Universidad Politécnica
de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Algoritmos de Generación de Música Fractal

Autor: Miguel Jaime Gonzalo Antón

Tutor(a): María del Carmen Escribano Iglesias

Madrid, Enero 2022

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Algoritmos de Generación de Música Fractal
Enero 2022

Autor: Miguel Jaime Gonzalo Antón

Tutor:
María del Carmen Escribano Iglesias
Departamento de Matemática Aplicada a las Tecnologías de la Información y las Comunicaciones
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

La música es uno de los elementos más importantes de una cultura, de hecho, aunque esta sea cada vez más accesible para las personas, la composición musical ha sido y sigue siendo, una actividad compleja reservada para las personas con gran conocimiento en la materia. El pensamiento matemático ha sido un componente esencial en los orígenes y desarrollo de la creación musical y en cierto sentido, una partitura se podría interpretar como una estructura matemática que se desarrolla mediante ciertos algoritmos.

Gracias al rápido desarrollo de los ordenadores, desde hace décadas comenzó el estudio de la automatización de la composición para la ágil generación de música, mediante inteligencias artificiales e, incluso programas basados en modelos matemáticos que provienen de Sistemas Dinámicos, Caos y Fractales.

En este trabajo se estudian algunos de los métodos existentes para la generación de música mediante algoritmos basados en la Geometría Fractal y Sistemas Dinámicos. Se presenta, además, el desarrollo y aportación de nuevas técnicas añadidas para conseguir una automatización de la labor de composición musical más eficiente y efectiva en comparación con otras técnicas y trabajos.

Abstract

Music is one of the most important elements of a culture, in fact, although it is becoming more and more accessible to people, musical composition has been, and still is, a complex activity reserved for people with great knowledge in the field. Mathematical thinking has been an essential component throughout the origins and development of musical creation and in a certain sense, a music score could be interpreted as a mathematical structure that is developed through certain algorithms.

Thanks to the rapid development of computers, decades ago began the study of the automatization of composition in order to generate music in a more agile way, through artificial intelligences and even programs based on mathematical models that come from Dynamical Systems, Chaos and Fractals.

In this work we study some of the existing methods for the generation of music through algorithms based on Fractal Geometry and Dynamical Systems. It also presents the development and contribution of new techniques added to achieve a more efficient and effective automatization of the music composition process in comparison with other techniques and researches.

Índice

1 Introducción

1.1 Contexto

A lo largo de la historia de la humanidad ha existido de manera imperante el gusto por las repeticiones de sonidos de manera secuencial para el disfrute personal, esto es, la música. Además, a medida que ha ido avanzando el tiempo, la creación de música o composición no ha hecho otra cosa que aumentar su complejidad, existiendo numerosas reglas o directrices para una composición musical de "calidad". Dicha composición se ha realizado históricamente de forma manual, perteneciendo al arte de cada cultura. Sin embargo, hasta el siglo XX no aparecen formas de automatización para la creación musical.

En la actualidad la mayoría de las formas de composición musical automatizada se basan en la inteligencia artificial, utilizando, por ejemplo, grandes recursos para el entrenamiento y validación de redes neuronales. Pero esta no es la única manera de generar música de forma artificial, ya que a lo largo del siglo XX comienza el estudio de la Geometría Fractal, término acuñado por Benoît Mandelbrot en 1975 [1].

Tan solo un año después Voss y Clarke pusieron de manifiesto en su artículo *"1/f noise" in music: Music from 1/f noise* [2], que la densidad espectral de la señal de audio de varias emisoras de diferentes estilos de música se acercaba notoriamente a ser inversamente proporcional a la frecuencia de las mismas. Posteriormente sería relacionado con la geometría fractal de la música por parte de Kenneth J. Hsü y Andreas J. Hsü en 1989 [3].

Demostrada la relación entre composiciones musicales y geometría fractal, analizando la primera para encontrar la segunda en dichas composiciones, ha habido ciertos acercamientos a la generación de notas musicales e incluso a composiciones musicales completas como la aplicación FractMus 2000. Aventajando en cierto sentido a las redes neuronales utilizando menos recursos pesados para la generación artificial de música. Aun así, no se trata de un método popular para este tipo de creaciones.

Este Trabajo se propone analizar e implementar algoritmos para la generación de música utilizando geometría fractal y sistemas dinámicos, comenzando por los existentes y diseñando nuevas técnicas.

1.2 Objetivos

El objetivo principal de este Trabajo es la generación de piezas musicales mediante algoritmos basados en la geometría fractal y sistemas dinámicos comenzando por las técnicas conocidas y diseñando nuevas técnicas para este tipo de desarrollo. Para ello me serviré de conocimientos adquiridos durante el grado de Ingeniería Informática, en concreto, sentando las bases matemáticas recibidas en el curso de Sistemas Dinámicos, Caos y Fractales; también aplicando los conocimientos de programación adquiridos a lo largo de asignaturas como Algorítmica Numérica I y II o Aplicaciones Numéricas de la Informática.

La lista de objetivos modular sería pues:

1. Revisión de las técnicas de simulación de Geometría Fractal y Sistemas Dinámicos

2. Recopilación de información sobre algoritmos existentes sobre generación de música fractal.
3. Diseño e implementación de programas que generen música mediante algoritmos conocidos de generación de fractales
4. Diseño e implementación de nuevos algoritmos para la generación de música utilizando geometría fractal
5. Redacción de la Memoria Final y Presentación de resultados.

1.3 Estructura de la Memoria

Esta memoria comenzará por una breve introducción sobre el interés histórico del tema del Trabajo y una breve introducción a las técnicas informáticas de creación de música.

Los siguientes pasos a seguir son contextualizar de manera más concreta en los conocimientos necesarios adquiridos para la comprensión de la Geometría Fractal y una breve explicación de sistemas dinámicos, resaltando su relación con ciertos fractales clásicos. Finalmente explicando técnicas y algoritmos existentes para la obtención de conjuntos fractales.

En el siguiente capítulo se hablará de la relación entre la música y la geometría fractal, comenzando con breves definiciones sobre el sonido y las composiciones musicales necesarias para el contexto del Trabajo. Se tratará desde ciertas similitudes fractales de la música convencional hasta trabajos previos en los que se pretende también generar música aleatoria, pseudoaleatoria o la música determinista de manera automatizada y utilizando la geometría fractal o sistemas dinámicos.

En el cuarto capítulo se procede al desarrollo de algoritmos existentes y no existentes de generación de música mediante fractales, comenzando por explicar los *mappings* utilizados por trabajos previos y los elegidos para crear nuevas técnicas. Finalmente se procede a la revisión y validación de dichos algoritmos.

Por último, se presentan las conclusiones, aprendizaje y aportaciones.

2 Geometría Fractal y Sistemas Dinámicos

Muchos compositores han demostrado que se pueden modelar diferentes características de la música de acuerdo con una determinada función matemática y computarla a través del ordenador. Esta función podría tomarse de cualquier modelo o fenómeno físico que se encuentre en la naturaleza, como en "Earth Magnetic Field" (1970), del compositor estadounidense Charles Dodge, traduce a tonos las variaciones diarias del campo magnético terrestre.

En las últimas décadas, uno de los modelos científicos que ha merecido más atención por parte de músicos e investigadores ha sido el concepto de Geometría Fractal (Mandelbrot, 1975, 1983). Las espectaculares imágenes fractales han conquistado un importante espacio dentro de las artes gráficas e incluso en el cine fantástico.

El término "Fractal" fue introducido por primera vez por el matemático e investigador de IBM Benoit Mandelbrot en su libro "Les Objets Fractals" (Mandelbrot, 1975) [1], y ampliamente conocido con la publicación de su famoso The Fractal Geometry of Nature (Mandelbrot, 1983), cuando a finales del siglo XIX comienzan a aparecer los primeros conjuntos que posteriormente se clasificarían como fractales. La palabra fractal proviene de la palabra latina fractus, que significa fracción. Mandelbrot quería clasificar y nombrar un tipo de conjunto que poseía una dimensión fraccionaria a diferencia de los conjuntos de la geometría euclídea que poseen una dimensión entera, como los puntos que tienen dimensión cero, segmentos, con dimensión uno, un área tiene dimensión dos. Mandelbrot llama a esta dimensión topológica. En el caso de los objetos fractales, su dimensión es mayor que su dimensión topológica.

En el próximo punto se comienza con algunos ejemplos de estos conjuntos para poner en contexto y enseñar las formas fractales más básicas.

Y, como veremos más adelante, la definición que el mismo Mandelbrot da a estas figuras geométricas es:

> *Fractal es un conjunto para el cual la dimensión de Hausdorff-Besicovitch es superior a su dimensión topológica. [1]*

El mismo Mandelbrot definiría posteriormente los fractales de manera más intuitiva, aunque menos concreta matemáticamente:

> *Fractal es un conjunto formado por partes semejantes al conjunto completo. [4]*

2.1 Conjuntos Fractales

De manera coloquial, los fractales se definen como objetos geométricos que repiten patrones de sí mismos a diferentes escalas y con diferentes transformaciones generalmente contractivas. En los Fractales clásicos, como se mostrará en el punto 2.1.2, las transformaciones serán semejanzas, dando lugar a los fractales autosemejantes. Aun así, la definición de "¿Qué es fractal?" Ha dado lugar a continuo debate y hasta el día de hoy sigue siendo controvertida.

Antes de comenzar a definir el concepto de dimensión en el que se basa el de medida, presentaremos algunos de los fractales clásicos que aparecen en el estudio de este tipo de geometría, así como los algoritmos de generación.

2.1.1 Fractales Clásicos

Para proponer un contexto con el que explicar partes de la teoría de la Geometría Fractal tales como la autosemejanza y las dimensiones fractales se comienza por mostrar conjuntos clásicos que, aunque propuestos mucho antes de los estudios de B.B Mandelbrot y, por lo tanto, de la definición de fractal; se ha visto que son, efectivamente, fractales.

2.1.1.1 Conjunto de Cantor

Propuesto por el matemático George Cantor en 1883, de dónde obtiene su nombre, es un conjunto que se genera comenzando con una recta o intervalo como el [0,1]. Esta se divide en tres partes iguales $\left[0,\frac{1}{3}\right]$, $\left[\frac{1}{3},\frac{2}{3}\right]$ y $\left[\frac{2}{3},1\right]$; eliminándose el intervalo central de la recta, resultando en que solo quedan los laterales. Posteriormente estos segmentos también se dividen en tres y se elimina el intervalo central. Esta operación se repite de forma iterativa con los segmentos resultantes de la iteración anterior y el conjunto resultante se denomina Conjunto de Cantor.

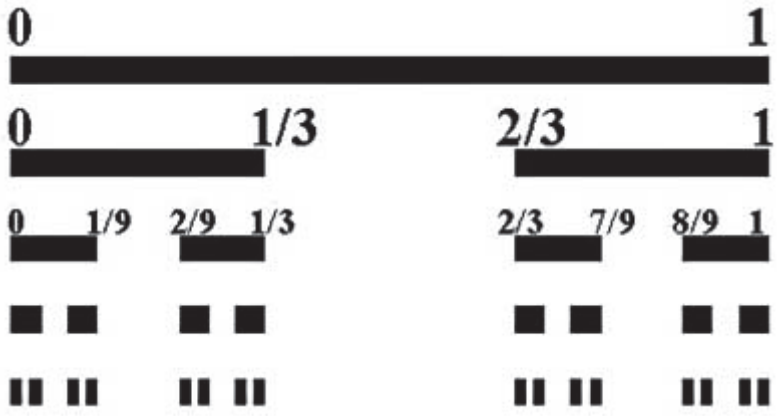


Figura 2.1 : Conjunto de Cantor a las 4 iteraciones.

2.1.1.2 Triángulo de Sierpinsky

Se trata de un patrón que aparece mucho antes de la acuñación de su nombre por el matemático polaco Waclaw Sierpinsky, que fue quien lo describió en 1915.

Existen varios algoritmos para obtener un triángulo de Sierpinsky, se comienza con un triángulo cualquiera, este se divide en cuatro triángulos iguales cuyos nuevos vértices se encuentran en los puntos medios de los segmentos que conformaban el primer triángulo, eliminándose el triángulo central (boca abajo). De este modo quedan tres triángulos de lado ½ del triángulo original. A continuación, y de forma iterativa este método se sigue con cada uno de los triángulos que se han quedado.

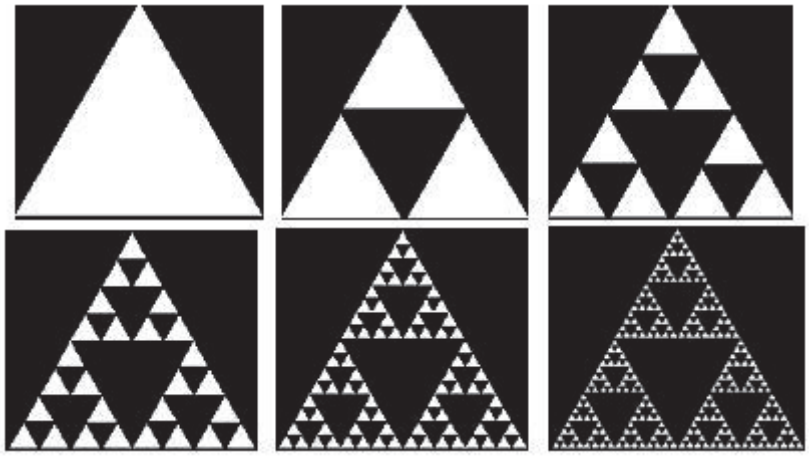


Figura 2.2 : 5 iteraciones del triángulo de Sierpinsky

2.1.2 Fractales Autosemejantes

En la geometría euclídea se dice que dos figuras son semejantes cuando mantienen las mismas proporciones entre sus distancias (como los lados de un triángulo), así como los ángulos que las conforman. Una semejanza es la composición de una isometría con una homotecia, es decir, se puede cambiar el tamaño y la orientación de una figura sin alterar la forma. Existen una serie de lo que se denominan transformaciones geométricas en las que se mantiene la semejanza entre figuras, estas son: homotecia, rotación, traslación, reflexión y simetría. Cualquier combinación de este tipo de transformaciones a una figura A da lugar a una figura A' semejante a la primera.

Homotecia se refiere, a efectos prácticos, un reescalado de la figura, ya se vea aumentada o disminuida la figura.

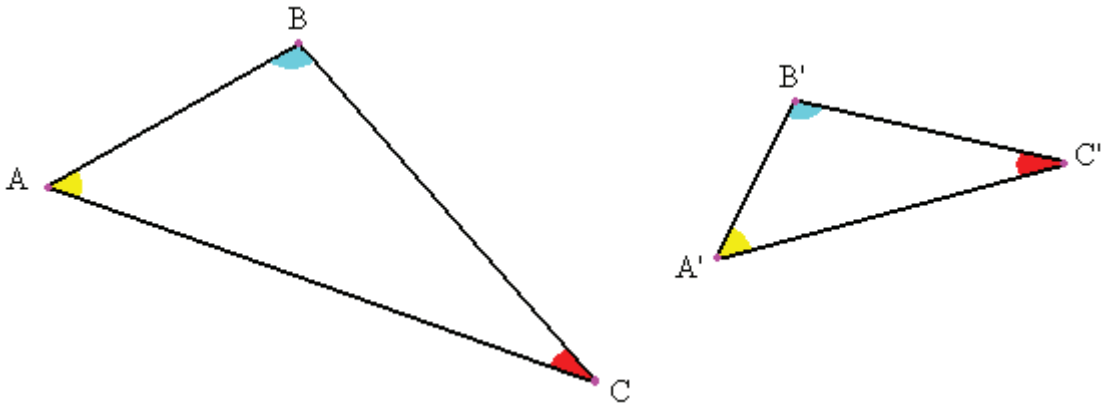


Figura 2.3 : Triángulos semejantes, transformación combinada de homotecia, rotación y traslación.

Una característica esencial de los conjuntos anteriores y que se puede apreciar de forma visual muy fácilmente, es que se generan a través de algoritmos recursivos, a cada iteración del algoritmo, los conjuntos utilizados para construir el nuevo paso de cada figura son semejantes a los de la iteración anterior y, además, la figura en una escala grande se ve repetida cuando uno se acerca a escalas más pequeñas, es decir, que se trata de conjuntos autosemejantes. Esto será una de las características básicas para la

identificación de fractales. De hecho, estos algoritmos son producidos por lo que se llama un "sistema de funciones iteradas".

Además de los fractales clásicos, otras muchas estructuras matemáticas son fractales, en particular, aquellas que se obtienen como atractores de sistemas dinámicos como por ejemplo el conjunto de Mandelbrot y atractor de Lorenz. Los fractales también describen muchos objetos del mundo real que no tienen formas geométricas simples, como nubes, montañas, turbulencias y costas.

2.1.3 Dimensiones y medidas básicas

Antes de hablar de la dimensión fractal, hay que hablar primero de la medida de Labesgue ya que es la medida "clásica" para asignar una longitud, área o volumen a conjuntos en el espacio euclídeo. Según la teoría de la medida si el conjunto tiene dimensión 1 se miden longitudes, con dimensión 2 se miden áreas y con dimensión 3 se miden volúmenes.

La medida de Labesgue se aplica para medir en dimensiones enteras. Pero como veremos más adelante, hay conjuntos, como el conjunto de Cantor, cuya medida de Labesgue es 0 pero es evidente que no se puede comparar su estructura con otros de la misma medida como puede ser un punto. En este sentido su medida y dimensión fractal va a poder distinguirlos.

Este tipo de conjuntos, entonces, advierten que puedan existir dimensiones cuyas medidas no sean números enteros, sino que puedan ser fraccionarios, por lo tanto, ahora se debe también definir la medida de Hausdorff, que permite este tipo de dimensiones.

Imaginemos un cubo descompuesto en copias iguales de sí mismo hasta formar el mismo cubo, N serían el número de copias en las que se divide, r la escala y d la dimensión. Si dividimos en 2 el lado x de un cubo, obtenemos 8 copias semejantes de lado $^{x}/_{2}$ que "cubren" el cubo. Por lo tanto, intuitivamente, podríamos decir que la relación entre la dimensión del cubo, la escala y el número de copias es:

$$d = \log_r N = \log_2 8 = 3$$

Igualmente, para un segmento que se divide en 2 a razón $^{1}/_{2}$ el cálculo es:

$$d = \log_2 2 = 1$$

Ahora observemos el conjunto de Cantor, que tiene una peculiaridad, y es que se trata de un conjunto que sabemos que no es vacío ya que el primer y el último punto del conjunto siempre van a ser escogidos por el algoritmo, pero se trata de un conjunto de longitud nula dado que se obtienen 2^n segmentos con n iteraciones de longitud $\frac{1}{3^n}$ por lo tanto, tomando la medida 1-dimensional de Labesgue del conjunto calcularíamos que: $\lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0$

Se divide a razón de $^{1}/_{3}$ y se obtienen 2 copias iguales al conjunto inicial, calculamos su dimensión y obtenemos:

$$d = \log_3 2 \approx 0{,}63$$

Para el caso más general se propone la siguiente fórmula para el cálculo de la **dimensión de semejanza** de un conjunto C a partir de N copias a escala *r*:

$$d_s(C) = \log_{1/r} N$$

Por lo tanto, también se obtiene que: $s^D = N$; dicha definición encaja con la **autosemejanza** de los fractales mencionada en el punto anterior.

Llamamos entonces a un conjunto A fractal autosemejante si existe una serie de semejanzas contractivas finitas, es decir, de transformaciones $\varphi_1, \varphi_2, \varphi_3 \dots \varphi_p$ al conjunto inicial que "dividan" el conjunto inicial en un número p de copias (con cierta propiedad de solapamiento) de él mismo y cuya razón r_i cumpla que $0 \leq r_i < 1$ tales que:

$$\cup_{i=1}^{p} \varphi_i(A) = \varphi_1(A) + \varphi_2(A) + \cdots + \varphi_p(A) = A$$

Y se llama dimensión de semejanza de A al único número $s > 0$ que verifica:

$$\sum_{i=1}^{p} r_i^s = r_1^s + r_2^s + \cdots + r_p^s = 1$$

Cuando las razones son todas igual a r, despejando la fórmula de la dimensión de semejanza, se tiene de forma general:

$$d_s(C) = \frac{\log N}{\log\left(1/r\right)}$$

Esta autosemejanza con razones iguales para todas las transformaciones ocurre también con el triángulo de Sierpinsky, cuya área, dividiéndose en 3 triángulos de la mitad de tamaño cuando las iteraciones tienden a infinito es 0. Y si calculamos su dimensión obtenemos que:

$$d_s(C) = \frac{\log(3)}{\log(2)} \approx 1{,}584$$

2.1.3.1 Dimensión de Hausdorff

Continuando con el estudio de la dimensión, en concreto la dimensión de los fractales, se puede calcular la dimensión de cierto conjunto C recubriéndolo con conjuntos a diferentes escalas, en concreto, se escoge un número $d>0$ y se define un ***d*-cubrimiento de C** como:

$$C \subseteq \cup_{i=1}^{\infty} B_i$$

Continuando, se recubre C con conjuntos de diámetro menor que d. Calculando para cada d el valor más ajustado para el que:

$$H_d^s(C) = \inf\{\sum_{i=1}^{\infty} |B_i|^s \,;\, C \subseteq \cup_{i=1}^{\infty} B_i\}$$

Sobre todos los d-cubrimientos de C, con $s>0$ y siendo $|B|$ el diámetro de B se define finalmente la **medida de Hausdorff** como:

$$H^s(C) = \lim_{d \to 0} H_d^s(C)$$

Debido a que este es un proceso de cálculo muy complicado y costoso se han desarrollado algoritmos que aproximan esta medida utilizando el mismo principio de recubrimiento. Uno de estos algoritmos es el **Algoritmo de Conteo de Cajas o Box-Counting**.

2.1.3.2 Dimensión por Box-Counting

Para definir este algoritmo para aproximar la dimensión de un conjunto se ejemplificará con el cálculo de la dimensión de la longitud de la costa de Reino Unido.

Este algoritmo consiste en tomar una malla en el plano dividida en cuadrados iguales de lado no nulo de manera que cubran la figura (A) a medir completamente en un polígono rectangular subdividido en estos cuadrados. El algoritmo consiste en recoger el número de cuadrados que son intersectados por A y luego dividirlo por el número total de "cajas" que conforman la malla.

Figura 2.4 : Box-Counting de la costa de Reino Unido.

A medida que se disminuye el lado de las cajas se aproxima mejor la dimensión de dicha figura. Esto se puede generalizar matemáticamente tomando el número de cajas intersectadas *N(A)* y l longitud del lado de las cajas obteniendo la aproximación de la dimensión de A con:

$$\dim(A) = \lim_{l \to 0} \frac{\log N(A)}{-\log(l)}$$

Esto es una aproximación muy cercana a la **dimensión de Hausdorff** cuando este límite existe. En el caso del ejemplo, Reino Unido tiene una dimensión aproximada de 1,21.

