.slice(0);
    var newVoiceScore = [];
    newVoiceScore.push(lowScore[0], lowScore[1])
    // 7x7 table to calculate new interval (interval from -3 to 3)
    var biggestDescendingInterval = 0;
    var biggestAscendingInterval = 14;
    var rangeOfIntervalsInMatrix = 15;
    var lastInterval, prelastInterval;
    var nextIntervalMatrix = matrixUnidim(225, biggestDescendingInterval, biggestAscendingInterval);
// table to calculate new interval
    post("MATRIX = " + nextIntervalMatrix + "\n");
    var lastInterval = Math.round((lowScore[lowScore.length - 1] - lowScore[lowScore.length
- 2]) % 7);
    var prelastInterval = Math.round((lowScore[lowScore.length - 2] - lowScore[lowScore.length
- 3]) % 7);
    for ( newEl = 2; newEl < score.length; newEl++ ) {
        lastInterval = Math.round((lowScore[newEl] - lowScore[newEl - 1]) % 7);
        prelastInterval = Math.round((lowScore[newEl - 1] - lowScore[newEl - 2]) % 7);
        newInterval = nextIntervalMatrix[(prelastInterval+7)*15 + lastInterval+7];
        if ( newInterval == undefined ) { newInterval = 0 };
        newVoiceScore.push(lowScore[newEl] + newInterval);
    }
    return newVoiceScore;
}
addFunctionsTypeItem("scoADAnewVoice", "score");

// returns a second voice, with another matrix with vertical intervals
function scoBABBAGEnewVoice(score) {
    var newInterval;
    var lowScore = score.slice(0);
    var newVoiceScore = [];
    newVoiceScore.push(lowScore[0], lowScore[1])
    // 7x7 table to calculate new interval (interval from -3 to 3)
    var biggestDescendingInterval = 0;
    var biggestAscendingInterval = 20;
    var rangeOfIntervalsInMatrix = 21;
    var lastInterval, prelastInterval;
    var nextIntervalMatrix = matrixUnidim(441, biggestDescendingInterval, biggestAscendingInterval);
// table to calculate new interval
    post("MATRIX = " + nextIntervalMatrix + "\n");
    var lastInterval = Math.round((lowScore[lowScore.length - 1] - lowScore[lowScore.length
- 2]) % 12);
    var prelastInterval = Math.round((lowScore[lowScore.length - 2] - lowScore[lowScore.length
```

```
- 3]) % 12);
    for ( newEl = 2; newEl < score.length; newEl++ ) {
        lastInterval = Math.round((lowScore[newEl] - lowScore[newEl - 1]) % 12);
        prelastInterval = Math.round((lowScore[newEl - 1] - lowScore[newEl - 2]) % 12);
        newInterval = nextIntervalMatrix[(prelastInterval+10)*21 + lastInterval+10];
        if ( newInterval == undefined ) { newInterval = 7 };
        newVoiceScore.push(lowScore[newEl] + newInterval);
    }
    return newVoiceScore;
}
addFunctionsTypeItem("scoBABBAGEnewVoice", "score");

// scans 2 scores, and return a new 3rd voice, considering only vertical intervals between
2 initial voices to determine third voice
function scoBABBAGEharmonicVoice(score_1, score_2) {
    var newScore = [];
    var biggestDescendingInterval = -15;
    var biggestAscendingInterval = 15;
    var nextIntervalMatrix = matrixUnidim(12, biggestDescendingInterval, biggestAscendingInterval);
// table to calculate new interval
    post("MATRIX = " + nextIntervalMatrix + "\n");
    for (d=0; d<Math.min(score_1.length, score_2.length); d++) {
        newScore[d] = Math.max(score_1[d], score_2[d]) + nextIntervalMatrix[Math.round(Math.abs(score_1[d]
- score_2[d])) % 12];
    }
    return newScore;
}
addFunctionsTypeItem("scoBABBAGEharmonicVoice", "score");

// identical as precedent function, but using a global vertical intervals array
function scoBABBAGEharmonicVoiceGlobal(score_1, score_2) {
    var newScore = [];
    post("MATRIX = " + intervalMatrixGlobal + "\n");
    for (d=0; d<Math.min(score_1.length, score_2.length); d++) {
        newScore[d] = Math.max(score_1[d], score_2[d]) + intervalMatrixGlobal[Math.round(Math.abs(score_1[d]
- score_2[d])) % 12];
    }
    return newScore;
}
addFunctionsTypeItem("scoBABBAGEharmonicVoiceGlobal", "score");

// returns a score alternating notes from score_1 y score_2 (both of identical length)
function scoJoin2Scores(score_1, score_2) {
    var joinScoreLength = Math.min(score_1.length, score_2.length);
    if (joinScoreLength > maxScoreSize) {
        goodScore = 0;
        return [-1];
    }
    var composedScore = [];
    for (i=0; i<joinScoreLength; i++) {
        composedScore.push(score_1[i], score_2[i]);
    }
    return composedScore;
}
addFunctionsTypeItem("scoJoin2Scores", "score");

// returns a score alternating notes from score_1, score_2 and score_3 (all of identical
length)
function scoJoin3Scores(score_1, score_2, score_3) {
    var joinScoreLength = Math.min(score_1.length, score_2.length, score_3.length);
    if (joinScoreLength > maxScoreSize) {
        goodScore = 0;
```

```
        return [-1];
    }
    var composedScore = [];
    for (i=0; i<joinScoreLength; i++) {
        composedScore.push(score_1[i], score_2[i], score_3[i]);
    }
    return composedScore;
}
addFunctionsTypeItem("scoJoin3Scores", "score");

// returns a score alternating notes from score_1, score_2 and score_3 (all of identical
length)
function scoJoin4Scores(score_1, score_2, score_3, score_4) {
    var joinScoreLength = Math.min(score_1.length, score_2.length, score_3.length, score_4.length);
    if (joinScoreLength > maxScoreSize) {
        goodScore = 0;
        return [-1];
    }
    var composedScore = [];
    for (i=0; i<joinScoreLength; i++) {
        composedScore.push(score_1[i], score_2[i], score_3[i], score_4[i]);
    }
    return composedScore;
}
addFunctionsTypeItem("scoJoin4Scores", "score");

// returns a score alternating notes from score_1, score_2 and score_3 (all of identical
length)
function scoJoin5Scores(score_1, score_2, score_3, score_4, score_5) {
    var joinScoreLength = Math.min(score_1.length, score_2.length, score_3.length, score_4.length,
score_5.length);
    if (joinScoreLength > maxScoreSize) {
        goodScore = 0;
        return [-1];
    }
    var composedScore = [];
    for (i=0; i<joinScoreLength; i++) {
        composedScore.push(score_1[i], score_2[i], score_3[i], score_4[i], score_5[i]);
    }
    return composedScore;
}
addFunctionsTypeItem("scoJoin5Scores", "score");

// ADA 2 voices generator
function scoADA2voices(score, value) {
    var lowVoice = scoADA(score, value);
    var upperVoice = scoADAnewVoice(lowVoice);
    return scoJoin2Scores(lowVoice, upperVoice);
}
addFunctionsTypeItem("scoADA2voices", "score");

// ADA 3 voices generator
function scoADA3voices(score, value) {
    var lowVoice = scoADA(score, value);
    var interVoice = scoADAnewVoice(lowVoice);
    var upperVoice = scoADAnewVoice(interVoice);
    return scoJoin3Scores(lowVoice, interVoice, upperVoice);
}
addFunctionsTypeItem("scoADA3voices", "score");

// BABBAGE 2 voices generator
function scoBABBAGE2voices(score, value) {
```

```
    var lowVoice = scoBABBAGE(score, value);
    var upperVoice = scoBABBAGEnewVoice(lowVoice);
    return scoJoin2Scores(lowVoice, upperVoice);
}
addFunctionsTypeItem("scoBABBAGE2voices", "score");

// BABBAGE 3 voices generator
function scoBABBAGE3voices(score, value) {
    var lowVoice = scoBABBAGE(score, value);
    var interVoice = scoBABBAGEnewVoice(lowVoice);
    var upperVoice = scoBABBAGEnewVoice(interVoice);
    return scoJoin3Scores(lowVoice, interVoice, upperVoice);
}
addFunctionsTypeItem("scoBABBAGE3voices", "score");

// BABBAGE 2 voices + harmonic voice generator
function scoBABBAGE3voicesHarmonic(score, value) {
    var lowVoice = scoBABBAGE(score, value);
    var interVoice = scoBABBAGEnewVoice(lowVoice);
    var upperVoice = scoBABBAGEharmonicVoice(lowVoice, interVoice);
    return scoJoin3Scores(lowVoice, interVoice, upperVoice);
}
addFunctionsTypeItem("scoBABBAGE3voicesHarmonic", "score");

// BABBAGE 2 voices + 2 harmonic voices generator
function scoBABBAGE4voicesHarmonic(score, value) {
    var lowVoice = scoBABBAGE(score, value);
    var interVoice = scoBABBAGEnewVoice(lowVoice);
    var upperVoice = scoBABBAGEharmonicVoice(lowVoice, interVoice);
    var upperUpperVoice = scoBABBAGEharmonicVoice(interVoice, upperVoice);
    return scoJoin4Scores(lowVoice, interVoice, upperVoice, upperUpperVoice);
}
addFunctionsTypeItem("scoBABBAGE4voicesHarmonic", "score");

// BABBAGE 2 voices + 3 harmonic voices generator
function scoBABBAGE5voicesHarmonic(score, value) {
    var lowVoice = scoBABBAGE(score, value);
    var interVoice = scoBABBAGEnewVoice(lowVoice);
    var upperVoice = scoBABBAGEharmonicVoice(lowVoice, interVoice);
    var upperUpperVoice = scoBABBAGEharmonicVoice(interVoice, upperVoice);
    var topVoice = scoBABBAGEharmonicVoice(upperVoice, upperUpperVoice);
    return scoJoin5Scores(lowVoice, interVoice, upperVoice, upperUpperVoice, topVoice);
}
addFunctionsTypeItem("scoBABBAGE5voicesHarmonic", "score");

// BABBAGE 2 voices + 3 harmonic voices generator using identical matrix interval (as
global matrix)
function scoBABBAGE5voicesHarmonicGlobal(score, value) {
    var lowVoice = scoBABBAGE(score, value);
    var interVoice = scoBABBAGEnewVoice(lowVoice);
    //intervalMatrixGlobal = matrixUnidim(12, 1, 16);
    var upperVoice = scoBABBAGEharmonicVoiceGlobal(lowVoice, interVoice);
    var upperUpperVoice = scoBABBAGEharmonicVoiceGlobal(interVoice, upperVoice);
    var topVoice = scoBABBAGEharmonicVoiceGlobal(upperVoice, upperUpperVoice);
    return scoJoin5Scores(lowVoice, interVoice, upperVoice, upperUpperVoice, topVoice);
}
addFunctionsTypeItem("scoBABBAGE5voicesHarmonic", "score");

// adds a second voice following a rule based in precedents intervals
function addBABBAGEvoice(score) {
    var newVoice = scoBABBAGEnewVoice(score);
    return scoJoin2Scores(score, newVoice);
```

```
}
addFunctionsTypeItem("addBABBAGEvoice", "score");

// adds a second voice following a rule based in precedents intervals
function addADAvoice(score) {
    var newVoice = scoADAnewVoice(score);
    return scoJoin2Scores(score, newVoice);
}
addFunctionsTypeItem("addADAvoice", "score");

// returns score with value new notes, calculated according to a 2-dim. matrix of the
last two intervals
function scoBABBAGE(score, value) {
    var newScore = score.slice(0);
    var biggestDescendingInterval = -11;
    var biggestAscendingInterval = 11;
    var rangeOfIntervalsInMatrix = 23;
    var lastInterval, prelastInterval;
    var nextIntervalMatrixBABBAGE = matrixUnidim(529, biggestDescendingInterval, biggestAscendingInterval);
// table to calculate new interval
    post("BABBAGE = " + nextIntervalMatrixBABBAGE + "\n");
    // introduces the constants for BABBAGE motif
    nextIntervalMatrixBABBAGE[( -1 +11)*23 + 1 +11] = 0; // BAB -> B
    nextIntervalMatrixBABBAGE[( 1 +11)*23 + 0 +11] = -1; // ABB -> A
    nextIntervalMatrixBABBAGE[( 0 +11)*23 + -1 +11] = -2; // BBA -> G
    nextIntervalMatrixBABBAGE[( -1 +11)*23 + -2 +11] = -3; // BAG -> E

    // introduces the constants for BABBAGE inverted motif
    nextIntervalMatrixBABBAGE[( 1 +11)*23 + -1 +11] = 0; // invert(BAB -> B)
    nextIntervalMatrixBABBAGE[( -1 +11)*23 + 0 +11] = 1; // invert(ABB -> A)
    nextIntervalMatrixBABBAGE[( 0 +11)*23 + 1 +11] = 2; // invert(BBA -> G)
    nextIntervalMatrixBABBAGE[( 1 +11)*23 + 2 +11] = 3; // invert(BAG -> E)

    // ADA loop: DAD -> A
    nextIntervalMatrixBABBAGE[( -7 +11)*23 + 7 +11] = -7; // ADA -> D
    nextIntervalMatrixBABBAGE[( 7 +11)*23 + -7 +11] = 7; // DAD -> A

    var lastInterval = Math.round((newScore[newScore.length - 1] - newScore[newScore.length
- 2]) % 12);
    var prelastInterval = Math.round((newScore[newScore.length - 2] - newScore[newScore.length
- 3]) % 12);
    for ( newEl = 0; newEl < Math.abs(value); newEl++ ) {
        lastInterval = Math.round((newScore[newScore.length - 1] - newScore[newScore.length
- 2]) % 12);
        prelastInterval = Math.round((newScore[newScore.length - 2] - newScore[newScore.length
- 3]) % 12);
        newInterval = nextIntervalMatrixBABBAGE[(prelastInterval+11)*23 + lastInterval+11]
+ mod(newScore[newEl-1], 3);
        if ( isNaN(newInterval) == true ) { newInterval = 0 };
        newScore.push( newScore[newScore.length - 1] + newInterval );
    }
    return newScore;
}
addFunctionsTypeItem("scoBABBAGE", "score");

// returns the value-th last interval from score
function intrv(score, value) {
    if ( value >= score.length ) {
        goodScore = 0;
        return [-1];
    }
    return score[score.length - value] - score[score.length - value - 1];
```

```
}
addFunctionsTypeItem("intrv", "value");

// returns the value-th last note from score
function nthLastNote(score, value) {
    if ( value >= score.length ) {
        goodScore = 0;
        return [-1];
    }
    return score[score.length - value];
}
addFunctionsTypeItem("nthLastNote", "value");

// returns a recursive score with value notes, using recursive encoded expression recur
with score as initial conditions
function scoRecursio(score, recur, value) {
    if ( value + score.length >= maxScoreSize ) {
        goodScore = 0;
        return [-1];
    }
    //post("RECIBI recur = " + recur + "\n");
    var recursio = decodeExpr(recur);
    var recursioPrinted = expandExpr( "encodeExpr(\"" + recursio + "\")" );
    outlet(4, recursioPrinted);
    //post("RECURSIO es " + recursio + "\n");
    var s = score.slice(0);
    //post("SCORE es " + s + "\n");
    for ( newEl = 0; newEl < Math.abs(value); newEl++ ) {
        //var("nuevo elem = " + eval(recursio) + "\n");
        s.push( s[s.length - 1] + eval(recursio) );
    }
    return s;
}
function preScoRecursio(value) {
}
addFunctionsTypeItem("scoRecursio", "score");
addFunctionsTypeItem("preScoRecursio", "score");

// returns a recursive score with value notes, using recursive encoded expression recur
with score as initial conditions
function scoRecursioBABBAGE(recur) {
    post("RECUR recibido es " + recur + "\n");
    var recursio = decodeExpr(recur);
    var recursioPrinted = expandExpr( "encodeExpr(\"" + recursio + "\")" );
    outlet(4, recursioPrinted);
    var s = MIDI2genomusPitch([58,57,58,58,57,54,51]);
    for ( newEl = 0; newEl < 200; newEl++ ) {
        s.push( s[s.length - 1] + eval(recursio) );
    }
    post("NO CONVERT = " + s + "\n");
    post("MOD = " + scoMod(s,1) + "\n");
    post("CONVERT = " + genomusPitch2MIDI(scoMod(s,1)) + "\n");
    return genomusPitch2MIDI(scoMod(s,1));
}
function preScoRecursioBABBAGE(value) {
}
addFunctionsTypeItem("scoRecursioBABBAGE", "score");
addFunctionsTypeItem("preScoRecursioBABBAGE", "score");

//mask functions for sustitute with recursions
function preEncRecursio() {
}
```

```
function preEncRecursioBABBAGE() {
}
addFunctionsTypeItem("preEncRecursio", "prov");
addFunctionsTypeItem("preEncRecursioBABBAGE", "prov");

// reuses a subExpression from the currentExpression
function scoInter(internalExpressionAddress) {
    if ( insco_primitive.length > 0 ) {
        return eval( extractSubExpression( growingExpression, internalExpressionAddress)
);
    }
    else {
        goodScore = 0;
        return [-1];
    }
}
addFunctionsTypeItem("scoInter", "score");

function preScoInter() {
}
addFunctionsTypeItem("preScoInter", "prov");

// evaluates value times expre, used as a literal expression (returns different results
in every eval.)
function scoRepExpr(internalExpressionAddress, numberIter) {
    if ( insco_primitive.length == 0 ) {
        goodScore = 0;
        return [-1];
    }
    expri_2 = (numberIter % 10) + 1;
    expressionEvaluated = extractSubExpression( growingExpression, internalExpressionAddress
);
    if ( eval(expressionEvaluated).length * numberIter > 10000 ) {
        goodScore = 0;
        return [-1];
    }
    var renderedSco = [];
    for(t=0; t < numberIter; t++) {
        renderedSco = renderedSco.concat(eval(expressionEvaluated));
    }
    return renderedSco;
}
function preScoRepExpr() {
}
addFunctionsTypeItem("scoRepExpr", "score");
addFunctionsTypeItem("preScoRepExpr", "prov");

// reindexes the internal addresses according to a prefix
// if address is [0,0,1] and prefix is "0,1", reindexed address will be [0,1,0,0,1]
function reindexInternalAddresses(expressionToIndex, prefix) {
    expressionToIndex = compressExpr(expressionToIndex);
    var replaceStr = "scoInter([" + prefix;
    expressionToIndex = expressionToIndex.replace(/scoInter\(\[/g, replaceStr);
    replaceStr = "scoRepExpr([" + prefix;
    expressionToIndex = expressionToIndex.replace(/scoRepExpr\(\[/g, replaceStr);
    return expressionToIndex;
}


////////////////// functions directories

var integ_functions = ["rInt", "randInt", "intPrim"];
```

```
var integ_primitive = ["intPrim"];
var float_functions = ["rFlo", "floPrim", "sqr", "sum", "dif", "absDif", "prod", "ratio",
"mod", "invers", "root", "pyth", "log", "sin", "cos", "asin", "acos", "tan", "atan", "abs",
"randFlo"];
var float_primitive = ["floPrim"];
var value_functions = [];
value_functions = value_functions.concat(integ_functions, float_functions);
var value_primitive = [];
value_primitive = value_primitive.concat(integ_primitive, float_primitive);
var score_functions = ["scoTransp", "scoRep", "scoConcat", "scoRand", "scoRandFlo", "scoEnumInt",
"scoMutate", "scoRender", "scoRetrog", "scoInvert", "scoExpand", "scoExcerpt", "scoPermut",
"scoPermutIter", "scoSteps", "scoExpandIter", "scoMod", "scoTranspMatrix", "scoPitchClass",
"scoPCGen", "scoSin", "scoCos", "scoExcerptMulti", "scoExcerptMulti2", "scoExcerptMultiTransp",
"addBABBAGEvoice", "scoJoin2Scores", "scoJoin3Scores", "scoJoin4Scores", "scoJoin5Scores"];
var score_primitive = ["scoRandPrim", "scoPCGenPrim", "preScoInter", "preScoRepExpr",
"scoNameADA", "scoNameADAintervals", "scoNameBABBAGE", "scoNameBABBAGEintervals"];
var insco_functions = ["scoInterSco"];
var insco_primitive = []; // each expression includes here the tree address of internal
subscores
var pcset_functions = ["scoPCGen"];
var pcset_primitive = ["scoPCGenPrim"];

var recur_functions = ["preEncRecursio", "preEncRecursioBABBAGE"];
var recur_primitive = ["preEncRecursio", "preEncRecursioBABBAGE"];

var all_reducible_functions = ["intPrim", "floPrim", "scoRandPrim", "scoPCGenPrim"];
var reducible_score_functions = ["scoRandPrim", "scoPCGenPrim"]; // function with output
array, for adding [ ] to array

var all_evolvFunctions = ["scoConcat", "scoTranspMatrix", "scoMutate", "scoMod", "scoTransp",
"scoRep", "scoPermut", "scoPermutIter", "scoExpand", "scoExpandIter", "scoExcerpt", "scoRetrog",
"scoInvert", "addBABBAGEvoice"]; // functions for evolution of an expression
var grow_evolvFunctions = ["scoConcat", "scoTranspMatrix", "scoExpandIter", "scoPermutIter",
"scoRep", "scoExcerptMultiTransp", "addBABBAGEvoice"];
var transform_evolvFunctions = ["scoMutate", "scoMod", "scoTransp", "scoPermut", "scoExpand",
"scoExcerpt", "scoRetrog", "scoInvert", "scoPitchClass", "scoExcerptMulti", "scoExcerptMulti2",
"scoExcerptMultiTransp"];

var available_functions = []; // not including primitive functions
available_functions = available_functions.concat(value_functions, score_functions);

// list of functions under testing
var tests_functions = ["scoRecursio"];

// list available functions
function getFunctionsLibrary() {
    available_functions.sort(); // reorder alphabet.
    post("library of available functions:\n-------------------------\n");
    for (i=0; i<available_functions.length; i++) {
        post(available_functions[i] + " (" + functionsType[available_functions[i]] + ")
- index " + function2index[available_functions[i]] + "\n");
    }
}

getFunctionsLibrary();

/////// auxiliary functions

// convert GenoMus native pitch scale to MIDI scale: 8.0 (GenoMus) = 60 (MIDI), 9.0 (GenoMus)
= 72 (MIDI)
function genomusPitch2MIDI (genomusPitchScore) {
    midiScore = [];
```

```
    for (a=0; a<genomusPitchScore.length; a++) {
        midiScore.push(genomusPitchScore[a]*120);
    }
    return midiScore;
}
addFunctionsTypeItem("genomusPitch2MIDI", "score");

// converts GenoMus native pitch scale to MIDI scale: 8.0 (GenoMus) = 60 (MIDI), 9.0 (GenoMus)
= 72 (MIDI)
function MIDI2genomusPitch (MIDIScore) {
    genomusPitch = [];
    for (a=0; a<MIDIScore.length; a++) {
        genomusPitch.push( MIDIScore[a]/120 );
    }
    return genomusPitch;
}
addFunctionsTypeItem("MIDI2genomusPitch", "score");

// extracts a pitch class from a score (respecting chromatic content of notes, and avoiding
repetitions
function pcsFromScore(score_array) {
    // number of elements of PCS from 1 to chromaticQuantiz
    var numberPitchClasses = (score_array.length % chromaticQuantiz) + 1;
    var pitchClassSet = [];
    var newElem;
    var iter = 0;
    do {
        newElem = Math.round(( score_array[iter] % 1 ) * chromaticQuantiz);
        if ( pitchClassSet.indexOf(newElem) == -1 ) {
            pitchClassSet.push(newElem);
        }
        iter++;
    } while ( pitchClassSet.length < numberPitchClasses && iter < score_array.length );
    return pitchClassSet.sort(function(a,b){return a-b});
}
addFunctionsTypeItem("pcsFromScore", "score");

// returns the sum of absolute values of arr
function absSumArray(arr) {
    var total = 0;
    for(a=0; a<arr.length; a++) {
        total = total + Math.abs(arr[a]);
    }
    return total;
}
addFunctionsTypeItem("absSumArray", "score");

// EOF
```

A.3 Archivo auxiliar include_lh.js

```
/*
* lh.jsextensions
*
* first argument must be "this" to set scope
* second argument specifies the file to include
* relative paths must start wih a "/" separator
* this function uses eval() so please be careful
*
* include(this,"/local.js"); // same directory
* include(this,"/lib/functions/below.js"); // navigate lower
* include(this,"/../../above.js"); // navigate higher
* include(this,"/../../sub1/sub2/both.js"); // combine both
* include(this,"Macintosh HD:/Users/directory/sub-directory/other.js"); // absolute path
* extracted from: http://hallluke.wordpress.com/2010/10/31/including-extra-javascript-files/
*/

function include(scope,dest) {
var mem = "";
var struct = /((\/\.\.)*)(.+$)/;
var parent = /(.+)\/[^.]+\.\w+$/;
var levels = struct.exec(dest)[1].length/3;
var name = struct.exec(dest)[3];
var target;
if (dest.charAt(0) != "/") {
target = dest;
} else if (!levels) {
target = scope.patcher.filepath.match(parent)[1]+name;
} else {
target = scope.patcher.filepath;
for (i=-1; i<levels; i++) {
target = target.slice(0,target.lastIndexOf("/",target.length-2));
}
target += name;
}
var f = new File(target,"read","TEXT");
f.open();
if (f.isopen) {
while(f.position<f.eof) {
mem += f.readline(800)+"\n";
}
f.close()
} else {
var er_file = target.slice(target.lastIndexOf("/",target.length-2)+1);
var er_path = target.slice(0,target.lastIndexOf("/",target.length-2)+1);
post("Error importing file \""+er_file+"\" from \""+er_path+"\"\n");
}
scope.eval(mem);
}

