

Doble Grado en Ingeniería Informática y Matemáticas
Curso 2021/2022
Trabajo de Fin de Grado

Fractales y Geometría Fractal

Fractales, geometría fractal y aplicaciones en la ciencia. Visualización de fractales con Ray-Tracing

Autor: Juan Antonio Villegas Recio

Autor: Juan Antonio Villegas Recio
Tutor de Matemáticas: Manuel Ruiz Galán, Catedrático de Universidad
Departamento de Matemática Aplicada, Universidad de Granada
Tutor de Informática: Carlos Ureña Almagro, Profesor Titular de Universidad
Departamento de Lenguajes y Sistemas Informáticos, Universidad de Granada

 AUTORÍA

I hereby affirm that this Master thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text. This work has not been submitted for any other degree or professional qualification except as specified; nor has it been published.

City, date



Juan Antonio Villegas Recio



ABSTRACT

(no more than 250-300 words)

Background

Describe background shortly

Aim

Describe the aim of your study

Method

Describe your methods

Results

Describe the main results of your study

Conclusion

State your conclusion

Keywords:

No more than six keywords, preferably MeSH terms

ÍNDICE GENERAL

Lista de Abreviaturas	II
Lista de Imágenes	III
Lista de Tablas	IV
1. El concepto de <i>fractal</i>	2
1.1. Ejemplos clásicos	3
1.1.1. El conjunto de Cantor	3
1.1.2. El triángulo de Sierpinski	4
1.1.3. La alfombra de Sierpinski y la esponja de Menger	5
1.1.4. La curva de Koch	5
1.1.5. El copo de nieve de Koch	6
1.2. Conceptos de dimensión fractal	7
1.2.1. Dimension fractal autosimilar	8
1.2.2. Dimensión por cajas	10
1.2.3. Medida y dimensión de Hausdorff	11
1.2.4. Dimensión topológica	12
1.2.5. Relación entre los distintos tipos de dimensión fractal	13
2. Iteración	15
2.1. Iteración de funciones	15
2.1.1. Convergencia a un punto fijo	16
2.1.2. Velocidad de convergencia	17
2.2. El método de Newton y cuencas de atracción	18
2.2.1. Autosimilaridad	21
3. Conjuntos de Julia y Mandelbrot	23
3.1. Iteración convergente y no convergente	24
3.2. Conjuntos de Julia	25
3.2.1. Representación gráfica de los conjuntos de Julia	25
3.3. Distinción entre conjuntos de Julia conexos y polvaredas	27
3.4. El conjunto de Mandelbrot	28
3.4.1. Representación gráfica del conjunto de Mandelbrot	29
3.5. Autosimilaridad de los conjuntos de Julia y Mandelbrot	30
3.5.1. Autosimilaridad en conjuntos de Julia	30

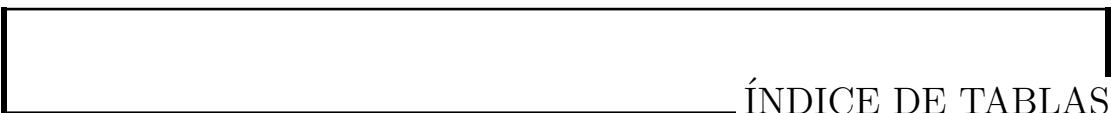
3.5.2. Autosimilaridad en el conjunto de Mandelbrot	31
3.6. Conjuntos de Julia y Mandelbrot generalizados	32
3.6.1. Familia $\{z^N + c\}_{c \in \mathbb{C}}$	32
3.6.2. Conjuntos de Julia con funciones no polinómicas	35
4. Introducción a las herramientas de renderización	37
4.1. Componentes de WebGL	38
4.1.1. Contexto de WebGL	38
4.1.2. El programa <i>Shader</i>	38
4.1.3. Los <i>Buffer</i>	40
4.2. Primera imagen renderizada	41
5. visualización de fractales en 2D	44
5.1. Objetivo	44
5.2. Estructurando el código	45
Appendices	48
A. Appendix title	48
B. Another Appendix	49