2.2 Sistemas Dinámicos

Antes de mostrar una definición de sistema dinámico se define un espacio métrico como un conjunto X junto con una métrica o distancia d definida sobre X donde d es una aplicación:

$$d : X \times X \to \mathbb{R}$$

Ahora, sea X un espacio métrico y f una aplicación de X en sí mismo, el estudio del Sistema Dinámico abarca el comportamiento de las iteraciones de f según los diferentes valores de un punto inicial cualquiera introducido. La sucesión $f^n(x)$ de las iteraciones de f en un punto dado se denomina órbita de ese punto.

Un ejemplo de un Sistema Dinámico unidimensional es la ecuación de Maltus, que monitoriza la evolución de una población de una determinada especie definiendo la relación de la población en un "instante" k con el instante siguiente. Esta relación fue definida como:

$$x_{k+1} = x_k + \mathrm{d}x_k = (1 + \mathrm{d})x_k = \mathrm{c}x_k$$

Uno de los elementos más destacables de cara al estudio de un sistema dinámico es la **órbita**. Tomando valor inicial x_0, se llama órbita de dicho punto al conjunto de valores obtenidos por el sistema en cada fase con valor inicial x_0. Estas pueden ser constantes, dando lugar a **puntos de equilibrio atractivos**, si a lo largo de las fases del sistema, la órbita acaba en la repetición de los mismos valores en cada fase.

Estos puntos de equilibrio pueden ser también **repulsivos** si en vez de acabar en ellos la órbita, todos los valores iniciales cercanos provocan órbitas que escapan de estos puntos de equilibrio.

Finalmente, también existen **k-ciclos**, que son conjuntos de puntos atractivos que se repiten de forma constante a lo largo de la órbita.

2.2.1 Sistemas Dinámicos Cuadráticos

El ejemplo visto anteriormente se trataba de una aplicación lineal simple, para poder definir ciertos tipos de fractales, es importante el estudio de aplicaciones cuadráticas.

En este espacio de aplicaciones existen dos familias de Sistemas Dinámicos muy importantes, siendo estas:

- Familia Cuadrática: es la formada por aplicaciones de la forma
- $f_p(x) = x^2 + p$ con p real.
-
- Familia Logística: formada por aplicaciones $X: [0,1] \rightarrow [0,1]$ de la forma $X(n+1) = X(n) * c * (1 - X(n))$ con c real.

Esta última familia es especialmente interesante ya que varios algoritmos de generación de fractales se basan en ella para dibujarlos.

Si se asigna c valor real en el intervalo [0,4], si se dibujan los valores que toman todas las aplicaciones tras muchas iteraciones se obtiene el llamado mapa logístico o diagrama de Feigenbaum de la Figura 2.5.

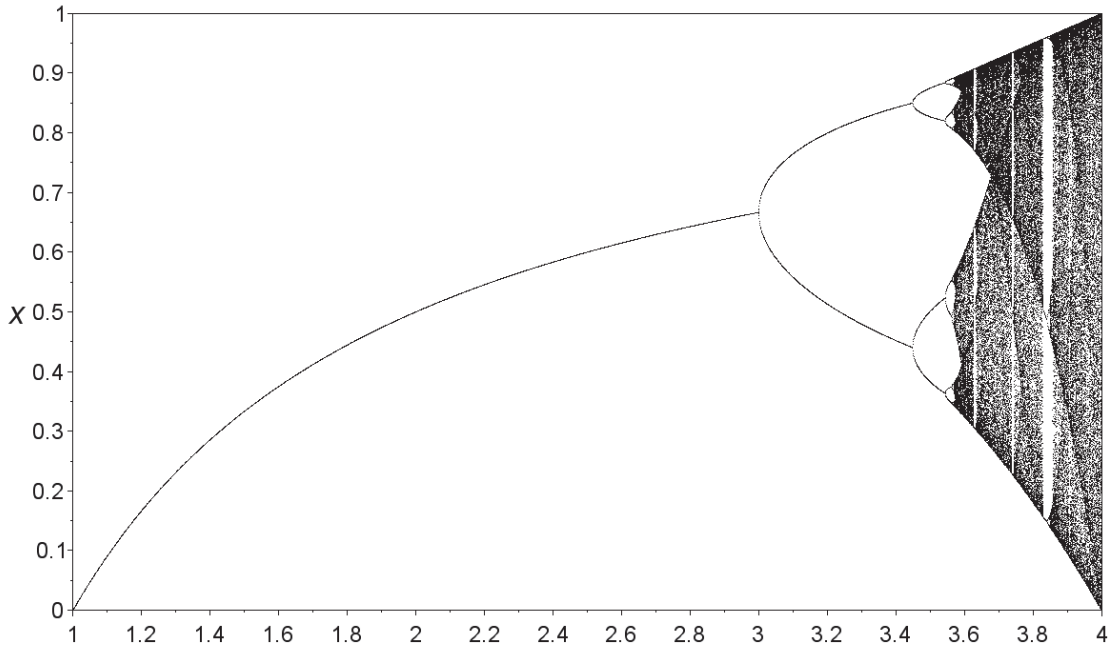


Figura 2.5: Diagrama de Feigenbaum

Lo que muestra que, por ejemplo, del 1 al 3, todos los valores de c hacen que el sistema se estabilice en un único valor, del 3 al 3.4 diverge en un 2-ciclo, etc. Las zonas "coloreadas" se trata de zonas caóticas, que no acaban en un valor repetido hasta la iteración infinita. Volviendo a converger en ciclos, por ejemplo, en el punto 3.9.

De hecho, existe una constante en relación con las bifurcaciones que duplican el periodo en el diagrama; esta constante se llama Constante de Feigenbaum y se calcula:

$$\delta = \lim_{n\to\infty} \frac{a_{n-1}-a_{n-2}}{a_n-a_{n-1}} = 4.669201\ldots$$

2.2.2 Algoritmos de Escape

También conocidos como Algoritmos de Tiempo de Escape, se trata de una familia de algoritmos que se sirven por lo general de Sistemas Dinámicos en los que se introduce el valor que se quiere comprobar si pertenece al fractal que se pretende dibujar. Una vez introducido el punto *P* a evaluar se realizan *n* iteraciones en el Sistema Dinámico y si el punto enésimo resultante se encuentra a menos de una cierta distancia (considerada previamente) del punto central del sistema, es decir, **no escapa** de dicho rango; entonces *P* pertenece al fractal que se pretende dibujar y se introduce en el conjunto.

En cuanto a los Algoritmos de Tiempo de Escape, se trata de una variación de los previamente explicados, en estos, se guarda también la iteración en la que "escapan" del rango y, usualmente, se dibujan con distinto color en según qué iteración hayan escapado. Para más variaciones de estos algoritmos también se puede tener en cuenta en la forma de dibujar los puntos el ángulo de escape o la órbita en la que queda atrapado el Sistema Dinámico, esto último ocurre cuando un Sistema Dinámico queda "atrapado" en una serie de valores concretos saltando de uno a otro de forma secuencial.

2.2.2.1 Conjunto de Mandelbrot

Este conjunto se representa mediante un Algoritmo de Escape, la forma más sencilla de representarlo es tomar el Sistema Dinámico:

$$\mathbb{C} \to \mathbb{C}$$

$$Z_{n+1} \to Z_n^2 + C$$

Se evalúa desde el origen de coordenadas (representación de coordenadas complejas), posteriormente se pasa a evaluar puntos *P* en *n* iteraciones, si el valor resultante tras las iteraciones se encuentra a una distancia menor que 2 entonces el punto *P* pertenece al fractal de Mandelbrot. El conjunto resultante es el mostrado en la Figura 2.6.

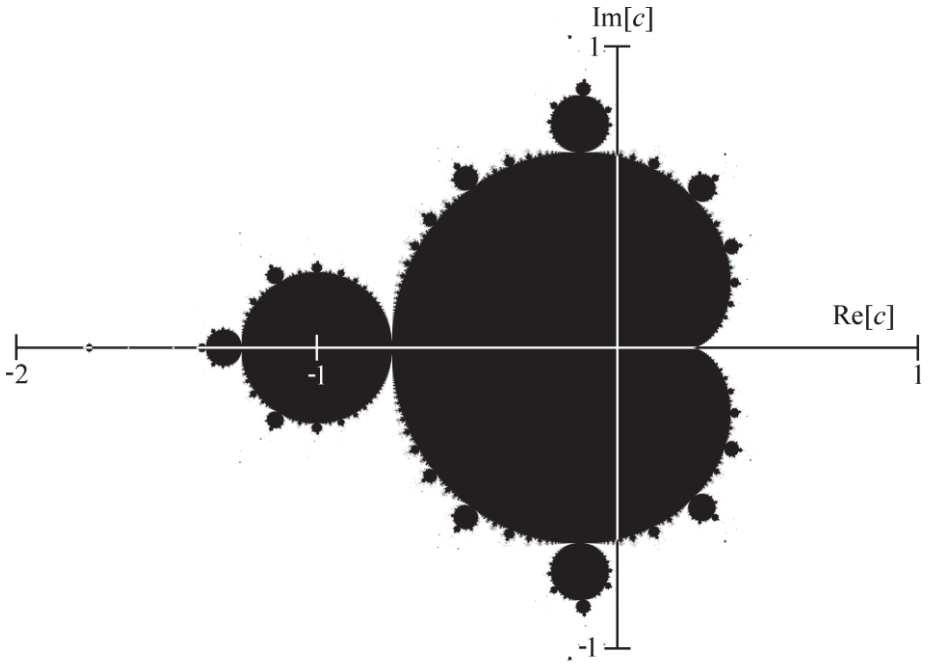


Figura 2.6: Conjunto de Mandelbrot

2.2.2.2 Conjuntos de Julia

Esta familia de conjuntos también se obtiene mediante la misma familia de aplicaciones que el conjunto de Mandelbrot, esta es:

$$\mathbb{C} \to \mathbb{C}$$

$$Z_{n+1} \to Z_n^2 + C$$

Para un número C, el conjunto de Julia de C es el conjunto frontera de las regiones de atracción de los puntos periódicos fijos o periódicos atractivos.

Una de las propiedades del del conjunto de Julia es que el conjunto de Mandelbrot son aquellos C, para los que el conjunto de Julia es conexo.

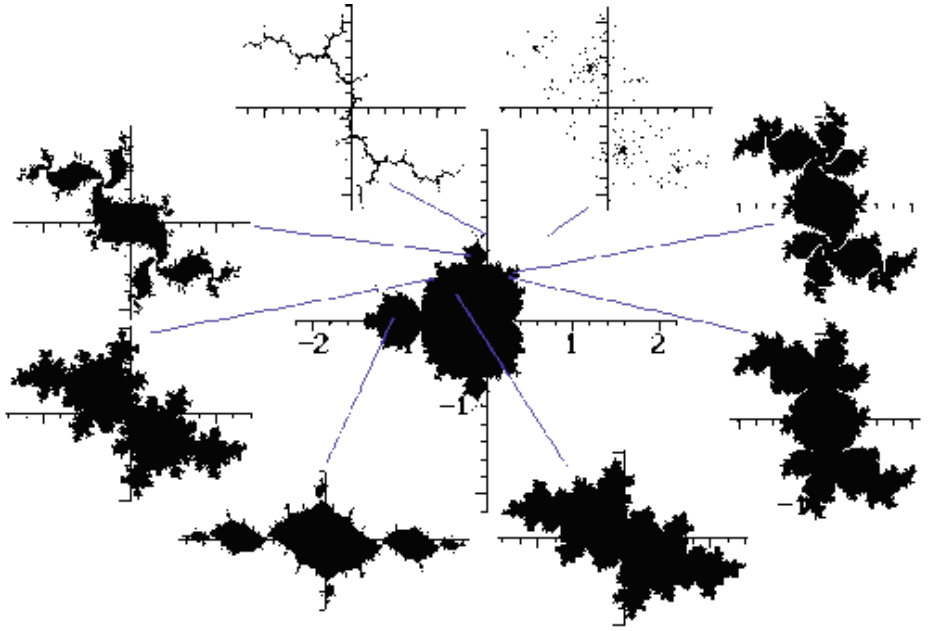


Figura 2.7: Conjuntos de Julia para varios C en el conjunto de Mandelbrot.

2.3 Algoritmos para la Generación de Fractales Autosemejantes

Antes de comenzar con los algoritmos definimos un **Sistema Dinámico** como un sistema que evoluciona con el tiempo, en concreto, hablaremos de Sistemas Dinámicos que se sirven de ecuaciones cuya variable depende de los valores anteriores, es decir, **recursivos**.

2.3.1 Sistemas de Funciones Iteradas (IFS)

Son una herramienta para construir fractales que presenten autosemejanza mediante aplicaciones afines. En concreto, se trata de sistemas de funciones que utilizan aplicaciones afines de carácter contractivo que son aplicadas recursivamente a un conjunto inicial o semilla. Estas aplicaciones contractivas son una composición de isometrías, es decir, traslaciones, giros y simetrías; y una homotecia contractiva que hace que el proceso iterativo converja.

Estos sistemas de funciones se aplican sobre un conjunto de datos de manera recursiva, resultando en un conjunto límite fractal que es el atractor del sistema y siendo una de las maneras más sencillas de obtener los fractales clásicos como el triángulo de Sierpinsky o la curva de Koch.

2.3.1.1 Definiciones

- Aplicación Afín:
  Una aplicación afín en $\mathbb{R}^n$ es cualquier aplicación de la forma:

  $$f(x) = Ax + b$$

  Con A $\epsilon \mathbb{R}^{nxn}$ y $b \in \mathbb{R}^n$
- Norma de A en $\mathbb{R}^n$:

  $$\|A\| = \sup_{x \neq y} \frac{d(Ax, Ay)}{d(x, y)}$$
- Contractividad de una Aplicación Afín:
  En $\mathbb{R}^n$, una aplicación afín es contractiva si y solo si $\|A\| < 1$

2.3.1.2 IFS

Se llama un sistema de funciones iteradas a cualquier conjunto finito de aplicaciones afines contractivas tal que:

$$\{f_1, f_2, \dots, f_N\} = \{f_i : 1 \leq i \leq N\} = \{f_i\}_{i=1}^{N}$$

Si $0 \leq r_i < 1$ es la razón de contractividad de f_i, $1 \leq i \leq N$ se llama razón del IFS a:

$$r = máx\{r_1, r_2, \dots, r_N\} \quad (0 \leq r < 1)$$

Para todos los sistemas de funciones iteradas que toman dichos valores de r entre 0 y 1 existe un único fractal llamado **atractor del IFS**.

Para los sistemas de funciones iteradas existen dos tipos de algoritmos que dibujan el atractor de dicho sistema.

Algoritmo Determinista

De forma general, para dibujar el fractal A mediante un algoritmo determinista, se elige un conjunto inicial no vacío B al que llamamos semilla.

Posteriormente se borra B y se calculan las funciones del IFS, asignando B a la unión de los conjuntos resultantes; se prosigue dibujando B. Este proceso se repite *n* iteraciones para obtener un fractal a medida que se va avanzando la *n*.

Un ejemplo de algoritmo determinista es el del Triángulo de Sierpinsky de la Figura 2.2, cuyo sistema de funciones iteradas sería:

$$f_1(\mathrm{x},\mathrm{y}) = \frac{1}{2}\cdot(\mathrm{x},\mathrm{y})$$

$$f_2(\mathrm{x},\mathrm{y}) = \frac{1}{2}\cdot(x,y) + \left(\frac{1}{2},0\right)$$

$$f_3(\mathrm{x},\mathrm{y}) = \frac{1}{2}\cdot(x,y) + \left(\frac{1}{4},\frac{\sqrt{3}}{4}\right)$$

Algoritmo Aleatorio

Se trata de un tipo de algoritmo en el que se le asignan ciertas probabilidades a cada función del IFS.

Comienza tomando un punto *x* arbitrario inicial y se comienza un bucle de 1 hasta *n* iteraciones en las que primero se elige un número natural *j* del *1* a *N* con probabilidades *p* asignadas a cada *j* donde *N* es el número de funciones del IFS, se sigue calculando la función $f_j(x)$, se guarda la solución *y* y se repite el bucle utilizando esta solución para el próximo *x*. Siempre se suelen descartar los primeros 50 o 100 puntos y representando del orden de 5000 se obtiene una buena representación del atractor.

Un ejemplo de este algoritmo es el llamado Juego del Caos, que acaba representando el Triángulo de Sierpinsky de esta manera:

Se definen los tres vértices *A*, *B* y *C* del triángulo, se escoge un punto *x* cualquiera en el plano y se lanza un dado, si sale 1 o 2 se elige *A*, 3 o 4 se elige *B* y 5 o 6 se elige *C*. Una vez lanzado imaginemos que ha salido 2, en ese caso se guarda el punto medio *y* entre el punto *x* y el punto *A*, se dibuja, se asigna *x=y* y se repite el proceso calculando para la iteración *i+1* el punto medio entre el punto dibujado en la iteración *i* y el punto aleatorio que haya salido en el dado.

3 Fractales y Música

Como se introducía en el punto 1.1 existe una gran relación entre la Geometría Fractal y la Música, en este punto se propondrán estudios en la geometría fractal de algunas piezas musicales, así como algoritmos utilizados para generar composiciones a partir de fractales y sistemas dinámicos conocidos.

Desde la publicación de "La Geometría Fractal de la Naturaleza" de Mandelbrot, algunos músicos e investigadores han estudiado y utilizado las ideas detrás de la teoría. La generación de datos simbólicos, el modelado de estructuras musicales y de sonido están entre las aplicaciones de esa teoría. El concepto de autosemejanza, está detrás de este modelado.

3.1 Sonido y Composición

Antes de relacionar la Geometría Fractal con la Música se debe presentar el sonido, ciertas características físicas que presenta, así como ciertos tipos. Posteriormente se presentarán algunos conceptos de Teoría Musical básicos que serán de utilidad para la comprensión de ciertos estudios, además de ciertos algoritmos y su implementación.

3.1.1 El Sonido en la Música

El sonido se puede definir según la RAE como vibración mecánica transmitida por un medio elástico, esto no tiene por qué significar que cierta vibración sea audible por el ser humano, pero nos ayuda a explicar ciertas propiedades de este fenómeno físico.

Sin entrar en mucho detalle, el sonido se puede interpretar como una onda mecánica clásica cuyas variables principales son:

- Amplitud de la onda: esta es la que indicará la energía del sonido y, por lo tanto, la intensidad con la que lo percibe el ser humano.
- Frecuencia: depende de la velocidad angular de la onda y dependiendo de esta los seres humanos perciben un tono u otro. Esta se mide en Hercios (Hz).

Además de estas variables se debe mencionar que, según la teoría de Fourier, cualquier función periódica se puede descomponer en suma de otras funciones simples sinusoidales cuya frecuencia es múltiplo de la primera. Estas "ondas secundarias" se llaman armónicos.

Enlazando con las composiciones musicales, estas se sirven esencialmente de sonidos y silencios, que, combinados de diferentes maneras, conforman lo que llamamos música. Existen también ciertas propiedades o características que se tienen en cuenta para la música, siendo estas:

- Tono: Está relacionado con la frecuencia de la onda sonora y se traduce en una audición para el ser humano más aguda cuanto más aumente la frecuencia y más grave cuanto menor sea. El rango auditivo del ser humano se encuentra entre 20 y 20000 Hz.
- Duración: Se refiere al tiempo en la que una nota concreta, un sonido, se encuentra vibrando, este se mide en lo largo o corto del sonido y, como veremos, se traduce en las figuras musicales de negra, blanca, corchea, etc.

- Intensidad: Igual que como ocurría con el tono, la intensidad es directamente proporcional a la amplitud de onda. Esta se puede medir en decibelios (dB) y en notación musical vendrá indicada como forte, mezzoforte, piano, etc.
- Timbre: Se trata de una cualidad del sonido que permite la diferenciación de la fuente del sonido, en un ejemplo en el caso musical, sería la diferenciación de dos notas iguales emitidas por un piano o por un violín. Esto se debe a las distintas ondas armónicas que conforman el sonido de, en este caso, cada uno de los instrumentos. Por lo tanto, distintos armónicos generarán distintos timbres, aunque el tono, la duración y la intensidad sean las mismas.

3.1.2 Notación Musical

En la música, para la comprensión e interpretación, se deben definir ciertas reglas sobre la representación de esta. Generalmente, la música se representa en un pentagrama, que se trata de cinco líneas horizontales, equidistantes y paralelas. El elemento básico que aparece en un pentagrama son las denominadas notas, que representan el tono (es decir, la frecuencia) y la duración del sonido a representar. La primera figura que aparece en un pentagrama es lo que se llama la clave, que es la referencia inicial para las notas que posteriormente se representarán a lo largo del pentagrama, esto es, si aparece una "clave de sol" significa que la segunda barra del pentagrama empezando desde abajo representará la nota sol.

Las notas actuales son do, re, mi, fa, sol, la y si; divididas en 12 semitonos distintos (existen "notas" intermedias llamadas fa#, si$b$, etc.). En el estándar internacional que se utiliza en la mayoría de los países, se puso como referencia el llamado la_4 cuyo tono son 440Hz y a partir de este se calculan el resto de las notas, sabiendo que el la_3 son 220Hz y el la_5, 880Hz. Todas las notas están escaladas en este sentido, teniendo cualquier nota a X Hz su equivalente una octava superior está a $2X$ y así sucesivamente. Teniendo esto en cuenta y asignando a cada nota un número entero n comenzando por el do (1) y a cada octava otro x (de 1 a 8 debido al rango de audición) se puede proponer la siguiente fórmula para saber la frecuencia asignada de cada nota existente:

$$f(n,x) = 440 \cdot \left(\sqrt[12]{2}\right)^{(x-3)\cdot 12+(n-10)}$$

Esta relación será muy importante en algunos estudios mostrados posteriormente.

Las notas en un pentagrama no solo representan un tono, sino que también deben decir cuánto tiempo debe durar el sonido de dicha nota, esto se representa mediante la forma que tiene la nota en el pentagrama, en contrapunto al tono, que se representa en el lugar en el que está colocada la nota. Las figuras en orden decreciente son *redonda, blanca, negra, corchea, semicorchea, fusa* y *semifusa.* Comenzando con la *redonda* cada figura representa el doble de duración que la siguiente.

3.2 Ruido 1/f

En este punto revisamos uno de los primeros ejemplos históricos en los que se relacionan los fractales y la música, este es, el ruido 1/f.

Ruido, antes de especificar, se denomina a una señal con cierto componente aleatorio. El caso más obvio sería el de una señal acústica, pero el "ruido" puede aparecer en las mediciones de numerosas magnitudes físicas, e incluso medidas en la naturaleza tales como en la lluvia o el sonido del viento.