// EOF
```

Anexo B

Threnody for Dimitris Christoulas

Encargo de Taller Sonoro

Instrumentación:
- flauta (alt. piccolo)
- clarinete en si bemol (alt. clarinete bajo)
- violonchelo
- piano
- cinta (estéreo)
- (los intérpretes llevan auriculares y se sincronizan con la cinta con 4 claquetas diferentes)

Duración: 16 min.
Período de composición: abril 2012 — octubre 2012
Estreno: 4 noviembre 2012, Teatros del Canal (Sala Verde), Madrid
Intérpretes:
- Taller Sonoro
- Jesús Sánchez, flauta
- Camilo Irizo, clarinete
- Mery Coronado, violonchelo
- Ignacio Torner, piano
- Javier Campaña, electrónica

Primeras interpretaciones de la obra a cargo de Taller Sonoro:
- 15 noviembre 2012, Auditorio de Zaragoza
- 3 diciembre 2012, Auditori de Barcelona
- 11 diciembre 2012, Sala Joaquín Turina de Sevilla

Página web de la obra: http://www.lopezmontes.es/obra/threnody.html

B.1 Concepto

Dimitris Christoulas es el nombre de un farmacéutico jubilado griego que se disparó en la cabeza en la plaza Syntagma de Atenas, frente al Parlamento griego, el 4 de abril de 2012. Dejó una nota de suicidio explicando que la disminución de su pensión le obligaba a buscar alimentos en la basura, por lo que prefería terminar su vida con dignidad, haciendo un llamamiento a la rebelión de los jóvenes. El planteamiento de la pieza tiene esta temática como telón de fondo extramusical, aunque sin intención programática ni ilustrativa. La referencia a Dimitris Christoulas establece una conexión con el mundo emocional y el contexto histórico en que tuvo lugar la tarea de composición. En algún momento este suceso quedó ligado a la obra. Su influencia en la pieza y en la percepción del público está en un ámbito subjetivo, y queda fuera del alcance de este trabajo.

Threnody for Dimitris Christoulas es la primera obra que explora las técnicas de metaprogramación de *GenoMus*. Sus materiales melódicos proceden casi íntegramente de los genotipos generados con el primer prototipo operativo del programa.

B.2 Método compositivo

B.2.1 El atractivo conceptual de la recursión

La recursión permite conseguir una gran variedad de comportamientos a partir de fórmulas de gran simplicidad. Un repaso a los genotipos y fenotipos que se detallan en la sección B.4 muestra la riqueza de texturas posibles.

La idea de que cada elemento de una composición se deduzca del anterior es muy antigua, pero en el caso de la recursión su significado es literal. Las secuencias así generadas tienen algo de esa *verdad*, en un sentido cercano a Cage, que se opone a la *belleza* fabricada: son objetos con una existencia independiente e inalterable, que no se doblegan ante los gustos del compositor.

B.2.2 Espacio melódico explorado

Cada recursión parte de un *motivo* de un máximo de 4 notas, las cuales constituyen las *condiciones iniciales* de cada recursión. A partir de estas alturas iniciales se deducen el resto. Estas ecuaciones recursivas son el primer prototipo operativo de genotipo. El algoritmo de creación de expresiones válidas era una primera versión menos eficaz que la actual.[67]

Las funciones que se combinan en estos genotipos son operadores aritméticos básicos y trigonométricos, constantes numéricas y las variables que referencian a las condiciones iniciales. Este tipo de genotipos es muy similar al que *GenoMus* usa en las recursiones comentadas en la sección 6.5.2.

[67] Esto es evidente en los niveles de paréntesis innecesarios de estas expresiones.

B.2.3 Criterios para la musicalización de los fenotipos

El momento de máxima libertad creativa tuvo lugar durante las sesiones de producción y selección de genotipos y fenotipos. Las reglas del juego que se establecieron para esta composición dieron total libertad para la producción y modelado manual de los pares genotipo–fenotipo, pero una vez que estos pares eran seleccionados obligaron a adaptarse a ellos de manera férrea, a pesar de las dificultades de tesitura, y en algunos casos, de rítmica.

La plasmación de estas secuencias abstractas de alturas en texturas rítmicas, contrapuntísticas o tímbricas se hizo con el criterio principal de dejarse arrastrar hacia el tipo de movimiento que en muchos casos la propia secuencia sugería claramente, y siempre tratando de que el *arreglo* instrumental revelase sus detalles formales de la manera más perceptible.

En algunos casos se optó por una rítmica estirada que enfatizase los aspectos armónicos de las alturas; en otros se optó por un rápido ritmo constante de casi *motto perpetuo* puntillista para hacer patentes detalles estructurales mucho más dilatados.

B.2.4 El doble papel de la electrónica

Para la producción de la parte electrónica se partió de estas premisas:

- La síntesis de sonido se usó como banco de pruebas de un prototipo de algoritmo genético en *MaxMSP*, el cual se aplicó en la manipulación de un sintetizador programado a tal fin.

- Los sonidos electrónicos tienen dos funciones: ampliar la sonoridad de los instrumentos, coloreando o ampliando su timbre, y ser un instrumento virtual más, que entra en contrapunto como una parte más. En muchos casos la línea que divide estas dos funciones es confusa.

- Todos los sonidos de la electrónica proceden de un único instrumento virtual que funciona como una batería de filtros que actúan sobre una fuente de audio (generalmente ruido o sonidos reales con muchas impurezas).

B.2.5 Un algoritmo genético en *MaxMSP*

Un algoritmo genético es un programa informático de búsqueda heurística que trata de encontrar soluciones a problemas concretos mediante técnicas inspiradas en la evolución y selección natural. En la composición de *Threnody for Dimitris Christoulas* se utilizó un algoritmo genético programado en *JavaScript* para ser insertado como parte del sintetizador de *MaxMSP* descrito arriba. Dado que el comportamiento de este sintetizador depende de una gran cantidad de variables, cuya interacción es además difícilmente

previsible, se optó por usar un algoritmo genético porque en la práctica se comprobó que tenía dos consecuencias de gran interés para el compositor:

- Propició la rápida creación de una amplia paleta de timbres, puesto que este sistema de búsqueda es inmune a los prejuicios del compositor, que tiende a manipular los controles del sintetizador conforme a una idea preconcebida del tipo de sonidos que busca,
- y agilizó el proceso de modelado de timbres, ya que se pudo trabajar desde una perspectiva global sin preocuparse de los detalles de la parametrización de cada timbre.

El algoritmo genético produjo una fértil interacción entre la riqueza de propuestas iniciales (independientes de la mano del compositor) y su capacidad para ir restringiendo y mutando los resultados seleccionados en función de las decisiones artísticas, que pueden tomarse sin atender a los detalles técnicos de la implementación. Este algoritmo no está documentado en este trabajo, pero sirvió como una experiencia previa a la programación de *GenoMus*, ya que en cierto modo reprodujo el mismo tipo de aproximación a la creatividad asistida en el campo del timbre y la síntesis de sonido.

B.2.6 *eHayku (Study for Threnody)*

Como preparación de la parte electrónica se compuso una miniatura electrónica de dos minutos, titulada *eHayku (Study for Threnody)* [34]. Esta pieza exploró la paleta tímbrica del instrumento creado. La tímbrica de la electroacústica de *Threnody for Dimitris Christoulas* es muy similar a la de esta miniatura.

B.3 Crítica

La única crítica aparecida hasta la fecha es la del crítico Pablo J. Payón, aparecida en el Diario de Sevilla tras el concierto de Taller Sonoro en la Sala Joaquín Turina de Sevilla:

> [...] *Taller Sonoro cerró su propio festival con la propuesta más arriesgada de cuantas se han visto en él. El conjunto sevillano ha cultivado siempre repertorios muy cercanos a la experimentación, y ese terreno en el que el arte convencional se aproxima al llamado arte sonoro, cuando no se cruza con él, es el que pisaron ayer.*
>
> [...] *Para el final quedó la, en mi opinión, más ambiciosa obra del programa. Originales del granadino José López-Montes (Guadix, 1977), esos Threnody están concebidos como un virtuoso diálogo instrumental, con un trabajo refinadísimo sobre las texturas y un empleo muy sutil de la electrónica.* [46]

B.4 Genotipos y fenotipos seleccionados

B.4.1 *Recursio I–a*

Genotipo

```
(vAdd(x1,(vSin((vDiv((vAdd(x1,(vAdd(x3,x1)))),(vDiv((vAdd(x2,0.48196260851064854)),x3)))),(vSin(x1,x2))))))

mapping: (-1, 1) -> (36, 96)
initial conditions:
in. -4: -0.4826  ->  52  |  164.814 Hz  |  E2
in. -3: 0.4338  ->  79  |  783.991 Hz  |  G4
in. -2: 0.6630  ->  86  |  1174.659 Hz  |  D5
in. -1: -0.7349  ->  44  |  103.826 Hz  |  G#1
generated score:
ev. 0: -1.1174  ->  32  |  51.9131 Hz  |  G#0
ev. 1: -1.9471  ->  8  |  12.9783 Hz  |  G#-2
ev. 2: -1.1456  ->  32  |  51.9131 Hz  |  G#0
ev. 3: -1.6614  ->  16  |  20.6017 Hz  |  E-1
ev. 4: -1.9063  ->  9  |  13.7500 Hz  |  A-2
ev. 5: -0.9116  ->  39  |  77.7817 Hz  |  D#1
ev. 6: -0.1142  ->  63  |  311.1270 Hz  |  D#3
ev. 7: -0.0686  ->  64  |  329.6276 Hz  |  E3
ev. 8: 0.4471  ->  79  |  783.9909 Hz  |  G4
ev. 9: 0.2333  ->  73  |  554.3653 Hz  |  C#4
...
```

$$n_0 = n_{-1} + \sin\left(\frac{2n_{-1} + n_{-3}}{\frac{n_{-2}+0{,}482}{n_{-3}}}\right)$$

Condiciones iniciales:
$n_{-4} = -0{,}4826$
$n_{-3} = 0{,}4338$
$n_{-2} = 0{,}663$
$n_{-1} = -0{,}7349$

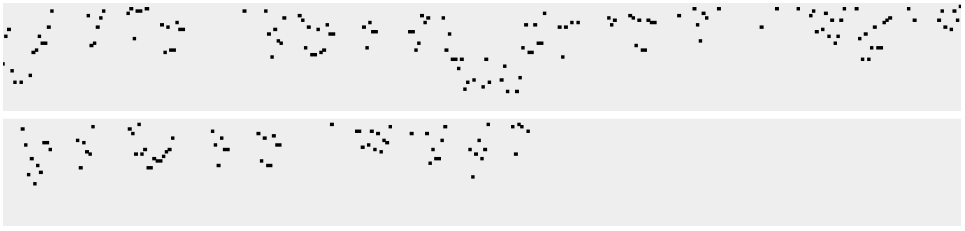
Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/threnody/recursio_I-a.mid
txt ⟶ http://www.lopezmontes.es/obra/threnody/recursio_I-a.txt

Gráfico de la secuencia de alturas



B.4.2 *Recursio II*

Genotipo

```
(vSin((vAdd((vDiv(0.4224016310636478,(vAdd(x1,x3)))),(vDiv((vAdd(x2,(vMult(x2,x1)))),(vDiv(x1,(vMult(x3,x2
))))))))),(vAdd((vDif((vDiv((vDiv(x2,x2)),(vSin(x1,x4))))),0.600562209309903)),(vDiv(x3,(vDif(x4,(vAdd(x2,x4
))))))))))

mapping: (-1, 1) -> (36, 96)
initial conditions:
in. -4: -0.8572  ->  40  |  82.407 Hz  |  E1
in. -3: -0.3279  ->  56  |  207.652 Hz  |  G#2
in. -2: -0.7214  ->  44  |  103.826 Hz  |  G#1
in. -1: -0.4249  ->  53  |  174.614 Hz  |  F2
generated score:
ev. 0: -0.3242  ->  56  |  207.6523 Hz  |  G#2
ev. 1: -0.1321  ->  62  |  293.6648 Hz  |  D3
ev. 2: -0.4484  ->  53  |  174.6141 Hz  |  F2
ev. 3: -0.5140  ->  51  |  155.5635 Hz  |  D#2
ev. 4: -0.5881  ->  48  |  130.8128 Hz  |  C2
ev. 5: -0.3189  ->  56  |  207.6523 Hz  |  G#2
ev. 6: -0.1272  ->  62  |  293.6648 Hz  |  D3
ev. 7: -0.1790  ->  61  |  277.1826 Hz  |  C#3
ev. 8: -0.7343  ->  44  |  103.8262 Hz  |  G#1
ev. 9: -0.4696  ->  52  |  164.8138 Hz  |  E2
...
```

$$n_0 = \sin\left(\frac{0{,}422}{n_{-1}+n_{-3}} + \frac{n_{-2}(n_{-1}+1)}{\frac{n_{-1}}{n_{-2}n_{-3}}}\right)$$

Condiciones iniciales:
$n_{-4} = -0{,}8572$
$n_{-3} = -0{,}3279$
$n_{-2} = -0{,}7214$
$n_{-1} = -0{,}4249$

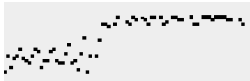
Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/threnody/recursio_II.mid
txt ⟶ http://www.lopezmontes.es/obra/threnody/recursio_II.txt

Gráfico de la secuencia de alturas



B.4.3 *Recursio III–a*

Genotipo

```
(vDif(-0.0335660162618765,(vMult(x1,(vAdd(x2,(vDiv(x4,x3))))))))

mapping: (-1, 1) -> (36, 96)
initial conditions:
in. -4: -0.1753  ->  61  |  277.183 Hz  |  C#3
in. -3: 0.3490  ->  76  |  659.255 Hz  |  E4
in. -2: 0.3341  ->  76  |  659.255 Hz  |  E4
in. -1: 0.3205  ->  76  |  659.255 Hz  |  E4
generated score:
ev. 0: 0.0203  ->  67  |  391.9954 Hz  |  G3
ev. 1: -0.0613  ->  64  |  329.6276 Hz  |  E3
ev. 2: 0.0316  ->  67  |  391.9954 Hz  |  G3
ev. 3: -0.5298  ->  50  |  146.8324 Hz  |  D2
ev. 4: -0.1925  ->  60  |  261.6256 Hz  |  C3
ev. 5: -0.5090  ->  51  |  155.5635 Hz  |  D#2
ev. 6: -0.1619  ->  61  |  277.1826 Hz  |  C#3
ev. 7: 0.3297  ->  76  |  659.2551 Hz  |  E4
ev. 8: -0.1049  ->  63  |  311.1270 Hz  |  D#3
ev. 9: 0.3306  ->  76  |  659.2551 Hz  |  E4
...
```

$$n_0 = -0{,}0336 - n_{-1}\left(n_{-2} + \frac{n_{-4}}{n_{-3}}\right)$$

Condiciones iniciales:
$n_{-4} = -0{,}1753$
$n_{-3} = 0{,}349$
$n_{-2} = 0{,}3341$
$n_{-1} = 0{,}3205$

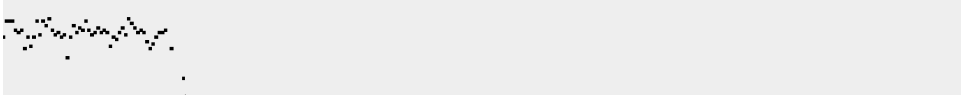
Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Fenotipos

MIDI $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_III-a.mid
txt $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_III-a.txt

Gráfico de la secuencia de alturas



B.4.4 *Recursio III–b*

Genotipo

```
(vDif(-0.0335660162618765,(vMult(x1,(vAdd(x2,(vDiv(x4,x3))))))))

mapping: (-1, 1) -> (36, 96)
initial conditions:
in. -4: -0.7869  ->  42  |  92.499 Hz  |  F#1
in. -3: -0.3248  ->  56  |  207.652 Hz  |  G#2
in. -2: -0.4391  ->  53  |  174.614 Hz  |  F2
in. -1: 0.0958  ->  69  |  440.000 Hz  |  A3
generated score:
ev. 0: -0.2236  ->  59  |  246.9417 Hz  |  B2
ev. 1: 0.1532  ->  71  |  493.8833 Hz  |  B3
ev. 2: 0.7031  ->  87  |  1244.5079 Hz  |  D#5
ev. 3: 0.1599  ->  71  |  493.8833 Hz  |  B3
ev. 4: 0.0873  ->  69  |  440.0000 Hz  |  A3
ev. 5: -0.0666  ->  64  |  329.6276 Hz  |  E3
ev. 6: 0.2649  ->  74  |  587.3295 Hz  |  D4
ev. 7: -0.5010  ->  51  |  155.5635 Hz  |  D#2
ev. 8: -0.5582  ->  49  |  138.5913 Hz  |  C#2
ev. 9: -0.4535  ->  52  |  164.8138 Hz  |  E2
...
```

$$n_0 = -0{,}0336 - n_{-1}\left(n_{-2} + \frac{n_{-4}}{n_{-3}}\right)$$

Condiciones iniciales:
$n_{-4} = -0{,}7869$
$n_{-3} = -0{,}3248$
$n_{-2} = -0{,}4391$
$n_{-1} = 0{,}0958$

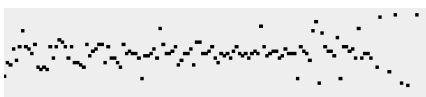
Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Fenotipos

MIDI $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_III-b.mid
txt $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_III-b.txt

Gráfico de la secuencia de alturas



B.4.5 *Recursio IV–a*

Genotipo

```
(vTan((vAdd((vDiv(x4,x2)),(vDiv((vAdd(x2,x2)),(vAdd(x4,x2))))))),(vMult(x2,(vAdd(x4,x1))))))

mapping: (-1, 1) -> (36, 96)
initial conditions:
in. -4: -0.1020  ->  63  |  311.127 Hz  |  D#3
in. -3: -0.6598  ->  46  |  116.541 Hz  |  A#1
in. -2: -0.6955  ->  45  |  110.000 Hz  |  A1
in. -1: -0.7776  ->  43  |  97.999 Hz  |  G1
generated score:
ev. 0: -3.0170  ->  -25  |  1.9292 Hz  |  C#-4
ev. 1: -2.6594  ->  -14  |  3.6419 Hz  |  D-3
ev. 2: -3.4126  ->  -36  |  1.0220 Hz  |  C-5
ev. 3: -3.6258  ->  -43  |  0.6821 Hz  |  G-5
ev. 4: -2.5419  ->  -10  |  4.5885 Hz  |  A#-2
ev. 5: -3.0540  ->  -26  |  1.8210 Hz  |  D-4
ev. 6: -1.3846  ->  24  |  32.7032 Hz  |  C0
ev. 7: -1.7035  ->  15  |  19.4454 Hz  |  D#-1
ev. 8: -0.6848  ->  45  |  110.0000 Hz  |  A1
ev. 9: -0.7332  ->  44  |  103.8262 Hz  |  G#1
...
```

$$n_0 = \tan\left(\frac{n_{-4}}{n_{-2}} + \frac{n_{-2} + n_{-3}}{n_{-2} + n_{-4}}\right)$$

Condiciones iniciales:
$n_{-4} = -0{,}102$
$n_{-3} = -0{,}6598$
$n_{-2} = -0{,}6955$
$n_{-1} = -0{,}7776$

Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Fenotipos

MIDI $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_IV-a.mid
txt $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_IV-a.txt

Gráfico de la secuencia de alturas

B.4.6 *Recursio V*

Genotipo

```
(vSin((vDiv(x3,x1)),x4))

mapping: (-1, 1) -> (36, 96)
initial conditions:
in. -4: 0.1965  ->  72  |  523.251 Hz  |  C4
in. -3: -0.2052  ->  60  |  261.626 Hz  |  C3
in. -2: -0.1703  ->  61  |  277.183 Hz  |  C#3
in. -1: 0.1859  ->  72  |  523.251 Hz  |  C4
generated score:
ev. 0: -0.8929  ->  39  |  77.7817 Hz  |  D#1
ev. 1: 0.1896  ->  72  |  523.2511 Hz  |  C4
ev. 2: 0.8309  ->  91  |  1567.9817 Hz  |  G5
ev. 3: -0.8794  ->  40  |  82.4069 Hz  |  E1
ev. 4: -0.2139  ->  60  |  261.6256 Hz  |  C3
ev. 5: 0.6765  ->  86  |  1174.6591 Hz  |  D5
ev. 6: -0.9635  ->  37  |  69.2957 Hz  |  C#1
ev. 7: 0.2202  ->  73  |  554.3653 Hz  |  C#4
ev. 8: 0.0686  ->  68  |  415.3047 Hz  |  G#3
ev. 9: -0.9952  ->  36  |  65.4064 Hz  |  C1
 ...
```

$$n_0 = \sin\left(\frac{n_{-3}}{n_{-1}}\right)$$

Condiciones iniciales:
$n_{-4} = 0{,}1965$
$n_{-3} = -0{,}2052$
$n_{-2} = -0{,}1703$
$n_{-1} = 0{,}1859$

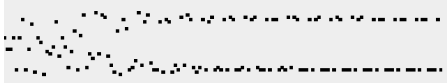
Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Fenotipos

MIDI $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_V.mid
txt $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_V.txt

Gráfico de la secuencia de alturas



B.4.7 *Recursio VI*

Solo de electrónica no compuesto según recursiones.

B.4.8 *Recursio IV–b*

Genotipo

```
(vTan((vAdd((vDiv(x4,x2)),(vDiv((vAdd(x2,x3)),(vAdd(x4,x2))))))),(vMult(x2,(vAdd(x4,x1))))))

mapping: (-1, 1) -> (36, 96)
initial conditions:
in. -4: 0.0387  ->  67  |  391.995 Hz  |  G3
in. -3: 0.0131  ->  66  |  369.994 Hz  |  F#3
in. -2: -0.1265  ->  62  |  293.665 Hz  |  D3
in. -1: -0.3769  ->  55  |  195.998 Hz  |  G2
generated score:
ev. 0: 1.5093  ->  111  |  4978.0317 Hz  |  D#7
ev. 1: 4.4338  ->  199  |  802806.6530 Hz  |  G14
ev. 2: 0.9042  ->  93  |  1760.0000 Hz  |  A5
ev. 3: 5.1754  ->  221  |  2860877.6722 Hz  |  F16
ev. 4: 0.9121  ->  93  |  1760.0000 Hz  |  A5
ev. 5: 12.2582  ->  434  |  630640287249.4790 Hz  |  D34
ev. 6: 2.5819  ->  143  |  31608.5313 Hz  |  B9
ev. 7: 2.4112  ->  138  |  23679.6431 Hz  |  F#9
ev. 8: 8.9018  ->  333  |  1845493760.3241 Hz  |  A25
ev. 9: -1.1590  ->  31  |  48.9994 Hz  |  G0
 ...
```

$$n_0 = \tan\left(\frac{n_{-4}}{n_{-2}} + \frac{n_{-2} + n_{-3}}{n_{-2} + n_{-4}}\right)$$

Condiciones iniciales:
$n_{-4} = 0{,}0387$
$n_{-3} = 0{,}0131$
$n_{-2} = -0{,}1265$
$n_{-1} = -0{,}3769$

Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/threnody/recursio_IV-b.mid
txt ⟶ http://www.lopezmontes.es/obra/threnody/recursio_IV-b.txt

Gráfico de la secuencia de alturas

B.4.9 *Recursio VII*

Genotipo

```
(vTan((vDif((vMult((vAdd((vMult(x4,x3)),-0.21903280447337825)),x2)),x1)),x4))

mapping: (-1, 1) -> (36, 96)
initial conditions:
in. -4: -0.5382  ->  50  |  146.832 Hz  |  D2
in. -3: -0.4377  ->  53  |  174.614 Hz  |  F2
in. -2: 0.3421  ->  76  |  659.255 Hz  |  E4
in. -1: 0.7316  ->  88  |  1318.510 Hz  |  E5
generated score:
ev. 0: -0.8876  ->  39  |  77.7817 Hz  |  D#1
ev. 1: 0.7107  ->  87  |  1244.5079 Hz  |  D#5
ev. 2: -0.9102  ->  39  |  77.7817 Hz  |  D#1
ev. 3: 0.3017  ->  75  |  622.2540 Hz  |  D#4
ev. 4: 0.5102  ->  81  |  880.0000 Hz  |  A4
ev. 5: -0.9725  ->  37  |  69.2957 Hz  |  C#1
ev. 6: 0.8782  ->  92  |  1661.2188 Hz  |  G#5
ev. 7: -1.0607  ->  34  |  58.2705 Hz  |  A#0
ev. 8: 0.4618  ->  80  |  830.6094 Hz  |  G#4
ev. 9: 0.8027  ->  90  |  1479.9777 Hz  |  F#5
 ...
```

$$n_0 = \tan((n_{-2}((n_{-4}n_{-3}) + 0{,}219)) - n_{-1})$$

Condiciones iniciales:
$n_{-4} = 0{,}579$
$n_{-3} = 0{,}4377$
$n_{-2} = 0{,}3421$
$n_{-1} = 0{,}7316$

Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Fenotipos

MIDI $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_VII.mid
txt $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_VII.txt

Gráfico de la secuencia de alturas

B.4.10 *Recursio VIII*

Genotipo

```
(vCos((vDiv(x1,x4)),(vLog(x3,x4))))

mapping: (-1, 1) -> (36, 96)
initial conditions:
in. -4: 0.5260  ->  82  |  932.328 Hz  |  A#4
in. -3: 0.6570  ->  86  |  1174.659 Hz  |  D5
in. -2: -0.4000  ->  54  |  184.997 Hz  |  F#2
in. -1: -0.8900  ->  39  |  77.782 Hz  |  D#1
generated score:
ev. 0: -0.1209  ->  62  |  293.6648 Hz  |  D3
ev. 1: 0.9831  ->  95  |  1975.5332 Hz  |  B5
ev. 2: -0.7752  ->  43  |  97.9989 Hz  |  G1
ev. 3: 0.6441  ->  85  |  1108.7305 Hz  |  C#5
ev. 4: 0.5762  ->  83  |  987.7666 Hz  |  B4
ev. 5: 0.8331  ->  91  |  1567.9817 Hz  |  G5
ev. 6: 0.4759  ->  80  |  830.6094 Hz  |  G#4
ev. 7: 0.7392  ->  88  |  1318.5102 Hz  |  E5
ev. 8: 0.2839  ->  75  |  622.2540 Hz  |  D#4
ev. 9: 0.9425  ->  94  |  1864.6550 Hz  |  A#5
 ...
```

$$n_0 = \cos\left(\frac{n_{-1}}{n_{-4}}\right)$$

Condiciones iniciales:
$n_{-4} = 0{,}526$
$n_{-3} = 0{,}657$
$n_{-2} = -0{,}8$
$n_{-1} = -0{,}4$

Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Fenotipos

MIDI $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_VIII.mid
txt $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_VIII.txt

Gráfico de la secuencia de alturas

B.4.11 *Recursio IX*

Genotipo