LISTA DE ABREVIATURAS

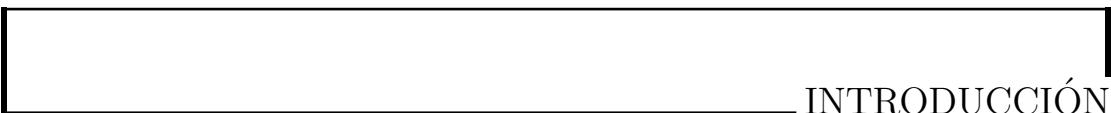
- API: Application Programming Interface (Interfaz de Programación de Aplicaciones)
- CPU: Central Processing Unit (Unidad Central de Procesamiento)
- GPU: Graphics Processing Unit (Unidad de Procesamiento Gráfico)
- SDF: Signed Distance Function
- SFI: Sistema de Funciones Iteradas

LISTA DE IMÁGENES

1.1.	Objetos de la naturaleza con estructura fractal	2
1.2.	Iteraciones del proceso de generación del conjunto de Cantor	4
1.3.	Generación del triángulo de Sierpinski	5
1.4.	Generación de la alfombra de Sierpinski	5
1.5.	Esponja de Menger	6
1.6.	Iteraciones del proceso de generación de la curva de Koch	6
1.7.	Generación del copo de nieve de Koch	7
1.8.	Curvas de Koch que componen el copo de Koch	7
1.9.	Possible movimiento de un punto en posibles objetos de \mathbb{R}^n	8
1.10.	Segmento, cuadrado y cubo divididos en objetos a razón $\frac{1}{2}$	9
1.11.	Una forma de calcular la dimensión por cajas de la curva de Koch .	11
1.12.	Figuras representativas de los ejemplos	13
2.1.	Representación de dos órbitas en \mathbb{C}	16
2.2.	Cuencas de atracción de $f(z) = z^2 - 1$	19
2.3.	Cuencas de atracción de distintas funciones coloreadas.	20
2.4.	Evaluación de la velocidad de convergencia en cada punto	21
2.5.	Cuencas de atracción de $f(z) = z^3 - 1$	21
2.6.	Diferentes regiones ampliadas de la figura 2.5	22
3.1.	Primeras imágenes de conjuntos de Julia	23
3.2.	Primeras imágenes del conjunto de Mandelbrot	24
3.3.	Conjuntos de Julia graficados con Mathematica	27
3.4.	Resultados de la orden ‘JuliaSetPlot’	27
3.5.	Representación de \mathcal{M} y detalle en $[-0.65, -0.4] \times [0.47, 0.72]$	30
3.6.	Diferentes regiones ampliadas de la figura 3.3 (b)	31
3.7.	Detalles autosimilares de algunos conjuntos de Julia	31
3.8.	Detalles autosimilares de \mathcal{M}	32
3.9.	Ampliaciones del bulbo principal de \mathcal{M}	33
3.10.	Conjuntos de Julia con $P_{-0.55+0.48i, N}$ para distintos valores de N . .	33
3.11.	Conjuntos de Mandelbrot \mathcal{M}_N para $N = 3, 4, 8, 10$	34
4.1.	<i>Clip space de WebGL</i>	39
4.2.	Cuadrado de colores renderizado con WebGL	42
4.3.	Componentes de WebGL e interacciones entre ellos	43



ÍNDICE DE TABLAS



INTRODUCCIÓN

TODO

CAPÍTULO 1

EL CONCEPTO DE *FRACTAL*

Las primeras preguntas que se pueden plantear son ¿qué es un fractal? ¿Qué tienen de especial estas figuras? ¿Qué las diferencia de un objeto no fractal? ¿Por qué es necesaria una geometría fractal? Trataremos de responder a cada una de estas preguntas a lo largo de este capítulo, comenzando por la primera de ellas. En realidad, hay distintas definiciones de *fractal*, pero todas utilizan dos conceptos como base: la **autosimilaridad** y la **dimensión**. La primera de ellas es más cercana para nosotros de lo que en un principio podemos pensar, fijémonos en los ejemplos de la imagen 1.1.