Se definen varios tipos de ruido dependiendo de la densidad espectral de la potencia de su señal, tres de ellos serán los más relevantes para este estudio, estos son: ruido blanco, ruido rosa o 1/f y ruido marrón, también llamado browniano o $1/f^2$. Esta densidad muestra la distribución de las frecuencias de una señal frente a la potencia que cada una presenta. Los tres tipos de ruido muestran, por lo tanto, diferentes tipos de dependencia de la densidad espectral, siendo:

- Ruido Blanco: Sonido completamente aleatorio, todas las frecuencias presentan la misma densidad espectral, por lo que se encuentran igualmente presentes en todos los rangos energéticos (potencia).
- Ruido $1/f^2$: Ruido muy dependiente de la frecuencia, estas muestran una relación fuerte con la potencia, desde las frecuencias menores existe mayor densidad acabando en menor a medida en que la frecuencia aumenta. Esto se ve representado en un sonido que varía muy poco entre frecuencias cercanas, sin saltos abruptos.
- Ruido 1/f: Intermedio entre los anteriores, no presenta una dependencia tan marcada como el ruido $1/f^2$, pero sí ciertos atisbos de correlación. Este fenómeno aparece, por ejemplo, en las crecidas del río Nilo [5] o incluso en los latidos del corazón humano [6].

Esto se puede observar en la Figura 3, con el Ruido Blanco aleatorio, el 1/f con saltos menos pronunciados y cierta coherencia entre puntos; y el $1/f^2$ sin ningún salto entre puntos más pronunciado que otro.

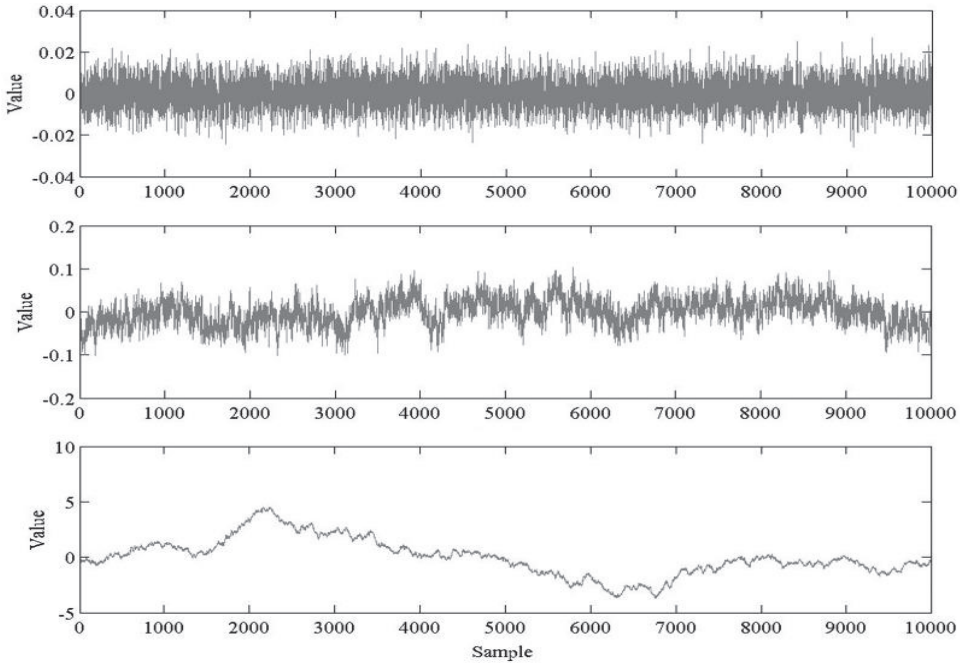


Figura 3.1: Por orden: Ruido Blanco, Ruido 1/f y Ruido $1/f^2$

Como cabría esperar, si alguno de estos tipos de ruido está relacionado con la música, se trata del 1/f, ya que el ruido blanco traducido, por ejemplo, en notas musicales sería demasiado aleatorio y no se alcanzaría ningún sentido musical; ni el ruido $1/f^2$, ejemplificando subidas y bajadas de las escalas musicales

secuenciales y, por lo tanto, “aburrido”. El Rosa, efectivamente, es el que tiene sentido para una generación aleatoria de música y, de hecho, se ha comprobado en ambos sentidos.

En [2], se estudió primero la aparición del ruido 1/f en diferentes estaciones de radio, piezas musicales e incluso con el propio habla; probando que, en efecto, esta clase de ruido aparece en casi la totalidad de las fuentes probadas. Posteriormente también realizaron una prueba de generación estocástica musical en la que crearon varias piezas musicales utilizando los tres tipos de ruido antes mostrados, siendo las más apreciadas las generadas mediante el 1/f.

3.3 Fractales en la Música

Lo que no se logra probar en ese último estudio es la naturaleza fractal de la música, comenzando en 1989, en [3] se realiza un análisis matemático con varias piezas de música clásica, basándose precisamente en los estudios presentados en el punto anterior que ponen el foco en el estudio de las frecuencias. Presentando un estudio que enfrenta la incidencia de cada intervalo entre notas sucesivas y calculando dimensiones fractales; todo ello obteniendo relaciones fractales muy altas en la mayoría de las composiciones.

Tan sólo un año más tarde, los mismos autores de este estudio, mostrarían en [7] la relación entre la autosemejanza y el ruido 1/f, mostrando la densidad espectral primero de obras ya demostradas fractales por ellos mismos anteriormente; luego se escriben de forma secuencial las notas musicales de cada obra (sus frecuencias) y se hacen estudios a 6 escalas diferentes, realizando una reducción desde la obra completa hasta escala 32:1, simplificando y fusionando frecuencias. El resultado es autosemejanza en todas las escalas, probando la relación entre el ruido 1/f, los fractales y la música.

Esto quiere decir, que en [2], se consigue por primera vez generación de música en relación con la Geometría Fractal.

3.4 Programas y Algoritmos Existentes para la Generación de Música Fractal

3.4.1 MusiNum

Comenzando por el programa MusiNum [8], este utiliza un principio simple para la creación de secuencias numéricas, luego traducidas en secuencias musicales. Este método se vale, entonces, de la llamada secuencia de Morse-Thue (que otros programas también utilizarán), esta es una secuencia binaria que puede generarse de manera recursiva, un ejemplo sería utilizando números en base 2, comenzando con un 0 y continuando cada iteración añadiendo el complementario al número anterior. Esto sería: 0, 01, 0110, 01101001, 0110100110010110, ...

La autosemejanza de los conjuntos es dada por la propia definición de la secuencia, que con ciertas modificaciones puede convertirse en una melodía. Tomando por ejemplo la secuencia 0, 1, 2, 3... se traduce a binario y se escribe en la melodía la suma de los dígitos de cada número, esto se puede aplicar a cualquier tipo de secuencia, dando lugar a melodías de carácter autosemejante.

3.4.2 FractMus 2000

Se trata de un programa desarrollado en C++ con una enorme variedad de algoritmos implementados, además de numerosos tipos de instrumentos, canales, escalas, notas y, en general, funcionalidades.

Algunos de los algoritmos que este programa implementa y que se implementarán más adelante en este Trabajo se muestran a continuación.

3.4.2.1 Morse-Thue

Igual que el anterior programa MusiNum, este implementa de forma casi completa este tipo de algoritmo, permitiendo introducir valores como la base en la que se elige trabajar y calculando con el número de notas en la escala también elegida; por lo tanto se debe elegir bien la escala (es decir, 8 notas concretas dadas por la Teoría Musical), el número de escalas o multiplicador, ya que 8 quizás no generan tanta variedad como se desearía (se escogen 2, 3, 4, …) y la base en la que se quiere fundamentar la secuencia.

3.4.2.2 Secuencia Earthworm

Se trata de una secuencia muy simple en la que se toma un número inicial entre un rango de valores y se elige un multiplicador *X*; la secuencia consiste en multiplicar el valor en cada iteración por *X*, si el valor obtenido sobrepasa el techo del rango elegido, se realiza el módulo de este valor y se reintroduce en el algoritmo "recortado".

3.4.2.3 Autómata Celular Unidimensional de Wolfram

Un autómata celular es una serie de reglas aplicadas a un conjunto de interruptores o "células" que tienen varios estados, en general, encendido o apagado, o, "viva" o "muerta". Las reglas que se aplican definen el comportamiento de cada célula en cada iteración dependiendo de las células que la rodean e incluso de su propio valor. Un autómata celular sería, por ejemplo, un cuadrado dividido en N cuadrados iguales (células) y cuya regla para el valor de cada célula sería que, si el valor de las células encima y a la derecha de la célula a calcular tienen el mismo valor, entonces la célula en cálculo cambia de estado, sea cual sea este; si los valores de las adyacentes son distintos, la célula mantiene su estado. Se propone entonces un estado inicial del autómata y se comienza la ejecución. La Figura 3.2 muestra el comportamiento de este autómata.

Figura 3.2: Autómata Celular durante 4 iteraciones

Estos autómatas se encuentran enormemente relacionados con los fractales, de hecho, muchos pueden ser representados por este tipo de autómatas escogiendo las reglas y el estado inicial adecuados, este es el caso, por ejemplo, del Triángulo de Sierpinsky.

En cuanto al autómata de FractMus 2000, trata cada nota que se va a representar como una célula con dos células adyacentes, una a la izquierda y

otra a la derecha, se introducen las reglas y el estado inicial y se puede representar cada estado del autómata de forma secuencial.

3.4.2.4 Sistema Dinámico *3n + 1*

Se comienza eligiendo cualquier número entero mayor que 1, y se definen dos funciones, si es par, el número se divide entre 2; si es impar, el número se multiplica por 3 y se le suma 1. Este proceso se conoce que en algún momento siempre acaba en 1 se escoja el número que se escoja, pero no se ha podido probar matemáticamente.

3.4.2.5 Mapa Logístico

Se trata de una familia de Sistemas Dinámicos caóticos de la forma:

$$X(n+1) = X(n) * a * (1 - X(n))$$

El *mapping* en este caso se obtendrá de recoger el valor entre 0 y 1 resultado del sistema y se escalará asignándole el valor entero que le corresponda en las escalas musicales elegidas.

3.4.2.6 Ruido 1/f

Históricamente ya se ha probado que existen métodos para generar música mediante este tipo de ruido. Una función que simula muy bien el ruido 1/f es:

$$X(n+1) = (X(n) * n) + r * (\sqrt{1 - n^2})$$

En la que se obtienen, igual que con el Mapa Logístico, valores en el intervalo [0,1], realizando el *mapping* de la misma forma que en el caso anterior, escalándolo según las variables escogidas.

4 Implementación de Algoritmos

4.1 Introducción

En este punto se comenzará a dar forma a algunos de los algoritmos de generación de música fractal, además de la implementación e hibridación de nuevos algoritmos de manera que se consiga cierta calidad musical.

Los algoritmos que se presentarán están basados en los siguientes algoritmos o sistemas dinámicos, adaptados para la generación de música:

1. Juego del Caos/Triángulo de Sierpinsky
2. Morse Thue Sequence
3. Earthworm Sequence
4. 3n+1
5. Mapa Logístico
6. Ruido 1/f

Concretamente, en el punto 4.2 se evaluará la evolución que ha llevado al desarrollo final de los algoritmos y su *mapping*, esta se concretará en este punto describiéndola para el algoritmo del Mapa Logístico, ya que la estructura de todos los algoritmos es similar y sería redundante.

Más adelante, en el punto 4.3 se analizarán y validarán los resultados obtenidos para cada algoritmo desarrollado. Y finalmente, en el punto 4.4 se describirá el desarrollo y pruebas de algoritmos híbridos y nuevas técnicas.

A lo largo de los siguientes puntos, se han insertado algunos audios con unos segundos de reproducción de la pista, para que los archivos no fueran demasiado pesados se han introducido tan solo unos pocos. Todos los audios completos de todas las ejecuciones mostradas se encuentran en:

https://drive.google.com/drive/u/2/folders/1i0ZuCfLUfab2dcx6LsgfLimCz6R5I7iU

Nota: Para poder escuchar los audios insertados en la memoria es necesario abrir este PDF con Adobe Acrobat y seleccionar la opción de "Considerar este documento de confianza".

4.1.1 Elección de Interfaz y Entorno de programación

Con FractMus [9] como ejemplo, se puede apreciar que la programación de anteriores aplicaciones para la generación de música utilizando Fractales y/o Sistemas Dinámicos, se encuentra algo desactualizada, haciendo uso de tecnología como C++ y enormes cantidades de código, resultando en aplicaciones pesadas, por lo tanto, tras la investigación previa de paquetes de programación de tratamiento de audio se ha elegido la librería *open source* "*pydub*" [10] de python para la composición y edición de audio con el que se desarrollarán todos los algoritmos presentados. Se trata de una librería que permite, como elementos clave para el desarrollo de este trabajo, la modificación del tono de un sonido, la sobreposición de sonidos en una misma pista y la elección de posición del sonido a insertar en cualquier lugar de la pista.

Debido a problemas surgidos a lo largo de la programación de las versiones iniciales de los algoritmos (Punto 4.2), se investigan nuevas librerías que ofrezcan tratamiento de audio y mejor gestión de los cambios de tono e introducción de figuras musicales, encontrando la librería *librosa* [11], que ofrece funcionalidades que complementan las de *pydub* en las versiones finales de los algoritmos.

Para el entorno de programación y para facilitar la sencillez de ejecución, se ha elegido Google Colaboratory, permitiendo la subida y tratamiento de archivos de audio en la nube, además del acceso a todas las librerías públicas de python.

4.1.2 Diseño

Previamente a comentar la estructura generalizada de los programas realizados que implementan cada algoritmo, se comenzará por presentar las decisiones de diseño sobre el tratamiento de archivos de audio.

4.1.2.1 Nota de referencia

Como ya se ha comentado desde el comienzo, uno de los problemas que tiene la creación musical por medio de inteligencias artificiales es la gran carga que conlleva el entrenamiento y, por lo tanto, tener una cantidad de datos significativa en un entorno activo para poder llevar a cabo el entrenamiento de la IA con eficacia; e incluso programas como [9] que se sirven de una librería de archivos enorme con todas las notas de cada instrumento presentes en el programa, haciéndolo pesado aunque efectivo en cuanto a ejecución. Debido a este problema, en los algoritmos desarrollados en este Trabajo, gracias a la librería "pydub", solo es necesaria una nota de cada instrumento para el cálculo del resto de notas, haciendo posible un programa ligero y con recursos optimizados.

A continuación, se ejemplifica una transformación estándar de la nota referencia, para ello tomaremos como nota de referencia el La Mayor de la 3ª escala (A3) de piano, que se caracteriza por tener una frecuencia de 440Hz.

Una de las funcionalidades de la librería es poder "estirar" y "acortar" un archivo de audio acelerándolo o ralentizándolo, cambiando así la frecuencia de los sonidos que aparecen en este. Por lo tanto, en un principio, se utiliza la fórmula del punto 3.1.2 para traducir A3 en otra nota cualquiera. Siguiendo con el ejemplo, para conseguir A4 (La Mayor de la 4ª escala), es decir, la nota 10 de la escala 4; la fórmula entonces será:

$$f(n,x) = 440 \cdot \left(\sqrt[12]{2}\right)^{(x-3)\cdot 12+(n-10)} = 440 \cdot \left(\sqrt[12]{2}\right)^{(4-3)\cdot 12+(10-10)} =$$

$$= 440 \cdot \left(\sqrt[12]{2}\right)^{12} = 880Hz$$

Así pues, cada vez que el algoritmo ejecutado devuelva una nota, esta se calcula a partir de la de referencia en cada iteración del algoritmo.

Más adelante en el desarrollo, este cálculo se realiza de manera externa ya que se obtiene acceso a la librería *librosa* [11], que permite de igual manera un cambio de tono mediante una única función.

4.1.2.2 Estructura de los Algoritmos

Todos los programas asociados a cada algoritmo tienen una estructura similar, que varía según la fórmula única de cada Fractal o Sistema Dinámico, además de diferenciarse en las técnicas de *mapping* en algunos de los casos.

Dicha estructura se puede observar en el código de una de las primeras pruebas realizadas, se trata de la versión 0 del algoritmo *3n+1*:

```
#INICIALIZACION DE VARIABLES#
seed = 27; notab = 28; notaB = 45;
Lugar = 1000; escala = notaB - notab;
Pista = AudioSegment.silent(duration=34000)

#BUCLE DEL ALGORITMO#
for i in range(135):
  #MAPPING#
  nota = seed
  nota = nota%escala + notab
  #TRANSFORMACION DE LA NOTA#
  Tono = int(A4.frame_rate * (2.0 ** ((nota-37)/12)))
  Tono = A4._spawn(A4.raw_data, overrides={'frame_rate': Tono})
  Tono = Tono.set_frame_rate(44100)
  #POSICION EN LA PISTA#
  Lugar = Lugar + 250
  #INSERCION DE LA NOTA#
  Pista = Pista.overlay(Tono, position=Lugar)
  #FUNCION DEL SISTEMA DINAMICO PARA#
  # CALCULAR LA SIGUIENTE ITERACION #
  if (seed % 2) == 0:
    seed = seed//2;
  else:
    seed = 3*seed + 1;

#EXPORT DE LA PISTA
Pista.export("pista.wav",format="wav")
```

Por lo tanto, se diferencian las fases del programa en:

1. Inicialización
2. Bucle
3. Mapping

4. Transformación de Tono
5. Posición en Pista
6. Inserción
7. Cálculo de la siguiente nota
8. Ir a 2 si la condición se cumple
9. Fin del bucle
10. Exportación de la Pista final

4.2 Evolución de los Algoritmos

En este punto analizaremos las decisiones tomadas en cuanto al diseño y *mapping* ejemplificados en el Algoritmo de la Familia Logística desde la versión 0 experimental hasta su versión final para un instrumento. Además, se analizarán los resultados obtenidos para compararlos entre versiones.

Antes de comenzar con el algoritmo, se muestra una pequeña traducción de las notas de piano convertidas en valores enteros únicos que permitirán la transformación directa del tono. Dicho *mapping* intermedio es:

```
def mapping(x):
    return {
        'A0': 1,'Bb0': 2,'B0': 3,
        'C1': 4, 'Db1': 5, 'D1': 6, 'Eb1': 7, 'E1': 8, 'F1': 9, 'Gb1': 10,'G1': 11,'A
b1': 12,'A1': 13, 'Bb1': 14, 'B1': 15,
        'C2': 16,'Db2': 17,'D2': 18,'Eb2': 19,'E2': 20,'F2': 21,'Gb2': 22,'G2': 23,'A
b2': 24,'A2': 25, 'Bb2': 26, 'B2': 27,
        'C3': 28,'Db3': 29,'D3': 30,'Eb3': 31,'E3': 32,'F3': 33,'Gb3': 34,'G3': 35,'A
b3': 36,'A3': 37, 'Bb3': 38, 'B3': 39,
        'C4': 40,'Db4': 41,'D4': 42,'Eb4': 43,'E4': 44,'F4': 45,'Gb4': 46,'G4': 47,'A
b4': 48,'A4': 49, 'Bb4': 50, 'B4': 51,
        'C5': 52,'Db5': 53,'D5': 54,'Eb5': 55,'E5': 56,'F5': 57,'Gb5': 58,'G5': 59,'A
b5': 60,'A5': 61, 'Bb5': 62, 'B5': 63,
        'C6': 64,'Db6': 65,'D6': 66,'Eb6': 67,'E6': 68,'F6': 69,'Gb6': 70,'G6': 71,'A
b6': 72,'A6': 73, 'Bb6': 74, 'B6': 75,
        'C7': 76,'Db7': 77,'D7': 78,'Eb7': 79,'E7': 80,'F7': 81,'Gb7': 82,'G7': 83,'A
b7': 84,'A7': 85, 'Bb7': 86, 'B7': 87,
        'C8': 88
    }.get(x,"error")
```

Donde se puede observar que se definen todas las notas desde el La Mayor más grave hasta el Do Mayor más agudo, siendo una aplicación biyectiva de las teclas de un piano tradicional hacia los números enteros del 1 al 88. Este *mapping* también es característico por ser el mismo principio que el utilizado por las tecnologías MIDI [12].

4.2.1 Logistic_v0

4.2.1.1 Algoritmo v0

Haciendo uso del Sistema Dinámico con el que se obtiene el Mapa Logístico, es decir, la Familia Logística vista en el punto 2.2.1, se utilizará un *mapping* muy simple basado en comenzar con un rango de notas dado en la inicialización por *notab* y *notaB*, con el que se obtiene la *escala*.

Además, como se vio en 3.4.2.5, los resultados de la función logística se encuentran en el intervalo (0,1), resultado que, por lo tanto, multiplicado por la *escala* y redondeado, obtiene un número entero en el intervalo (0, *escala*), sumado a la *notab* (donde comienza la escala), se obtiene una nota entre *notab* y *notaB* que será la introducida en la Pista Final.

Esta versión inicial se trata de una función $\mathbb{R}^1 \to \mathbb{R}^1 \to \mathbb{N}^1$ ya que existe otra variable a tener en cuenta, que es el *Lugar* en el que se situará la nota en la Pista, introduciendo una nota cada 250ms, haciendo que la pista sea una secuencia de *negras* a 240 golpes por minuto (bpm).

Un pequeño ejemplo del algoritmo es:

```
seed = 3.91; notab = 28; notaB = 45; x = 0.1;
Lugar = 1000; escala = notaB - notab;
Pista = AudioSegment.silent(duration=34000)
Puntos = np.array([])

for i in range(135):
  x = x * seed * (1 - x)
  nota = round(x * escala)
  nota = nota%escala + notab
  Tono = int(A2.frame_rate * (2.0 ** ((nota-37)/12)))
  Tono = A2._spawn(A2.raw_data, overrides={'frame_rate': Tono})
  Tono = Tono.set_frame_rate(44100)
  Pfijo = nota
  Lugar = Lugar + 250
  Pista = Pista.overlay(Tono, position=Lugar)
  Puntos = np.concatenate((Puntos,[Pfijo]),axis=0)

Pista.export("pista.wav",format="wav")
```

4.2.1.2 Resultados v0

Ejecución v0_r0

En una primera ejecución *r0* se ha comenzado por dar un valor al coeficiente de la función cercano al 4 para obtener un resultado “pseudoaleatorio” aunque con cierta geometría fractal, ya que es a partir de 4 cuando los resultados sí que se denominan caóticos.