```
(vDiv((vAdd(x1,-6)),x1))

mapping: (-1, 1) -> (58, 61)
initial conditions:
in. -4: -0.4849  ->  59  |  246.942 Hz  |  B2
in. -3: -0.7870  ->  58  |  233.082 Hz  |  A#2
in. -2: 0.1491  ->  60  |  261.626 Hz  |  C3
in. -1: 0.4678  ->  60  |  261.626 Hz  |  C3
generated score:
ev. 0: -11.8260  ->  42  |  92.4986 Hz  |  F#1
ev. 1: 1.5074  ->  62  |  293.6648 Hz  |  D3
ev. 2: -2.9805  ->  55  |  195.9977 Hz  |  G2
ev. 3: 3.0131  ->  64  |  329.6276 Hz  |  E3
ev. 4: -0.9913  ->  58  |  233.0819 Hz  |  A#2
ev. 5: 7.0526  ->  70  |  466.1638 Hz  |  A#3
ev. 6: 0.1493  ->  60  |  261.6256 Hz  |  C3
ev. 7: -39.2000  ->  1  |  8.6620 Hz  |  C#-2
ev. 8: 1.1531  ->  61  |  277.1826 Hz  |  C#3
ev. 9: -4.2035  ->  53  |  174.6141 Hz  |  F2
 ...
```

$$n_0 = \frac{n_{-1} + 6}{n_{-1}}$$

Condiciones iniciales:
$n_{-4} = -0{,}4849$
$n_{-3} = -0{,}787$
$n_{-2} = 0{,}1491$
$n_{-1} = 0{,}4678$

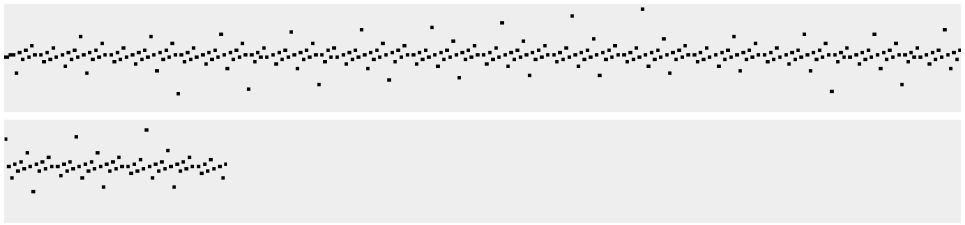
Intervalo de conversión:
$[-1, 1] \longrightarrow [58, 61]$

Función de conversión a alturas MIDI:
$n = \frac{3n_0+119}{2}$

Fenotipos

MIDI $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_IX.mid
txt $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_IX.txt

Gráfico de la secuencia de alturas



B.4.12 *Recursio X*

Genotipo

```
(vAtan2((vAdd(x2,x3)),vAdd((vDiv((vSin(x1,x2)),x3)),vMult((vSin((vDiv(x2,x3)),(vLog(x2,x1)))),(vSin((vDiv
(x4,x2)),x4)))))))

mapping: (-1, 1) -> (36, 96)
initial conditions:
in. -4: -0.2143  ->  60  |  261.626 Hz  |  C3
in. -3: -0.4355  ->  53  |  174.614 Hz  |  F2
in. -2: -0.0314  ->  65  |  349.228 Hz  |  F3
in. -1: 0.9293  ->  94  |  1864.655 Hz  |  A#5
generated score:
ev. 0: -1.8242  ->  11  |  15.4339 Hz  |  B-2
ev. 1: 1.5413  ->  112  |  5274.0409 Hz  |  E7
ev. 2: 2.2721  ->  134  |  18794.5451 Hz  |  D9
ev. 3: -1.8947  ->  9  |  13.7500 Hz  |  A-2
ev. 4: -0.3358  ->  56  |  207.6523 Hz  |  G#2
ev. 5: 0.8058  ->  90  |  1479.9777 Hz  |  F#5
ev. 6: -2.9371  ->  -22  |  2.2943 Hz  |  A#-3
ev. 7: 1.1620  ->  101  |  2793.8259 Hz  |  F6
ev. 8: 2.6310  ->  145  |  35479.3768 Hz  |  C#10
ev. 9: -2.9132  ->  -21  |  2.4307 Hz  |  A-3
 ...
```

$$n_0 = \tan\left(\frac{n_{-2}+n_{-3}}{\frac{\sin(n_{-1})}{n_{-3}}+\sin\left(\frac{n_{-2}}{n_{-3}}\right)\sin\left(\frac{n_{-4}}{n_{-2}}\right)}\right)$$

Condiciones iniciales:
$n_{-4} = -0{,}2143$
$n_{-3} = -0{,}4355$
$n_{-2} = -0{,}0314$
$n_{-1} = 0{,}9293$

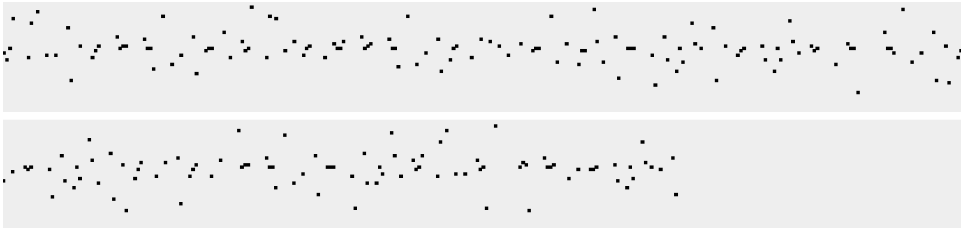
Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/threnody/recursio_X.mid
txt ⟶ http://www.lopezmontes.es/obra/threnody/recursio_X.txt

Gráfico de la secuencia de alturas



B.4.13 *Recursio I–b*

Genotipo

```
(vAdd(x1,(vSin((vDiv((vAdd(x1,(vAdd(x3,x1)))),(vDiv((vAdd(x2,0.48196260851064854)),x3)))),(vSin(x1,x2))))))

mapping: (-1, 1) -> (36, 96)
initial conditions:
in. -4: 0.5790  ->  83  |  987.767 Hz  |  B4
in. -3: 0.2416  ->  73  |  554.365 Hz  |  C#4
in. -2: -0.0063  ->  66  |  369.994 Hz  |  F#3
in. -1: -0.3749  ->  55  |  195.998 Hz  |  G2
generated score:
ev. 0: -0.6302  ->  47  |  123.4708 Hz  |  B1
ev. 1: -0.5557  ->  49  |  138.5913 Hz  |  C#2
ev. 2: 0.0238  ->  67  |  391.9954 Hz  |  G3
ev. 3: 0.9886  ->  96  |  2093.0045 Hz  |  C6
ev. 4: -0.0113  ->  66  |  369.9944 Hz  |  F#3
ev. 5: -0.0113  ->  66  |  369.9944 Hz  |  F#3
ev. 6: 0.8854  ->  93  |  1760.0000 Hz  |  A5
ev. 7: 0.8431  ->  91  |  1567.9817 Hz  |  G5
ev. 8: 0.8293  ->  91  |  1567.9817 Hz  |  G5
ev. 9: 1.8210  ->  121  |  8869.8442 Hz  |  C#8
 ...
```

$$n_0 = n_{-1} + \sin\left(\frac{2n_{-1} + n_{-3}}{\frac{n_{-2}+0{,}482}{n_{-3}}}\right)$$

Condiciones iniciales:
$n_{-4} = 0{,}579$
$n_{-3} = 0{,}2416$
$n_{-2} = -0{,}0063$
$n_{-1} = -0{,}3749$

Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Fenotipos

MIDI $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_I-b.mid
txt $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_I-b.txt

Gráfico de la secuencia de alturas

B.4.14 *Recursio XI*

Genotipo

```
(vTan((vDif((vAdd((vDiv((vDiv(0.12,0.83)),(vDiv(x2,x1)))),x1)),x1)),(vAdd(-0.02,(vDiv((vPow(x1,(vLog(x1,x2
))))),x2))))))

mapping: (-1, 1) -> (36, 96)
initial conditions:
in. -4: -0.0822  ->  64  |  329.628 Hz  |  E3
in. -3: 0.7865  ->  90  |  1479.978 Hz  |  F#5
in. -2: 0.1410  ->  70  |  466.164 Hz  |  A#3
in. -1: 0.1962  ->  72  |  523.251 Hz  |  C4
generated score:
ev. 0: 0.2039  ->  72  |  523.2511 Hz  |  C4
ev. 1: 0.1514  ->  71  |  493.8833 Hz  |  B3
ev. 2: 0.1078  ->  69  |  440.0000 Hz  |  A3
ev. 3: 0.1033  ->  69  |  440.0000 Hz  |  A3
ev. 4: 0.1394  ->  70  |  466.1638 Hz  |  A#3
ev. 5: 0.1977  ->  72  |  523.2511 Hz  |  C4
ev. 6: 0.2080  ->  72  |  523.2511 Hz  |  C4
ev. 7: 0.1532  ->  71  |  493.8833 Hz  |  B3
ev. 8: 0.1069  ->  69  |  440.0000 Hz  |  A3
ev. 9: 0.1012  ->  69  |  440.0000 Hz  |  A3
 ...
```

$$n_0 = \tan\left(\frac{0{,}145}{\frac{n_{-2}}{n_{-1}}}\right)$$

Condiciones iniciales:
$n_{-4} = -0{,}0822$
$n_{-3} = 0{,}7865$
$n_{-2} = 0{,}141$
$n_{-1} = 0{,}1962$

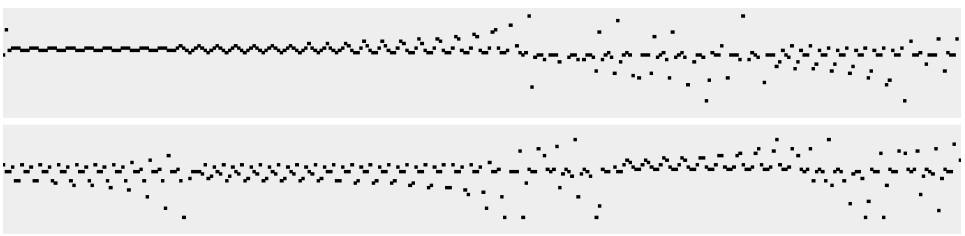
Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

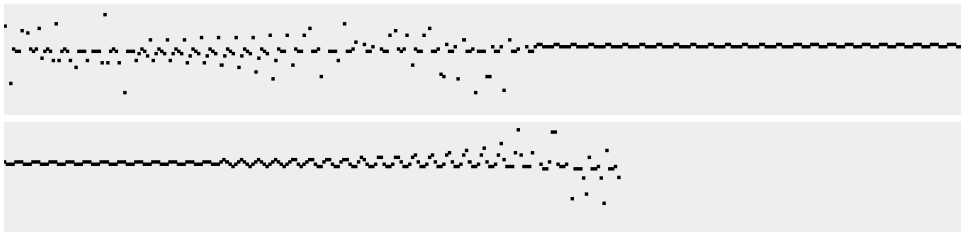
Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Fenotipos

MIDI $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_XI.mid
txt $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_XI.txt

Gráfico de la secuencia de alturas





B.4.15 *Recursio XII*

Genotipo

```
(vDiv(x3,(vDiv((vSin(x4,(vAdd(x2,x3)))),x4))))

mapping: (-1, 1) -> (36, 96)
initial conditions:
in. -4: 0.5400  ->  82  |  932.328 Hz  |  A#4
in. -3: 0.4281  ->  79  |  783.991 Hz  |  G4
in. -2: 0.8000  ->  90  |  1479.978 Hz  |  F#5
in. -1: 0.2970  ->  75  |  622.254 Hz  |  D#4
generated score:
ev. 0: 0.4496  ->  79  |  783.9909 Hz  |  G4
ev. 1: 0.8250  ->  91  |  1567.9817 Hz  |  G5
ev. 2: 0.3312  ->  76  |  659.2551 Hz  |  E4
ev. 3: 0.4563  ->  80  |  830.6094 Hz  |  G#4
ev. 4: 0.8534  ->  92  |  1661.2188 Hz  |  G#5
ev. 5: 0.3720  ->  77  |  698.4565 Hz  |  F4
ev. 6: 0.4648  ->  80  |  830.6094 Hz  |  G#4
ev. 7: 0.8838  ->  93  |  1760.0000 Hz  |  A5
ev. 8: 0.4213  ->  79  |  783.9909 Hz  |  G4
ev. 9: 0.4757  ->  80  |  830.6094 Hz  |  G#4
 ...
```

$$n_0 = \frac{n_{-3}}{\frac{\sin(n_{-4})}{n_{-4}}}$$

Condiciones iniciales:
$n_{-4} = 0{,}54$
$n_{-3} = 0{,}4281$
$n_{-2} = 0{,}8$
$n_{-1} = 0{,}2970$

Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Fenotipos

MIDI $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_XII.mid
txt $\longrightarrow$ http://www.lopezmontes.es/obra/threnody/recursio_XII.txt

Gráfico de la secuencia de alturas

B.4.16 *Recursio XIII–a — XIII–d6*

Genotipo

```
(vDif((vDif(x2,x4)),(vSin(-0.033,x1))))

mapping: (-1, 1) -> (36, 96)
initial conditions a:
in. -4: -0.3047  ->  57  |  220.000 Hz  |  A2
in. -3: -0.6255  ->  47  |  123.471 Hz  |  B1
in. -2: -0.6307  ->  47  |  123.471 Hz  |  B1
in. -1: -0.4024  ->  54  |  184.997 Hz  |  F#2
generated score:
ev. 0: -0.2930  ->  57  |  220.0000 Hz  |  A2
ev. 1: 0.2561  ->  74  |  587.3295 Hz  |  D4
ev. 2: 0.3707  ->  77  |  698.4565 Hz  |  F4
ev. 3: 0.6915  ->  87  |  1244.5079 Hz  |  D#5
ev. 4: 0.6967  ->  87  |  1244.5079 Hz  |  D#5
ev. 5: 0.4684  ->  80  |  830.6094 Hz  |  G#4
ev. 6: 0.3590  ->  77  |  698.4565 Hz  |  F4
ev. 7: -0.1901  ->  60  |  261.6256 Hz  |  C3
ev. 8: -0.3047  ->  57  |  220.0000 Hz  |  A2
ev. 9: -0.6255  ->  47  |  123.4708 Hz  |  B1
 ...

initial conditions b:
in. -4: 0.6362  ->  85  |  1108.731 Hz  |  C#5
in. -3: -0.3537  ->  55  |  195.998 Hz  |  G2
in. -2: 0.3535  ->  77  |  698.456 Hz  |  F4
in. -1: -0.4419  ->  53  |  174.614 Hz  |  F2
generated score:
ev. 0: -0.2497  ->  59  |  246.9417 Hz  |  B2
ev. 1: -0.0552  ->  64  |  329.6276 Hz  |  E3
ev. 2: -0.5702  ->  49  |  138.5913 Hz  |  C#2
ev. 3: 0.4197  ->  79  |  783.9909 Hz  |  G4
ev. 4: -0.2875  ->  57  |  220.0000 Hz  |  A2
ev. 5: 0.5079  ->  81  |  880.0000 Hz  |  A4
ev. 6: 0.3157  ->  75  |  622.2540 Hz  |  D#4
ev. 7: 0.1212  ->  70  |  466.1638 Hz  |  A#3
ev. 8: 0.6362  ->  85  |  1108.7305 Hz  |  C#5
ev. 9: -0.3537  ->  55  |  195.9977 Hz  |  G2
...

initial conditions c:
in. -4: -0.2095  ->  60  |  261.626 Hz  |  C3
in. -3: -0.5310  ->  50  |  146.832 Hz  |  D2
in. -2: -0.0433  ->  65  |  349.228 Hz  |  F3
in. -1: 0.1802  ->  71  |  493.883 Hz  |  B3
generated score:
ev. 0: 0.1992  ->  72  |  523.2511 Hz  |  C4
ev. 1: 0.7442  ->  88  |  1318.5102 Hz  |  E5
ev. 2: 0.2755  ->  74  |  587.3295 Hz  |  D4
ev. 3: 0.5970  ->  84  |  1046.5023 Hz  |  C5
ev. 4: 0.1093  ->  69  |  440.0000 Hz  |  A3
ev. 5: -0.1142  ->  63  |  311.1270 Hz  |  D#3
ev. 6: -0.1332  ->  62  |  293.6648 Hz  |  D3
ev. 7: -0.6782  ->  46  |  116.5409 Hz  |  A#1
ev. 8: -0.2095  ->  60  |  261.6256 Hz  |  C3
ev. 9: -0.5310  ->  50  |  146.8324 Hz  |  D2
...

initial conditions d1:
in. -4: -0.1742  ->  61  |  277.183 Hz  |  C#3
in. -3: -0.3265  ->  56  |  207.652 Hz  |  G#2
```

```
in. -2: -0.7603  ->  43  |  97.999 Hz  |  G1
in. -1: -0.1980  ->  60  |  261.626 Hz  |  C3
generated score:
ev. 0: -0.5531  ->  49  |  138.5913 Hz  |  C#2
ev. 1: 0.1615  ->  71  |  493.8833 Hz  |  B3
ev. 2: 0.2402  ->  73  |  554.3653 Hz  |  C#4
ev. 3: 0.3925  ->  78  |  739.9888 Hz  |  F#4
ev. 4: 0.8263  ->  91  |  1567.9817 Hz  |  G5
ev. 5: 0.2640  ->  74  |  587.3295 Hz  |  D4
ev. 6: 0.6191  ->  85  |  1108.7305 Hz  |  C#5
ev. 7: -0.0955  ->  63  |  311.1270 Hz  |  D#3
ev. 8: -0.1742  ->  61  |  277.1826 Hz  |  C#3
ev. 9: -0.3265  ->  56  |  207.6523 Hz  |  G#2
...

initial conditions d2:
in. -4: -0.0800  ->  64  |  329.628 Hz  |  E3
in. -3: -0.2800  ->  58  |  233.082 Hz  |  A#2
in. -2: -0.7600  ->  43  |  97.999 Hz  |  G1
in. -1: -0.1900  ->  60  |  261.626 Hz  |  C3
generated score:
ev. 0: -0.6470  ->  47  |  123.4708 Hz  |  B1
ev. 1: 0.1230  ->  70  |  466.1638 Hz  |  A#3
ev. 2: 0.1460  ->  70  |  466.1638 Hz  |  A#3
ev. 3: 0.3460  ->  76  |  659.2551 Hz  |  E4
ev. 4: 0.8260  ->  91  |  1567.9817 Hz  |  G5
ev. 5: 0.2560  ->  74  |  587.3295 Hz  |  D4
ev. 6: 0.7130  ->  87  |  1244.5079 Hz  |  D#5
ev. 7: -0.0570  ->  64  |  329.6276 Hz  |  E3
ev. 8: -0.0800  ->  64  |  329.6276 Hz  |  E3
ev. 9: -0.2800  ->  58  |  233.0819 Hz  |  A#2
...

initial conditions d3:
in. -4: -0.0800  ->  64  |  329.628 Hz  |  E3
in. -3: -0.2480  ->  59  |  246.942 Hz  |  B2
in. -2: -0.7300  ->  44  |  103.826 Hz  |  G#1
in. -1: -0.2200  ->  59  |  246.942 Hz  |  B2
generated score:
ev. 0: -0.6170  ->  47  |  123.4708 Hz  |  B1
ev. 1: 0.0610  ->  68  |  415.3047 Hz  |  G#3
ev. 2: 0.1460  ->  70  |  466.1638 Hz  |  A#3
ev. 3: 0.3140  ->  75  |  622.2540 Hz  |  D#4
ev. 4: 0.7960  ->  90  |  1479.9777 Hz  |  F#5
ev. 5: 0.2860  ->  75  |  622.2540 Hz  |  D#4
ev. 6: 0.6830  ->  86  |  1174.6591 Hz  |  D5
ev. 7: 0.0050  ->  66  |  369.9944 Hz  |  F#3
ev. 8: -0.0800  ->  64  |  329.6276 Hz  |  E3
ev. 9: -0.2480  ->  59  |  246.9417 Hz  |  B2
...

initial conditions d4:
in. -4: -0.0800  ->  64  |  329.628 Hz  |  E3
in. -3: -0.2480  ->  59  |  246.942 Hz  |  B2
in. -2: -0.7300  ->  44  |  103.826 Hz  |  G#1
in. -1: -0.2600  ->  58  |  233.082 Hz  |  A#2
generated score:
ev. 0: -0.6170  ->  47  |  123.4708 Hz  |  B1
ev. 1: 0.0210  ->  67  |  391.9954 Hz  |  G3
ev. 2: 0.1460  ->  70  |  466.1638 Hz  |  A#3
ev. 3: 0.3140  ->  75  |  622.2540 Hz  |  D#4
ev. 4: 0.7960  ->  90  |  1479.9777 Hz  |  F#5
ev. 5: 0.3260  ->  76  |  659.2551 Hz  |  E4
ev. 6: 0.6830  ->  86  |  1174.6591 Hz  |  D5
ev. 7: 0.0450  ->  67  |  391.9954 Hz  |  G3
ev. 8: -0.0800  ->  64  |  329.6276 Hz  |  E3
ev. 9: -0.2480  ->  59  |  246.9417 Hz  |  B2
...

initial conditions d5:
in. -4: -0.0800  ->  64  |  329.628 Hz  |  E3
in. -3: -0.2480  ->  59  |  246.942 Hz  |  B2
in. -2: -0.7300  ->  44  |  103.826 Hz  |  G#1
in. -1: -0.3000  ->  57  |  220.000 Hz  |  A2
generated score:
ev. 0: -0.6170  ->  47  |  123.4708 Hz  |  B1
ev. 1: -0.0190  ->  65  |  349.2282 Hz  |  F3
ev. 2: 0.1460  ->  70  |  466.1638 Hz  |  A#3
ev. 3: 0.3140  ->  75  |  622.2540 Hz  |  D#4
ev. 4: 0.7960  ->  90  |  1479.9777 Hz  |  F#5
ev. 5: 0.3660  ->  77  |  698.4565 Hz  |  F4
ev. 6: 0.6830  ->  86  |  1174.6591 Hz  |  D5
```

```
ev. 7: 0.0850  ->  69  |  440.0000 Hz  |  A3
ev. 8: -0.0800  ->  64  |  329.6276 Hz  |  E3
ev. 9: -0.2480  ->  59  |  246.9417 Hz  |  B2
...