(a) Romanescu



(b) Rayo

Imagen 1.1: Objetos de la naturaleza con estructura fractal

Observemos que el romanescu, que es un tipo de coliflor, pareciera que está formado de pequeños trozos que recuerdan el objeto original, mientras que estos pequeños trozos a su vez también están formados de pequeños trozos similares al objeto inicial, y así sucesivamente. Por su parte, el rayo se compone de un destello principal del que salen ramificaciones de las que a su vez se originan otras divisiones, formando pequeños rayos semejantes al rayo primitivo.

Esta idea de objetos prácticamente iguales al original salvo cambios de escala es la subyacente al concepto de autosimilaridad.

Definición 1.0.1 (Autosimilaridad). Una figura o subconjunto A de \mathbb{R}^n es **autosimilar** si está compuesto por copias de sí mismo reducidas mediante un

factor de escala y desplazadas por un movimiento rígido. Es decir,

$$A = \bigcup_{i=1}^n f_i \circ h_i(A),$$

donde cada $h_i, i = 1, \dots, n$ es una homotecia de razón menor que 1 y $f_i, i = 1, \dots, n$ es un movimiento rígido.

En futuras ocasiones se utilizarán indistintamente los términos de «reducción por un factor de escala» e «imagen vía una homotecia», evidenciando el movimiento rígido y queriendo en ambos casos referirnos a este concepto.

Para afianzar y formalizar conceptos y con el objetivo de introducir una definición de la dimensión, estudiaremos algunos ejemplos clásicos de objetos fractales.

1.1. Ejemplos clásicos

En adelante, y salvo que se indique lo contrario, cuando hablamos en términos topológicos de \mathbb{R}^n o subconjuntos suyos nos estaremos refiriendo al espacio topológico \mathbb{R}^n dotado de la topología usual o la topología inducida por la usual en el caso de subconjuntos de \mathbb{R}^n .

1.1.1. El conjunto de Cantor

Creado por el célebre matemático *George Cantor*, este fractal se construye a partir de un segmento de línea recta aplicando el siguiente proceso iterativo:

1. Partimos del segmento de recta compuesto por el intervalo cerrado $[0, 1]$, aunque realmente es indiferente qué intervalo se escoja, pues el resultado final será el mismo salvo homotecia. Dividimos dicho segmento en tres segmentos iguales y extraemos el intervalo central, manteniendo los extremos. Es decir, extraemos el intervalo abierto $(\frac{1}{3}, \frac{2}{3})$ y mantenemos el segmento $[0, \frac{1}{3}]$ y el $[\frac{2}{3}, 1]$. Nótese que obtenemos $2 = 2^1$ segmentos, cada uno a escala $\frac{1}{3} = (\frac{1}{3})^1$ del original.
2. Aplicamos el mismo proceso a los segmentos $[0, \frac{1}{3}]$ y $[\frac{2}{3}, 1]$. Esto es, se dividen ambos en tres partes iguales y se extrae el intervalo abierto central de cada uno de ellos, manteniendo los extremos. En este caso obtendríamos $4 = 2^2$ segmentos iguales, cada uno de ellos a escala $\frac{1}{3}$ de los dos obtenidos en el primer paso y a escala $\frac{1}{9} = (\frac{1}{3})^2$ del original.
3. Repetimos este proceso de manera indefinida, de manera que en el n -ésimo paso se obtendrían 2^n segmentos de recta a escala $(\frac{1}{3})^n$. Denotemos como C_n al conjunto unión de los 2^n segmentos de recta que se generan en el paso n del proceso.

Los puntos del intervalo inicial $[0, 1]$ que restan tras las infinitas iteraciones son los que conforman el *conjunto de Cantor*, que denotamos con \mathbf{C} , de forma que $\mathbf{C} = \bigcap_{n \in \mathbb{N}} C_n$.

El conjunto de Cantor es además un conjunto compacto, pues cada C_n es una unión finita de intervalos cerrados y acotados de \mathbb{R} , y por tanto compactos, como



Imagen 1.2: Iteraciones del proceso de generación del conjunto de Cantor

sabemos gracias al *teorema de Heine-Borel*. Sabiendo que la intersección arbitraria de conjuntos cerrados es cerrada, y que cada C_n está acotado, tenemos pues que \mathbf{C} es un conjunto compacto.