Gráfica de las notas obtenidas:

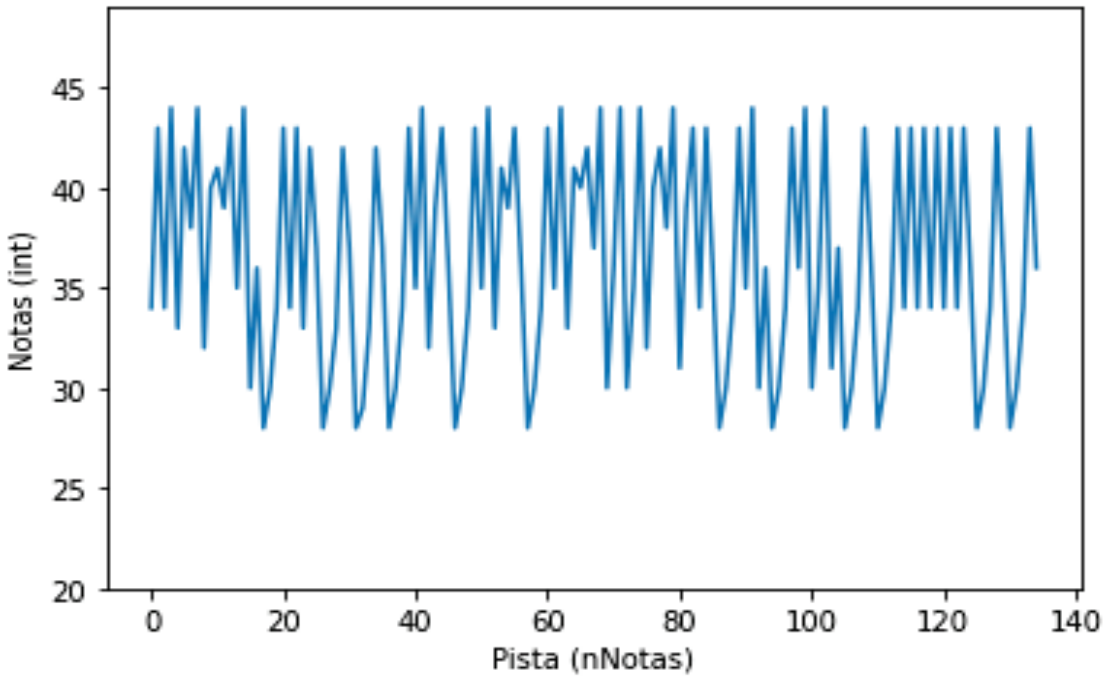


Figura 4.2.1: Notas en secuencia obtenidas por *logistic_v0_r0*

Tratándose de la primera versión del algoritmo, se han encontrado diversos problemas con el resultado obtenido, comenzando por una "calidad musical deficiente". La pista suena desafinada, esto se debe a que se han tomado las notas sin aplicar ninguna regla en cuanto a composición musical se refiere, es decir, las escalas musicales clásicas (elección de 7 de las 12 notas elementales que conforman una escala) como la escala de Do Mayor: Do, Re, Mi, Fa, Sol, La y Si; sin contar con los bemoles ni los sostenidos de cada nota.

Además, no se ha conseguido una secuencia de figuras *negras* a 240 bpm, insertando una nota cada 250ms; parece que hay variaciones de tiempo distintas entre cada nota. Esto es debido a la manera de transformar la nota de referencia en la nota calculada por parte de la librería *pydub*, ya que la forma en que se realiza es "estirando" o "aplastando" un archivo de audio de 4 segundos en el que la nota suena alrededor del segundo 1. Por lo que al estirar la pista, la nota comienza más tarde y al compactarla la nota comienza antes, haciendo que se desacompasen las notas aunque la inserción de cada fragmento sí sea cada 250ms. Esto se puede apreciar en la Figura 4.2.

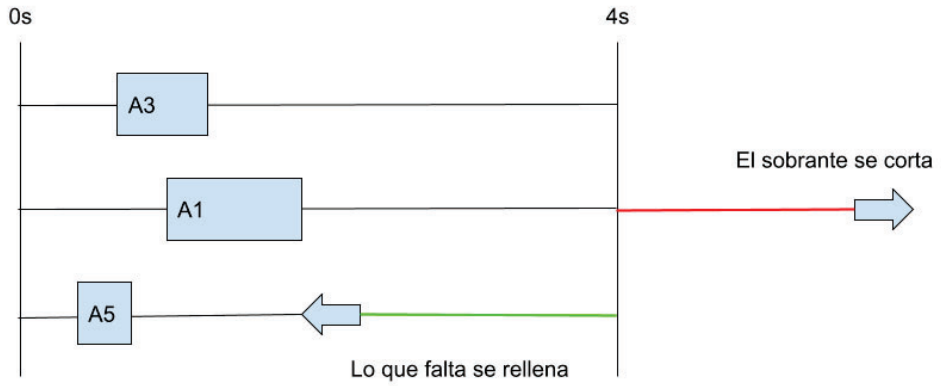


Figura 4.2: Traducción de notas mediante *pydub*

Ejecución v0_r1

Para la validación del algoritmo, se muestran también los resultados de la ejecución *r1*, en la que se escoge como coeficiente de la Función Logística un valor entre 3 y 3.5 ya que se conoce que entre estos valores se consiguen k-ciclos. En concreto se ha aplicado *seed = 3.44949*, el resto de los valores se mantienen igual que en la ejecución *r0*.

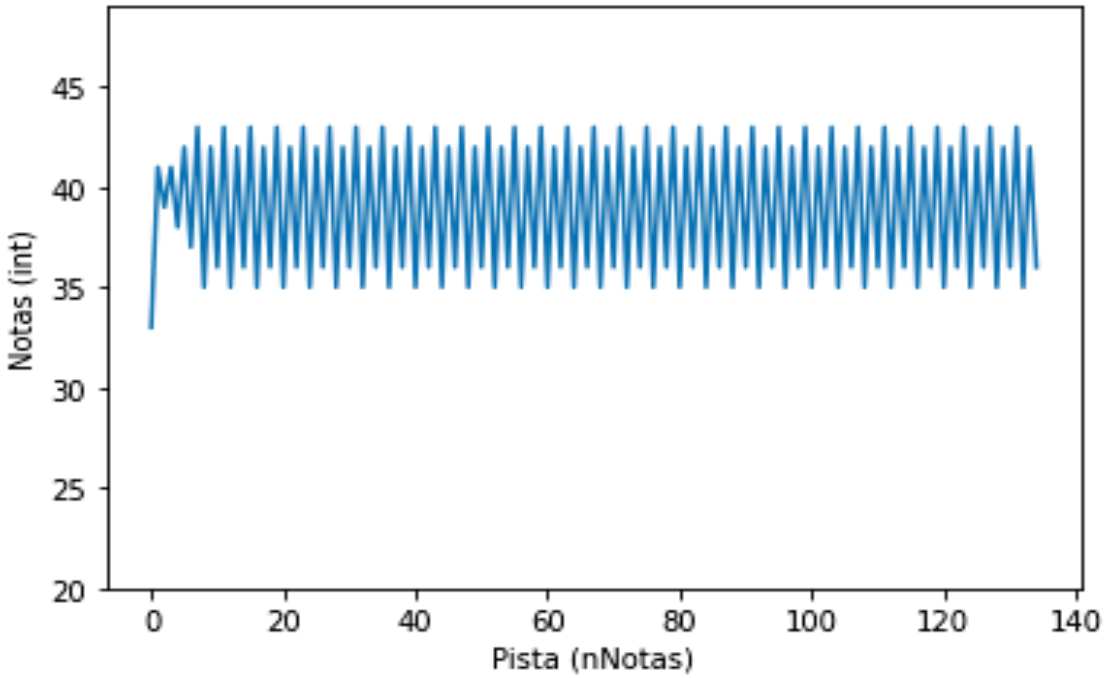


Figura 4.2.3: Notas obtenidas por *logistic_v0_r1*

Los problemas encontrados en la ejecución anterior se repiten también en esta, pero se ha conseguido validar el algoritmo en un 4-ciclo que se puede apreciar en la Figura 4.3.

4.2.2 Logistic_v1

4.2.2.1 Algoritmo v1

Para esta versión se trata de resolver el problema de la desafinación encontrado en la versión anterior, para ello, se comienzan a definir distintos tipos de escalas tales como la escala de Do Mayor (Do, Re, Mi, Fa, Sol, La, Si) o la de Fa Menor (Fa, Sol, La bemol, Si bemol, Do, Re bemol, Mi bemol). Lo que se espera con esto, es que la pieza deje de sonar "desafinada" para que tenga un sentido más musical aplicando una regla de composición simple. Para ello se definen las escalas como:

```
def escalas(x):
  return {
      'doM4': np.array(['C4','D4','E4','F4','G4','A4','B4']),
      'fam4': np.array(['F4','G4','Ab4','Bb4','C4','Db4','Eb4']),
      'doM2': np.array(['C2','D2','E2','F2','G2','A2','B2']),
  }.get(x,"error")
```

La forma de *mapping* en este caso será recoger el valor devuelto por la función logística y normalizarlo al número de notas totales elegidas, cada escala tiene 7 notas y se elige un número de escalas; a este valor se le aplica el módulo y la división entera, resultando en el nombre de la nota y el número de la escala en la que se encuentra respectivamente. Con ello se calcula, mediante la transformación habitual, el tono de la nota final.

En esta versión el problema de la posición de las notas todavía no ha podido ser resuelto, ya que la librería utilizada no ofrece opciones para su resolución.

Un ejemplo de v1 es:

```
seed = 3.82843; escala = escalas('doM2'); x = 0.1;
Lugar = 1000; nEscalas = 3;
Pista = AudioSegment.silent(duration=34000)

nNotas = nEscalas * 7 - 1

for i in range(450):
  x = x * seed * (1 - x)
  if i < 300:
    continue
  nota = round(x * nNotas)
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  nota = nota+(notaEscala)*12
  Tono = int(A3.frame_rate * (2.0 ** (((nota-37)/12))))
  Tono = A3._spawn(A3.raw_data, overrides={'frame_rate': Tono})
  Tono = Tono.set_frame_rate(44100)
  Lugar = Lugar + 200
  Pista = Pista.overlay(Tono, position=Lugar)

Pista.export("pista.wav",format="wav")
```

4.2.2.2 Resultados v1

Ejecución v1_r0

En esta ejecución se va a tomar el valor *seed* = *3.82843*, que es conocido que resulta en un 3-ciclo, se realiza esta ejecución para la validación del algoritmo. Además, debido a que el Sistema Dinámico tarda unas 300 iteraciones en llegar al 3-ciclo, para mostrar los resultados y la validación, se ejecuta una cláusula que se saltará las primeras 300 iteraciones. En la Figura 4.4 se observa que en torno a la iteración 300 + 20 comienza el k-ciclo, al no poder observarse fácilmente los 3 valores que se repiten, en la Figura 4.5 se realiza una ampliación en la que se aprecia 3-ciclo.

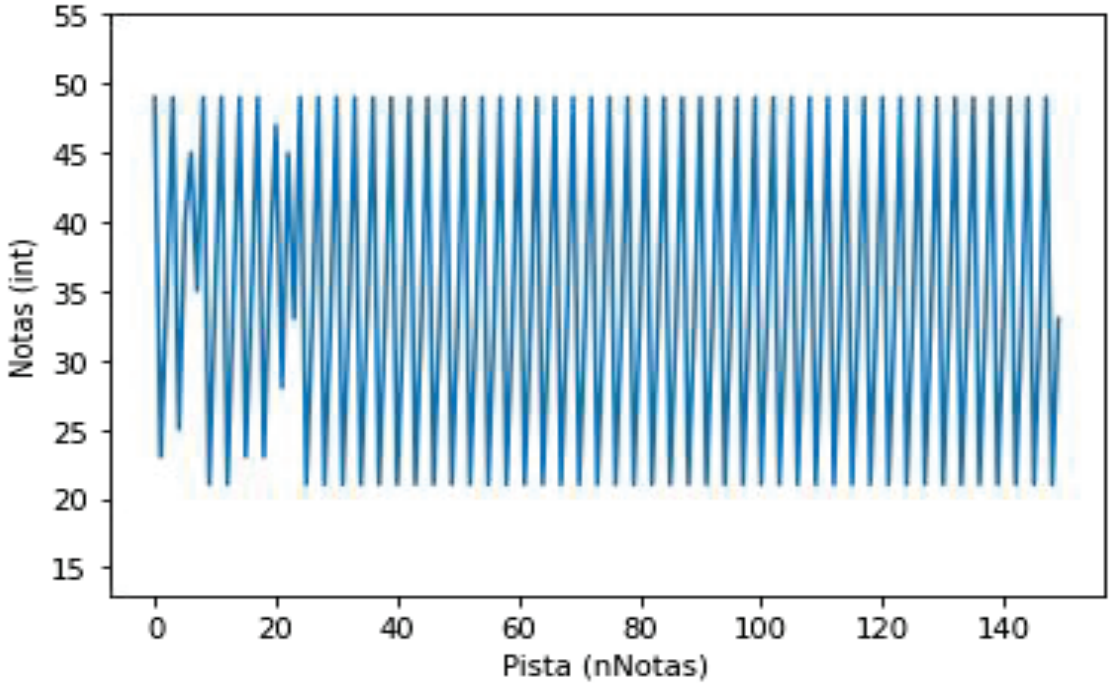


Figura 4.2.4: Notas obtenidas por *logistic_v1_r0*

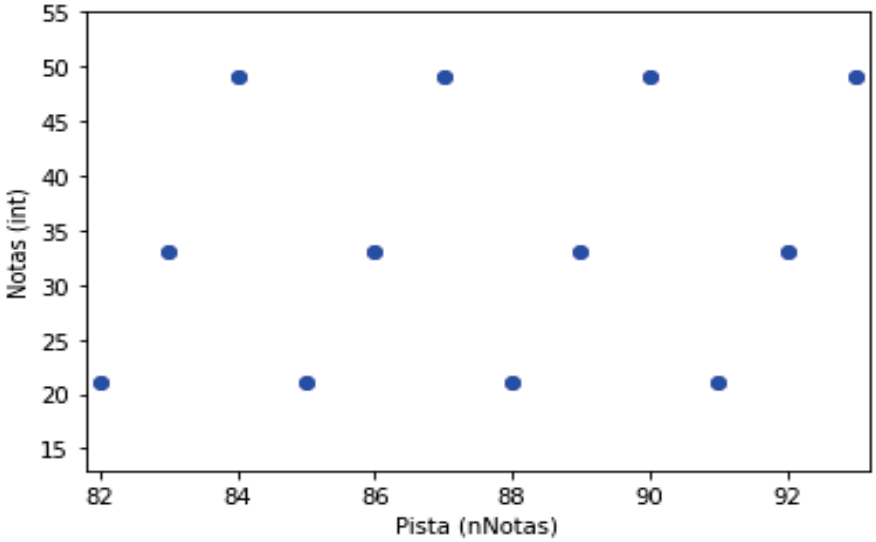


Figura 4.2.5: Ampliación de notas de *logistic_v1_r0*

Ejecución v1_r1

En esta ejecución se toma el valor *seed = 3.901029*. Se nota una notable mejora en cuanto a la tonalidad, las notas no se ven tan incoherentes entre ellas. Aún así sigue dando la sensación de notas aleatorias tocadas sin sentido, debido a que no se ha solucionado el problema del estiramiento y acortamiento de las notas. De hecho, se comienzan a ver patrones auto similares entre, por ejemplo, alrededor de la nota 30, 70 y la 80.

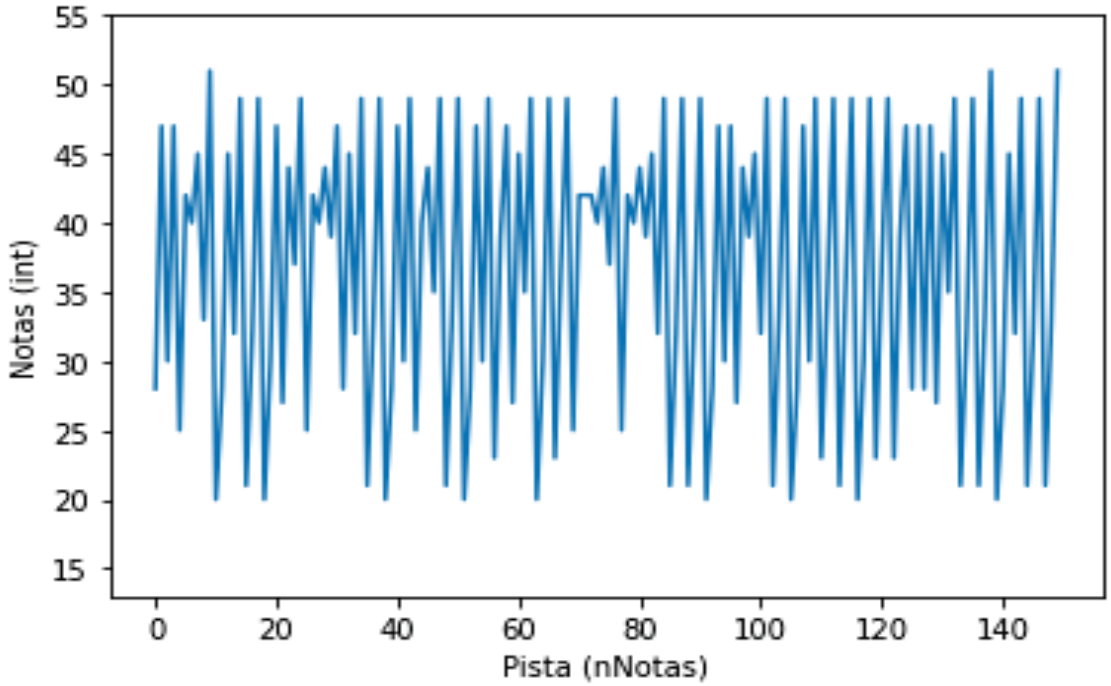


Figura 4.2.6: Notas obtenidas por *logistic_v1_r1*

4.2.3 Logistic_v2

4.2.3.1 Algoritmo v2

En esta versión se ha conseguido, tras la investigación de librerías de tratamiento de audio, la resolución del problema del posicionamiento impreciso de las notas. En la librería *librosa* [11] en python existe una función de cambio de tono que utiliza el mismo cálculo que el que se ha utilizado en versiones anteriores de este Trabajo, pero consigue que la nota comience en el punto exacto en la que se inserta, por lo que, a partir de ahora, se puede aplicar el algoritmo también a la variable del tiempo, es decir, tener *negras, corcheas, blancas,* etc. Para facilidad de uso del algoritmo se ha creado la función auxiliar *pitch_shift* que hace uso de la nueva librería y de la antigua para realizar el cambio de tono deseado:

```
def pitch_shift(sound, n_steps):
    y = np.frombuffer(sound._data, dtype=np.int16).astype(np.float32)/2**15
    y = librosa.effects.pitch_shift(y, sound.frame_rate, n_steps=n_steps)
    a  = AudioSegment(np.array(y * (1<<15), dtype=np.int16).tobytes(), frame_rate = s
ound.frame_rate, sample_width=2, channels = 1)
    return a
```

En el caso de esta versión se va a probar la nueva librería sin modificar la variable del tiempo, validando así el algoritmo.

En cuanto a las modificaciones hechas, se utiliza la misma escala que en *v1*, pero esta vez, en vez de calcularse la nota objetivo, se calcula el número de semitonos en el que difiere de la nota referencia. En el ejemplo mostrado posteriormente se toma de referencia la nota A2 (int 25) y para obtener A3 (int 37) se introducen como parámetros en la función *pitch_shift* el audio de A2 y 37-25=+12 semitonos de diferencia entre ambas.

Un ejemplo del programa resultado es:

```
seed = 3.91; escala = escalas('doM2'); x = 0.1;
Lugar = 100; nEscalas = 3;
Pista = AudioSegment.silent(duration=42000)
A2n = mapping('A2')
nNotas = nEscalas * 7 - 1

for i in range(150):
  x = x * seed * (1 - x)
  nota = round(x * nNotas)
  notaTono = nota%7
  Pfijo2 = notaTono
  notaEscala = nota//7
  Pfijo3 = notaEscala
  nota = mapping(escala[notaTono])
  n = nota + notaEscala*12 - A2n
  Tono = pitch_shift(A2, n)
  Lugar = Lugar + 250
  Pista = Pista.overlay(Tono, position=Lugar)

Pista.export("pista.wav",format="wav")
```

4.2.3.2 Resultados v2

Ejecución v2_r0

Esta ejecución se inicializará de la misma forma que en versiones anteriores para su validación, se tomará por lo tanto *seed = 3.82843* de forma que se muestre un 3-ciclo alrededor de la iteración 320.

El resultado de v2_r0 es el esperado, las notas se han ordenado completamente en el tiempo eliminando la sensación de aleatoriedad de tempo; además de cumplir y validar el algoritmo, encontrando el 3-ciclo igual que en la versión anterior, como se puede ver en la Figura 4.2.7, en la que se ha elegido mostrar

Como añadido, también se aprecia una eficiencia menor en el algoritmo, se han producido 150 notas en una ejecución de alrededor de 2 minutos, en contraparte con las versiones anteriores, que ejecutaban en tan solo unos segundos.

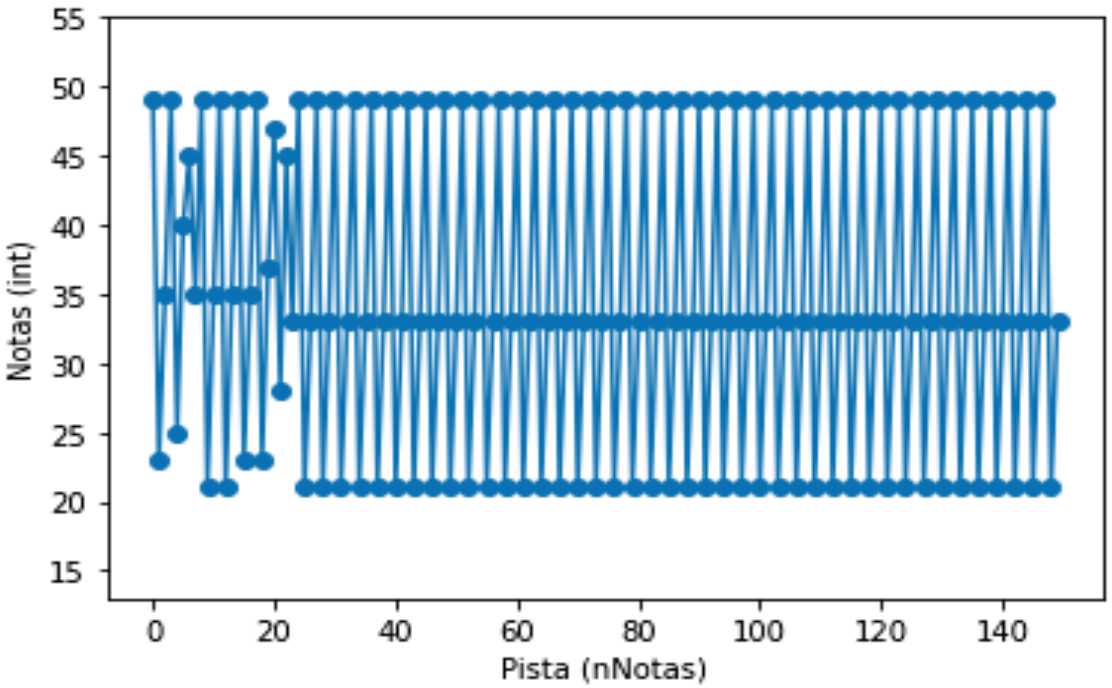


Figura 4.2.7: Notas obtenidas por *logistic_v2_r0*

Ejecución v2_r1

En esta ejecución se va a tomar el mismo *seed* que en la ejecución anterior, pero se van a mostrar los primeros datos de la ejecución, ya que se trata de la composición más interesante hasta el momento, se repiten patrones en ciertos momentos, pero cambian hacia otros patrones de forma que hacen una escucha mucho más agradable que las anteriores. En esta ejecución, además, se observan patrones auto similares y repeticiones en numerosas ocasiones, que se pueden ver en la Figura 4.8.