initial conditions d6:
in. -4: -0.0820  ->  64  |  329.628 Hz  |  E3
in. -3: -0.2640  ->  58  |  233.082 Hz  |  A#2
in. -2: -0.7630  ->  43  |  97.999 Hz  |  G1
in. -1: -0.3054  ->  57  |  220.000 Hz  |  A2
generated score:
ev. 0: -0.6480  ->  47  |  123.4708 Hz  |  B1
ev. 1: -0.0084  ->  66  |  369.9944 Hz  |  F#3
ev. 2: 0.1480  ->  70  |  466.1638 Hz  |  A#3
ev. 3: 0.3300  ->  76  |  659.2551 Hz  |  E4
ev. 4: 0.8290  ->  91  |  1567.9817 Hz  |  G5
ev. 5: 0.3714  ->  77  |  698.4565 Hz  |  F4
ev. 6: 0.7140  ->  87  |  1244.5079 Hz  |  D#5
ev. 7: 0.0744  ->  68  |  415.3047 Hz  |  G#3
ev. 8: -0.0820  ->  64  |  329.6276 Hz  |  E3
ev. 9: -0.2640  ->  58  |  233.0819 Hz  |  A#2
...
```

$$n_0 = n_{-2} - n_{-4} - \sin(0{,}033)$$

Condiciones iniciales:

Recursio XIII-a
$n_{-4} = -0{,}3047$
$n_{-3} = -0{,}6255$
$n_{-2} = -0{,}6307$
$n_{-1} = -0{,}4024$

Recursio XIII-b
$n_{-4} = 0{,}6362$
$n_{-3} = -0{,}3537$
$n_{-2} = 0{,}3535$
$n_{-1} = -0{,}4419$

Recursio XIII-d1
$n_{-4} = -0{,}1742$
$n_{-3} = -0{,}3265$
$n_{-2} = -0{,}7603$
$n_{-1} = -0{,}198$

Recursio XIII-d3
$n_{-4} = -0{,}08$
$n_{-3} = -0{,}248$
$n_{-2} = -0{,}73$
$n_{-1} = -0{,}22$

Intervalo de conversión:
$[-1, 1] \longrightarrow [36, 96]$

Función de conversión a alturas MIDI:
$n = 30n_0 + 66$

Recursio XIII-c
$n_{-4} = -0{,}2095$
$n_{-3} = -0{,}531$
$n_{-2} = -0{,}0433$
$n_{-1} = 0{,}1802$

Recursio XIII-d2
$n_{-4} = -0{,}08$
$n_{-3} = -0{,}28$
$n_{-2} = -0{,}76$
$n_{-1} = -0{,}19$

Recursio XIII-d4
$n_{-4} = -0{,}08$
$n_{-3} = -0{,}248$
$n_{-2} = -0{,}73$
$n_{-1} = -0{,}26$

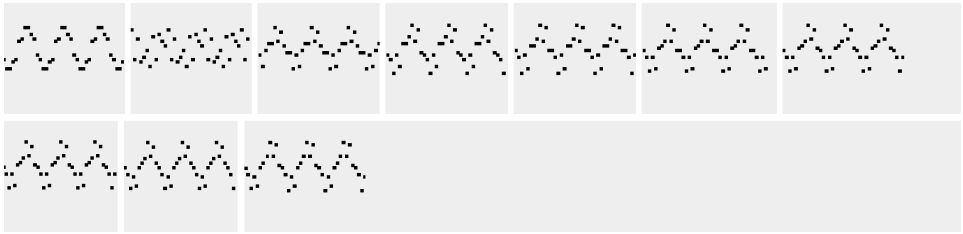
Recursio XIII-d5
$n_{-4} = -0{,}08$
$n_{-3} = -0{,}248$
$n_{-2} = -0{,}73$
$n_{-1} = -0{,}3$

Recursio XIII-d6
$n_{-4} = -0{,}082$
$n_{-3} = -0{,}264$
$n_{-2} = -0{,}763$
$n_{-1} = -0{,}3054$

Fenotipos

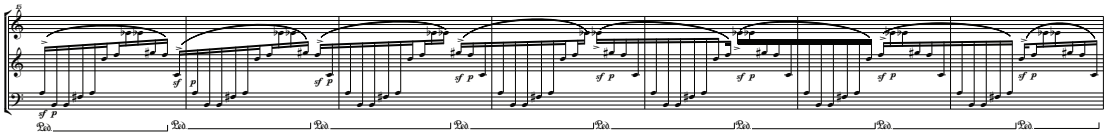
MIDI ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-a.mid
MIDI ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-b.mid
MIDI ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-c.mid
MIDI ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-d1.mid
MIDI ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-d2.mid
MIDI ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-d3.mid
MIDI ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-d4.mid
MIDI ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-d5.mid
MIDI ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-d6.mid
txt ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-a.txt
txt ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-b.txt
txt ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-c.txt
txt ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-d1.txt
txt ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-d2.txt
txt ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-d3.txt
txt ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-d4.txt
txt ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-d5.txt
txt ⟶ http://www.lopezmontes.es/obra/threnody/recursio_XIII-d6.txt

Gráfico de la secuencia de alturas

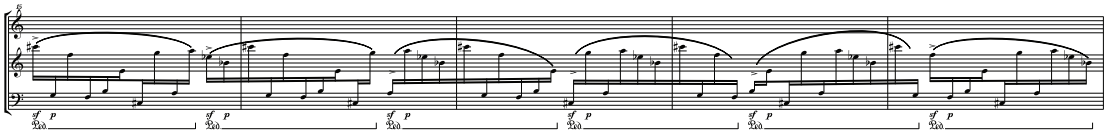


Transcripción

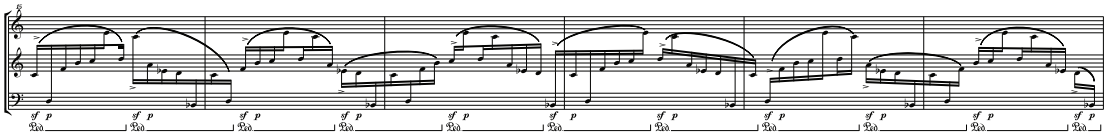
Recursio XIII-a



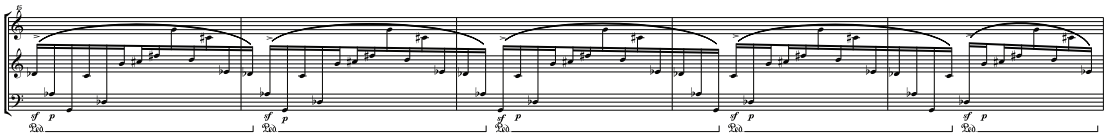
Recursio XIII-b



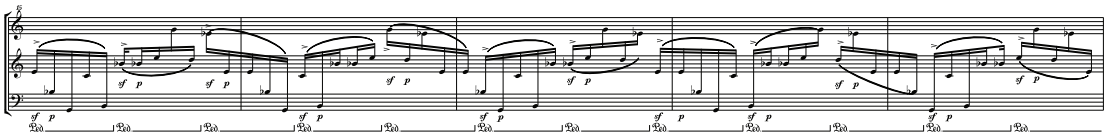
Recursio XIII-c



Recursio XIII-d1

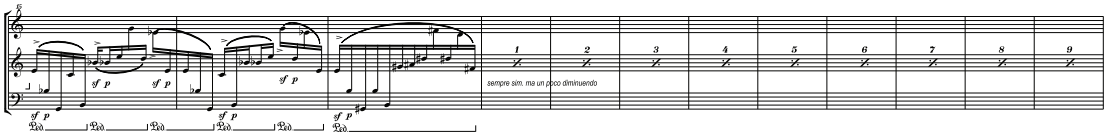


Recursio XIII-d2

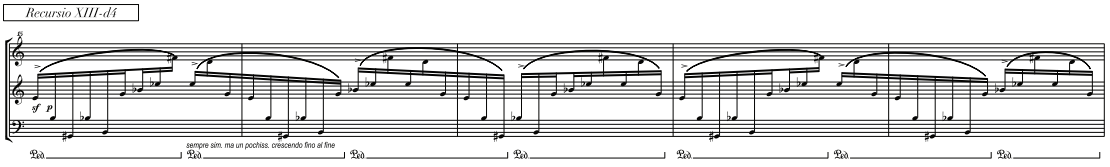


16

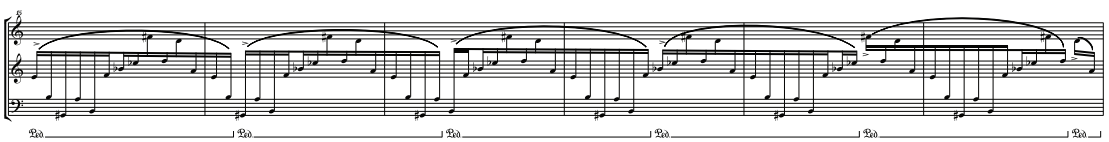
Recursio XIII-d3



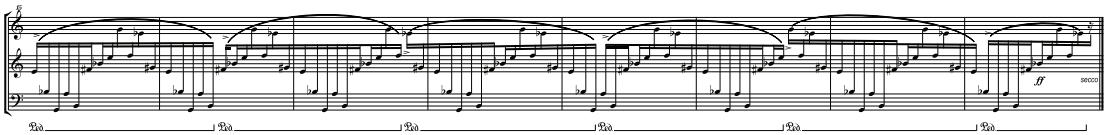
Recursio XIII-d4



Recursio XIII-d5



Recursio XIII-d6



Anexo C

Ada + Babbage — Capricci

Encargo de Trino Zurita

Instrumentación:
- violonchelo
- piano

Duración: 18 min.
Período de composición: enero 2013 — marzo 2013
Estreno: 15 marzo 2013, Conservatorio Superior de Música "Rafael Orozco" de Córdoba
Intérpretes:
- Trino Zurita, violonchelo
- Óscar Martín, piano

Página web de la obra: http:www.lopezmontes.es/obra/ada+babbage.html

C.1 Concepto

Ada + Babbage — Capricci es un dúo para violonchelo y piano. La obra está integrada por 16 caprichos sobre los nombres de Babbage y Ada,[68] pioneros de la moderna computación automatizada a mediados del siglo XIX. Charles Babbage fue célebre por sus ingenios mecánicos capaces de ejecutar complejos cálculos, al tiempo que Ada Lovelace ideó lo que luego serían los lenguajes de programación, y fue descrita a menudo como una virtuosa intérprete de las máquinas de Babbage. La obra no es en primera línea un homenaje, sino que la referencia a estos científicos se plasma en una analogía con los procedimientos de *GenoMus*: mientras el papel de Babbage consistió en la concepción de sistemas de gran potencialidad, Ada fue quien supo comprender y formalizar la manera de proceder para extraer brillantes resultados de estas máquinas.[69] La similitud entre el diseñador de un instrumento musical y el intérprete virtuoso es directa. En el caso de *GenoMus*, la máquina es el metalenguaje y sus posibles manipulaciones, y las decisiones y la habilidad del compositor los factores decisivos para conseguir resultados artísticos de valor.

Los nombres de Babbage y Ada son musicalizados y usados como material constructivo para la producción de los genotipos empleados. Se crearon una serie de funciones específicas para los genotipos de esta pieza. Partiendo de la función `scoNameADA` y `scoNameBABBAGE`, que devuelven respectivamente las notas LA–RE–LA y SI♭–LA–SI♭–SI♭–LA–SOL–MI, se derivan un grupo de funciones (identificables por usar también sus nombre propios) que se integraron en la librería de funciones estándar de *GenoMus*. Los genotipos de cada *capriccio* realizan muy diversas variaciones con estas células motívicas, que en muchos casos son fácilmente identificables en la transcripción en partitura. Los criterios de musicalización de las secuencias de alturas han sido muy cercanas a las empleadas antes en *Threnody for Dimitris Christoulas*.

Ada + Babbage — Capricci está compuesta con la versión de *GenoMus* documentada en el capítulo 6.

[68] La potencialidad musical de estos nombres fue ya utilizada por Hofstadter [24], quien juega con las iniciales de BACH en diálogo con las de CHarles BAbbage.

[69] No en vano se la considera la pionera de la programación, y existe un lenguaje de programación que lleva su nombre.

C.2 Genotipos y fenotipos seleccionados

C.2.1 *Capriccio I*

Genotipo

`seed` = 0,12267276066200561

```
scoBABBAGE2voices(
    scoPCGenPrim(),
    300)
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_01.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_01.txt

Gráfico de la secuencia de alturas

C.2.2 *Capriccio II*

Genotipo

`seed` = 0,14049629543007258

```
scoTransp(
    scoADA2voices(
        scoPCGenPrim(),
        300),
    35)
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_02.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_02.txt

Gráfico de la secuencia de alturas

C.2.3 *Capriccio III*

Genotipo

seed = 0,2366373293347337

```
scoTranspMatrix(
    scoRender(
        scoNameADA()),
    scoExcerptMulti(
        scoInvert(
            scoExcerpt(
                scoRandFlo(
                    94,
                    25.99744415283203,
                    pyth(
                        55.94232177734375,
                        101)),
                root(
                    randInt(
                        75,
                        54.761192321777344),
                    root(
                        94,
                        15)),
                sin(
                    50.85978317260742))),
        scoInter(
            [0,0,0])))
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_03.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_03.txt

Gráfico de la secuencia de alturas

C.2.4 *Capriccio IV*

Genotipo

seed = 0,06906914511016293

```
scoBABBAGE5voicesHarmonicGlobal(
    scoNameBABBAGE(),
    200)
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_04.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_04.txt

Gráfico de la secuencia de alturas

C.2.5 *Capriccio V*

Genotipo

seed = 0,06906914511016293

```
scoBABBAGE5voicesHarmonicGlobal(
    scoNameBABBAGE(),
    200)
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_05.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_05.txt

Gráfico de la secuencia de alturas

C.2.6 *Capriccio VI*

Genotipo

seed = 0,5024332028236203

```
scoTranspMatrix(
    scoExcerptMulti2(
        scoTranspMatrix(
            scoInvert(
                scoPermut(
                    scoNameADAintervals())),
            scoEnumInt(
                20,
                2)),
        scoNameADA(),
        scoNameADA()),
    scoRender(
        scoNameBABBAGEintervals()))
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_06.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_06.txt

Gráfico de la secuencia de alturas

C.2.7 *Capriccio VII*

Genotipo

seed = 0,6449608976441318

```
scoTransp(
    scoBABBAGE4voicesHarmonic(
        scoNameBABBAGE(),
        40),
    10)
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_07.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_07.txt

Gráfico de la secuencia de alturas

C.2.8 *Capriccio VIII*

Genotipo

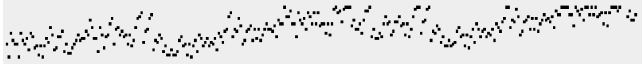
seed = 0,07204728925454051

```
scoBABBAGE5voicesHarmonic(
    scoExcerpt(
        scoExpand(
            scoNameADA(),
            floPrim()),
        intPrim(),
        ratio(
            sin(
                log(
                    floPrim(),
                    floPrim())),
            floPrim())),
    floPrim())
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_08.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_08.txt

Gráfico de la secuencia de alturas



C.2.9 *Capriccio IX*

Genotipo

seed = 0,07755023860351185

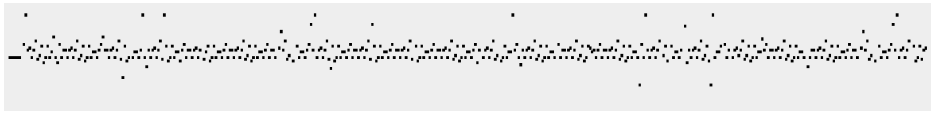
```
scoMutate(
    scoADA3voices(
        scoRep(
            scoPitchClass(
                scoNameBABBAGE(),
```

```
                scoPCGen(
                    intPrim())),
            intPrim()),
        cos(
            intPrim())),
    root(
        absDif(
            floPrim(),
            floPrim()),
        floPrim()),
    randInt(
        intPrim(),
        floPrim()))
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_09.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_09.txt

Gráfico de la secuencia de alturas



C.2.10 *Capriccio X*

Genotipo

seed = 0,010562005465045599

```
scoBABBAGE5voicesHarmonicGlobal(
    scoNameBABBAGE(),
    200)
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_10.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_10.txt

Gráfico de la secuencia de alturas

C.2.11 *Capriccio XI*

Genotipo

seed = 0,6912066419198322

```
scoExpandIter(
    scoRandPrim(),
    dif(
        atan(
            floPrim()),
        floPrim()),
    sqr(
        intPrim()),
    rFlo(
        floPrim(),
        sum(
            intPrim(),
            root(
                intPrim(),
                floPrim()))))
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_11.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_11.txt

Gráfico de la secuencia de alturas

C.2.12 *Capriccio XII*

Genotipo

seed = 0,30938694044888126

```
scoTranspMatrix(
    scoExcerptMulti2(
        scoTranspMatrix(
            scoInvert(
                scoPermut(
                    scoNameADAintervals())),
            scoEnumInt(
                46,
                32)),
        scoNameADA(),
        scoNameADA()),
    scoRender(
        scoNameBABBAGEintervals()))
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_12.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_12.txt

Gráfico de la secuencia de alturas

C.2.13 *Capriccio XIII*

Genotipo

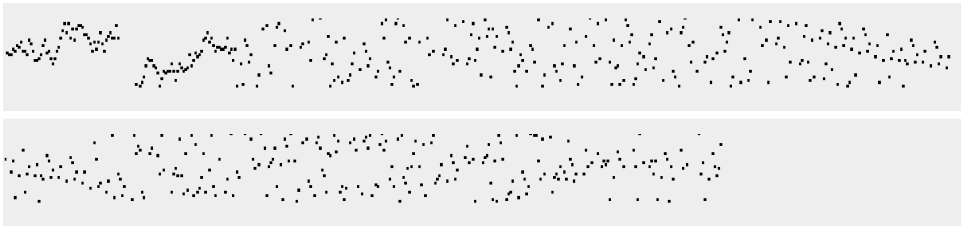
seed = 0,8853635125980808

```
scoMutate(
    scoConcat(
        scoConcat(
            scoADA(
                scoBABBAGE(
                    scoRand(
                        7,
                        60,
                        72),
                    50)),
            scoRetrog(
                scoInter(
                    [0,0,0,0,0]))),
        scoTranspMatrix(
            scoRandPrim(),
            scoInter(
                [0,0,0,1])))
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_13.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_13.txt

Gráfico de la secuencia de alturas



C.2.14 *Capriccio XIV*

Genotipo

seed = 0,6449608976441318

```
scoBABBAGE5voicesHarmonicGlobal(
    scoTransp(
        scoNameBABBAGE(),
        -12),
    40)
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_14.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_14.txt

Gráfico de la secuencia de alturas

C.2.15 *Capriccio XV*

Genotipo

seed = 0,7370028217611009

```
scoRender(
    scoExpandIter(
        scoNameBABBAGE(),
```

```
        1,
        6,
        60))
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_15.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_15.txt

Gráfico de la secuencia de alturas

C.2.16 *Capriccio XVI*

Genotipo

seed = 0,7370028217611009

```
scoBABBAGE5voicesHarmonic(
    scoNameADA(),
    40)
```

Fenotipos

MIDI ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_16.mid
txt ⟶ http://www.lopezmontes.es/obra/ada+babbage/capriccio_16.txt

Gráfico de la secuencia de alturas

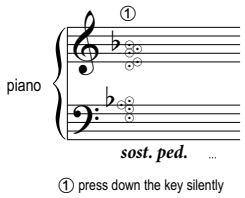
C.3 Partitura completa

to Gerald Bennett,
to answer his question: "Where the notes come from?"

A d a + B a b b a g e — C a p r i c c i

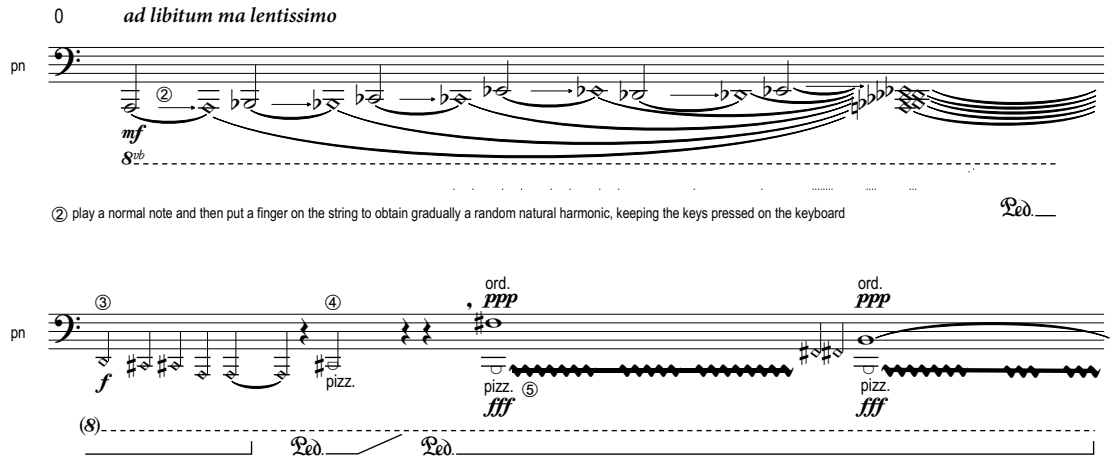
for cello and piano
(accidentals only affect the notes they inmmediately precede)

José López-Montes
(1977)



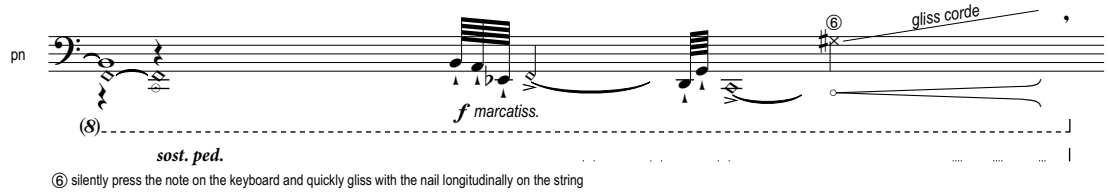
① press down the key silently

Capriccio I

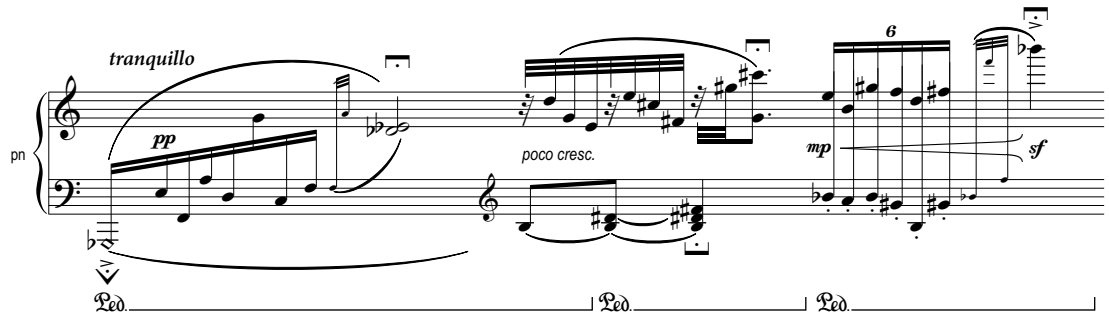


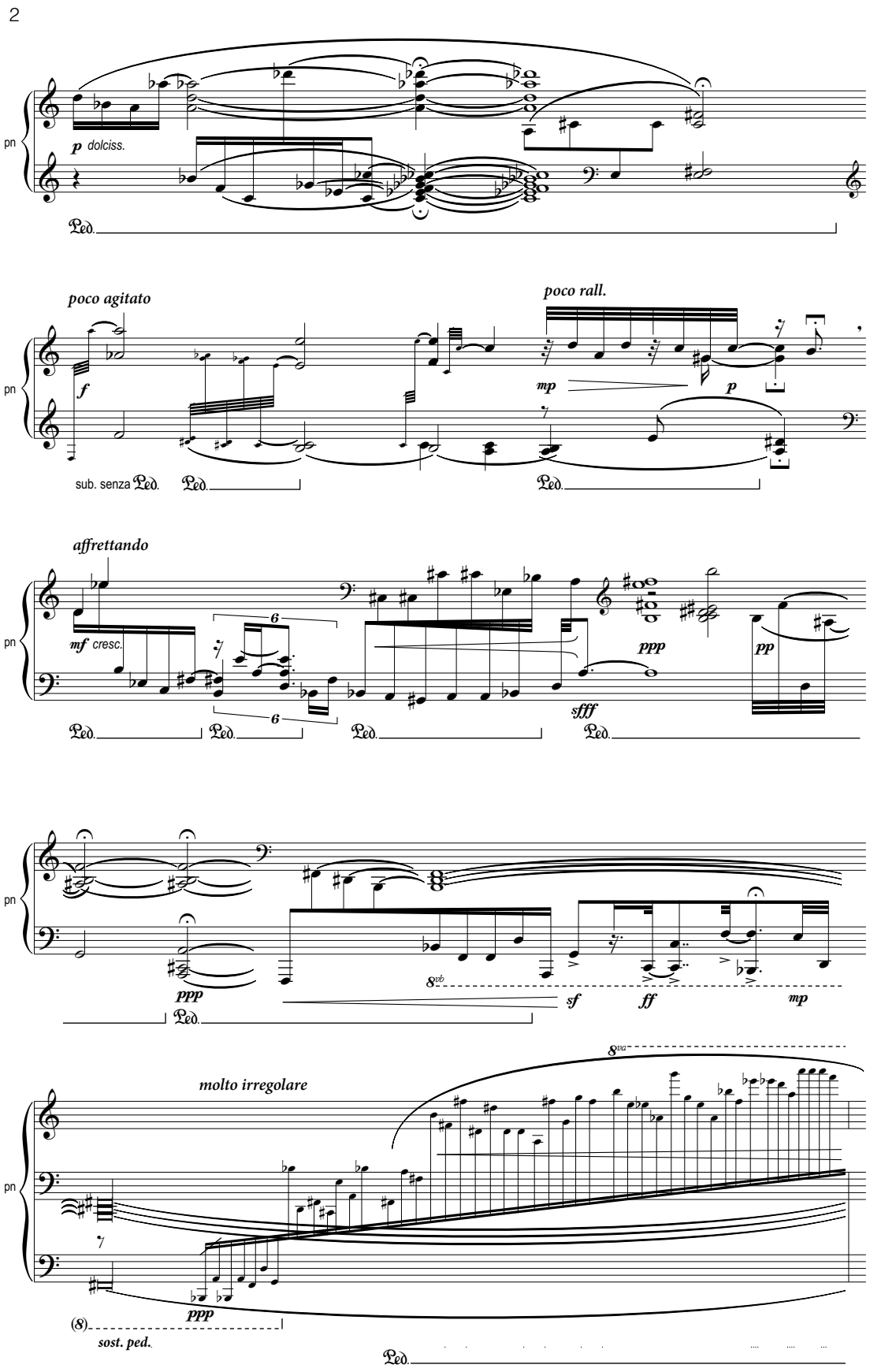
② play a normal note and then put a finger on the string to obtain gradually a random natural harmonic, keeping the keys pressed on the keyboard