Observemos ahora que en la primera iteración eliminamos 1 segmento de longitud $\frac{1}{3}$, en la segunda iteración se eliminan 2 segmentos de longitud $(\frac{1}{3})^2$ y en la n -ésima iteración extraemos 2^{n-1} segmentos de longitud $(\frac{1}{3})^n$. Si sumamos las longitudes de todos los segmentos que son eliminados en cada paso se obtiene:

$$\sum_{n=1}^{\infty} 2^{n-1} \left(\frac{1}{3}\right)^n = \frac{1}{3} \sum_{n=0}^{\infty} 2^n \left(\frac{1}{3}\right)^n = \frac{1}{3} \sum_{n=0}^{\infty} \left(\frac{2}{3}\right)^n = \frac{1}{3} \left(\frac{1}{1 - \frac{2}{3}}\right) = 1,$$

donde hemos usado que la suma de una serie geométrica de razón $|q| < 1$ es $\sum_{n=0}^{\infty} q^n = \frac{1}{1-q}$.

Esto nos lleva a concluir que la longitud eliminada es igual a la longitud original, es decir, la longitud de \mathbf{C} es nula y aún así tenemos puntos, por ejemplo los extremos de los intervalos que se van generando. Es decir, los puntos de \mathbf{C} no están agrupados, sino que forman una especie de *polvareda*.

1.1.2. El triángulo de Sierpinski

Esta figura, original del polaco *Waclaw Sierpinski*, es creada de una manera que evoca al conjunto de Cantor, pero en dos dimensiones. Veamos detalladamente el proceso (ver imagen 1.3):

1. Se parte de un triángulo equilátero de lado 1 (de nuevo la longitud inicial es irrelevante). Uniendo los puntos medios de cada lado obtenemos una partición del triángulo inicial en 4 triángulos equiláteros, del cual extraemos el interior del triángulo central. Tenemos por tanto $3 = 3^1$ triángulos a escala $\frac{1}{2} = (\frac{1}{2})^1$ del original.
2. En cada uno de estos tres triángulos equiláteros se repite la operación anterior, obteniendo por tanto $9 = 3^2$ triángulos, cada uno a escala $\frac{1}{4} = (\frac{1}{2})^2$ del original.
3. Repetimos este proceso indefinidamente, de forma que en el paso n -ésimo se tienen 3^n triángulos, cada uno de ellos a escala $(\frac{1}{2})^n$ del original.



Imagen 1.3: Generación del triángulo de Sierpinski

La figura a la que converge este proceso infinito se conoce como triángulo **S** de Sierpinski.

Si llamamos A al área del triángulo inicial, sabemos que en la primera iteración eliminamos un área de $\frac{1}{4}A$, en el segundo eliminamos $3\left(\frac{1}{4}\right)^2 A$ y en el n -ésimo $3^{n-1}\left(\frac{1}{4}\right)^n A$, de forma que si sumamos todo el área que eliminamos en cada paso obtenemos:

$$A \sum_{n=1}^{\infty} 3^{n-1} \left(\frac{1}{4}\right)^n = \frac{A}{4} \sum_{n=0}^{\infty} 3^n \left(\frac{1}{4}\right)^n = \frac{A}{4} \sum_{n=0}^{\infty} \left(\frac{3}{4}\right)^n = \frac{A}{4} \left(\frac{1}{1 - \frac{3}{4}}\right) = A,$$

En este caso ocurre algo parecido a lo que vimos que sucedía con el conjunto de Cantor en la sección 1.1.1. El área eliminada es igual al área total, es decir, su área es 0 y seguimos teniendo puntos (por ejemplo los vértices de los triángulos originados en cada iteración). Es decir, los puntos que forman **S** no están agrupados formando un área.

1.1.3. La alfombra de Sierpinski y la esponja de Menger

El propio Sierpinski se dio cuenta que con el mismo patrón utilizado para generar **S** se pueden obtener otras formas. Por ejemplo, pensemos que en lugar de comenzar con un triángulo equilátero partimos de un cuadrado, lo subdividimos en 9 cuadrados de lado $\frac{1}{3}$ y extraemos el cuadrado central. Repitiendo este proceso indefinidamente con cada uno de los cuadrados que se generan finalmente se obtiene la llamada alfombra de Sierpinski (ver imagen 1.4).