Pista Logistic_v2_r1:

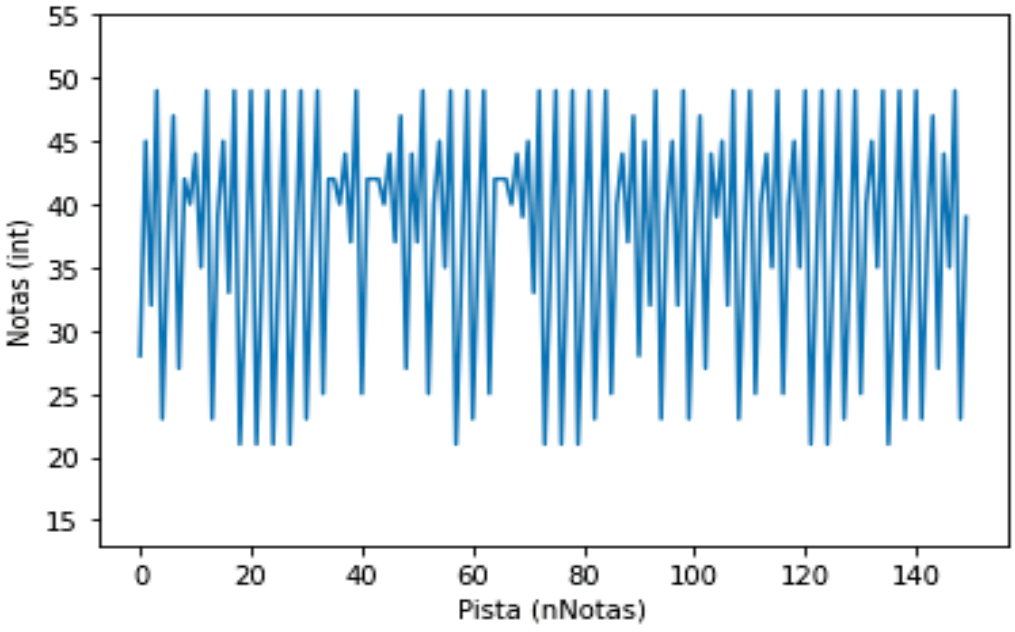


Figura 4.2.8: Notas obtenidas por *logistic_v2_r1*

4.2.4 Logistic v3

4.2.4.1 Algoritmo v3

Para esta versión, se ha tratado de comenzar a realizar cambios en la variable del tiempo, ya que, los datos obtenidos en *v2*, precisamente por la mejora que ha resultado ser la nueva librería utilizada, se pueden hacer monótonos y son inviables para una composición seria, aunque las notas obtenidas sí sean coherentes para ello.

De esta forma en esta versión se plantea utilizar la variable del tiempo de forma que se introduzcan las figuras musicales de *negra, corchea, semicorchea,* etc. Esto se va a conseguir en este caso mediante una función probabilística Gaussiana, de forma que, tomando valores iniciales de la media y la desviación estándar, se generarán los valores que, posteriormente, serán *mapeados* en las figuras musicales mencionadas.

Debido a que las figuras musicales se calculan como potencias de 2 entre ellas, es decir, que si *negra = 1* entonces *corchea=0.5, semicorchea=0.25* y *blanca=2*, se propone la función Gaussiana que si se inicializa con *media=2* y *desv=0.5*, se obtienen valores *R*, en general, en el intervalo [1,3], apareciendo a veces valores algo más extremos. Esos valores son redondeados como *R'* y se le aplica el siguiente *mapping*:

$$figura = 2^{R'-3}$$

Con esto, se obtienen valores del mismo tipo que los anteriormente mencionados (1, 0.5, 2, 0.25, etc.) donde las figuras más probables en aparecer son las *corcheas* (0.5) seguidas de *negras* (1) y *semicorcheas* (0.25), obteniendo, en raros casos *blancas* (2) o *fusas* (0.125).

Obtenidos los ratios de las figuras a introducir, también se tomarán como valor inicial el *tempo* (por ejemplo, *beats per minute = 100*) y proponiendo de referencia la figura de *negra = 1 segundo*, se obtiene el siguiente *mapping* para calcular el *Lugar* donde se insertará la nota:

$$Lugar = 1000 \cdot figura \cdot \left(\frac{60}{bpm}\right)$$

Un ejemplo del programa es:

```
seed = 3.901; x = 0.198; bpm = 100;
mu = 2; sigma = 0.5
Lugar = 0; nEscalas = 2;
Pista = AudioSegment.silent(duration=90000)
escala = escalas('fam3'); A2n = mapping('A2');
nNotas = nEscalas * 7 - 1

for i in range(150):
  x = x * seed * (1 - x)
  figura = round(random.gauss(mu, sigma))
  figura = 2**(figura - 3)
  nota = round(x * nNotas)
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
```

```
    nota = nota + notaEscala*12 - A2n
    Tono = pitch_shift(A2, nota)
    Pista = Pista.overlay(Tono, position=Lugar)
    Lugar = Lugar + 1000*figura*(60/bpm)

Pista.export("pista.wav",format="wav")
```

4.2.4.2 Resultados v3

Ejecución v3_r0

Se toman valores iniciales como los mostrados en el programa propuesto en el punto anterior, de forma que la melodía cambie con respecto a las versiones anteriores del algoritmo. De esta manera, esta vez se ha elegido la escala de Fa Menor a lo largo de dos octavas en vez de tres, esperando que el resultado tenga una tonalidad menor, asociada a composiciones más lúgubres o tristes.

Como se puede observar en la Figura 4.2.9, en cuanto a los tonos obtenidos se pueden apreciar zonas con cierta autosemejanza, como en el intervalo alrededor de (~15000, ~28000), o los picos iguales en el punto ~14000 y ~30000.

En cuanto a la nueva variable introducida, la composición obtenida se percibe algo extraña, aunque tiene zonas en las que las figuras encontradas adquieren coherencia, hay otras zonas en las que simplemente se pierde el *tempo*, haciendo que se obtenga una sensación similar que con las versiones v0 y v1.

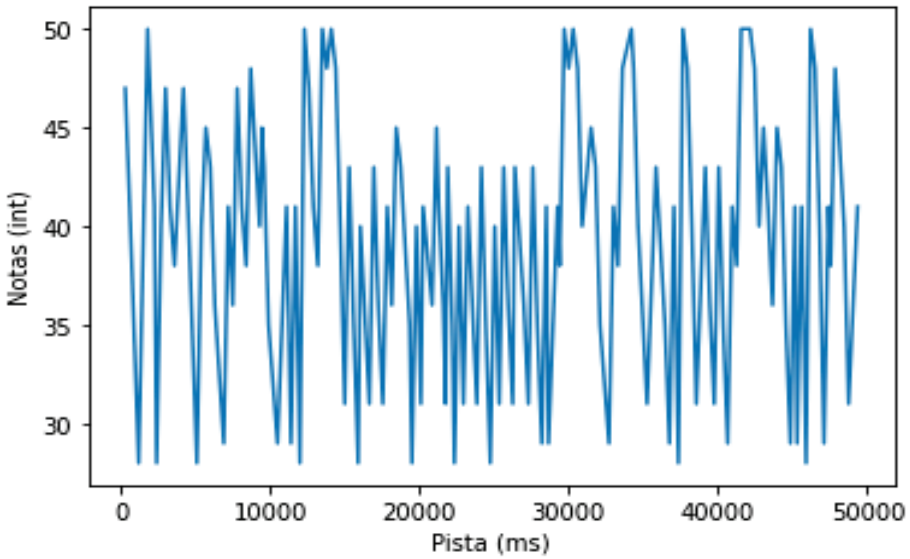


Figura 4.3.9: Notas obtenidas por *logistic_v3_r0*

Ejecución v3_r1

Se inicializan valores *seed = 3.877* y *x*, la escala y número de octavas se eligen las mismas que la ejecución anterior; en cuanto a los valores aleatorios para las *figuras* se va a tomar una función Gaussiana cuya media es 1, es decir, valores más bajos y por lo tanto *figuras* más centradas en las *semicorcheas* (más rápidas); y *desv = 0.75* permitiendo más variedad de figuras en torno a la media.

El resultado agrava más el problema de la coherencia en el tiempo, las notas no tienen por qué ser incoherentes en cuanto a sus tonos, pero la variedad de figuras en cada momento hace una escucha casi incómoda. Esto es debido

precisamente a la *desv* elegida, por lo que se observa que los valores mayores a 0.5 resultan en demasiada aleatoriedad en las figuras.

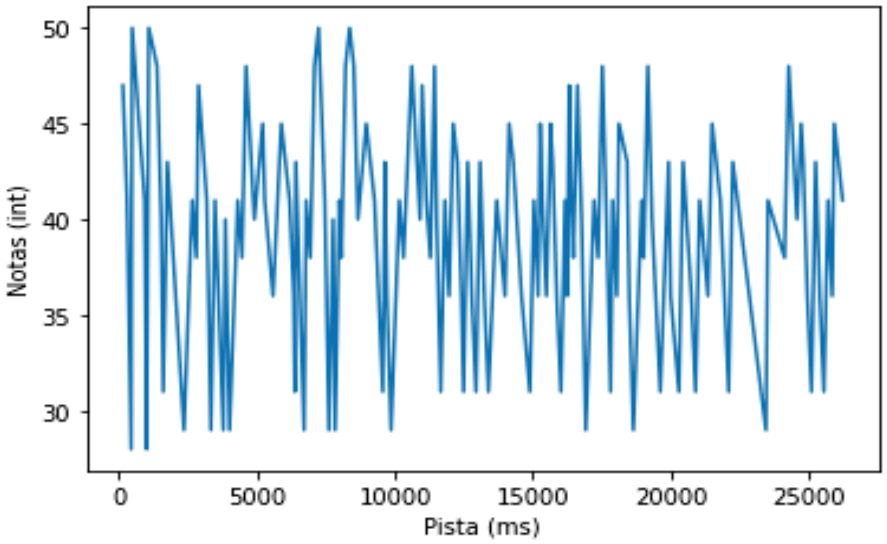


Figura 4.3.10: Notas obtenidas por *logistic_v3_r1*

Ejecución v3_r2

Para comprobar la premisa propuesta en la ejecución anterior, se realiza una adicional, con *seed = 3.929* y *desv = 0.35*.

Efectivamente se consigue una composición con una gran coherencia musical en ambos ámbitos, sustancialmente mayor que la obtenida en la ejecución anterior, obteniendo zonas autosemejantes como ~26000 y ~42000 en la Figura 4.3.11 y siempre manteniendo un *tempo* casi constante dando lugar también a patrones en cuanto a *figuras*.

Por lo tanto, los mejores resultados en cuanto a *figuras se refieren* son los obtenidos por el intervalo [0.3, 0.5] para la variable *desv*, con *media = 2* y 100 *bpm*.

Pista Logistic_v3_r2:

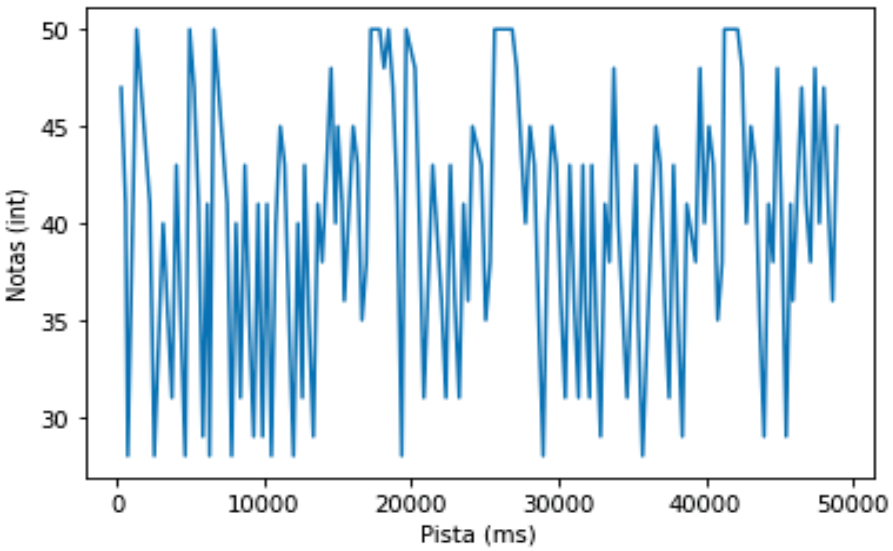


Figura 4.3.11: Notas obtenidas por *logistic_v3_r2*

4.3 Implementación de Algoritmos existentes

En este punto se realizará un repaso de algunos de los algoritmos existentes para la composición de música fractal, la mayoría de ellos recogidos y estudiados en [9], se ha tratado de simular los mismos resultados que los producidos por estos trabajos previos, con ciertos puntos en añadido en alguno de los casos.

La evolución de estos algoritmos ha sido pareja con la mostrada anteriormente, por lo tanto, no se mostrarán los resultados de cada versión de cada algoritmo, sino la versión o versiones finales para la recopilación de resultados y validación. Además, se realizará un breve análisis de la capacidad de hibridación de estos algoritmos y sus posibilidades.

4.3.1 Morse-Thue Sequence

4.3.1.1 Algoritmo morsethue_vf

Este algoritmo se basa en el utilizado por [9] para la generación de audio, necesitando previamente inicializar las variables *base*, *seed* (elemento inicial), un *multiplicador (mult)*, la *escala* y *numero de escalas*.

El algoritmo comienza por cambiar el valor de *seed* a la base introducida, se suman los dígitos del resultado, luego se realiza el *mapping* utilizando el módulo del resultado con el número de notas incluidas en todas las escalas escogidas para asociar el número a la nota final que representará mediante la función *pitch_shift*; finalmente se recalcula el nuevo *seed* multiplicando el antiguo por *mult*.

En anteriores versiones el *mapping* se trataba igual que en las primeras versiones de *logistic* comentadas en el punto 4.2, sin utilizar escalas musicales, sino un intervalo de notas; o la transformación de la nota referencia mediante *pydub*. Para la validación de este algoritmo se va a comparar con los resultados que se obtienen mediante el programa FractMus, por lo que se tomarán notas en secuencia sin realizar *mapping* a la variable del tiempo.

Un ejemplo del algoritmo es:

```
base = 3; seed = 2; mult = 17;
escala = escalas('doM2')
Lugar = 100; nEscalas = 3;
Pista = AudioSegment.silent(duration=34000)
A2n = mapping('A2')
nNotas = nEscalas * 7 - 1

for i in range(135):
  nota = getSum(cDBase(seed,base))
  nota = nota%nNotas
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  nota = nota + notaEscala*12 - A2n
  Tono = pitch_shift(A2, nota)
  Lugar = Lugar + 250
  Pista = Pista.overlay(Tono, position=Lugar)
```

```
  seed = seed * mult;

Pista.export("pista.wav",format="wav")
```

4.3.1.2 Resultados morsethue_vf

Ejecución vf_r0

Para esta ejecución se va a ejecutar el mismo código mostrado en el punto anterior, con una *base = 3, seed = 2* y *mult = 17*. El resultado obtenido es el esperado, en comparación con los datos obtenidos por FractMus, se observa que el programa desarrollado en este trabajo tiene la funcionalidad adicional de introducir el valor inicial de la nota, mientras que el mencionado comienza siempre por el valor 1. A diferencia de *logistic*, en este caso no se aprecian patrones autosimilares a simple vista en la Figura 4.3.1, aunque sí se aprecia coherencia musical en la composición.

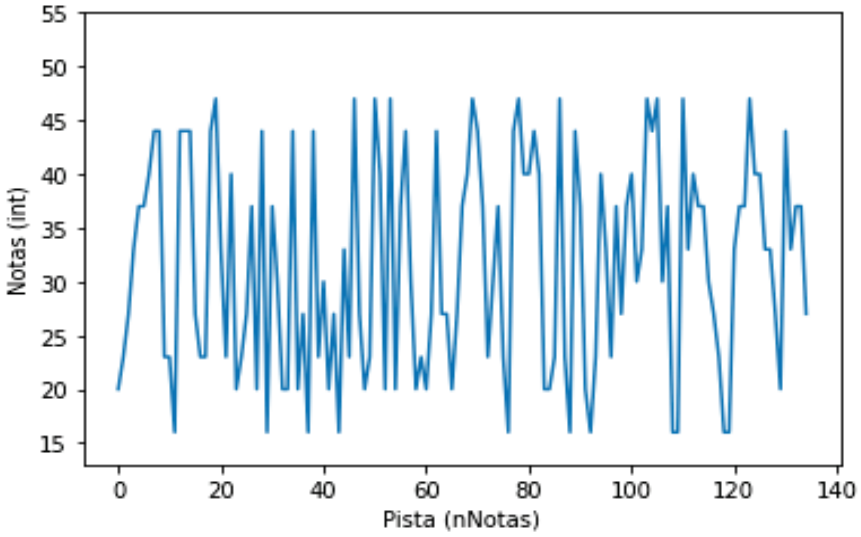


Figura 4.3.1: Notas obtenidas por *morsethue_vf_r0*

Ejecución vf_r1

Para esta ejecución se va a probar con valores *base = 11, seed = 2* y *mult = 23*, escogiendo base y multiplicador primos para evitar ciclos cortos y la mayor complejidad posible en el resultado. Los valores obtenidos, aunque difiriendo de los obtenidos en la ejecución anterior, se notan parecidos en la escucha. En este caso tampoco se observan patrones autosimilares en la Figura 4.3.2.

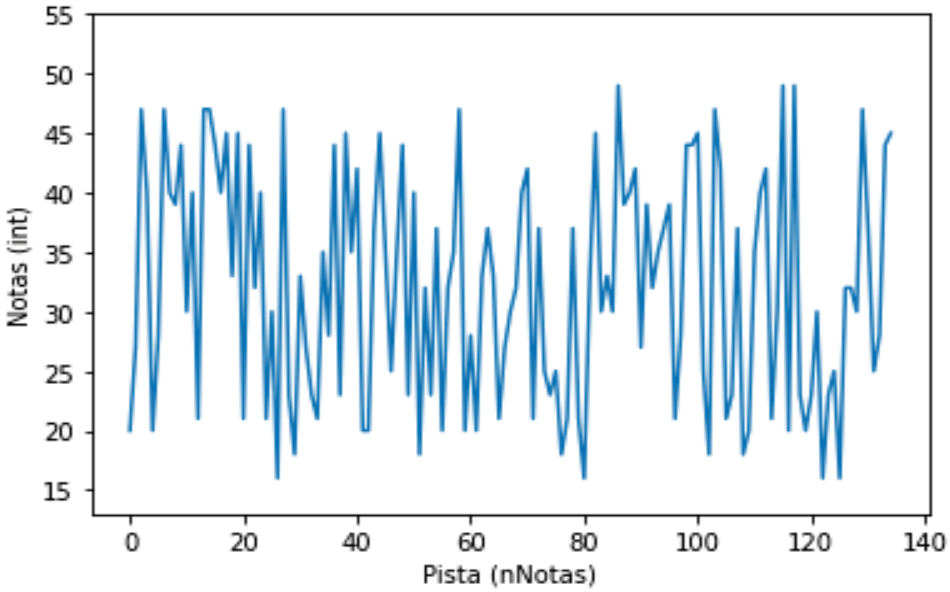


Figura 4.3.2: Notas obtenidas por *morsethue_vf_r1*

4.3.2 Earthworm Sequence

4.3.2.1 Algoritmo earthworm_vf

El caso de la implementación de este algoritmo es parecido al anterior, se trata de conseguir validar los resultados comparándolos con los resultados que obtiene FractMus, utilizando también el mismo método de *mapping*.

Para la ejecución de este algoritmo se parte de la entrada de tres variables: *worm*, *seed* y *mult*, números enteros. El algoritmo comienza por comprobar si *seed* excede el valor de *worm*, si lo hace, se calcula *seed* módulo *worm*, manteniendo siempre valores en el intervalo [0, *worm*]; posteriormente se realiza la transformación de la nota a partir del número entero resultante y se introduce en la pista. Finalmente se recalcula *seed* multiplicándolo con *mult*, de igual manera que para el algoritmo anterior.

Para este algoritmo, el tiempo de cada figura vuelve a ser constante, es decir, se trata de una nota cada 250ms como se muestra en el ejemplo de programa más adelante, tratándose de un *mapping* exclusivo hacia la tonalidad.

Un ejemplo del algoritmo es:

```
Worm = 100; seed = 6; mult = 3;
escala = escalas('doM2');A2n = mapping('A2');
Lugar = 100; nEscalas = 3; nNotas = nEscalas * 7 - 1;
Pista = AudioSegment.silent(duration=34000)
```

```
for i in range(135):
  if seed >= Worm :
    seed = seed%Worm
  nota = seed
  nota = nota%nNotas
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  nota = nota + notaEscala*12 - A2n
  Tono = pitch_shift(A2, nota)
  Lugar = Lugar + 250
  Pista = Pista.overlay(Tono, position=Lugar)
  seed = seed * mult;

Pista.export("pista.wav",format="wav")
```

4.3.2.2 Resultados morsethue_vf

Ejecución vf_r0

Se tomarán los valores *worm = 100, seed = 6* y *mult = 3*, de manera que facilite la validación del algoritmo en comparación con el producido por FractMus. De hecho, de manera general se formarán ciclos debido a la poca complejidad de las operaciones realizadas.

Efectivamente el algoritmo se valida ya que, observando la pista y la Figura 4.3.3, se trata de una repetición de 3 notas constante. Se trata de un ciclo simple debido a los valores elegidos, para la próxima ejecución se utilizarán valores coprimos para obtener ciclos más largos.