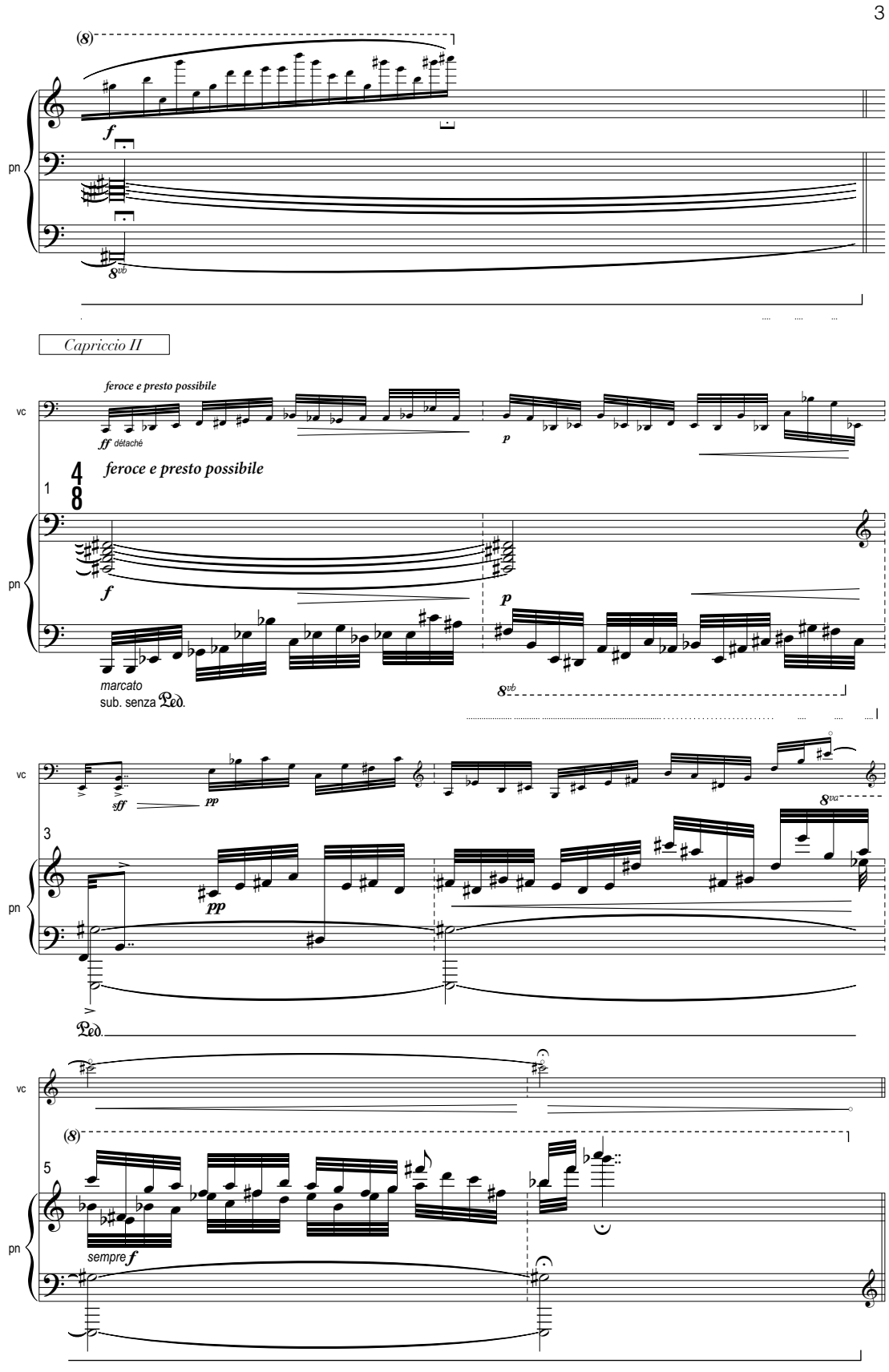
③ random natural harmonic (put the fingertip or a nail somewhere on the string, and press the note on the keyboard)
④ pizz. the string with the finger
⑤ push a string to collide with a contiguous vibrating string, sounding as a percussive thrill

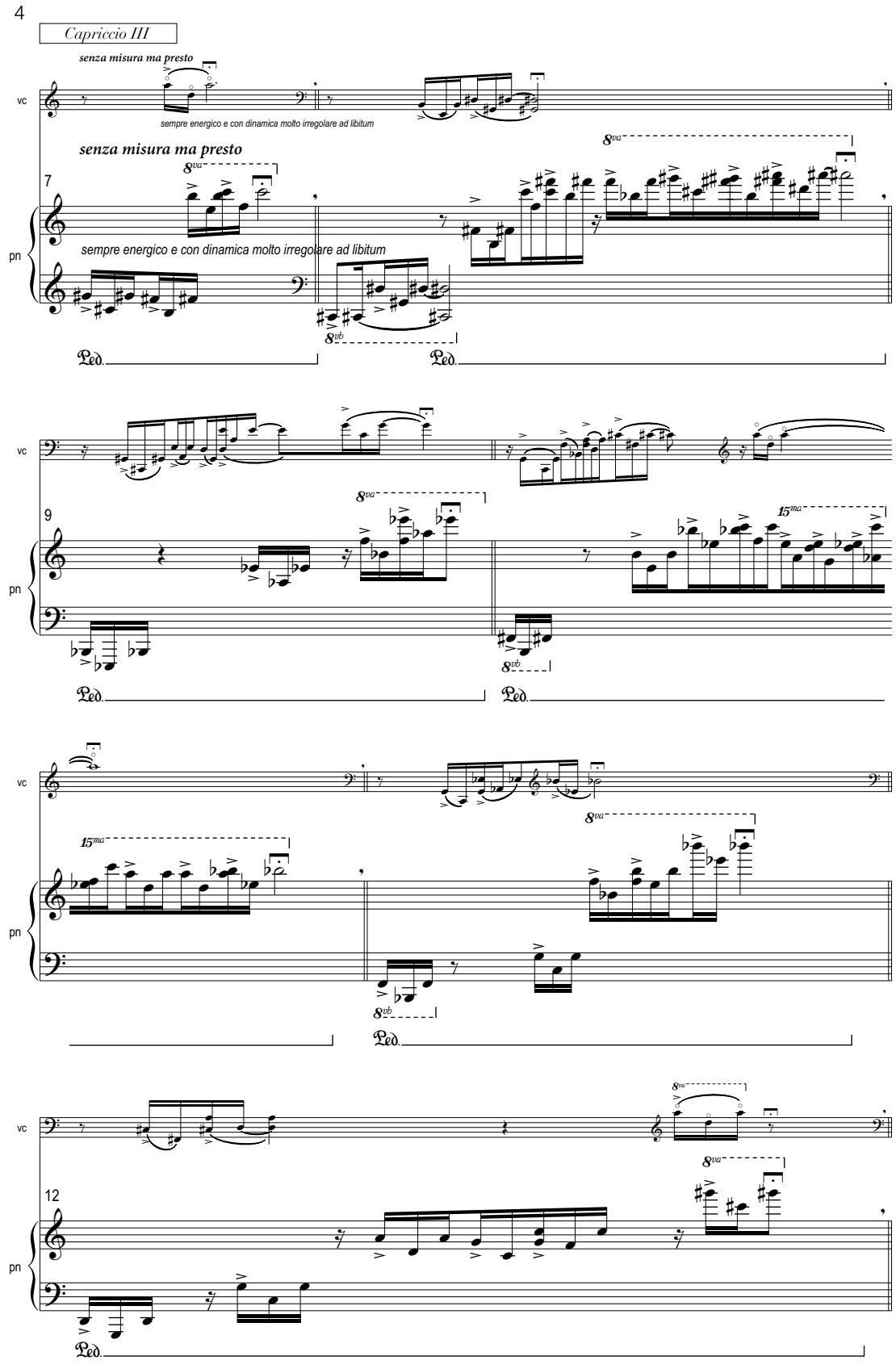


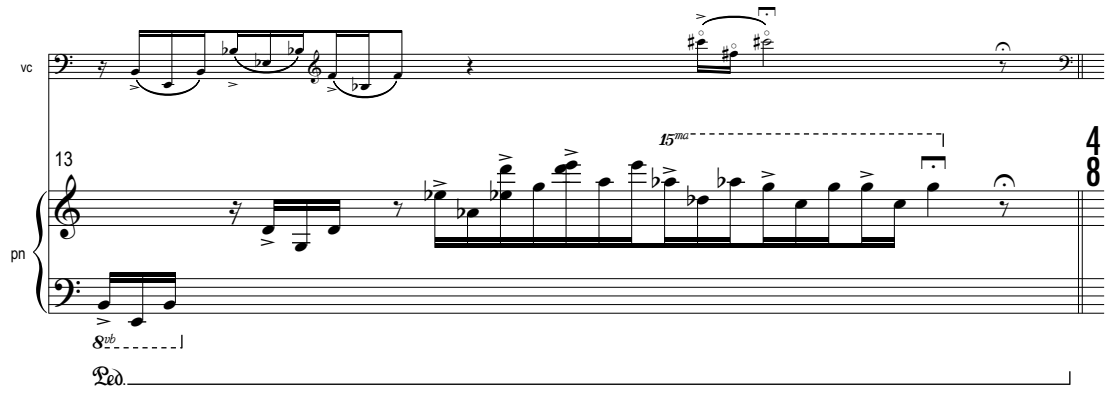
⑥ silently press the note on the keyboard and quickly gliss with the nail longitudinally on the string

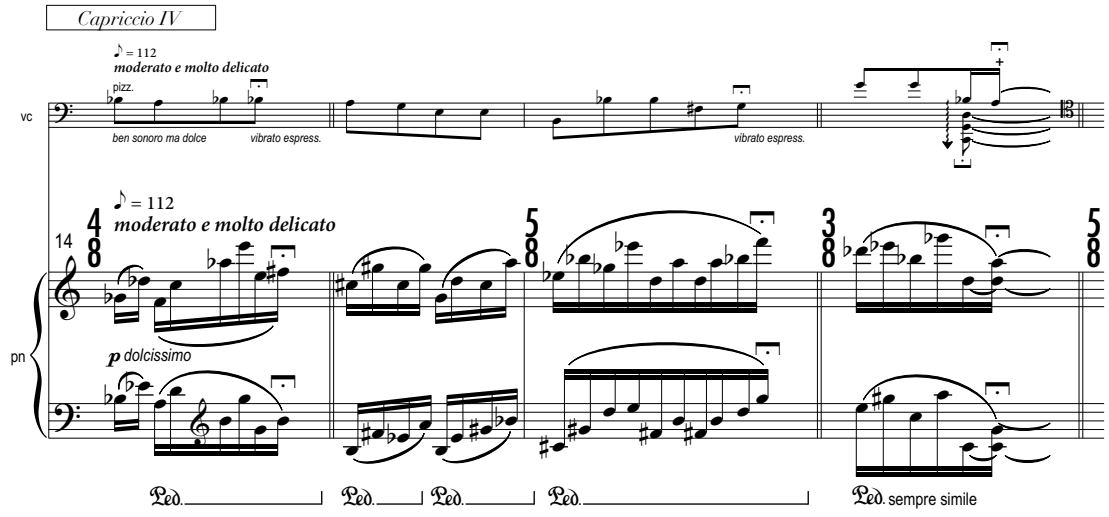


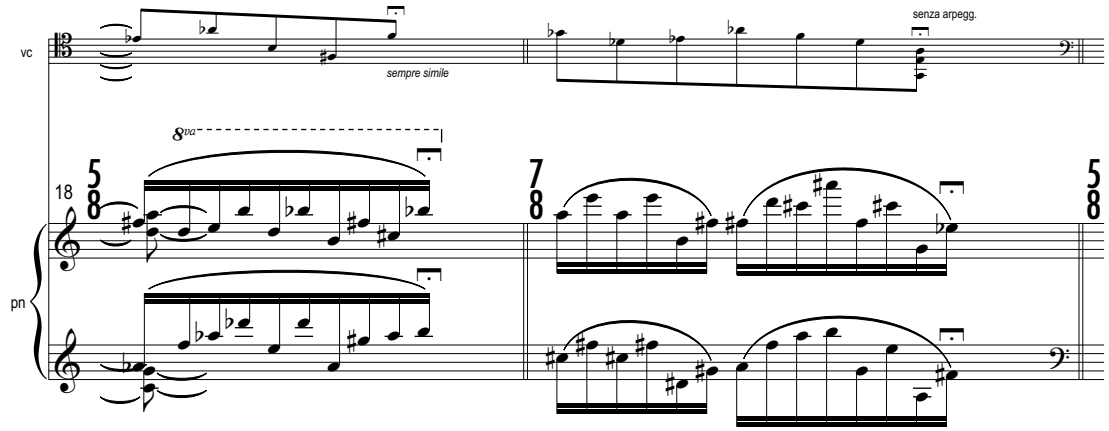

2
pn
p dolciss.
Ped.
poco agitato
poco rall.
f
mp
p
sub. senza Ped.
Ped.
affrettando
mf cresc.
6
ppp
pp
sfff
8va
sf
ff
mp
molto irregolare
(8)
sost. ped.


3
pn
Capriccio II
feroce e presto possibile
vc
ff détaché
p
feroce e presto possibile
1
4
8
f
p
marcato
sub. senza Ped.
8vb
sff
pp
8va
3
Ped.
sempre f
5

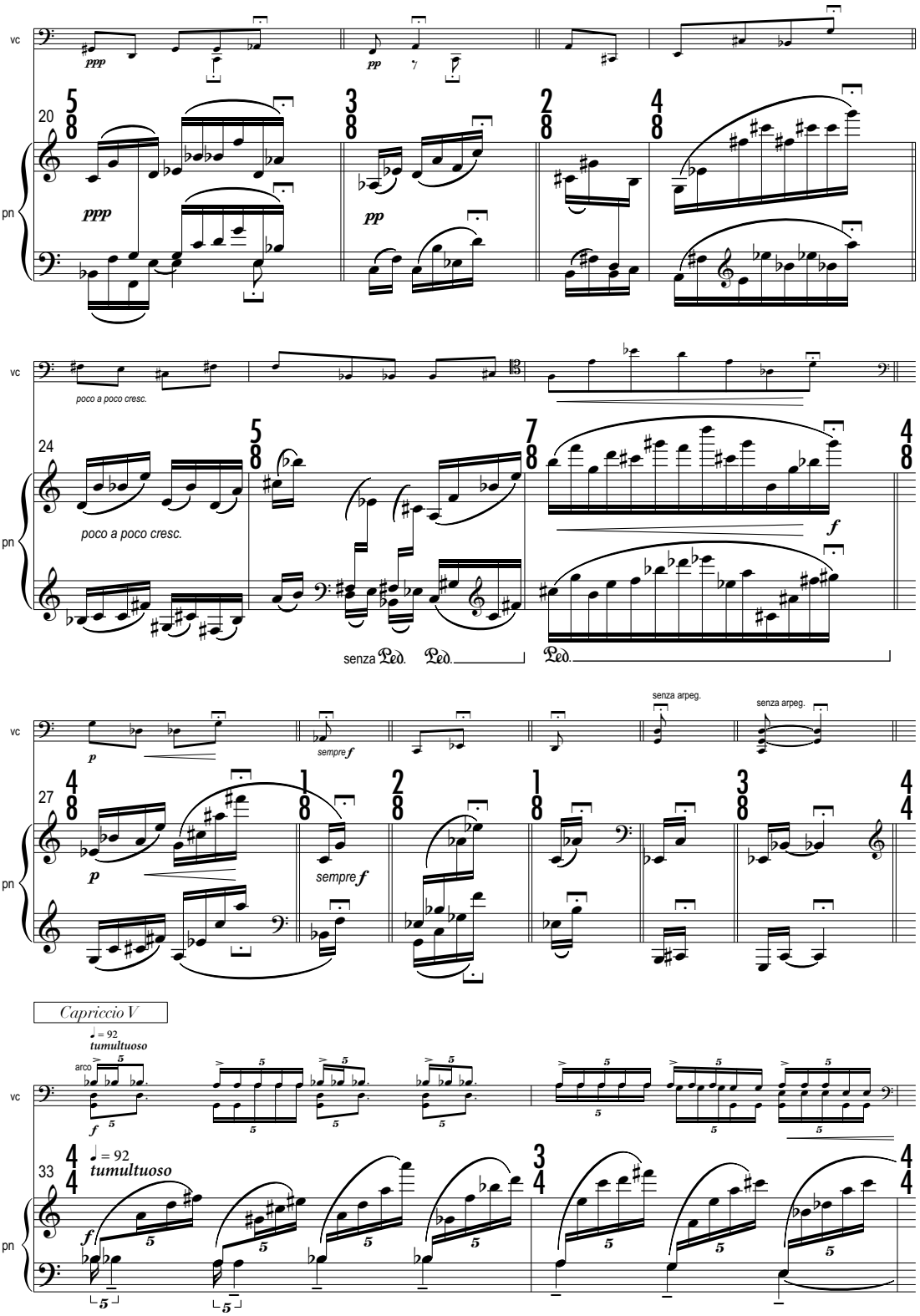

4
Capriccio III
senza misura ma presto
vc
sempre energico e con dinamica molto irregolare ad libitum
senza misura ma presto
7
pn
sempre energico e con dinamica molto irregolare ad libitum
Ped.
Ped.
vc
9
pn
Ped.
Ped.
vc
pn
Ped.
vc
12
pn
Ped.

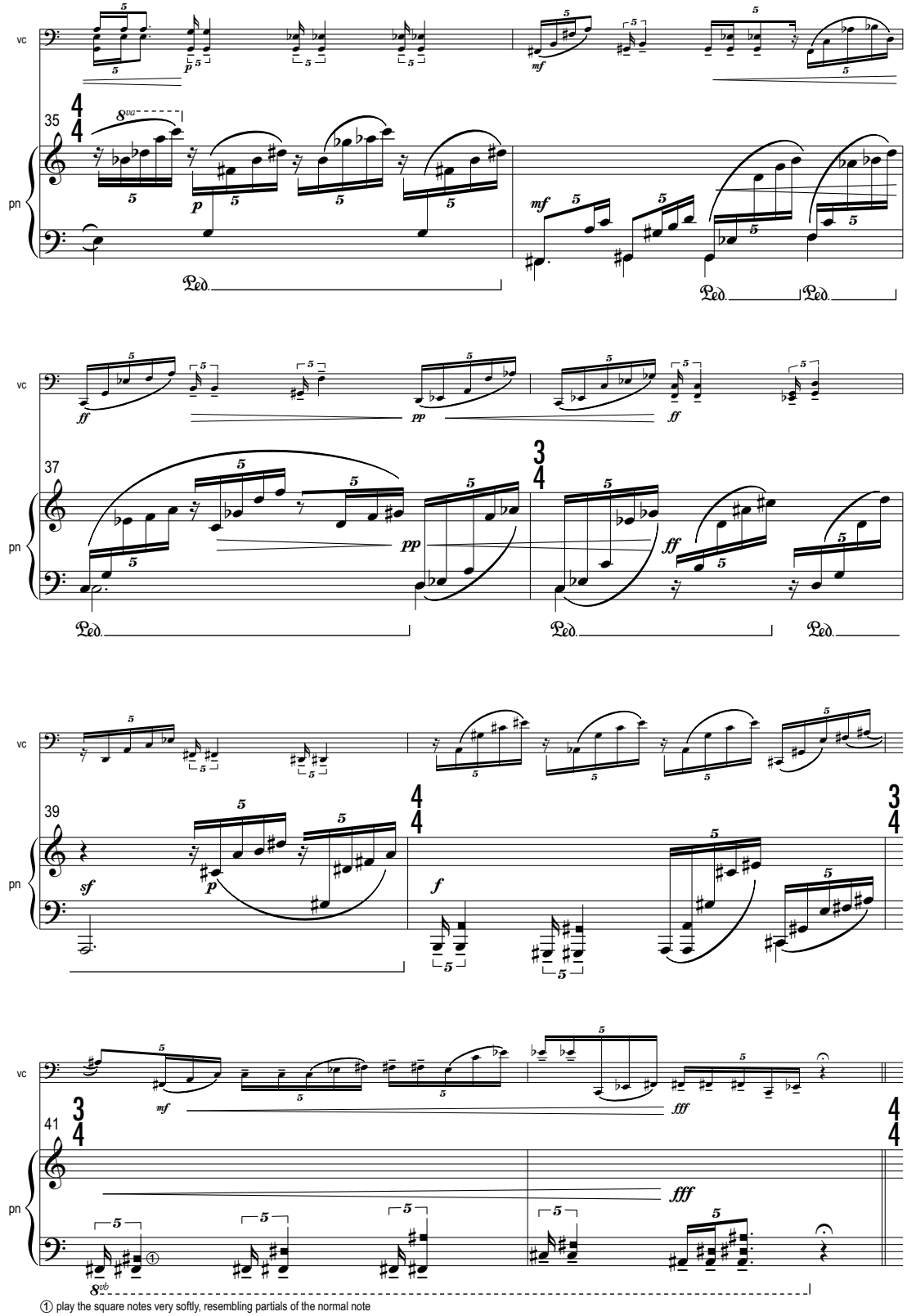

5
vc
13
15ma
pn
8vb
Ped.

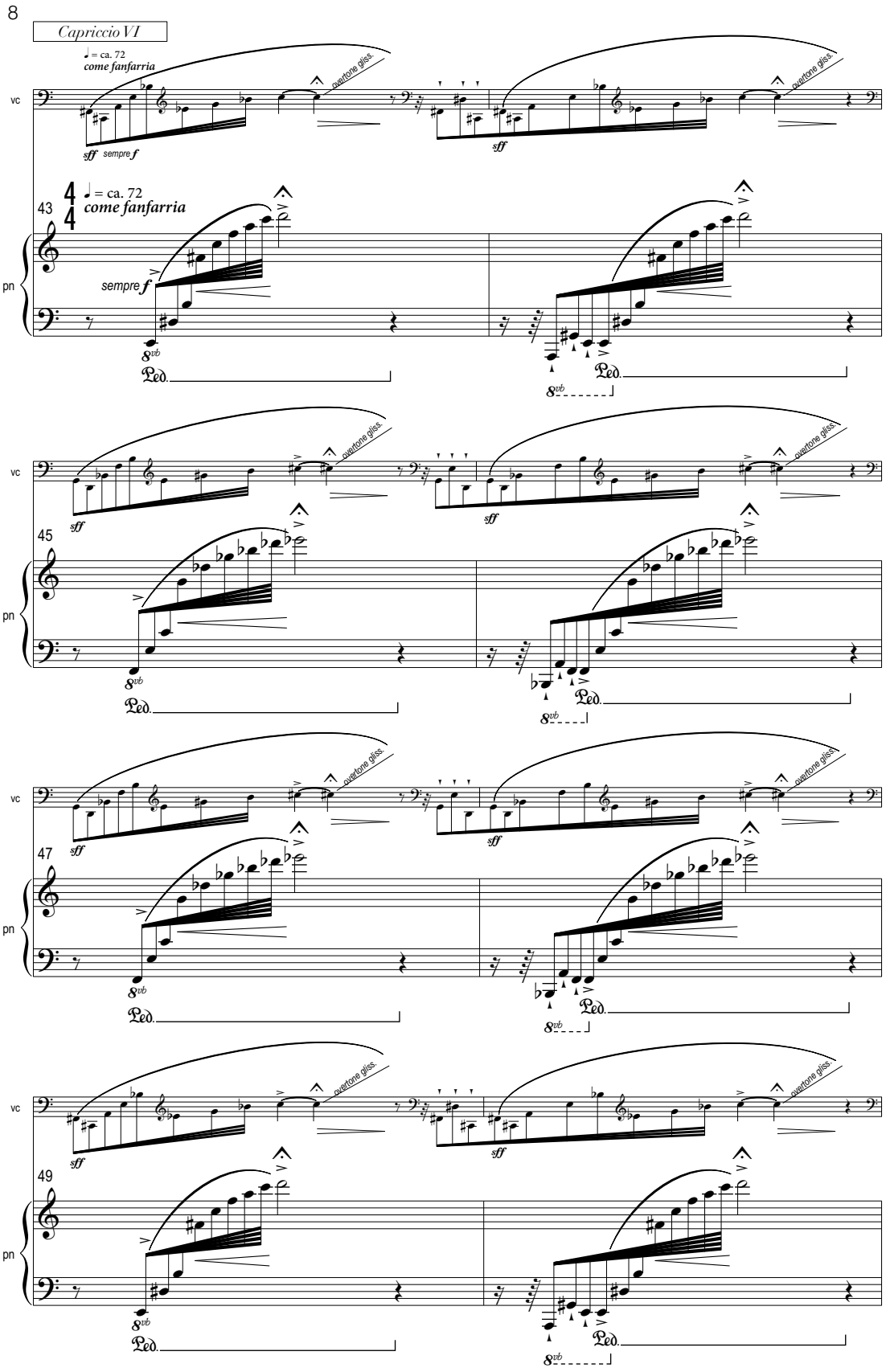

Capriccio IV
♪ = 112
moderato e molto delicato
pizz.
vc
ben sonoro ma dolce
vibrato espress.
vibrato espress.
♪ = 112
moderato e molto delicato
14
pn
p dolcissimo
Ped.
Ped.
Ped.
Ped.
Ped. sempre simile