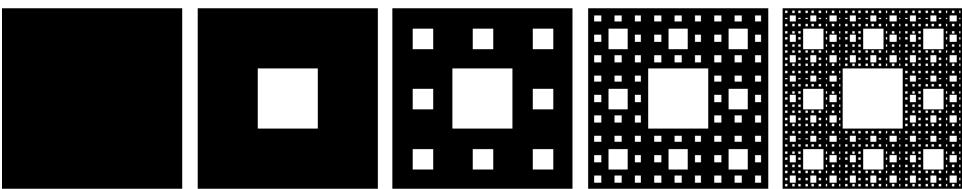


Imagen 1.4: Generación de la alfombra de Sierpinski

Este proceso también se puede modelar en 3D, obteniendo así la conocida como esponja de Menger o cubo de Magritte, que es una generalización en tres dimensiones de la alfombra de Sierpinski, la cual podemos ver en la imagen 1.5.

1.1.4. La curva de Koch

Esta figura fractal, creada por el sueco *N. F. Helge von Koch* sigue un proceso de construcción iterativo al igual que el conjunto de Cantor, pero en lugar de eliminar segmentos, se añaden de la siguiente manera (ver imagen 1.6):

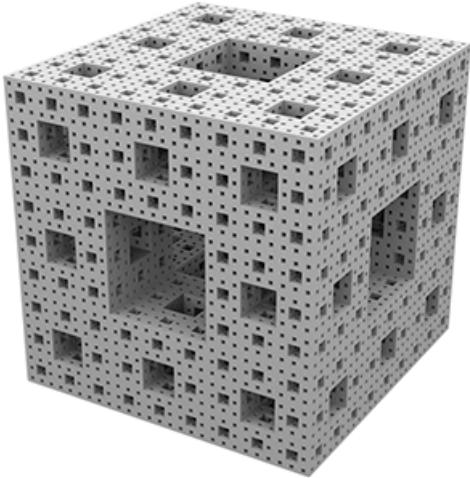


Imagen 1.5: Esponja de Menger

1. Partiendo de un segmento de recta de longitud 1 (al igual que en el conjunto de Cantor, la longitud del segmento inicial es irrelevante, pues la figura final es la misma salvo homotecia), se divide en tres partes iguales de longitud $\frac{1}{3}$ y la parte central se sustituye por un triángulo equilátero al que se le elimina la base. Esto da lugar a $4 = 4^1$ segmentos de recta de longitud $\frac{1}{3} = \left(\frac{1}{3}\right)^1$.
2. Repetimos este proceso en cada uno de los segmentos de recta obtenidos, colocando el triángulo siempre por encima de la recta, obteniendo así $16 = 4^2$ segmentos de recta de longitud $\frac{1}{9} = \left(\frac{1}{3}\right)^2$.
3. Aplicamos este proceso indefinidamente, obteniendo en el paso n -ésimo 4^n segmentos de longitud $\left(\frac{1}{3}\right)^n$.

El resultado final del proceso es lo que llamamos la *curva de Koch*, que denotamos como **K**.

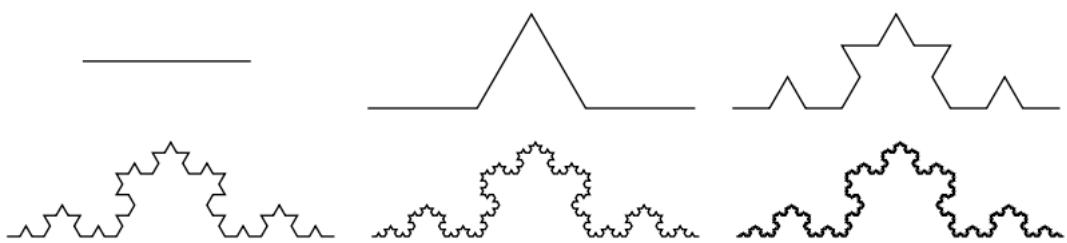


Imagen 1.6: Iteraciones del proceso de generación de la curva de Koch

1.1.5. El copo de nieve de Koch

A partir de la curva de Koch podemos generar un objeto matemático muy particular: el copo de nieve de Koch. Para crearlo, basta aplicar el proceso iterativo descrito para generar la curva de Koch a cada uno de los segmentos que componen un triángulo equilátero, de forma que los triángulos que se generan apunten hacia el exterior.