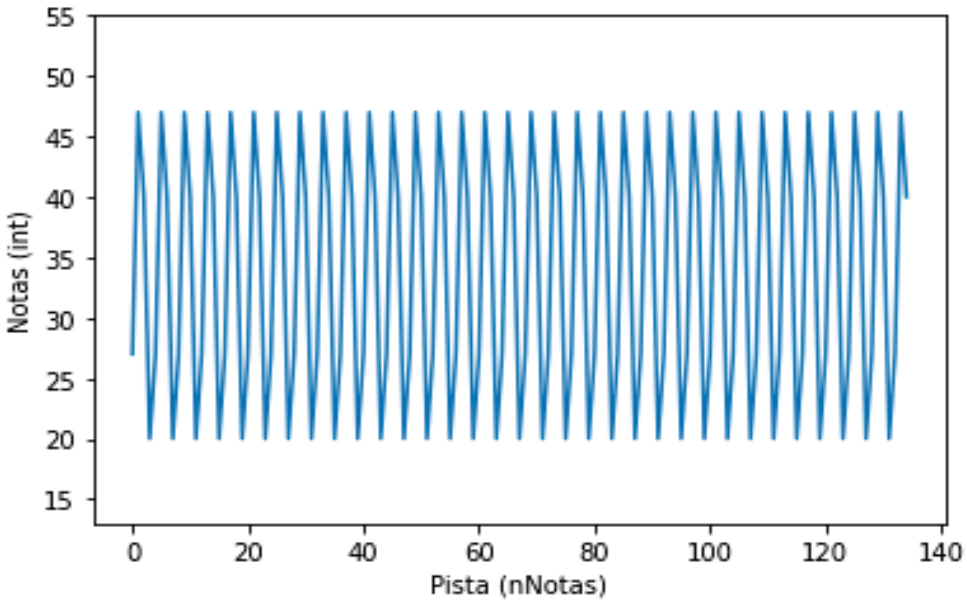


Figura 4.3.3: Notas obtenidas por *earthworm_vf_r0*

Ejecución vf_r1

Para esta ejecución se trata de que se emita el mayor número de notas antes de que comiencen ciclos. Para ello, se eligen *worm = 111, seed = 6* y *mult = 7*. Igualmente se mantiene la escala de Do Mayor de base, a lo largo de 3 octavas.

El resultado, otra vez, se trata de un bucle de notas repetidas, aunque en este caso se ha conseguido un ciclo más grande, de 8 notas, como aparece en la Figura 4.3.4.

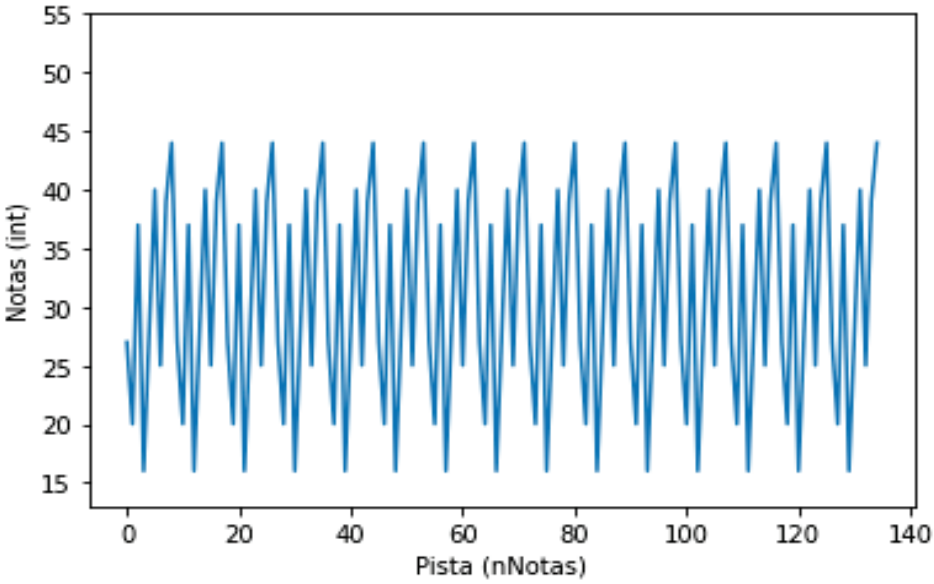


Figura 4.3.4: Notas obtenidas por *earthworm_vf_r1*

Como evaluación general de este algoritmo, las composiciones generadas, aunque coherentes con los tonos de las notas, dejan mucho que desear hacia una composición compleja y seria; aunque podría ser de ayuda en la hibridación de algoritmos, introduciendo acompañamientos, acordes e incluso otros instrumentos.

4.3.3 3n+1

4.3.3.1 Algoritmo *3n1_vf*

Otro ejemplo de algoritmo "simple" muy útil para comprobar la validez de Sistemas Dinámicos para la composición musical, es el basado en la Conjetura de Collatz, también conocido por 3n+1. En este caso sólo vamos a hacer uso de una variable inicial *seed*, la escala y el número de octavas a utilizar, ya que el Sistema Dinámico no requiere de más campos.

Se comienza por traducir *seed* a la nota musical correspondiente en el rango de notas elegidas, se inserta y finalmente se recalcula *seed* de la siguiente manera: si $seed_n$ es múltiplo de 2 entonces $seed_{n+1}$ se calcula dividiendo *seed* entre 2; en cambio, si $seed_n$ es impar, entonces $seed_{n+1}$ se calcula multiplicando $seed_n$ por 3 y sumándole 1.

Un ejemplo de programa es:

```
seed = 27;
escala = escalas('doM2');A2n = mapping('A2');
Lugar = 100; nEscalas = 3; nNotas = nEscalas * 7 - 1;
Pista = AudioSegment.silent(duration=34000)
```

```
for i in range(135):
  nota = seed
  nota = nota%nNotas
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  nota = nota + notaEscala*12 - A2n
  Tono = pitch_shift(A2, nota)
  Lugar = Lugar + 250
  Pista = Pista.overlay(Tono, position=Lugar)
  if (seed % 2) == 0:
    seed = seed//2;
  else:
    seed = 3*seed + 1;

Pista.export("pista.wav",format="wav")
```

4.3.3.2 Resultados *3n1_vf*

Ejecución vf_r0

Debido a que se cree que este Sistema Dinámico siempre acaba en un ciclo de 3 números (4,2,1,4,2,1,...) se va a intentar encontrar un valor que produzca el máximo número de notas diferentes antes de caer al ciclo. Por lo tanto, para esta ejecución se va a inicializar *seed = 27*.

El resultado es una composición con cierto sentido musical, en el que existen pequeños ciclos a lo largo de la secuencia (Figura 4.3.5), por lo que se producen patrones para luego salir de ellos y reproducir otros. Se trata de una serie más interesante de lo esperado, debido a la simplicidad del algoritmo y la complejidad del resultado. Al final de la pista se puede observar que se llega al bucle final donde los valores de *seed* se mantienen en 4,2 y 1.

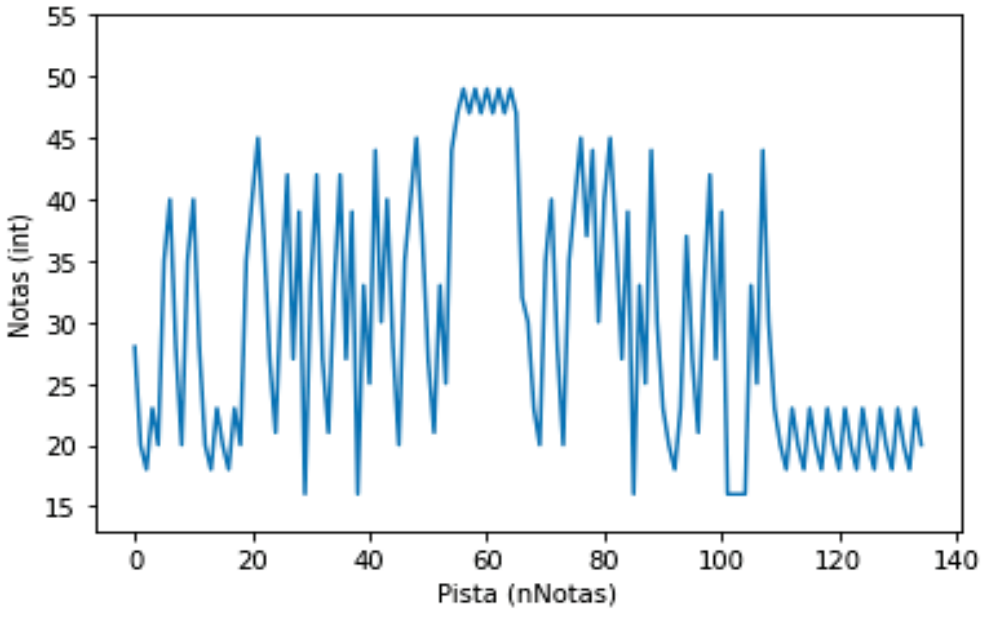


Figura 4.3.5: Notas obtenidas por *3n1_vf_r0*

Ejecución vf_r1

En esta ejecución se trata de conseguir más iteraciones únicas que para la anterior, por lo que se va a tomar *seed = 871* sabiendo que llega al final de sus iteraciones tras 178 pasos, más de las 135 notas que se introducen en la pista.

El resultado de esta ejecución tiene también una coherencia musical comprobable, pero, además, se ha obtenido al final de las iteraciones el mismo resultado de notas que en la ejecución r0, llegando a los mismos números desde otros distintos, pero debido a que las notas en r1 comienzan a ser las mismas que en r0 en torno a la iteración 120, se ha vuelto a ejecutar el algoritmo esta vez introduciendo más notas para apreciar también el parecido en todas las iteraciones finales.

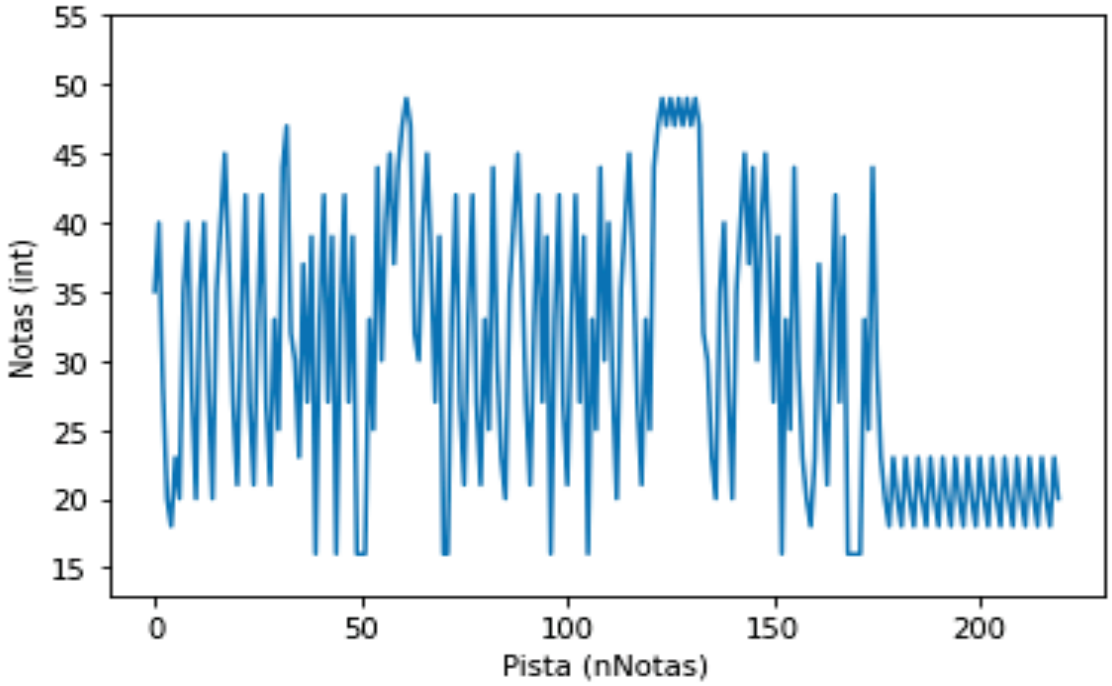


Figura 4.3.6: Notas obtenidas por *3n1_vf_r1*

4.3.4 Triángulo de Sierpinsky – Juego del Caos

4.3.4.1 Algoritmo *sierpinsky_vf*

A diferencia de los casos anteriores, el desarrollo de este algoritmo ha sido ideado de forma original. El objetivo de este algoritmo es dibujar el Triángulo de Sierpinsky con notas musicales mediante el algoritmo del Juego del Caos.

El Juego del Caos es un algoritmo determinista que permite dibujar una aproximación del Fractal del Triángulo de Sierpinsky. Definidos los tres vértices del Triángulo (A, B y C), la forma del algoritmo es:

1. Tomar un punto *X* inicial
2. Elegir aleatoriamente un vértice del Triángulo *V* (A, B o C)
3. Calcular el punto medio *P* entre *X* y *V*
4. Dibujar *P*
5. X = P
6. Volver a 2

Para el desarrollo del programa se utilizará un *mapping* que solo se encuentra en este algoritmo con respecto del resto de los tratados en este Trabajo. Este, consiste en obtener puntos de dos dimensiones del Triángulo de Sierpinsky y transformar la altura, o coordenada *y* de dichos puntos en la tonalidad de la nota a insertar, mientras que la coordenada *x* servirá para calcular en qué punto de la pista se inserta la nota. Tal y como se hace en el Juego del Caos, para mayor precisión del triángulo, las primeras iteraciones no se introducen en la pista, sino que se omiten., así se obtienen valores más aproximados al fractal.

Un ejemplo de programa es:

```
escala = escalas('doM2');A2n = mapping('A2');
nEscalas = 3; nNotas = nEscalas * 7 - 1;
Pista = AudioSegment.silent(duration=35000)

for i in range(2200):
  Prand = random.randint(1,3)
  Prand = swi(Prand)
  Prand = np.array([Prand])
  Dist = (Prand - Pfijo)/2
  Pfijo = Pfijo + Dist
  if i < 900:
    continue
  nota = nNotas*Pfijo[0,1]/2
  nota = round(nota)
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  nota = nota + notaEscala*12 - A2n
  Tono = pitch_shift(A2, nota)
  Lugar = round((Pfijo[0,0]/4)*30000)
  Pista = Pista.overlay(Tono, position=Lugar)

Pista.export("pista.wav",format="wav")
```

4.3.4.2 Resultados *sierpinsky_vf*

Ejecución vf_r0

Para esta ejecución se ha ejecutado el programa mostrado en el punto anterior. Se ha escogido un Triángulo de 35 segundos de "largo" y tres octavas, comenzando en la escala de Do Mayor, de "alto". Además, para mostrar de mejor forma de manera visual, se insertan 300 notas en la pista.

La composición resultante es un tanto negativa, ya que no se ha conseguido ninguna coherencia musical más allá de un "muro de sonido" en aras de mostrar bien la forma del Triángulo, como se muestra en la Figura 4.3.7. La razón de ser de esa incoherencia musical es que para dibujar el Triángulo de Sierpinsky mediante el Juego del Caos, se toman variables del mismo tipo y, estas, al ser separadas en tiempo y tono, pierden la esencia fractal del Triángulo, aunque estén correlacionadas en este caso.

Cabe destacar también el tiempo de ejecución en este caso, ya que ha llegado a sobrepasar los 15 minutos de ejecución, perdiendo eficiencia en cuanto a tiempo

con respecto de versiones anteriores con la antigua librería utilizada o, incluso, con respecto a trabajos anteriores que producen pistas de forma mucho más rápida.

Pista del Triángulo de Sierpinsky:

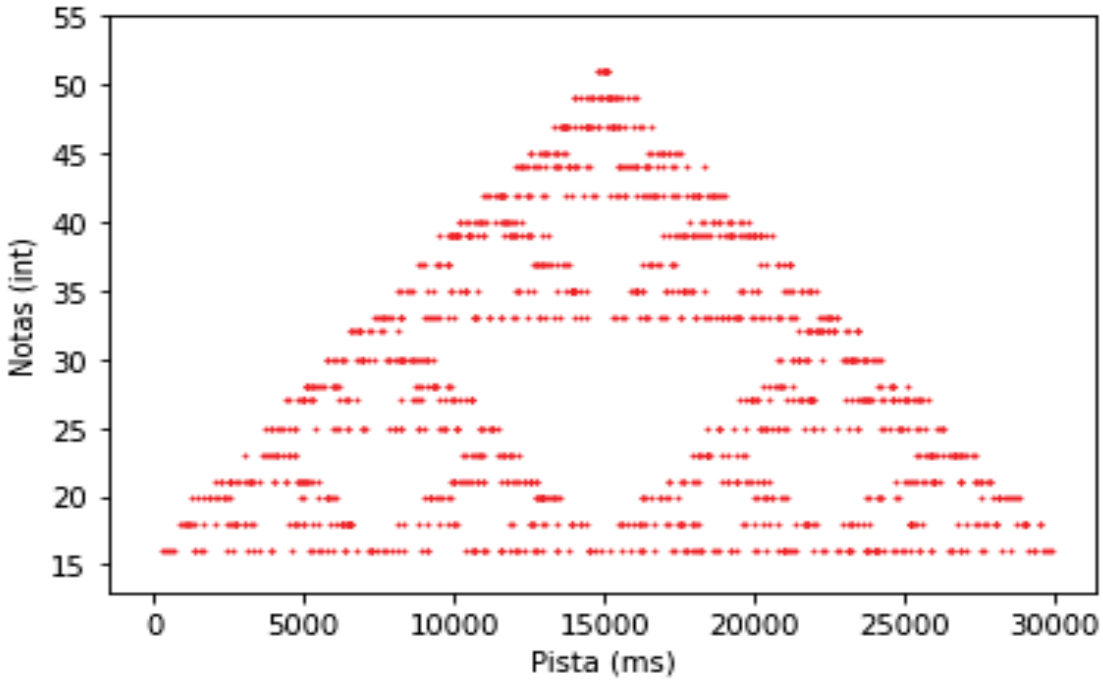


Figura 4.3.7: Notas obtenidas por *sierpinsky_vf_r0*

4.3.5 Ruido 1/f

4.3.5.1 Algoritmo *1fnoise_vf*

Se trata de un algoritmo muy parecido en forma a *logistic_v2*, ya que la única diferencia es el Sistema Dinámico que lo define. De hecho se escogerán los mismos valores iniciales *seed*, x_0 , escala y número de octavas. El Sistema Dinámico elegido se basa en la función:

$$x_{n+1} = x_n \cdot seed + r \cdot (1 - seed^2)^{1/2}$$

Esta, simula el llamado *ruido 1/f* o *ruido rosa*, de forma que x comienza con un valor inicial, igual que *seed*, pero r es un valor aleatorio en el intervalo (0,1), dando lugar a diferentes resultados cada ejecución.

También se ha decidido que el tiempo cada cual se insertan las notas sea constante para asegurar una calidad musical también constante. Habiendo recibido así los mejores resultados.

Un ejemplo del programa es:

```
seed = 0.85; escala = escalas('doM2'); x = 0.1;
Lugar = 100; nEscalas = 3;
Pista = AudioSegment.silent(duration=42000)
Puntos = np.array([]);
A2n = mapping('A2')
```

```
nNotas = nEscalas * 7 - 1

for i in range(135):
  r = random.random()
  x = x * seed + r * (1 - seed ** 2) ** (1/2)
  nota = round(x * nNotas)
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  Pfijo = nota+(notaEscala)*12
  n = nota + notaEscala*12 - A2n
  Tono = pitch_shift(A2, n)
  Lugar = Lugar + 250
  Pista = Pista.overlay(Tono, position=Lugar)
  Puntos = np.concatenate((Puntos,[Pfijo]),axis=0)

Pista.export("pista.wav",format="wav")
```

4.3.5.2 Resultados *1fnoise_vf*

Ejecución vf_r0

Se comienza probando con *seed = 0.85* y *x = 0.1* realizando una pequeña ejecución de prueba.

El resultado es muy parecido al obtenido por *logistic*, pero esta vez no se aprecian patrones autosimilares, aunque sí se ha conseguido coherencia musical. Debido a que se trata de un algoritmo con componente aleatoria, no se puede concluir que el resultado es el deseado.

Pista de 1fnoise:

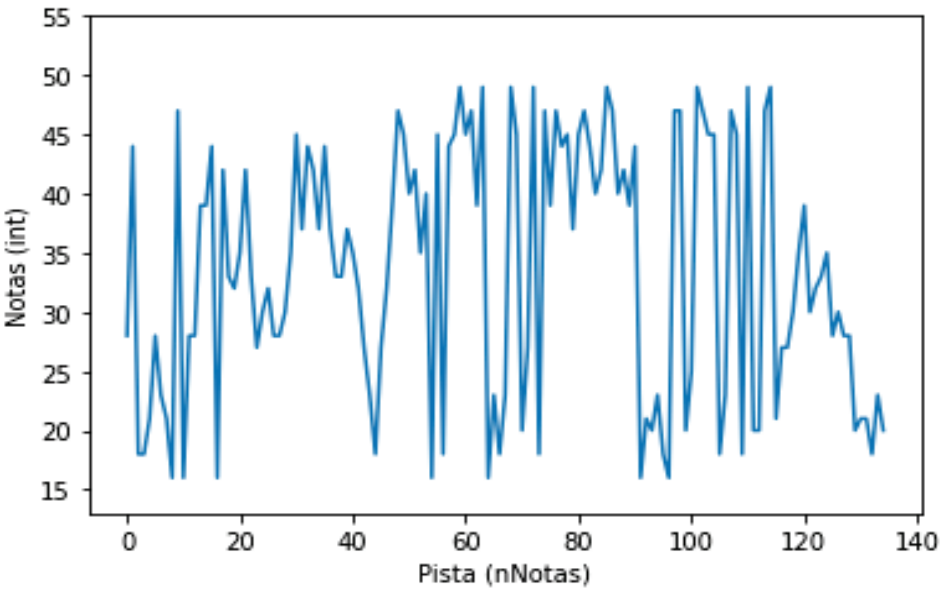


Figura 4.3.8: Notas obtenidas por *1fnoise_vf_r0*

Ejecución vf_r1

En esta ejecución se toma *seed = 0.99* para comprobar y validar los resultados comparándolos con FractMus, ya que se comenta en la documentación, que valores cercanos a los extremos de *seed* (0 y 1), producen composiciones mucho menos aleatorias, consiguiendo composiciones $1/f^2$ en vez de $1/f$, cuya diferencia viene comentada en el punto 3.2.

El resultado, de hecho, es una subida pseudo-constante de notas, que se retoma desde abajo cuando se alcanza el techo de las octavas elegidas.