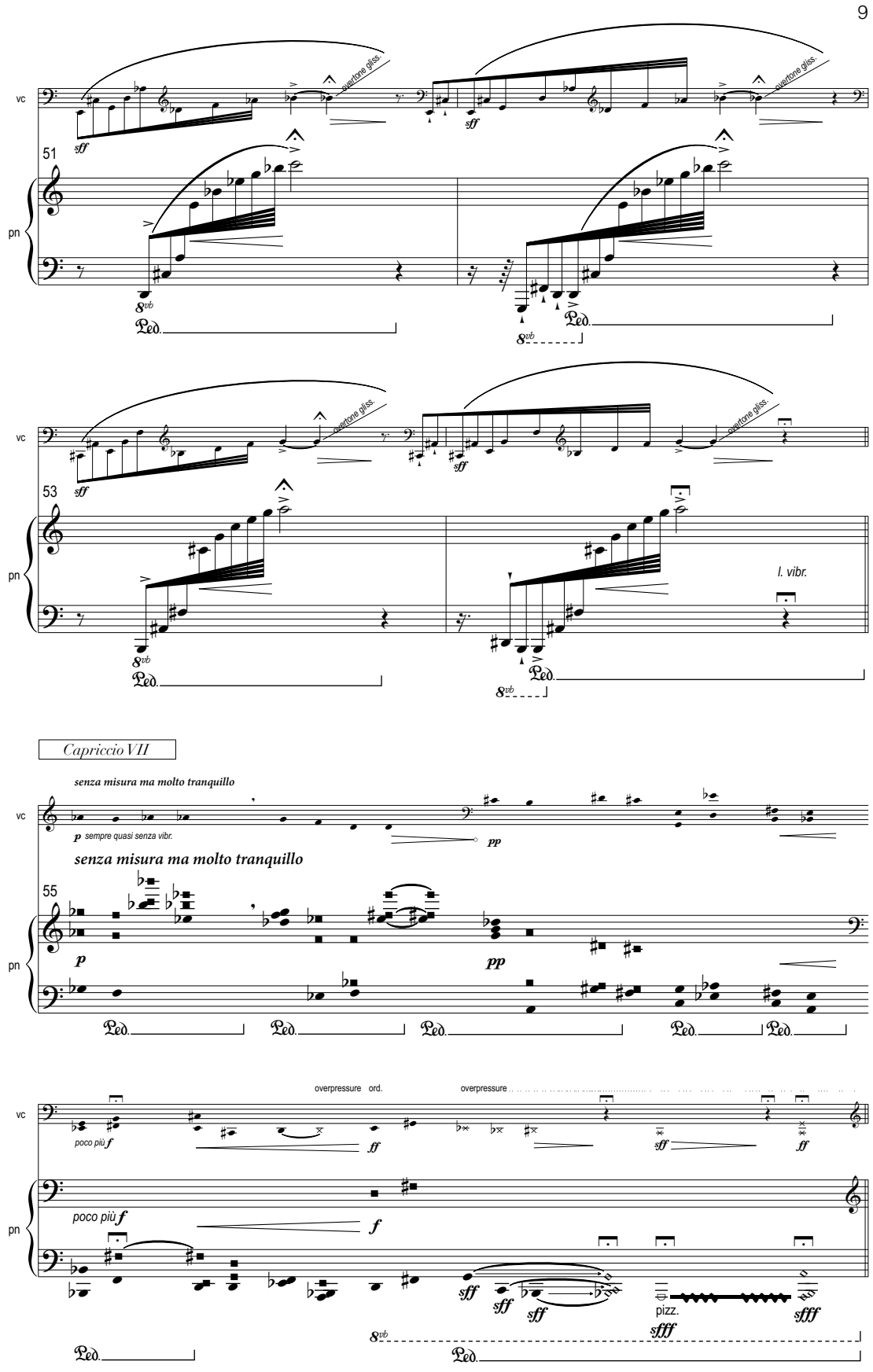

vc
sempre simile
senza arpegg.
8va
18
pn

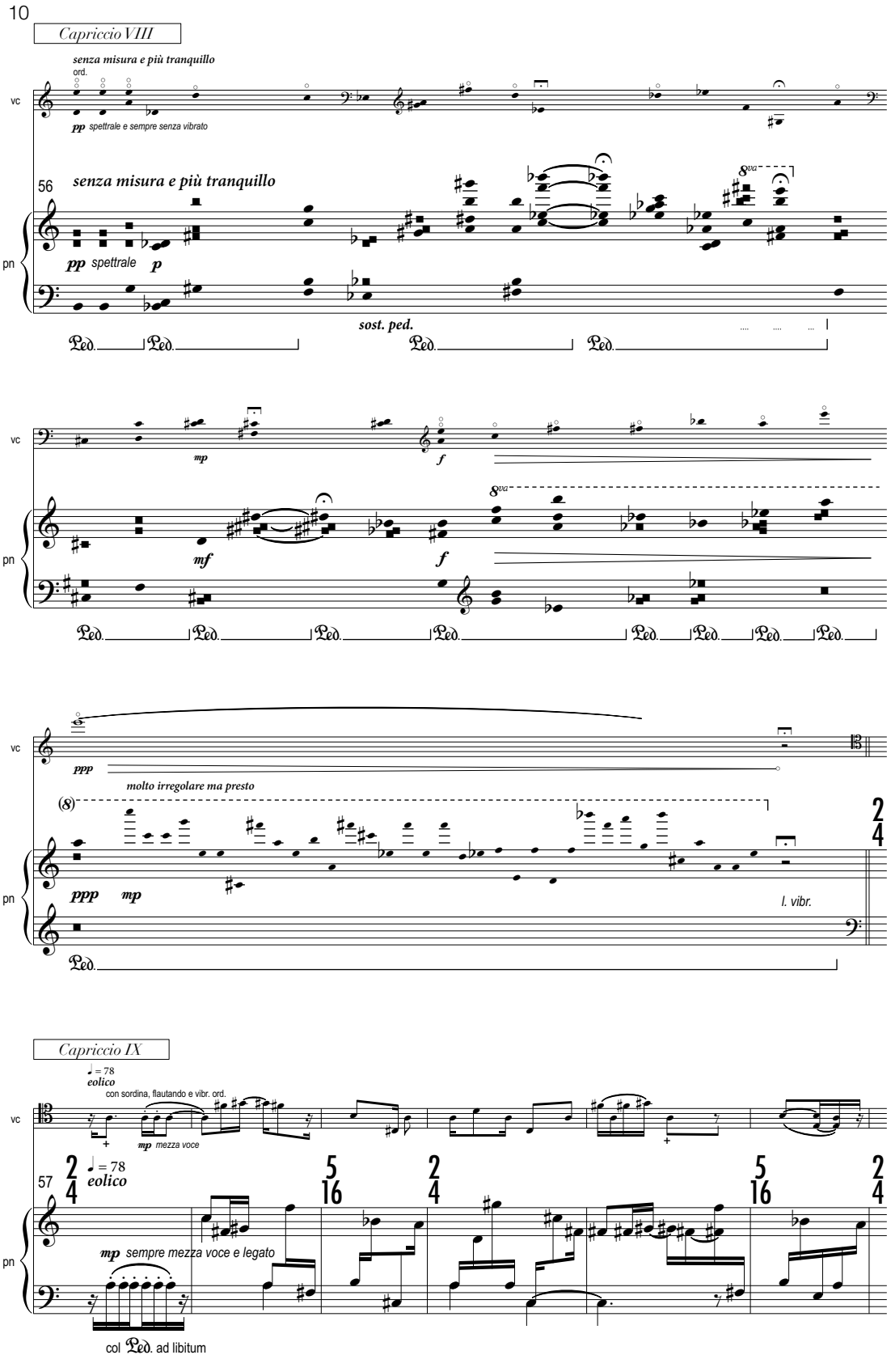
6

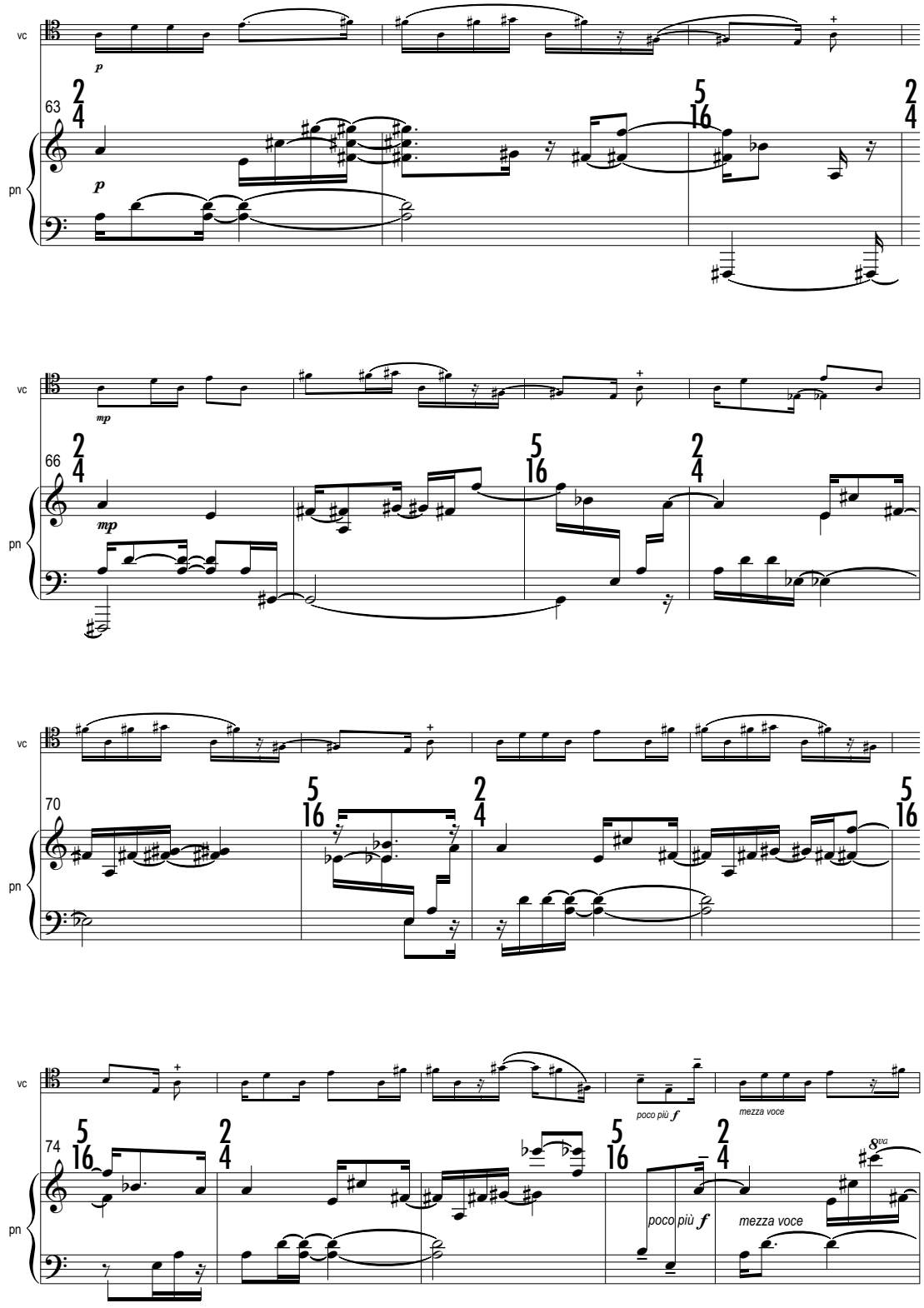

vc
ppp
pp
20
pn
ppp
pp
vc
poco a poco cresc.
24
pn
poco a poco cresc.
f
senza Ped. Ped. Ped.
vc
p
sempre f
senza arpeg.
senza arpeg.
27
pn
p
sempre f
Capriccio V
♩ = 92
tumultuoso
vc
arco
f
33
pn
♩ = 92
tumultuoso
f


vc
35
pn
Ped.
37
39
41
① play the square notes very softly, resembling partials of the normal note

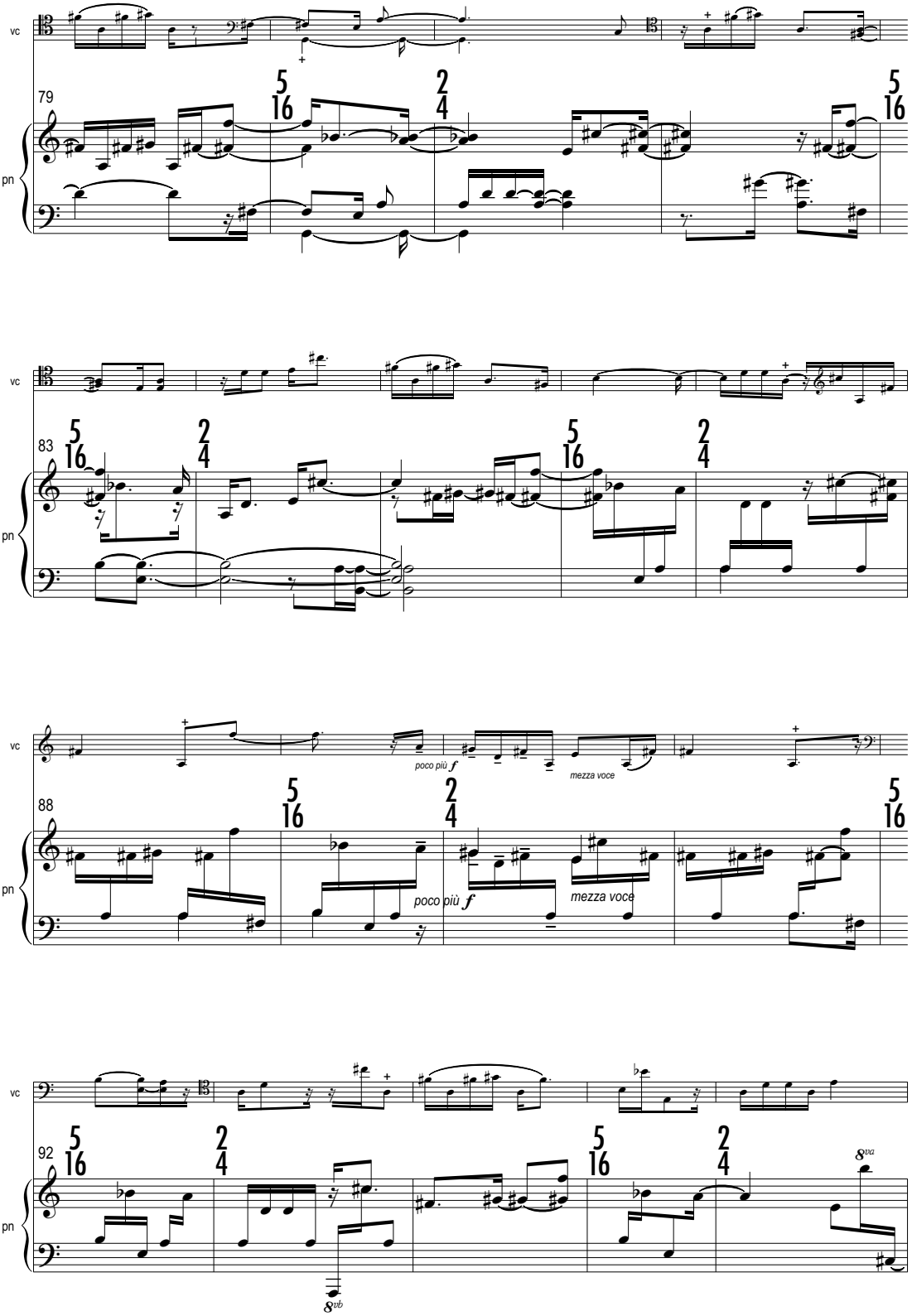

8
Capriccio VI
♩ = ca. 72
come fanfarria
vc
overtone gliss.
sff sempre f
43
4/4
♩ = ca. 72
come fanfarria
pn
sempre f
8vb
Ped.
45
47
49

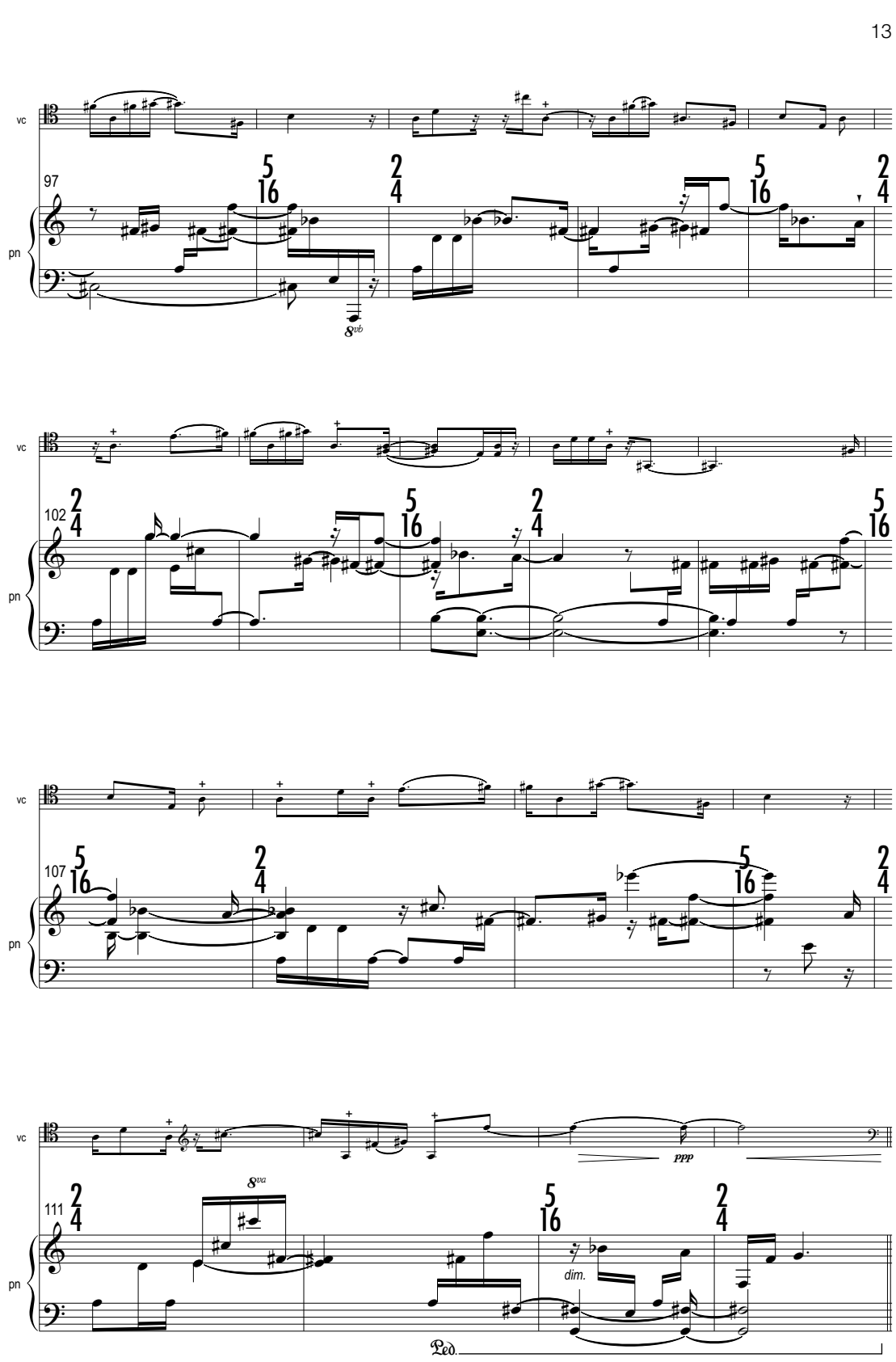

9
vc
pn
51
overtone gliss.
sff
8vb
Ped.
53
l. vibr.
Capriccio VII
senza misura ma molto tranquillo
p sempre quasi senza vibr.
pp
55
overpressure ord.
overpressure
poco più f
ff
sff
pizz.
sfff
sffff

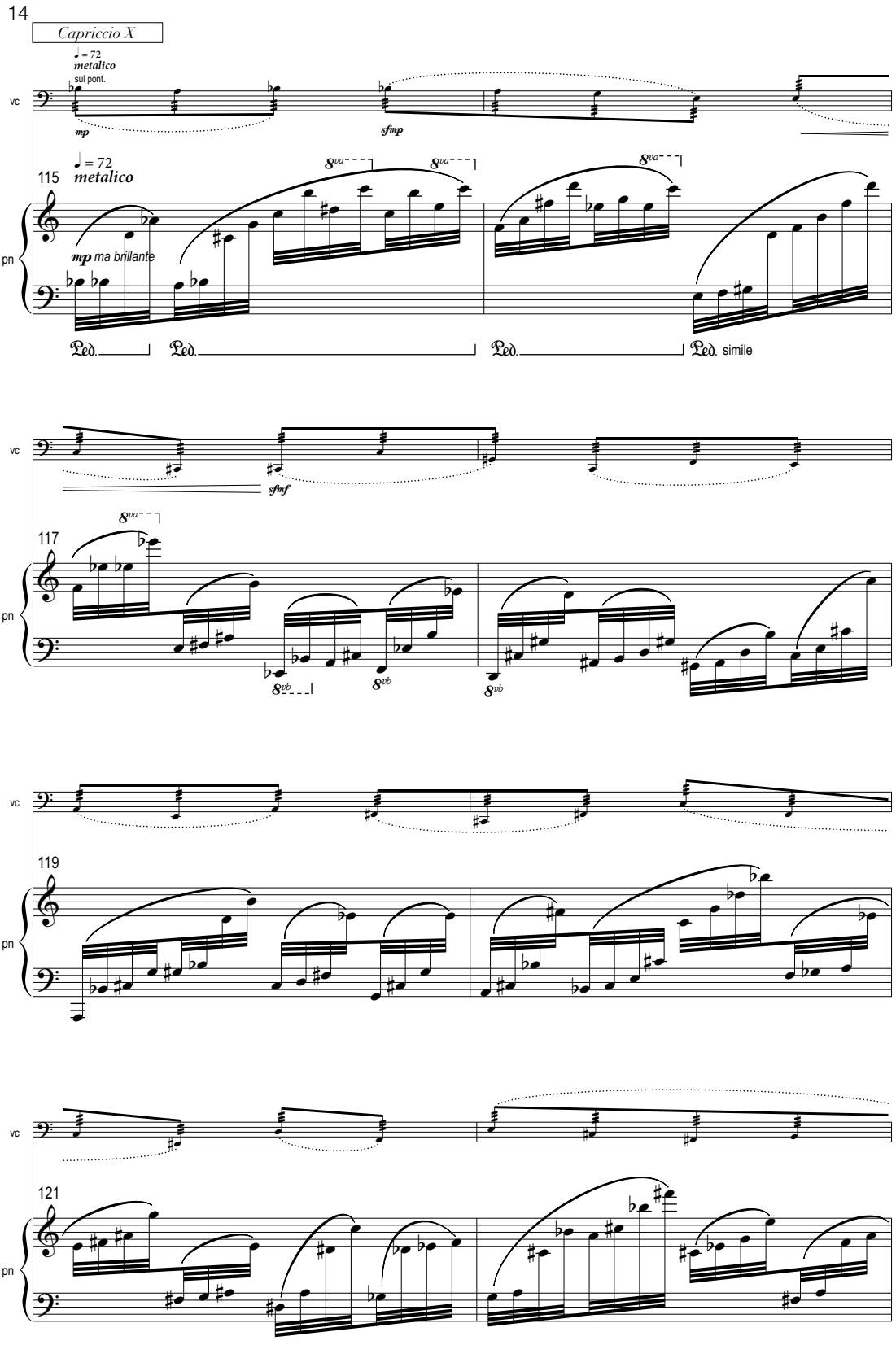

10
Capriccio VIII
senza misura e più tranquillo
ord.
vc
pp spettrale e sempre senza vibrato
56
senza misura e più tranquillo
8va
pn
pp spettrale p
sost. ped.
Ped. Ped. Ped. Ped.
mp
f
mf
f
8va
Ped. Ped. Ped. Ped. Ped. Ped. Ped. Ped.
ppp
molto irregolare ma presto
(8)
ppp mp
l. vibr.
Ped.
Capriccio IX
♩ = 78
eolico
con sordina, flautando e vibr. ord.
mp mezza voce
57
♩ = 78
eolico
mp sempre mezza voce e legato
col Ped. ad libitum

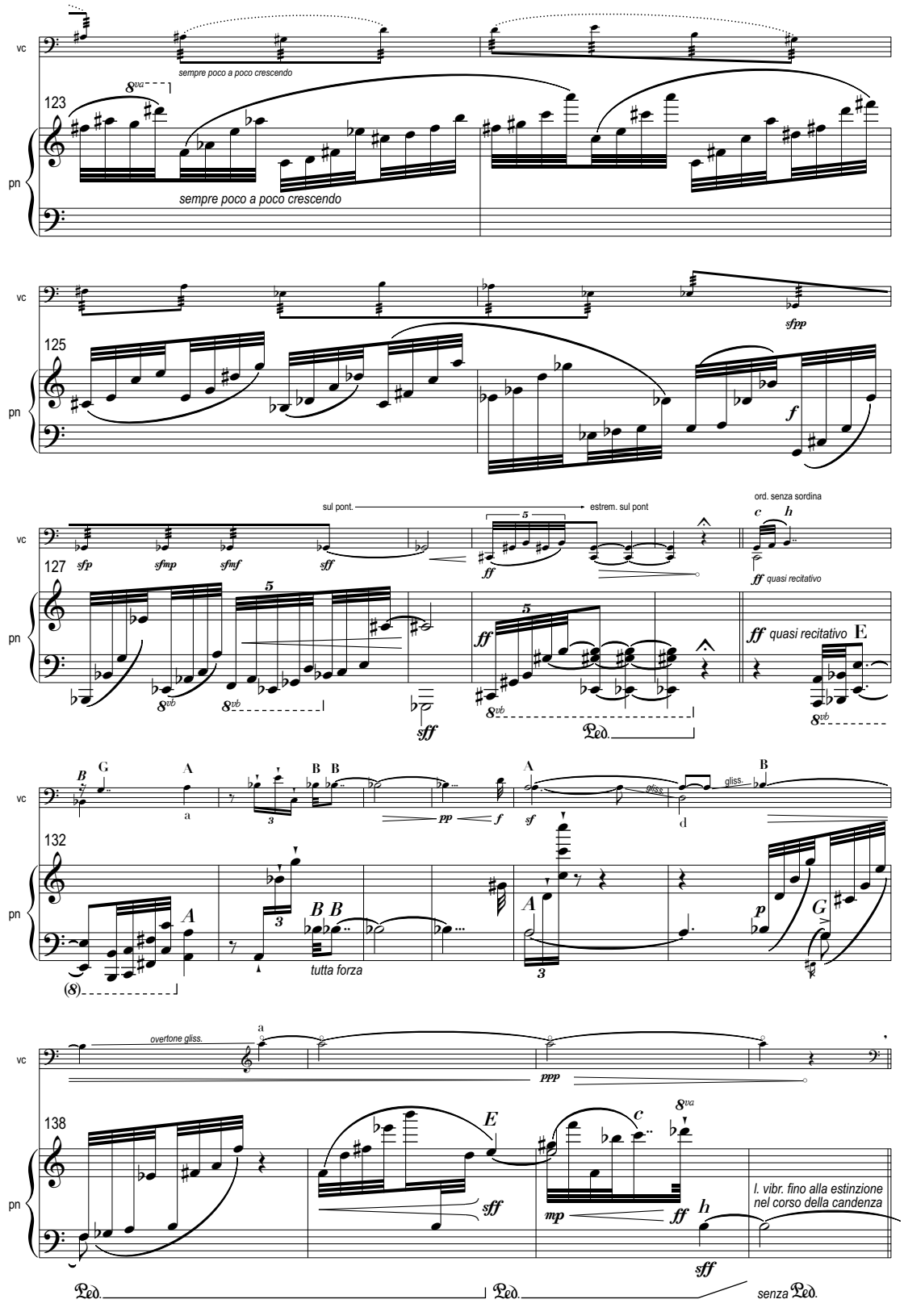

vc
p
63
2/4
5/16
2/4
pn
p
vc
mp
66
2/4
5/16
2/4
pn
mp
vc
70
5/16
2/4
5/16
pn
vc
poco più f
mezza voce
74
5/16
2/4
5/16
2/4
8va
pn
poco più f
mezza voce

12


vc
79
pn
5/16
2/4
5/16
vc
83
pn
5/16
2/4
5/16
2/4
vc
poco più f
mezza voce
88
pn
5/16
2/4
5/16
poco più f
mezza voce
vc
92
pn
5/16
2/4
5/16
2/4
8va
8vb


13
vc
97
pn
8vb
vc
102
pn
vc
107
pn
vc
111
8va
ppp
dim.
pn
Ped.