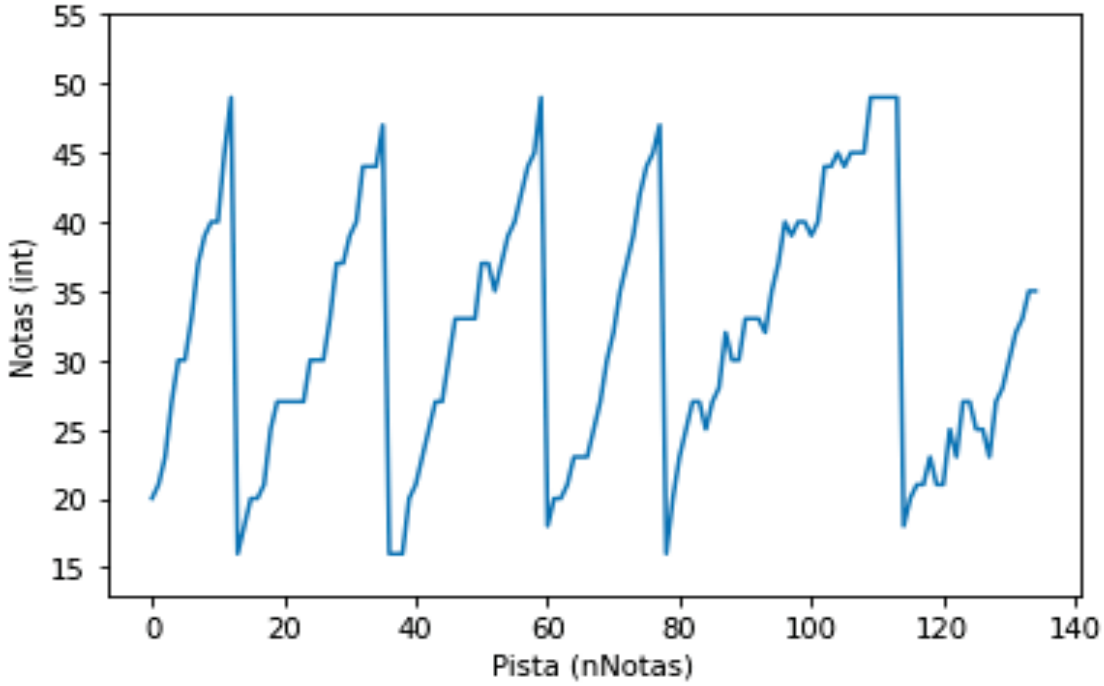


Figura 4.3.9: Notas obtenidas por *1fnoise_vf_r1*

4.4 Hibridación de Algoritmos

4.4.1 Algoritmo Híbrido de varios instrumentos

En este punto se tratará de mezclar los algoritmos comentados en los puntos 4.2 y 4.3 realizando modificaciones y adaptaciones en muchos de los casos para conseguir una composición musical que consiga hibridar dichos algoritmos y añadir varios instrumentos en la misma pieza. Por lo tanto, el método es ir adaptando algunos de los algoritmos vistos con anterioridad a los instrumentos que se vayan añadiendo al algoritmo conjunto.

Para ello, se comienza decidiendo una base para la composición, es decir, la estructura de esta será de un instrumento "solista" con una serie de acompañamientos acordes. Para el instrumento solista se mantiene el piano utilizado en los algoritmos desarrollados, y para los acompañamientos se utilizarán un piano producido por un algoritmo nuevo que inserta en acordes en vez de notas sueltas, y un violonchelo con el algoritmo *logistic* adaptado para él.

Primero, para crear una base musical se comienza por la introducción del violonchelo a la pieza. Para este instrumento, se ha escogido aplicar el algoritmo *logistic* debido a la gran calidad musical que ha mostrado. Además, se van a escoger valores que creen k-ciclos dado que se trata de un buen instrumento de acompañamiento marcando siempre las notas de la escala resultante. Además del tono, se modifica el algoritmo para que genere menos notas y más largas; esto es, cambiando las entradas de la función Gaussiana haciendo que genere figuras *blancas, negras* y *redondas* principalmente. Ya que se trata de un instrumento de acompañamiento, se escoge una escala baja con una sola octava.

Posteriormente, se pretende añadir un piano de acompañamiento para dar complejidad a los bajos del violonchelo y, igual que ocurre con este, se le van a dar valores de una escala baja en una sola octava aplicando también el mismo algoritmo *logistic*. Por lo tanto, los acordes de piano y el violonchelo se encontrarán en la misma zona de tonos, con diferentes timbres, dando consistencia a la pista. La diferencia principal con los acordes y el piano solista es que un acorde clásico incluye tres notas tocadas al mismo tiempo, por lo que las tres notas deben ser calculadas de alguna forma. Ya que las escalas de todos los instrumentos son las de Do Mayor (aunque estén en diferentes octavas), se elige utilizar Acordes Mayores cuya base es la nota recibida por el algoritmo en cada momento, esto es, se calcula la nota y, a partir de esta, se añaden las notas con +4 y +7 semitonos por encima de la calculada (esto es una regla de composición musical para calcular acordes). Obtenidas las tres notas del acorde se introducen todas en el mismo lugar de la pista resultando en el Acorde Mayor de la nota producida por el algoritmo.

Una vez vista la manera de introducir los acordes, la mejor forma de insertar estos en la pista es parecida a la del violonchelo, utilizando *figuras* largas como *blancas* o *negras* en k-ciclos, mediante la función Gaussiana con una desviación pequeña y la media en las *blancas*. En este caso, además, asignamos un 6-ciclo a los acordes dando lugar a variantes distintas entre estos y el violonchelo, ya que usando otro 4-ciclo los valores se iban a solapar prácticamente, dando lugar a un acompañamiento monótono. La producción de acordes es una mejora original en comparación con trabajos previos, ya que estos tomaban notas individuales en todos los casos.

Finalmente se configura el piano solista, al que se le asigna el algoritmo *1/f noise* ya que ha sido uno de los algoritmos con mejor resultado además de *logistic*. Para asignar las variables en este, se propone un valor de *seed* parecido al elegido en la ejecución r0 vista en el punto 4.3.5.2, ofreciendo un fractal estadístico en el tono de las notas. También se fijará la función Gaussiana para que produzca *corcheas* y *semicorcheas* con una media centrada en estas y desviación típica algo superior a la del violonchelo y los acordes, dando algo más de variedad en la pieza.

Pista de HybridMix:

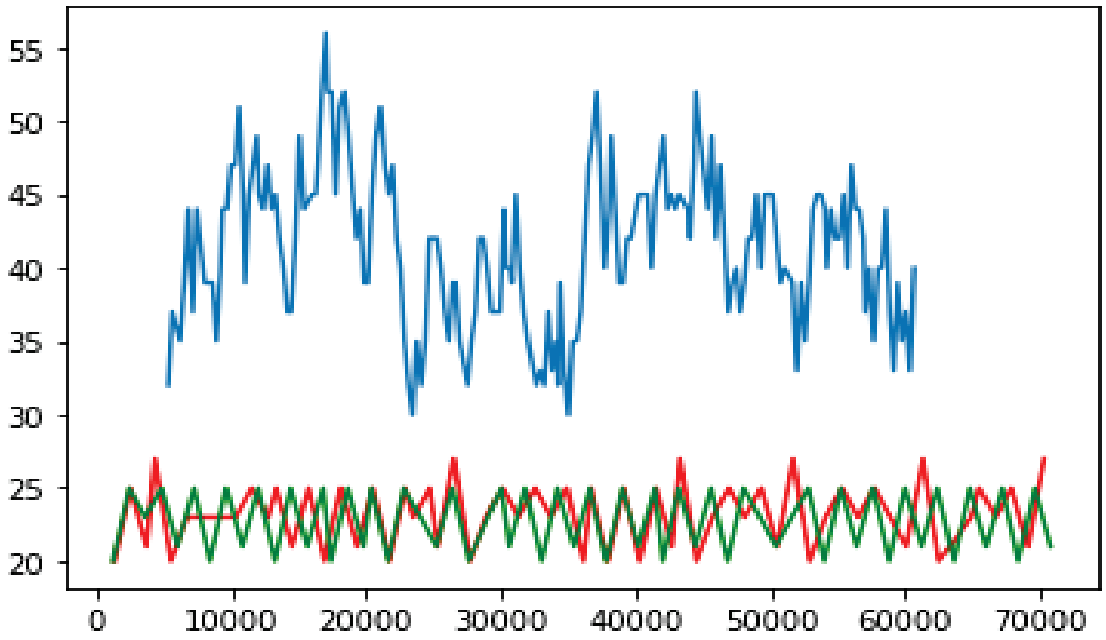


Figura 4.4.1: Verde: Violonchelo. Rojo: Acordes de Piano. Azul: Piano Solista

Como se observa en la Figura 4.4.1 se han obtenido los resultados esperados, a excepción de que algunas de las notas que se han producido para el violonchelo y los acordes coinciden en tono e, incluso, llegan a tocar la misma nota al mismo tiempo, agregando armonía y consistencia a la pieza.

Se han tomado también pequeñas decisiones adicionales de diseño para la estética de esta pista, tales como el comienzo en solitario del violonchelo, luego entrando los acordes y, finalmente el piano solista, que también es el primero que acaba, dejando a los acordes las últimas notas.

Para terminar es importante comentar que este tipo de hibridaciones se pueden realizar con cualquier instrumento y cualquier algoritmo a elección.

Dado que el código de este programa es muy extenso, se incluye solamente en el anexo.

4.4.2 Algoritmo fractal híbrido

Este algoritmo está basado en la mezcla entre los algoritmos deterministas de representación de fractales y el algoritmo *logistic* previamente utilizado.

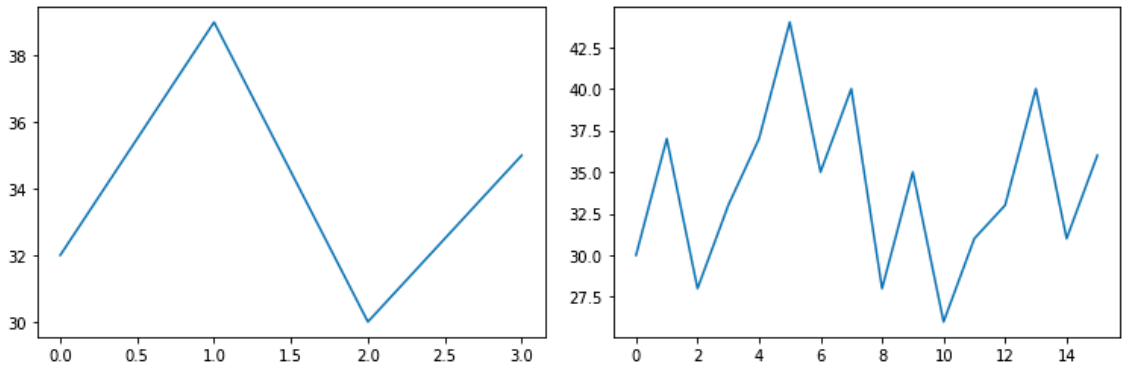
Si teniendo, entonces, un Sistema de Funciones Iteradas contractivo en el que en cada paso se obtienen más puntos y más aproximados a los pertenecientes al fractal, se observa que en la música esto no puede ocurrir, ya que el espacio entre notas es limitado, esto es, si se busca cierta calidad musical. Por lo tanto, se propone partir desde el punto "infinitesimal" del fractal, haciéndolo cada vez más grande. Este "punto" puede ser una nota o varias y, además, estas se pueden escoger de forma aleatoria o, como en este caso, las obtenemos de un algoritmo externo que tiene también una fuerte relación con la Geometría Fractal, *logistic* o *1fnoise*, por ejemplo, ya que han sido los que mejores resultados han mostrado.

A partir de ese grupo de notas base se "amplía" el fractal igual que en una iteración de un SFI, pero en vez de contractivo, es expansivo. Por lo tanto, si llegásemos a la n-iteración cuando $n \to \infty$ obtendríamos un fractal, aunque su escala fuera también infinita.

El algoritmo tiene cuatro variables que entran como parámetro del programa principal, *seed* y *x*, serán las mismas que en *logistic* o *1fnoise*; *i* será el número de iteraciones que se realizarán del algoritmo y *base* que es el número de notas que formarán parte del "punto" base. El algoritmo se llama de forma recursiva, comenzando por producir las notas *base*, devolviendo una pequeña pista de audio con esas notas y la nota media calculada entre las notas obtenidas; esta pista recibe la pista, vuelve a producir la primera nota del algoritmo, calcula la diferencia entre la nota calculada y la nota media y aplica esta diferencia a la pista recibida, por lo que todas las notas de la pista recibida se "desplazan"; esto ocurre con tantas notas como se haya introducido en *base* y se devuelve a la siguiente iteración la pista con la concatenación de todos los conjuntos "desplazados"; así hasta que se realicen todas las iteraciones.

Una explicación más visual se puede observar en las siguientes figuras:

Para simplicidad del algoritmo, se fija *logistic* para que sólo produzca notas en una escala, *base = 4* para observar lo mejor posible los patrones sin ejecuciones muy largas, *seed = 3.91, x = 0.12.*



Figuras 4.4.2 y 4.4.3: Notas obtenidas en las iteraciones 1 y 2 de *hybridfractal*

Como se puede ver en las Figuras 4.4.2 y 4.4.3, se tiene el patrón de la iteración 1 a la izquierda repetido cada zona de la iteración 2, desplazado de forma que se aprecian las mismas pendientes en la siguiente escala.

En la siguiente ejecución, en la Figura 4.4.4, se muestra la iteración 10 del algoritmo, por lo tanto, con $4^{10} = 2^{20} = 1048576\, notas$. Esta ejecución ha sido bastante más rápida de lo esperado, esto es debido a que la función de cambio de tono no es llamada tantas veces como antes, aplicándose de forma más eficiente a varias notas a la vez.

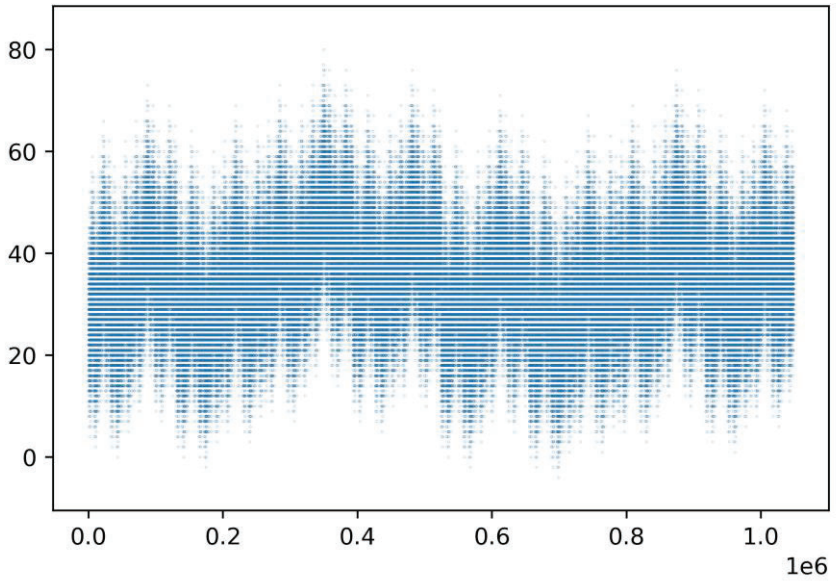


Figura 4.4.4: Notas obtenidas con 10 iteraciones de *hybridfractal*

Si ampliamos la imagen para observar las notas obtenidas más de cerca:

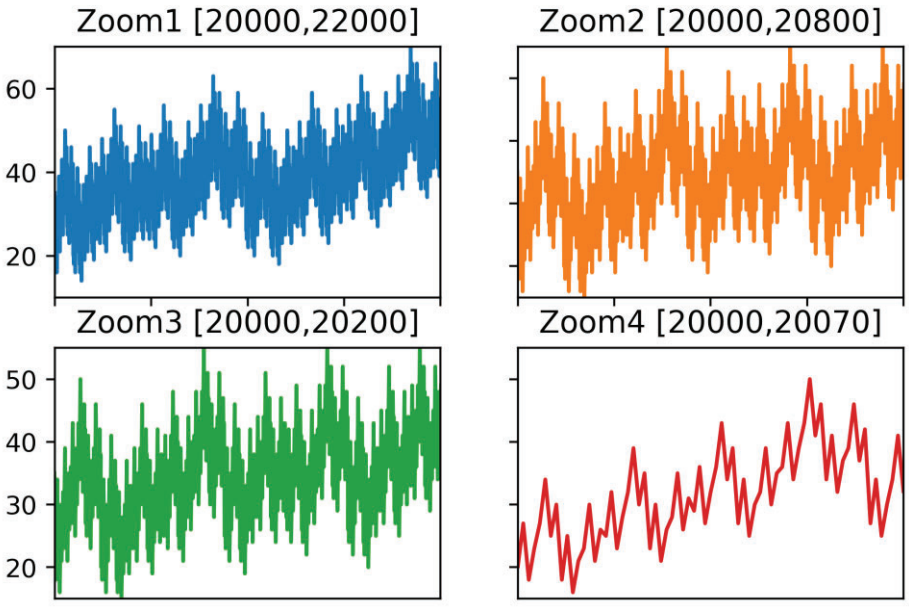


Figura 4.4.5: Ampliaciones de Figura 4.4.3

Los patrones que se muestran en la Figura 4.4.5, se repiten a todas las escalas, de forma que se obtiene un fractal con los tonos de las notas de la pieza. Debido a la dificultad y opciones de parametrización se trata de una pieza simple cuya

calidad musical no es la encontrada anteriormente, pero la razón de ser de los parámetros utilizados es la validación del algoritmo, objetivo que se ha cumplido. De hecho, si se toman mayores valores para *base*, la calidad aumenta, manteniendo su componente fractal, aunque el número de notas producido y, por lo tanto, la longitud de la pista aumente de forma exponencial con cada iteración.

Para una muestra de audio ejemplificando lo comentado, se han tomado valores *base = 20* y 2 iteraciones. El resultado es de mayor calidad que con 4 notas.

Pista de Hybridfractal: 

El código de la función recursiva es:

```
def hybridfractal(seed,x,i,base):
  if i == 1:
    nEscalas = 1; nNotas = nEscalas * 7 - 1;notaMedia = 0;
    Notas = np.array([]);NotasOG = np.array([]); Lugar = 0;
    pista = AudioSegment.silent(duration=3200)
    for j in range(base):
      nota,notaEscala,x = logistic(seed,x,nNotas)
      nota = nota + notaEscala*12
      Notas = np.concatenate((Notas,[nota]),axis=0)
      NotasOG = np.concatenate((NotasOG,[nota]),axis=0)
      notaMedia = notaMedia + nota
      Tono = pitch_shift(A2p02,nota-A2n)
      pista = pista.overlay(Tono, position=Lugar)
      Lugar = Lugar + 200
    notaMedia = round(notaMedia/base)
    return pista, notaMedia, Notas, NotasOG
  elif i > 1:
    Notas2 = np.array([]);
    pista,notaMedia,Notas,NotasOG = hybridfractal(seed,x,i-1,base)
    pista2 = AudioSegment.empty()
    nEscalas = 1; nNotas = nEscalas * 7 - 1;notaMedia2 = 0;
    for j in range(base):
      nota,notaEscala,x = logistic(seed,x,nNotas)
      nota = nota + notaEscala*12
      NotasAux = Notas + (nota - notaMedia)
      Notas2 = np.concatenate((Notas2,NotasAux),axis=0)
      Tono = pitch_shift(pista,nota-notaMedia)
      pista2 = pista2 + Tono
    return pista2, notaMedia, Notas2
```

5 Conclusiones

En este punto se discute la consecución de los objetivos propuestos, de qué forma se han conseguido y líneas futuras de mejora.

En general, se concluye de forma satisfactoria este trabajo ya que se ha conseguido el desarrollo de numerosos algoritmos de generación de música mediante Geometría Fractal y Sistemas Dinámicos, es decir, el objetivo principal, en concreto las últimas versiones de cada algoritmo y la hibridación de algoritmos para generar composiciones con valor musical destacado.

Incluso creando algoritmos originales como *hybridfractal* que mezcla *logistic* con un sistema de funciones iterado resultando en un fractal de tonos; o habiendo introducido elementos nuevos a los trabajos anteriores como la variabilidad de las figuras musicales, otros instrumentos y optimización de recursos.

De hecho, se ha conseguido la eficiencia que se buscaba, mediante una sola nota base como recurso de los algoritmos se produce el resto, en comparación con trabajos previos que se sirven de una gran base de datos para acceder a cada nota de cada instrumento cuando se necesita, o para realizar entrenamiento y validación, como es el caso de las Inteligencias Artificiales.

En cuanto al tiempo de ejecución, se comenzó con ejecuciones muy rápidas y eficientes en cuanto a tiempo pero poco efectivas, ya que se perdía el control sobre las *figuras (negras, blancas, corcheas, etc.)* que se estaban insertando; eficiencia que se perdió cierta medida, por el cambio de librería utilizado para el cambio de tono, más efectiva y precisa permitiendo variar las *figuras* intencionadamente, pero teniendo ejecuciones de, por ejemplo, 15 minutos para generación de composiciones de algo más de 1 minuto.

Por lo tanto y, finalmente, este trabajo permite posibilidades a la mejora, siendo el aspecto de los tiempos de ejecución uno de los problemas más visuales de este trabajo.

Se propone también el estudio de algoritmos que hagan uso de los números complejos para generar notas “dobles” sobre el mismo resultado del algoritmo. Otra posibilidad es tener en cuenta la estructura de la composición, como *versos-estribillo-versos-estribillo*, por ejemplo, y tomar esta estructura como otra variable del algoritmo, consiguiendo quizás composiciones dinámicas.

6 Bibliografía

[1] B.B Mandelbrot, *Les objets fractals, forme, hasard et dimension*, Flammarion, 1975

[2] Richard F. Voss y John Clarke, *"1/f noise" in music: Music from 1/f noise*, Nature, 1976.

[3] Kenneth J. Hsü y Andreas J. Hsü, *Fractal geometry of music*, Proc. Natl. Acad. Sci, 1989

[4] B.B Mandelbrot, *Self-affine Fractals and Fractal Dimension*, Phisica Scripta, 1985

[5] B.B Mandelbrot y James R. Wallis, *Some long-run properties of geophysical records*, Water Resources Research, 1969

[6] M. Kobayashi y T. Musha, 1/f Fluctuation of Heartbeat Period, *IEEE Transactions on Biomedical Engineering*, 1982

[7] Kenneth J. Hsü y Andreas J. Hsü, *Self-similarity of the "1/f noise" called music*, Proc. Natl. Acad. Sci, 1990

[8] Lars Kindermann, *MusiNum 2.08: the music in the numbers*, *documentación del programa*, https://reglos.de/musinum , 2000

[9] Gustavo Díaz Jerez, https://fractmus.com/

[10] https://github.com/jiaaro/pydub

[11] https://zenodo.org/record/4792298#.YdCDrGjMJPY

[12] https://newt.phys.unsw.edu.au/jw/notes.html

7 Figuras

2.1: Cantor Set: https://matemelga.wordpress.com/2018/09/29/el-conjunto-de-cantor/

2.2: Triángulo de Sierpinsky: https://repository.eafit.edu.co/bitstream/handle/10784/199/JorgeHumberto_RestrepoRestrepo_2011.pdf

2.3: Transformaciones Geométricas: http://institucional.us.es/olimpiada2006/talleres/talleres20-21/18%20DIC20-RPiedra-%20Geometria.pdf

2.4: UK Box-Counting: https://www.researchgate.net/figure/Box-counting-Method-applied-to-measure-roughness-of-Coastline-of-Britain-With-similar_fig10_326305093

2.5: Diagrama de Feigenbaum: https://commons.wikimedia.org/wiki/File:Logistic_map_bifurcation_diagram_from_1_to_4.png

2.6: Mandelbrot: https://commons.wikimedia.org/wiki/File:Mandelset_hires.png

2.7: Julia Set: http://www-old.dma.fi.upm.es/docencia/cursosanteriores/99-00/segundociclo/sistdin/sdmandelbrot.html

3.1: Ruidos: https://www.researchgate.net/figure/Sample-time-series-of-white-top-pink-middle-and-brown-bottom-noise_fig4_232236967

3.2: Autómata Celular: Original.

4.2.1: Notas en secuencia obtenidas por *logistic_v0_r0*: Original.

4.2.2: Traducción de notas mediante *pydub*: Original.

4.2.3: Notas obtenidas por *logistic_v0_r1*: Original.

4.2.4: Notas obtenidas por *logistic_v1_r0*: Original.

4.2.5: Ampliación de notas de *logistic_v1_r0*: Original.

4.2.6: Notas obtenidas por *logistic_v1_r1*: Original.

4.2.7: Notas obtenidas por *logistic_v2_r0*: Original.

4.2.8: Notas obtenidas por *logistic_v2_r1*: Original.

4.2.9: Notas obtenidas por *logistic_v3_r0*: Original.

4.2.10: Notas obtenidas por *logistic_v3_r1*: Original.

4.2.11: Notas obtenidas por *logistic_v3_r2*: Original.

4.3.1: Notas obtenidas por *morsethue_vf_r0*: Original.

4.3.2: Notas obtenidas por *morsethue_vf_r1*: Original.

4.3.3: Notas obtenidas por *earthworm_vf_r0*: Original.

4.3.4: Notas obtenidas por *earthworm_vf_r1*: Original.

4.3.5: Notas obtenidas por *3n1_vf_r0*: Original.

4.3.6: Notas obtenidas por *3n1_vf_r1*: Original.

4.3.7: Notas obtenidas por *sierpinsky_vf_r0*: Original.

4.3.8: Notas obtenidas por *1fnoise_vf_r0*: Original.

4.3.9: Notas obtenidas por *1fnoise_vf_r1*: Original.

4.4.1: Verde: Violonchelo. Rojo: Acordes de Piano. Azul: Piano Solista: Original.

4.4.2: Notas obtenidas en la iteracion 1 de *hybridfractal*: Original.

4.4.3: Notas obtenidas en la iteracion 2 de *hybridfractal*: Original.

4.4.4: Notas obtenidas con 10 iteraciones de *hybridfractal*: Original.

4.4.5: Ampliaciones de Figura 4.4.3: Original.

8 Anexo

Funciones Auxiliares:

```
# Mapping de las notas de piano
def mapping(x):
    return {
        'A0': 1,'Bb0': 2,'B0': 3,
        'C1': 4, 'Db1': 5, 'D1': 6, 'Eb1': 7, 'E1': 8, 'F1': 9, 'Gb1': 10,'G1': 11,'A
b1': 12,'A1': 13, 'Bb1': 14, 'B1': 15,
        'C2': 16,'Db2': 17,'D2': 18,'Eb2': 19,'E2': 20,'F2': 21,'Gb2': 22,'G2': 23,'A
b2': 24,'A2': 25, 'Bb2': 26, 'B2': 27,
        'C3': 28,'Db3': 29,'D3': 30,'Eb3': 31,'E3': 32,'F3': 33,'Gb3': 34,'G3': 35,'A
b3': 36,'A3': 37, 'Bb3': 38, 'B3': 39,
        'C4': 40,'Db4': 41,'D4': 42,'Eb4': 43,'E4': 44,'F4': 45,'Gb4': 46,'G4': 47,'A
b4': 48,'A4': 49, 'Bb4': 50, 'B4': 51,
        'C5': 52,'Db5': 53,'D5': 54,'Eb5': 55,'E5': 56,'F5': 57,'Gb5': 58,'G5': 59,'A
b5': 60,'A5': 61, 'Bb5': 62, 'B5': 63,
        'C6': 64,'Db6': 65,'D6': 66,'Eb6': 67,'E6': 68,'F6': 69,'Gb6': 70,'G6': 71,'A
b6': 72,'A6': 73, 'Bb6': 74, 'B6': 75,
        'C7': 76,'Db7': 77,'D7': 78,'Eb7': 79,'E7': 80,'F7': 81,'Gb7': 82,'G7': 83,'A
b7': 84,'A7': 85, 'Bb7': 86, 'B7': 87,
        'C8': 88
    }.get(x,"error")

# Mapping de escalas
def escalas(x):
  return {
      'doM4': np.array(['C4','D4','E4','F4','G4','A4','B4']),
      'fam4': np.array(['F4','G4','Ab4','Bb4','C4','Db4','Eb4']),
      'doM2': np.array(['C2','D2','E2','F2','G2','A2','B2']),
      'fam2': np.array(['F2','G2','Ab2','Bb2','C2','Db2','Eb2']),
  }.get(x,"error")

# Funcion auxiliar para switch
def swi(x):
    return {
        1: [0,0],
        2: [4,0],
        3: [2,2]
    }.get(x,"error")

# Funcion Cambio de Base
def cDBase(n,b):
    if n == 0:
        return '0'
    nums = []
    while n:
        n, r = divmod(n, b)
```

```
        nums.append(str(r))
    return ''.join(reversed(nums))

# Funcion Suma de Digitos
def getSum(n):
    sum = 0
    for digit in str(n):
      sum += int(digit)
    return sum

# Funcion Cambio de Tono
def pitch_shift(sound, n_steps):
    y = np.frombuffer(sound._data, dtype=np.int16).astype(np.float32)/2**15
    y = librosa.effects.pitch_shift(y, sound.frame_rate, n_steps=n_steps)
    a  = AudioSegment(np.array(y * (1<<15), dtype=np.int16).tobytes(), frame_rate = s
ound.frame_rate, sample_width=2, channels = 1)
    return a
```

Logistic_v0:

```
seed = 3.91; notab = 28; notaB = 45; x = 0.1;
Lugar = 1000; escala = notaB - notab;
Pista = AudioSegment.silent(duration=34000)
Puntos = np.array([])
for i in range(135):
  x = x * seed * (1 - x)
  nota = round(x * escala)
  nota = nota%escala + notab
  #Pfijo = nota
  Tono = int(A3.frame_rate * (2.0 ** ((nota-37)/12)))
  Tono = A3._spawn(A3.raw_data, overrides={'frame_rate': Tono})
  Tono = Tono.set_frame_rate(44100)
  Pfijo = nota
  Lugar = Lugar + 250
  Pista = Pista.overlay(Tono, position=Lugar)
  Puntos = np.concatenate((Puntos,[Pfijo]),axis=0)
Pista.export("pista.wav",format="wav")
```

Logistic_v1:

```
seed = 3.82843; escala = escalas('doM2'); x = 0.1;
Lugar = 1000; nEscalas = 3;
Pista = AudioSegment.silent(duration=34000)
Puntos4 = np.array([])
nNotas = nEscalas * 7 - 1
for i in range(450):
  x = x * seed * (1 - x)
  if i < 300:
    continue
  nota = round(x * nNotas)
```

```
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  nota = nota+(notaEscala)*12
  Pfijo4 = nota
  Tono = int(A3.frame_rate * (2.0 ** (((nota-37)/12))))
  #Pfijo5 = Tono
  Tono = A3._spawn(A3.raw_data, overrides={'frame_rate': Tono})
  Tono = Tono.set_frame_rate(44100)
  Lugar = Lugar + 200
  Pista = Pista.overlay(Tono, position=Lugar)
  Puntos4 = np.concatenate((Puntos4,[Pfijo4]),axis=0)
Pista.export("pista.wav",format="wav")
```

Plot de Figura 4.2.5:

```
plt.plot(Puntos4,'bo')
plt.xlabel('Pista (nNotas)')
plt.ylabel('Notas (int)')
C2 = mapping('C2')
bot = C2 - 3
top = C2 + nEscalas*12+3
plt.ylim(bot, top)
plt.xlim(81.8, 93.2)
plt.show()
```

Logistic v2:

```
seed = 3.82843; escala = escalas('doM2'); x = 0.1;
Lugar = 100; nEscalas = 3;
Pista = AudioSegment.silent(duration=42000)
Puntos = np.array([]);Puntos2 = np.array([]);Puntos3 = np.array([]);Puntos4 = np.arra
y([]);Puntos5 = np.array([])
A2n = mapping('A2')
nNotas = nEscalas * 7 - 1
for i in range(150):
  x = x * seed * (1 - x)
  nota = round(x * nNotas)
  Pfijo = nota
  notaTono = nota%7
  Pfijo2 = notaTono
  notaEscala = nota//7
  Pfijo3 = notaEscala
  nota = mapping(escala[notaTono])
  Pfijo5 = nota
  Pfijo4 = nota+(notaEscala)*12
  n = nota + notaEscala*12 - A2n
  Tono = pitch_shift(A2, n)
  Lugar = Lugar + 250
  Pista = Pista.overlay(Tono, position=Lugar)
```

```
  Puntos = np.concatenate((Puntos,[Pfijo]),axis=0)
  Puntos2 = np.concatenate((Puntos2,[Pfijo2]),axis=0)
  Puntos3 = np.concatenate((Puntos3,[Pfijo3]),axis=0)
  Puntos4 = np.concatenate((Puntos4,[Pfijo4]),axis=0)
  Puntos5 = np.concatenate((Puntos5,[Pfijo5]),axis=0)
Pista.export("pista.wav",format="wav")
np.set_printoptions(threshold=np.inf)
```

Logistic v3:

```
seed = 3.929; x = 0.198; bpm = 100;
mu = 2; sigma = 0.35
Lugar = 0; nEscalas = 2;
Pista = AudioSegment.silent(duration=40000)
Pfijo = np.array([0,0]);P2 = np.array([0.,0.]);Puntos = np.array([[0,0]])
escala = escalas('fam3'); A2n = mapping('A2');
nNotas = nEscalas * 7 - 1
for i in range(150):
  x = x * seed * (1 - x)
  figura = round(random.gauss(mu, sigma))
  figura = 2**(figura - 3)
  nota = round(x * nNotas)
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  P2[1] = nota+(notaEscala)*12
  nota = nota + notaEscala*12 - A2n
  Tono = pitch_shift(A2, nota)
  Pista = Pista.overlay(Tono, position=Lugar)
  Lugar = Lugar + 1000*figura*(60/bpm)
  P2[0] = Lugar
  Puntos = np.concatenate((Puntos,[P2]),axis=0)
Pista.export("pista.wav",format="wav")
```

Plot logistic_v3:

```
x_val = [x[0] for x in Puntos]
x_val = np.delete(x_val,0)
y_val = [x[1] for x in Puntos]
y_val = np.delete(y_val,0)
plt.plot(x_val,y_val)
plt.xlabel('Pista (ms)')
plt.ylabel('Notas (int)')
C2 = mapping('C2')
bot = C2 - 3
top = C2 + nEscalas*12+3
#plt.ylim(bot, top)
plt.show()
```

Morse-Thue Sequence:

```
base = 11; seed = 2; mult = 23;
escala = escalas('doM2')
Lugar = 100; nEscalas = 3;
Pista = AudioSegment.silent(duration=34000)
Puntos = np.array([])
A2n = mapping('A2')
nNotas = nEscalas * 7 - 1
for i in range(135):
  nota = getSum(cDBase(seed,base))
  nota = nota%nNotas
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  Pfijo = nota+(notaEscala)*12
  nota = nota + notaEscala*12 - A2n
  Tono = pitch_shift(A2, nota)
  Lugar = Lugar + 250
  Pista = Pista.overlay(Tono, position=Lugar)
  Puntos = np.concatenate((Puntos,[Pfijo]),axis=0)
  seed = seed * mult;
Pista.export("pista.wav",format="wav")
```

Earthworm Sequence:

```
Worm = 111; seed = 6; mult = 7;
escala = escalas('doM2');A2n = mapping('A2');
Lugar = 100; nEscalas = 3; nNotas = nEscalas * 7 - 1;
Pista = AudioSegment.silent(duration=34000)
Puntos = np.array([])
for i in range(135):
  if seed >= Worm :
    seed = seed%Worm
  nota = seed
  nota = nota%nNotas
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  Pfijo = nota+(notaEscala)*12
  nota = nota + notaEscala*12 - A2n
  Tono = pitch_shift(A2, nota)
  Lugar = Lugar + 250
  Pista = Pista.overlay(Tono, position=Lugar)
  Puntos = np.concatenate((Puntos,[Pfijo]),axis=0)
  seed = seed * mult;
Pista.export("pista.wav",format="wav")
```

3n+1:

```
seed = 871;
escala = escalas('doM2');A2n = mapping('A2');
Lugar = 100; nEscalas = 3; nNotas = nEscalas * 7 - 1;
Pista = AudioSegment.silent(duration=60000)
Puntos = np.array([])
for i in range(220):
  nota = seed
  nota = nota%nNotas
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  Pfijo = nota+(notaEscala)*12
  nota = nota + notaEscala*12 - A2n
  Tono = pitch_shift(A2, nota)
  Lugar = Lugar + 250
  Pista = Pista.overlay(Tono, position=Lugar)
  if (seed % 2) == 0:
    seed = seed//2;
  else:
    seed = 3*seed + 1;
  Puntos = np.concatenate((Puntos,[Pfijo]),axis=0)
Pista.export("pista.wav",format="wav")
```

Triángulo de Sierpinsky:

```
escala = escalas('doM2');A2n = mapping('A2');
nEscalas = 3; nNotas = nEscalas * 7 - 1;
Pista = AudioSegment.silent(duration=35000)
Pfijo = np.array([0,0]);
P2 = np.array([0.,0.]);
Puntos = np.array([[]])
for i in range(2200):
  Prand = random.randint(1,3)
  Prand = swi(Prand)
  Prand = np.array([Prand])
  Dist = (Prand - Pfijo)/2
  Pfijo = Pfijo + Dist
  if i < 900:
    continue
  P2[0] = (Pfijo[0,0]/4)*30000
  nota = nNotas*Pfijo[0,1]/2
  nota = round(nota)
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  P2[1] = nota+(notaEscala)*12
  nota = nota + notaEscala*12 - A2n
  #Tono = pitch_shift(A2, nota)
```

```
  Lugar = round((Pfijo[0,0]/4)*30000)
  #Pista = Pista.overlay(Tono, position=Lugar)
  Puntos = np.concatenate((Puntos,[P2]),axis=0)
Pista.export("pista.wav",format="wav")
```

Plot de Triángulo de Sierpinsky:

```
x_val = [x[0] for x in Puntos]
y_val = [x[1] for x in Puntos]
plt.plot(x_val,y_val,'ro',ms=1)
plt.xlabel('Pista (ms)')
plt.ylabel('Notas (int)')
C2 = mapping('C2')
bot = C2 - 3
top = C2 + nEscalas*12+3
plt.ylim(bot, top)
plt.show()
```

1fnoise:

```
seed = 0.99; escala = escalas('doM2'); x = 0.1;
Lugar = 100; nEscalas = 3;
Pista = AudioSegment.silent(duration=42000)
Puntos = np.array([]);
A2n = mapping('A2')
nNotas = nEscalas * 7 - 1
for i in range(135):
  r = random.random()
  x = x * seed + r * (1 - seed ** 2) ** (1/2)
  nota = round(x * nNotas)
  nota = nota%nNotas
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  Pfijo = nota+(notaEscala)*12
  n = nota + notaEscala*12 - A2n
  Tono = pitch_shift(A2, n)
  Lugar = Lugar + 250
  Pista = Pista.overlay(Tono, position=Lugar)
  Puntos = np.concatenate((Puntos,[Pfijo]),axis=0)
Pista.export("pista.wav",format="wav")
```

Hybrid Piano Solista, Piano Acordes y Violonchelo:

```
# PIANO 1/f - seed = "random" ; mu = corcheas ; sigma = cambios regulares ; 2 escalas
 desde C3 #
seed = 0.84; x = 0.90; bpm = 100;
mu = 2; sigma = 0.35
Lugar = 5000; nEscalas = 2;
duration = 70000;
Pista = AudioSegment.silent(duration=80000)
```

```
P2 = np.array([0.,0.]);piano = np.array([[0,0]]);pianoAcord = np.array([[0,0]]);cello
 = np.array([[0,0]]);
violinS = np.array([[0,0]]);violin = np.array([[0,0]]);
escala = escalas('doM3');
C3 = mapping('C3');
nNotas = nEscalas * 7 - 1
for i in range(180):
  if Lugar > duration+5000:
    break
  r = random.random()
  x = x * seed + r * (1 - seed ** 2) ** (1/2)
  figura = round(random.gauss(mu, sigma))
  figura = 2**(figura - 3)
  nota = round(x * nNotas)
  notaTono = nota%7
  notaEscala = nota//7 - 2
  if notaEscala < 0:
    continue
  nota = mapping(escala[notaTono])
  P2[1] = nota+(notaEscala)*12
  nota = nota + notaEscala*12 - A2n
  Tono = pitch_shift(A2p, nota)
  Pista = Pista.overlay(Tono, position=Lugar)
  Lugar = Lugar + 1000*figura*(60/bpm)
  P2[0] = Lugar
  piano = np.concatenate((piano,[P2]),axis=0)
############################################################
# PIANO ACORDES logistic - seed = 6 ciclo ; mu = redondas ; sigma = pocos cambios ; 1
 escala en bajo#
seed = 3.702; x = 0.1; bpm = 100;
mu = 4; sigma = 0.32
Lugar = 100; nEscalas = 1;
P3 = np.array([0.,0.]);
cello = np.array([[0,0]]);
escala = escalas('doM2')
nNotas = nEscalas * 7 - 1
for i in range(60):
  x = x * seed * (1 - x)
  figura = round(random.gauss(mu, sigma))
  figura = 2**(figura - 3)
  nota = round(x * nNotas)
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  P3[1] = nota+(notaEscala)*12
  nota = nota + notaEscala*12 - A2n
  Tono1 = pitch_shift(A2p, nota) - 15
  Tono2 = pitch_shift(A2p, nota+4) - 15
  Tono3 = pitch_shift(A2p, nota+7) - 15
```

```
  Pista = Pista.overlay(Tono1, position=Lugar)
  Pista = Pista.overlay(Tono2, position=Lugar)
  Pista = Pista.overlay(Tono3, position=Lugar)
  Lugar = Lugar + 1000*figura*(60/bpm)
  P3[0] = Lugar
  pianoAcord = np.concatenate((pianoAcord,[P3]),axis=0)
############################################################
# CELLO logistic - seed = 4 ciclo ; mu = redondas ; sigma = pocos cambios ; 1 escala
en bajo#
seed = 3.54409; x = 0.1; bpm = 100;
mu = 4; sigma = 0.32
Lugar = 0; nEscalas = 1;
P2 = np.array([0.,0.]);
cello = np.array([[0,0]]);
escala = escalas('doM2')
nNotas = nEscalas * 7 - 1
for i in range(80):
  if Lugar>duration:
    break
  x = x * seed * (1 - x)
  figura = round(random.gauss(mu, sigma))
  figura = 2**(figura - 3)
  nota = round(x * nNotas)
  notaTono = nota%7
  notaEscala = nota//7
  nota = mapping(escala[notaTono])
  P2[1] = nota+(notaEscala)*12
  nota = nota + notaEscala*12 - E2n
  Tono = pitch_shift(E2c, nota)
  Tono = Tono - 15
  Pista = Pista.overlay(Tono, position=Lugar)
  Lugar = Lugar + 1000*figura*(60/bpm)
  P2[0] = Lugar
  cello = np.concatenate((cello,[P2]),axis=0)
Pista.export("pista.wav",format="wav")
```

Funciones Hybrid Fractal:

```
# Sistema Dinamico 1fnoise
def noise1f(seed,x,nNotas):
  Nota = np.array([0.,0.])
  r = random.random()
  x = x * seed + r * (1 - seed ** 2) ** (1/2)
  nota = round(x * nNotas)
  notaTono = nota%7
  Nota[1] = nota//7 - 2 # -2 es para normalizar.
  Nota[0] = mapping(escala[notaTono])
  return Nota


# Sistema Dinamico logistic
def logistic(seed,x,nNotas):
  Nota = np.array([0.,0.]);escala = escalas('doM3');
  x = x * seed * (1 - x)
  nota = round(x * nNotas)
  notaTono = nota%7
  Nota[1] = nota//7
  Nota[0] = mapping(escala[notaTono])
  return Nota[0],Nota[1],x;

# Funcion recursiva hybridfractal
def hybridfractal(seed,x,i,base):
  if i == 1:
    nEscalas = 1; nNotas = nEscalas * 7 - 1;notaMedia = 0;
    Notas = np.array([]);NotasOG = np.array([]); Lugar = 0;
    pista = AudioSegment.silent(duration=1000)
    for j in range(base):
      nota,notaEscala,x = logistic(seed,x,nNotas)
      nota = nota + notaEscala*12
      Notas = np.concatenate((Notas,[nota]),axis=0)
      NotasOG = np.concatenate((NotasOG,[nota]),axis=0)
      notaMedia = notaMedia + nota
      Tono = pitch_shift(A2p05,nota-A2n)
      pista = pista + Tono
      pista = pista.overlay(Tono, position=Lugar)
      Lugar = Lugar + 250
    notaMedia = round(notaMedia/base)
    return pista, notaMedia, Notas, NotasOG
  elif i > 1:
    Notas2 = np.array([]);
    pista,notaMedia,Notas,NotasOG = hybridfractal(seed,x,i-1,base)
    pista2 = AudioSegment.empty()
    nEscalas = 1; nNotas = nEscalas * 7 - 1;notaMedia2 = 0;
    for j in range(base):
      nota,notaEscala,x = logistic(seed,x,nNotas)
```

```
        nota = nota + notaEscala*12
        NotasAux = Notas + (nota - notaMedia)
        Notas2 = np.concatenate((Notas2,NotasAux),axis=0)
        Tono = pitch_shift(pista,nota-notaMedia)
        pista2 = pista2 + Tono
    return pista2, notaMedia, Notas2, NotasOG

# Ejecucion
pistaX,notaMedia,Notas,NotasOG = hybridfractal(3.91,0.12,10,4)
pistaX.export("pistax.wav",format="wav")
pistaX


# Plot con mayor calidad
fig, ax = plt.subplots()
plt.plot(Notas,'o',ms=0.015)
plt.savefig('myimage.png', format='png', dpi=400)

# Plot de Figura 4.4.5:
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
ax1.set_ylim(10, 70)
ax1.set_xlim(20000, 22000)
ax1.set_title('Zoom1 [20000,22000]')
ax1.plot(Notas)
ax2.set_ylim(15, 55)
ax2.set_xlim(20000, 20800)
ax2.set_title('Zoom2 [20000,20800]')
ax2.plot(Notas, 'tab:orange')
ax3.set_ylim(15, 55)
ax3.set_xlim(20000, 20800)
ax3.axes.xaxis.set_visible(False)
ax3.set_title('Zoom3 [20000,20200]')
ax3.plot(Notas, 'tab:green')
ax4.set_ylim(15, 55)
ax4.set_xlim(20000, 20070)
ax4.axes.xaxis.set_visible(False)
ax4.set_title('Zoom4 [20000,20070]')
ax4.plot(Notas, 'tab:red')
for ax in fig.get_axes():
    ax.label_outer()
fig.savefig('myimage.png', format='png', dpi=400)
```

Este documento esta firmado por

Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
Fecha/Hora	Thu Jan 20 21:03:59 CET 2022
Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
Numero de Serie	561
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)