14
Capriccio X
♩ = 72
metalico
sul pont.
vc
mp
sfmp
115
♩ = 72
metalico
pn
mp ma brillante
8va
Ped.
Ped. simile
117
sfmf
8vb
119
121


15
vc
pn
sempre poco a poco crescendo
123
sempre poco a poco crescendo
125
sfpp
f
sul pont.
estrem. sul pont
ord. senza sordina
127
sfp
sfmp
sfmf
sff
ff
ff quasi recitativo
E
Ped.
132
tutta forza
pp
sf
gliss.
p
overtone gliss.
ppp
138
mp
l. vibr. fino alla estinzione
nel corso della cadenza
senza Ped.

16

Cruciverba

vc

durata ad libitum, ma sempre espressivo

gliss. gliss.

f *espress.*

dim.

pp *senza vibrato*

Capriccio XI

♩ = 102

sempre estr. sul pont.

vc

sempre ***f***

8va

143 5/16 ♩ = 102

pn

sempre ***f*** *staccatiss.*

sempre senza Ped.

vc

151

pn

8va 8va 8va 8va 8va 8vb 8vb 8vb

vc

158

pn

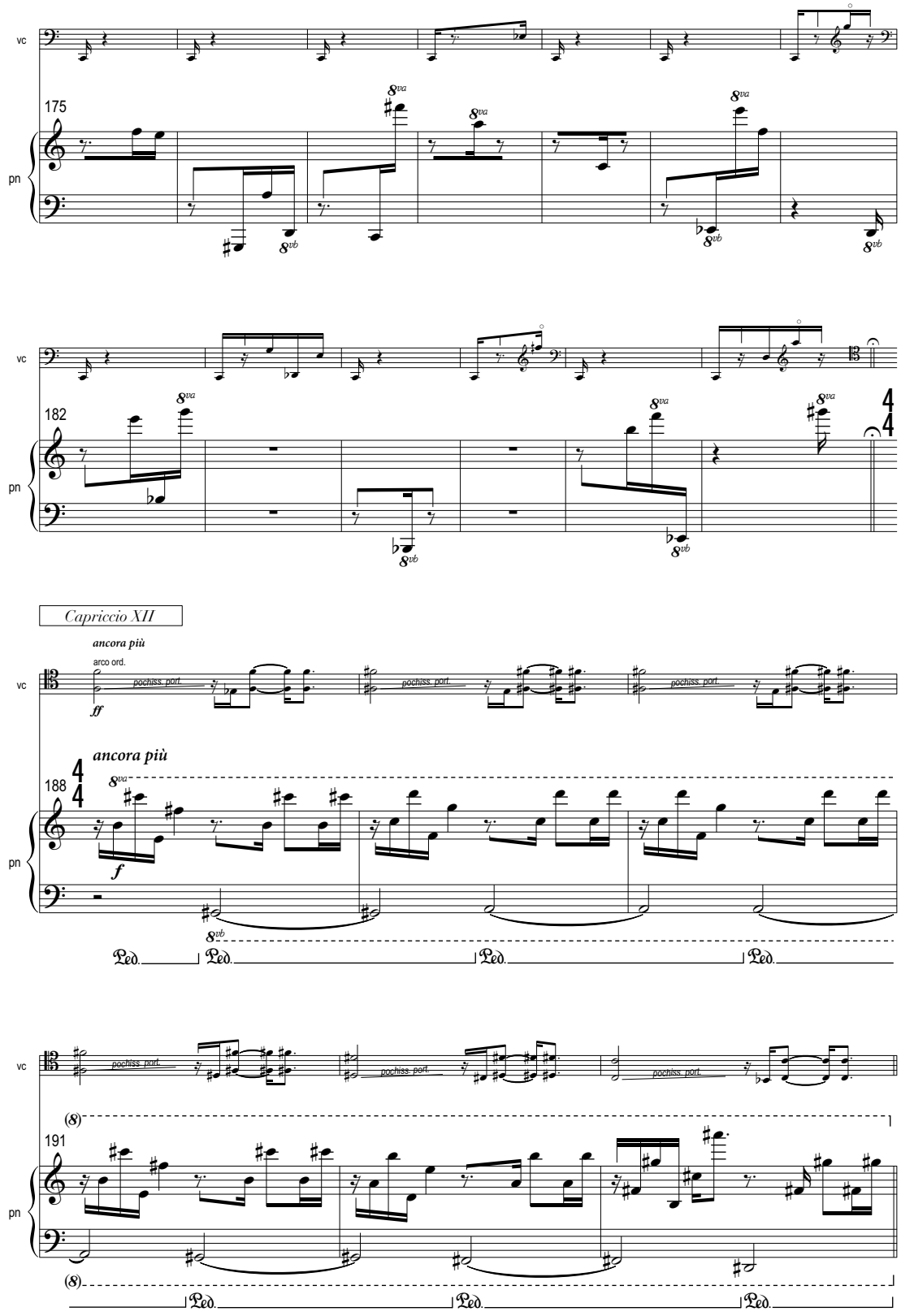
8va 8vb

vc

167

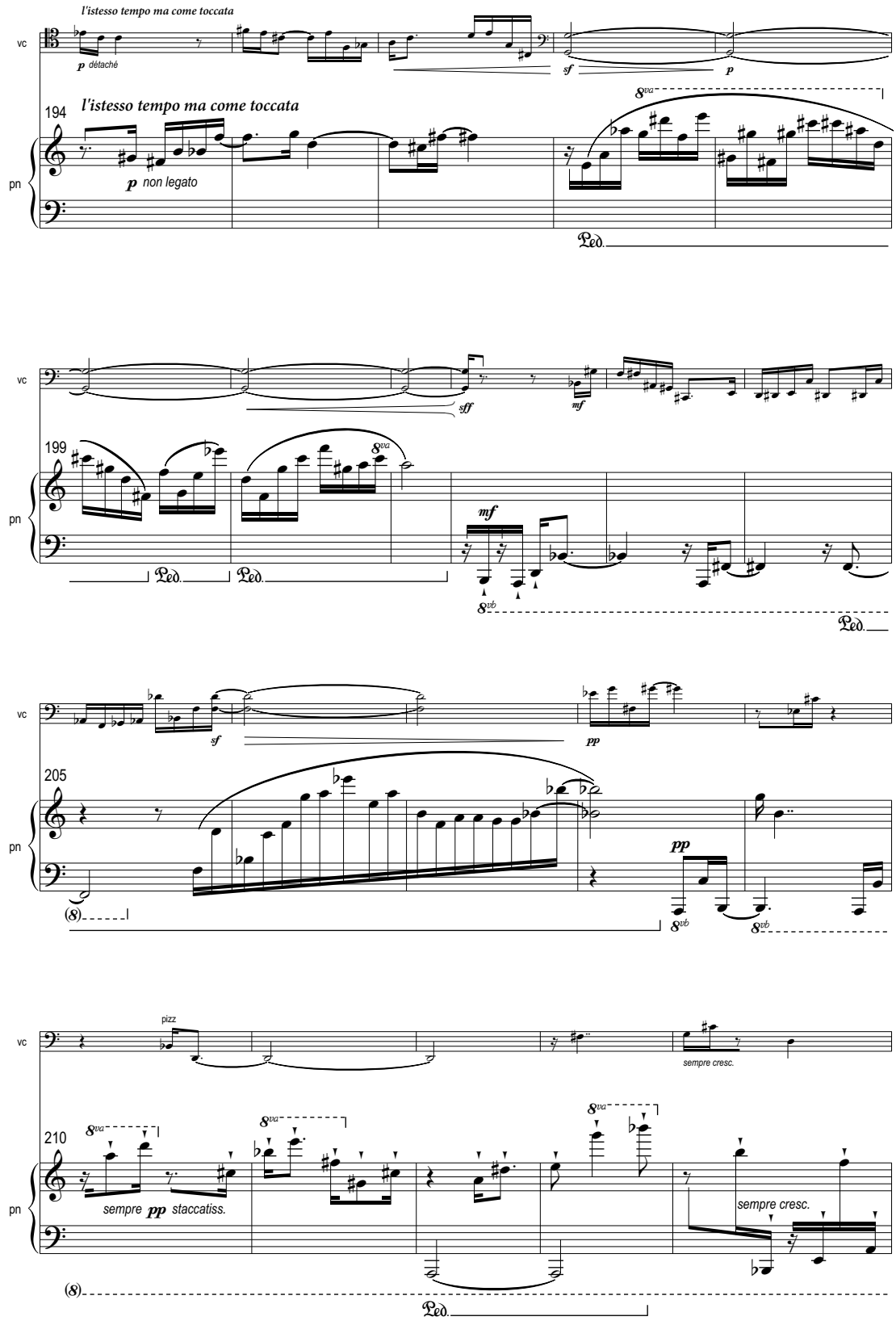
pn

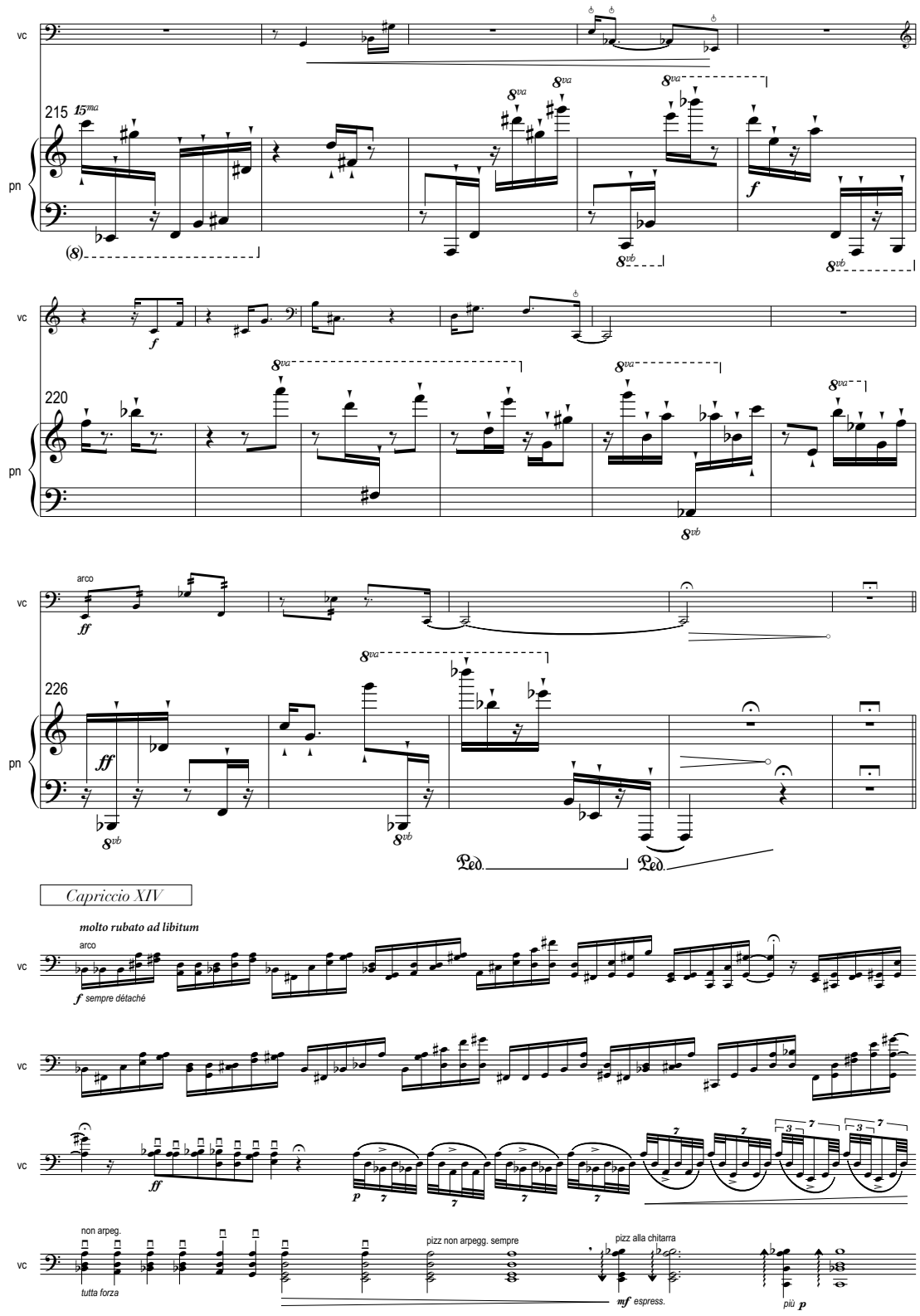
8va 8va 8va 8va 8va 8vb

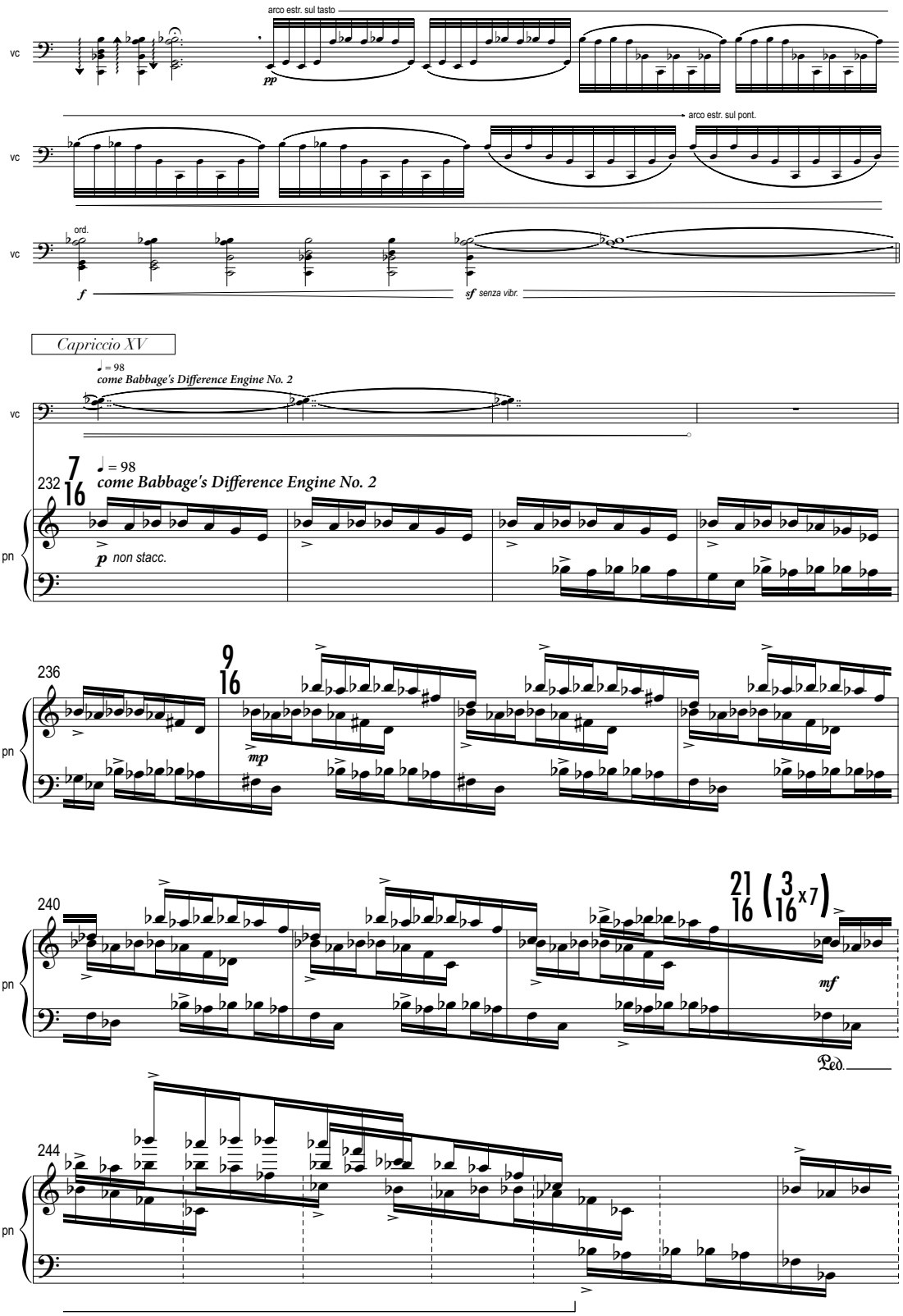

vc
175
pn
8va
8vb
182
4
4
Capriccio XII
ancora più
arco ord.
pochiss. port.
ff
188
f
Ped.
191

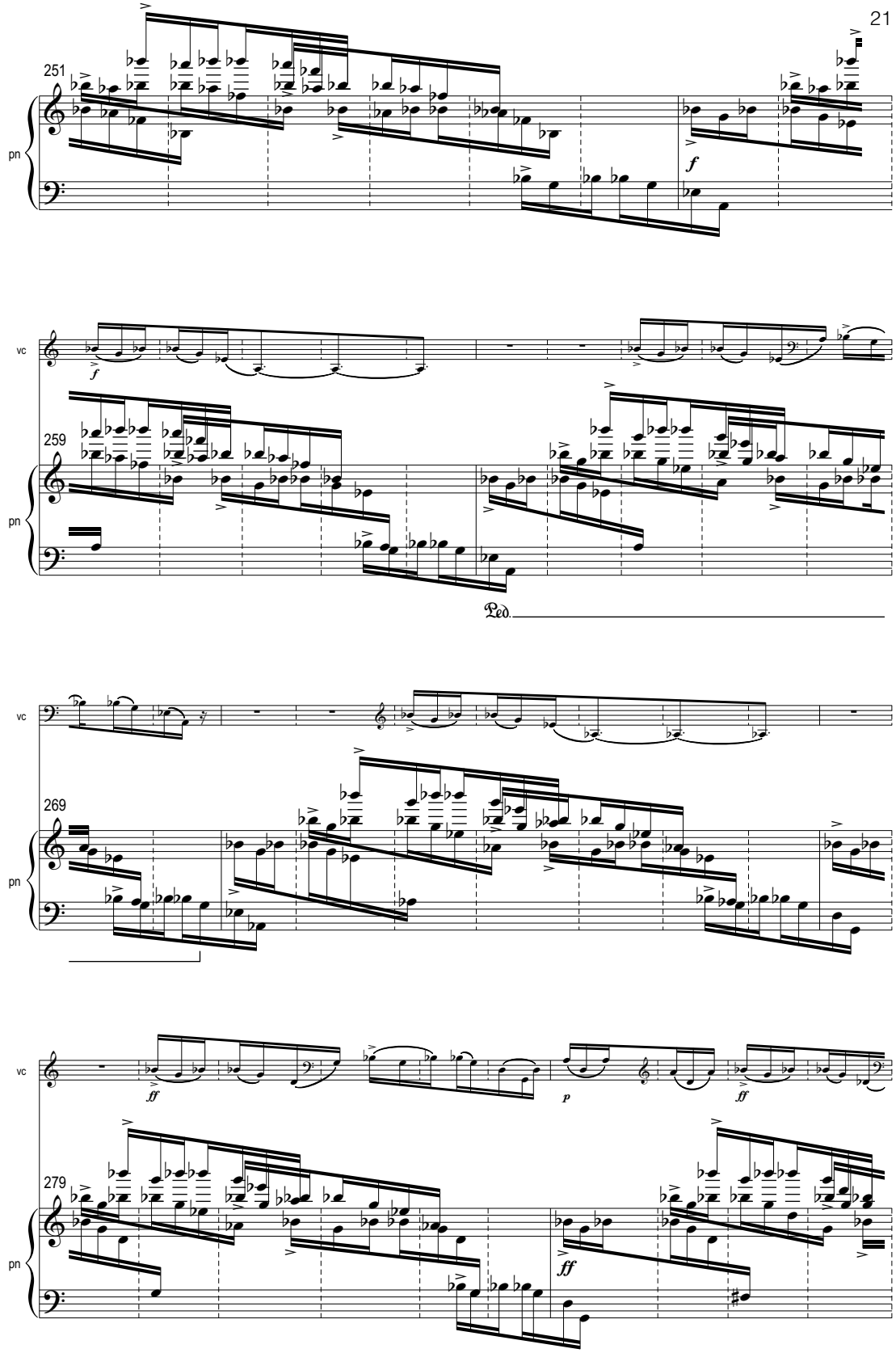
18

Capriccio XIII

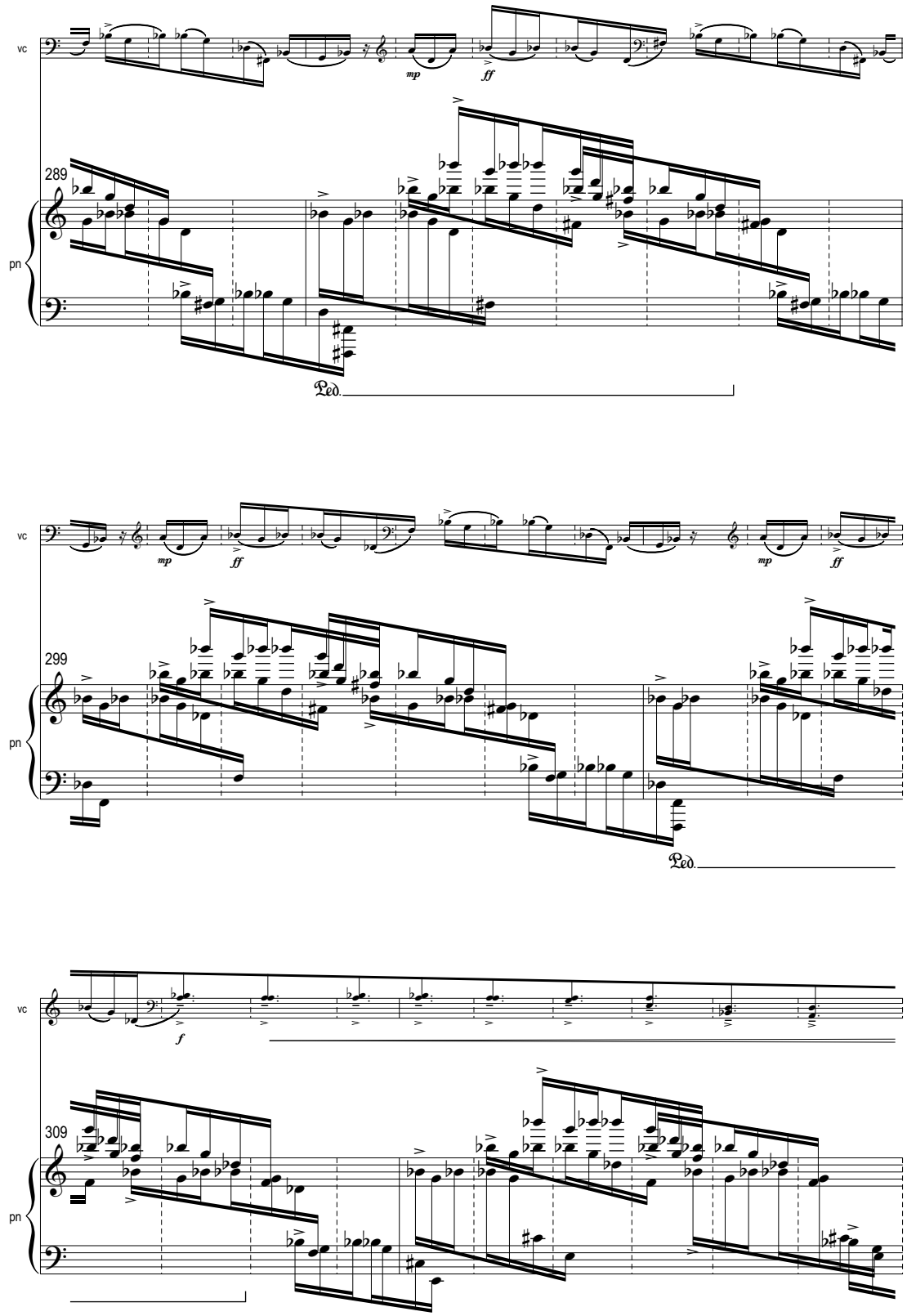


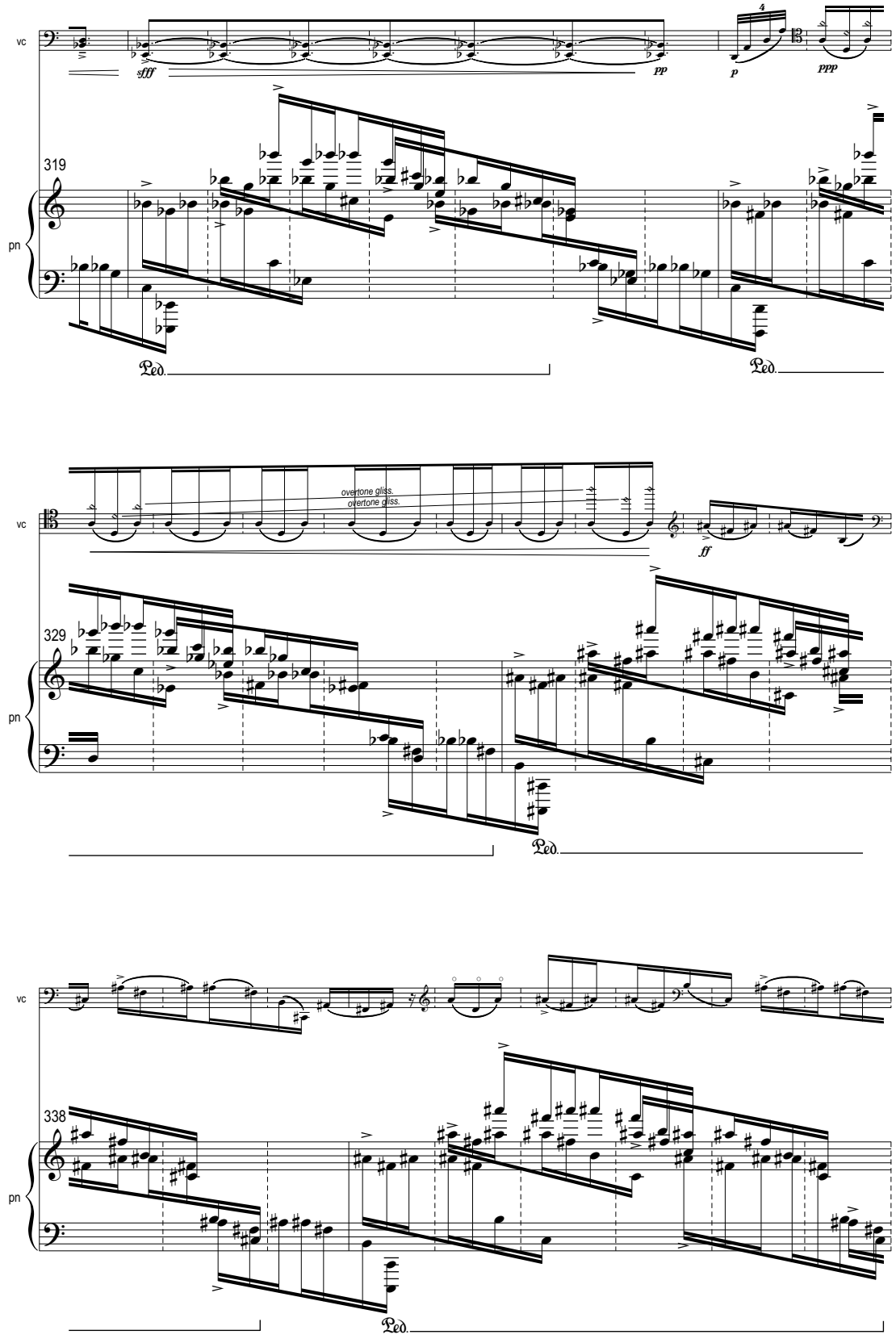

vc
pn
215
15ma
8va
8vb
f
220
ff
arco
226
Ped.
Capriccio XIV
molto rubato ad libitum
f sempre détaché
p
non arpeg.
tutta forza
pizz non arpegg. sempre
pizz alla chitarra
mf espress.
più p


20
arco estr. sul tasto
vc
pp
arco estr. sul pont.
vc
ord.
vc
f
sf senza vibr.
Capriccio XV
♩ = 98
come Babbage's Difference Engine No. 2
vc
232
7/16
♩ = 98
come Babbage's Difference Engine No. 2
pn
p non stacc.
236
9/16
pn
mp
240
21/16 (3/16 x 7)
pn
mf
Ped.
244
pn

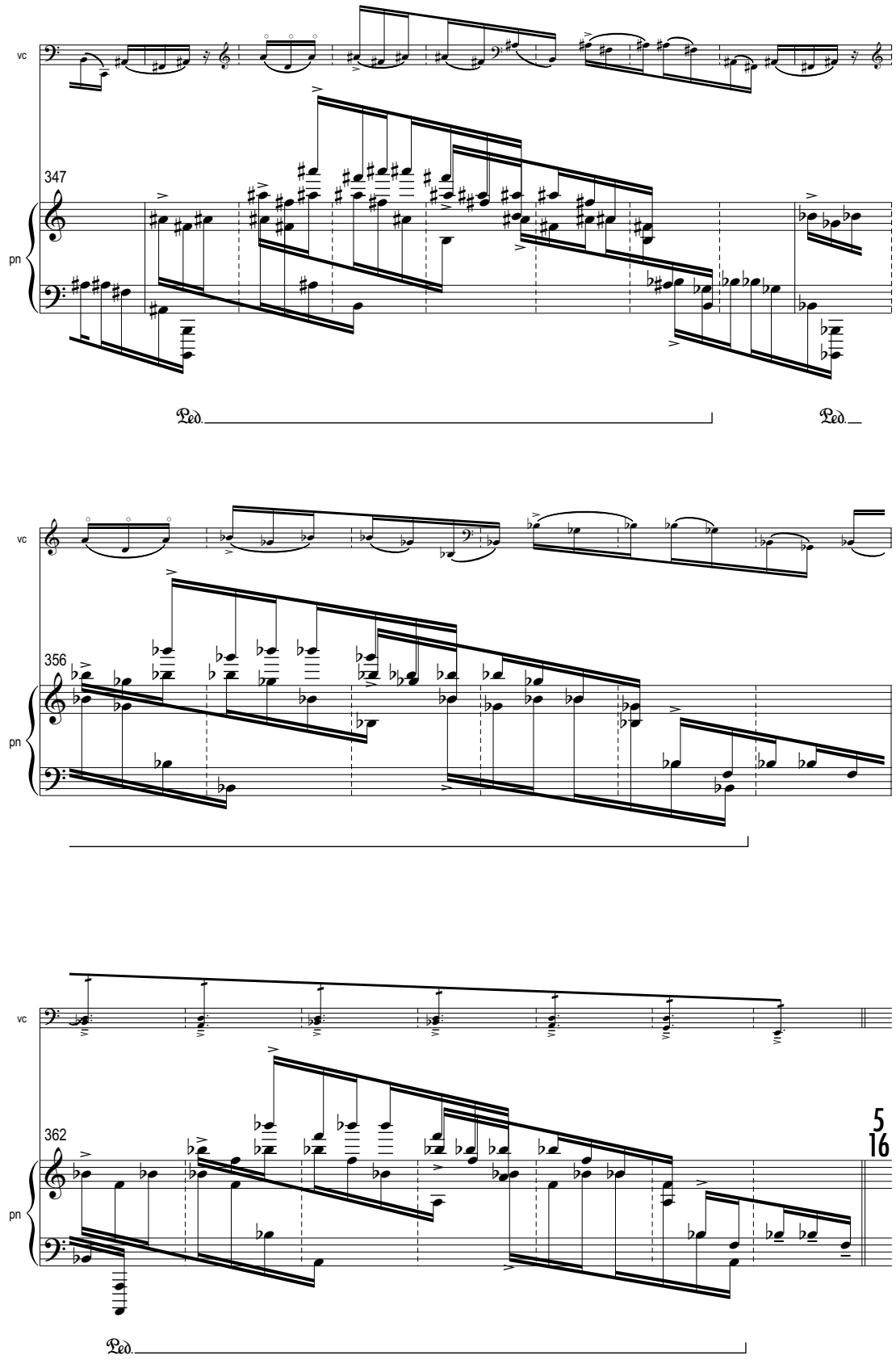

21
251
pn
f
vc
f
259
pn
Ped.
vc
269
pn
vc
ff
p
ff
279
pn
ff

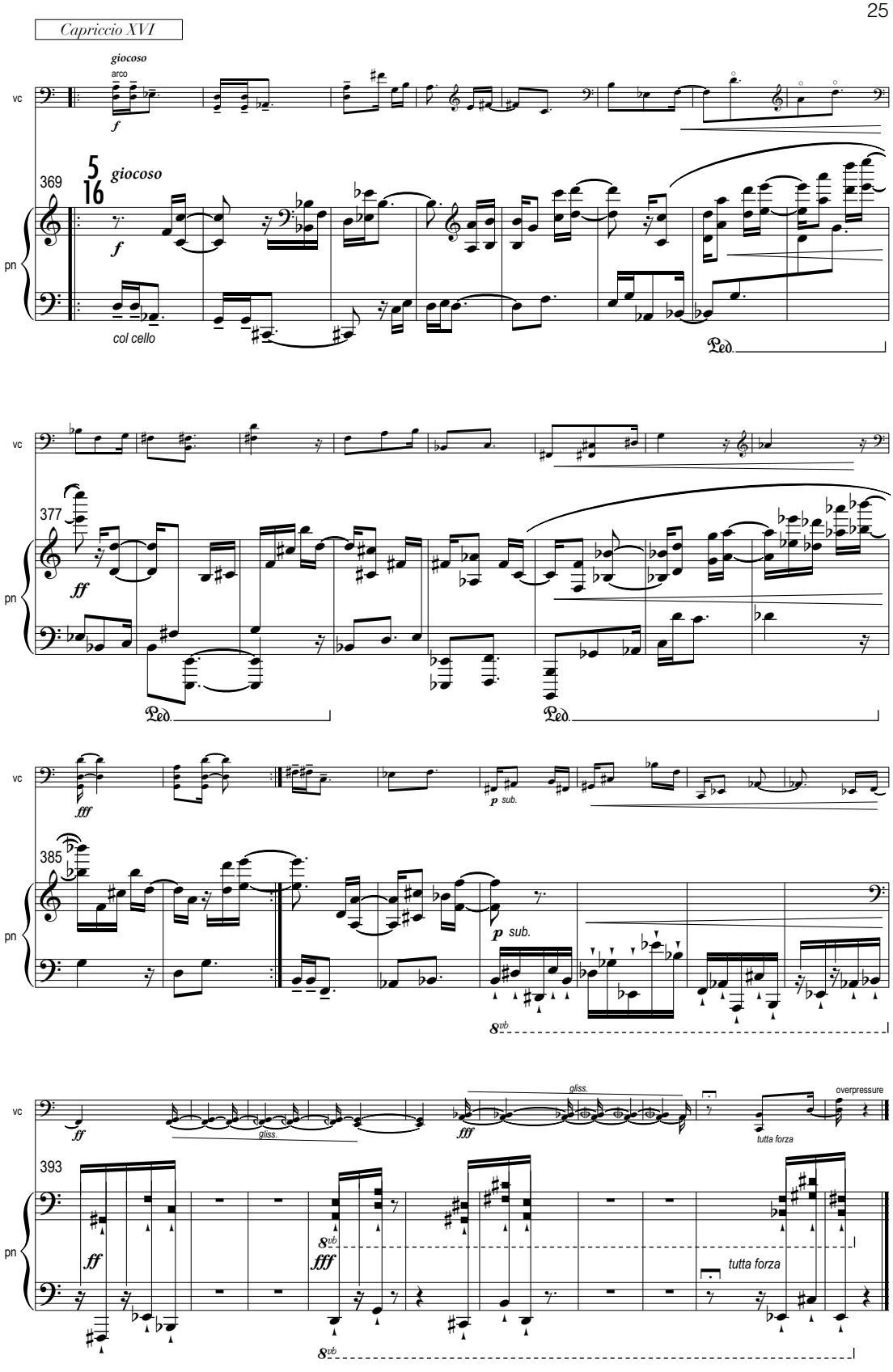
22




vc
sfff
pp
p
ppp
319
pn
Ped.
Ped.
overtone gliss.
overtone gliss.
ff
329
pn
Ped.
vc
338
pn
Ped.

24




25
Capriccio XVI
giocoso
arco
vc
f
369
5/16
giocoso
pn
f
col cello
Ped.
377
ff
Ped.
Ped.
fff
p sub.
385
p sub.
8vb
ff
gliss.
gliss.
fff
overpressure
tutta forza
393
ff
8vb
fff
tutta forza
8vb

Referencias

[1] Alivisatos, A. Paul *et al.*: "The Brain Activity Map Project and the Challenge of Functional Connectomics". *Neuron*, Ed. Elsevier (21 junio 2012), vol. 74, p. 970.

[2] Anders, Torsten: "Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System". Director: Alcorn, Michael; Anagnostopoulou, Christina. J. Queen's University Belfast, Faculty of Arts, Humanities and Social Sciences, 2007.

[3] Aranda, Joaquín *et al.*: *Fundamentos de Lógica Matemática y Computación*. Madrid: Ed. Sanz y Torres, 2006. 309 pp. [refer.: pp. 167, 281] isbn: 849609474X

[4] Assayag, Gérard *et al.*: "Computer Assisted Composition at Ircam: PatchWork & OpenMusic". *Computer Music Journal*, Ed. MIT Press (1999), vol. 23 (3).

[5] Ball, Philip: "Algorithmic rapture". *Nature*, Ed. Macmillan (23 agosto 2012), vol. 488, p. 458.

[6] Bennett, Gerald: *Chaos, Self-Similarity, Musical Phrase and Form* [en línea]. 1985. Disponible en:
http://www.icst.net/uploads/media/Chaos.pdf
[Consulta: 3 de junio de 2013]

[7] Berry, Wallace: *Structural functions in music*. Mineola (Nueva York): 1ª ed. Dover, 1987 (reedición revisada de la edición original de Prentice-Hall, 1976). 447 pp. [cita: p. 255]. isbn: 0-486-25384-8

[8] Beyls, Peter: "Selectionist musical automata: Integrating explicit instruction and evolutionary algorithms". En:
Proceedings of IX Brazilian Symposium on Computer Music, 2003.

[9] Bhattacharya, Shantanu: "Use functional programming techniques to write elegant JavaScript". IBM Developer Works, 2006, [en línea]. Disponible en:
http://www.ibm.com/developerworks/library/wa-javascript/index.html
[Consulta: 3 de junio de 2013]

[10] Boulez, Pierre: *Puntos de referencia*. Prieto, Eduardo J. (trad.). Barcelona: Ed. Gedisa, 1966. 496 pp. [cita: p. 17]. ISBN: 978-8474321975

[11] Burraston, Dave *et al.*: "Cellular Automata in MIDI-Based Computer Music.". *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, 2004, pp. 117-128.

[12] Cage, John: *Silence: Lectures and Writings*. 1ª ed. Wesleyan University Press, 1961. 312 pp. [cita: p. 274]. ISBN: 978-0819560285

[13] Cage, John: *Silencio: conferencias y escritos*. Pedraza, María (trad.). 4ª ed. Madrid: Árdora Ediciones, 2007. 288 pp. [cita: p. 274]. ISBN: 978-8488020345

[14] Cont, Arshia: "Modeling Musical Anticipation: From the time of music to the music of time". University of Paris 6 (UPMC), and University of California San Diego (UCSD), 2008. Disponible en:
http://articles.ircam.fr/textes/Cont08b/
[Consulta: 3 de junio de 2013]

[15] Cook, Matthew: "Universality in Elementary Cellular Automata". *Complex Systems*, 2004, vol. 15, pp. 1-40.

[16] Cope, David: *Computer Models of Musical Creativity*. Ed. MIT Press, 2006. 456 pp. ISBN: 978-0262033381

[17] Dannenberg, Roger: "The CANON score language". *Computer Music Journal*, Ed. MIT Press (1989), vol. 13 (1), pp. 47-56.

[18] de la Motte, Diether: *Contrapunto*. 1ª ed. Madrid: Ed. Idea Books, 1998. 420 pp. [ref.: p. XII-XIII] ISBN: 848236104X

[19] di Scipio, Agustino: *Pensare le tecnologie del suono e della musica*. Diana, Rosario (prol.) Ed. Scientifica, 2013. 265 pp. ISBN: 978-8863424706

[20] Forte, Allen; Gilbert, Steven E.: *Introducción al análisis schenkeriano*. Purroy Chicot, Pedro (trad.). 1ª ed. Barcelona: Ed. Labor, 1992. 451 pp. [cita: p. 291]. ISBN: 8433578693

[21] García Salas, Horacio Alberto; Gelbukh, Alexander; Calvo, Hiram: "Music Composition Based on Linguistic Approach". En: Sidorov, Grigori; Hernández Aguirre, Arturo; Reyes García, Carlos Alberto: *Advances in Artificial Intelligence*. Berlín/Heidelberg: Ed. Springer, 2010, pp. 117-128. ISBN: 978-3642167607

[22] Goertzel, Ben.: *From Complexity to Creativity: Computational Models of Evolutionary, Autopoietic and Cognitive Dynamics*. Nueva York: ed. Plenum, 1997. 376 pp. [cita: p. 362]. ISBN: 0306455188

[23] Hawking, Stephen W.: *The Theory of Everything: The Origin and Fate of the Universe*. Ed. Jaico Publishing House, 2007. 140 pp. ISBN: 978-8179925911

[24] Hofstadter, Douglas R.: *Gödel, Escher, Bach: un Eterno y Grácil Bucle*. Usabiaga Bandizzi, Mario Arnaldo; López Rousseau, Alejandro (trad.). 2ª ed. (Fábula) Barcelona: ed. Tusquets, 2009. 882 pp. [cita: p. 695]. ISBN: 978-8483830246

[25] Hofstadter, Douglas R.: *Yo soy un extraño bucle*. De Juan Vidales, Luis Enrique (trad.). 1ª ed. Barcelona: ed. Tusquets, 2008. 524 pp. ISBN: 978-8483830871

[26] Hudak, Paul: *The Haskell School of Music: From Signals to Symphonies*. Ed. Yale University, 2012. [*eBook* en línea] 353 pp. Disponible en:
www.cs.yale.edu/homes/hudak/Papers/HSoM.pdf
[Consulta: 3 de junio de 2013]

[27] Hughes, John: "Why Functional Programming Matters". *Computer Music Journal*, Ed. MIT Press (1989), vol. 32 (2), pp. 98-107.

[28] Jackson, Philip C. (Jr.): *Introduction to Artificial Intelligence* (Second, Enlarged Edition). Mineola (Nueva York): Ed. Dover, 1985. 512 pp. ISBN: 978-0486248646

[29] Järveläinen, Hanna: *Algorithmic musical composition* [en línea]. Disponible en:
http://www.tml.tkk.fi/Studies/Tik-111.080/2000/papers/hanna/alco.pdf
[Consulta: 3 de junio de 2013]

[30] Laske, Otto: "Considering Human Memory in Designing User Interfaces for Computer Music". *Computer Music Journal*, Ed. MIT Press (1978), vol. 2 (4), pp. 39-45.

[31] Lerdahl, Fred; Jackendoff, Ray: *Teoría generativa de la música tonal*. González-Castelao, Juan (trad.). Madrid: Ed. Akal, 2003. 407 pp. [cita: p. 333]. ISBN: 8446015986

[32] Lobo, Daniel: "Evolutionary Development based on Genetic Regulatory Models for Behavior-Finding". Director: Vico, Francisco J. Universidad de Málaga, Departamento de Lenguajes y Ciencias de la Computación, 2010.

[33] López-Montes, José: Catálago de obras [en línea]. Disponible en: http://www.lopezmontes.es/obra.html [Consulta: 3 de junio de 2013]

[34] López-Montes, José: "eHayku (Study for Threnody)". En: Asociación de Música Electroacústica de España: *AMEE 25: 25º Aniversario de la Asociación de Música Electroacústica de España*. [CD]. Ed. AMEE, 2012.

[35] Maconie, Robin: *La música como concepto*. Gil Aristu, José Luis (trad.). Barcelona: Ed. Acantilado, 2007. 298 pp. [citas: p. 125]. ISBN: 978-8496834248

[36] Mandelbrot, Benoît: *La geometría fractal de la naturaleza*. Llosa, Josep (trad.). 2ª ed. Barcelona: Ed. Tusquets, 2003. 662 pp. [ref: p. 347, 415] [citas: p. 523]. ISBN: 8483105497

[37] Mazzola, Guerino.: *The Topos of Music: Geometric Logic of Concepts, Theory, and Performance*. Göller, Stefan; Müller, Stefan (colab.). Basilea/Boston: Ed. Birkhäuser, 2002], 1335 pp. ISBN: 978-3764357312

[38] Meyer, Leonard B.: *El estilo en la música. Teoría musical, historia e ideología*. Angstadt, Michel (trad.). Madrid: Ed. Pirámide (Grupo Anaya), 2000. 548 pp. [citas: pp. 59-61, 65]. ISBN: 8436813669

[39] Miranda, Eduardo Reck: *Composing Music with Computers*. Ed. Focal Press, 2001. 256 pp. ISBN: 978-0240515670

[40] Ninagawa, Shigeru: "$1/f$ noise in elementary cellular automaton rule 110". En: *Proceedings of the 5th international conference on Unconventional Computation*. Berlin/Heidelberg: Ed. Springer, 2006, pp. 207–216. ISBN: 978-3-540-38593-6

[41] O'Brien, Larry; Eckel, Bruce: *Thinking in C#*. Ed. Prentice Hall, 2002. 1000 pp. ISBN: 978-0130385727

[42] Oliva, Tomasz; Wagner, Markus: "Composing Music with Neural Networks and Probabilistic Finite-State Machines". En:
Giacobini, Mario *et al.*: *Applications of Evolutionary Computing, EvoWorkshops 2008*. Ed. Springer, 2008, pp. 503-509.

[43] Ord-Hume, Arthur W. J. G.: *Joseph Haydn and the Mechanical Organ*, Ed. University College Cardiff, 1982], 185 pp. ISBN: 978-0906449370

[44] Ortega, Alfonso; Sánchez, Rafael; Alfonseca, Manuel: "Automatic composition of music by means of Grammatical Evolution". En: APL'2002 Madrid Proceedings.

[45] Paynter, John: *Sonido y estructura*. Hurquhart, Hamisch (trad.). Madrid: Ed. Akal, 1999. 216 pp. ISBN: 8446012790

[46] Payón, Pablo J.: "Entre la armonía y el ruido". *Diario de Sevilla* (Sevilla, 12 diciembre 2012) [online]. Disponible en:
http://www.diariodesevilla.es/article/ocio/1418078/entre/la/armonia/y/ruido.html
[Consulta: 3 de junio de 2013]

[47] Peitgen, Heinz-Otto; Hartmut, Jürgens; Saupe, Dietmar: *Fractals for the Classroom*, Part 2. 1ª ed. New York: Springer Verlag, 1992. 512 pp. ISBN: 978-0387977225

[48] Poe, Edgar Allan: *Maelzel's Chess Player*, Ed. Dodo, 2009, 48 pp. ISBN: 978-1409948506

[49] Puckette, Miller: idea extraída de la videoconferencia ofrecida en *II Congreso Internacional de Música y Tecnologías Contemporáneas*, Sevilla, noviembre 2008.

[50] Real Academia Española: *Diccionario de la lengua española*. 22ª ed. [en línea], 2001. Disponible en:
http://www.rae.es/
[Consulta: 3 de junio de 2013]

[51] Revesz, Gyorgy E.: *Lambda-Calculus Combinators and Functional Programming* Cambridge Tracts in Theoretical Computer Science vol. 4. Ed. Cambridge University Press, 1988. 192 pp. ISBN: 978-0521114295

[52] Rideau, François-René: "Metaprogramming and Free Availability of Sources" [en línea]. Disponible en:
http://fare.tunes.org/articles/ll99/mpfas.pdf
[Consulta: 3 de junio de 2013]

[53] Roads, Curtis: "Grammars as Representations for Music" (versión expandida del artículo originalmente publicado en *Computer Music Journal*, Ed. MIT Press (1979), vol. 3 (1), pp. 48-55). En:
Roads, Curtis; Strawn, John: *Foundations of Computer Music*, Ed. MIT Press, 1985], 730 pp. isbn: 978-0262680516

[54] Romero Cardala, Juan Jesús: "Metodología para la construcción de modelos cognitivos complejos: exploración de la "creatividad artificial" en composición musical". Director: Pazos Sierra, Alejandro; Santos del Riego, Antonino. Universidade da Coruña, Departamento de Tecnoloxías da Información e as Comunicacións, 2001. isbn: 978-8469288856

[55] Russomanno, Stefano: "Guerreros de Valencia". *ABC Cultural* (Madrid, 5 noviembre 2005), p 55.

[56] Satie, Erik: *Mémoires d'un amnésique*. Ed. Ombres, 2010. 190 pp. isbn: 978-2841421886

[57] Satie, Erik: *Memorias de un amnésico y otros escritos*. Casado, Loreto (trad.). 2ª ed. Madrid: Ed. Ardora, 1994. 143 pp. [cita: p. 25]. isbn: 8488020031

[58] Schaathun, Asbjørn: "Formula-composition modernism in music made audible". En:
Thommessen, Olav Anton: *Inspirator – Tradisjonsbærer – Rabulist*. Oslo: Ed. Norsk Musikforlag, 1996, pp. 132-147.

[59] Schacher, Jan C.: comunicación personal, 2009.

[60] Schmidhuber, Jürgen: "Low-Complexity Art". *Leonardo, Journal of the International Society for the Arts, Sciencies and Technology*, Ed. MIT Press, 1997, vol. 30 (2), pp. 97-103.

[61] Schmidhuber, Jürgen: "Gödel machines: self-referential universal problem solvers making provably optimal self-improvements". En:
Goertzel, Ben; Pennachin, Cassio *Artificial General Intelligence*, Berlín/Heidelberg: Ed. Springer, 2006, pp. 119-226.

[62] Schönberg, Arnold: *Harmonielehre*. Universal Edition Ag., 2005. 520 pp. isbn: 978-3702400293

[63] Searle, John R.: *Freedom and Neurobiology: Reflections on Free Will, Language, and Political Power* Ed. Columbia University Press, 2006. 128 pp. isbn: 978-0231137522

[64] Tannenbaum, Mya: *Stockhausen: entrevista sobre el genio musical*. Alonso, Carlos (trad.). Madrid: Ed. Turner, 1988. 110 pp. [cita: p. 83]. isbn: 8475062415

[65] Univ. Complutense de Madrid: *Del cálculo numérico a la creatividad abierta. El Centro de Cálculo de la Universidad de Madrid (1965-1982)*. Ed. Univ. Complutense de Madrid, Dep. de Humanidades, 2012. 317 pp. isbn: 978-8496701632

[66] von Neumann, John.: *The Collected Works of John von Neumann*. Taub, A. H. (ed.). Ed. Oxford, 1967.

[67] Voss, Richard F., Clarke, John: "'$1/f$ noise' in music and speech". *Nature*, Ed. Macmillan (27 noviembre 1975), vol. 258, p. 317-318.

[68] Wolfram, Stephen: *A New Kind of Sciencie*. Ed. Wolfram Media, 2001. 1280 pp. [cita: p. 844] isbn: 1579550088