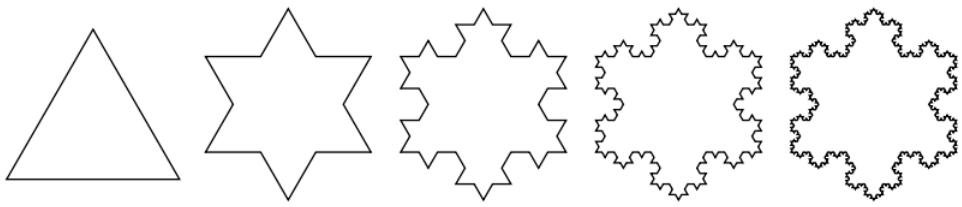


Imagen 1.7: Generación del copo de nieve de Koch

Esta curva posee la particularidad de tener longitud infinita y a su vez encerrar un área finita. Realmente el copo de Koch no es exactamente un fractal, pues no es totalmente autosimilar, aunque se compone de tres partes idénticas las cuales sí son autosimilares, como podemos ver en la imagen 1.8.

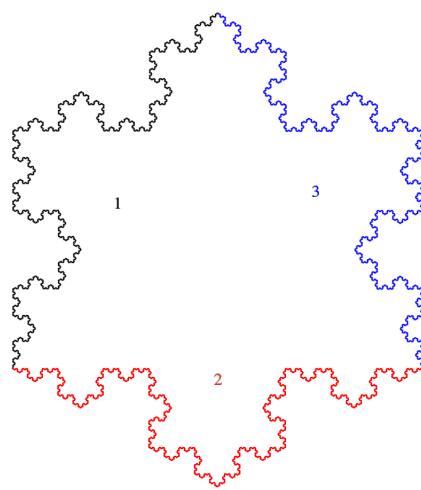


Imagen 1.8: Curvas de Koch que componen el copo de nieve de Koch

1.2. Conceptos de dimensión fractal

Al iniciar este capítulo mencionamos que las distintas definiciones de fractal utilizaban los conceptos de autosimilaridad y dimensión. Con los distintos ejemplos hemos entendido el primero de ellos, por lo que es momento de abordar el concepto de dimensión.

El concepto de dimensión más clara que tenemos es el de dimensión algebraica, esto es, la dimensión de un espacio vectorial. Sabemos que un espacio vectorial V se dice que tiene $\dim(V) = n$, con $n \in \mathbb{N}$ si, y solo si existe una base de V constituida por n vectores, de forma que cualquier elemento de V puede ser expresado como una combinación lineal de los n vectores de la base. Otra manera de ver esto es que para construir los vectores de V tenemos hasta n parámetros de libertad, esto es, $v = a_1v_1 + \dots + a_nv_n \quad \forall v \in V$, donde $\{v_1, \dots, v_n\}$ es una base de V y $a_1, \dots, a_n \in K$ siendo K el cuerpo sobre el que está construido el espacio vectorial.

En el caso de subconjuntos de \mathbb{R}^n como puede ser una curva parametrizada, si seguimos la analogía del número de parámetros que define un punto en este caso de una curva, podemos decir que su dimensión es 1, ya que un punto de una curva

parametrizada puede expresarse en función de un único parámetro. La variación de este parámetro nos daría otro punto de la curva, por lo que podemos decir que un punto puede moverse por la curva con un grado de libertad (ver imagen 1.9 (a)). Por su lado, una superficie regular de \mathbb{R}^3 puede localmente ser expresada como la imagen de una parametrización que depende de dos variables y la variación de estas mediante dicha parametrización resulta en otro punto de la superficie, pudiendo expresar esto como que un punto de una superficie regular tiene dos grados de libertad, lo que intuitivamente permite afirmar que una superficie regular tiene dimensión 2 (ver imagen 1.9 (b)).

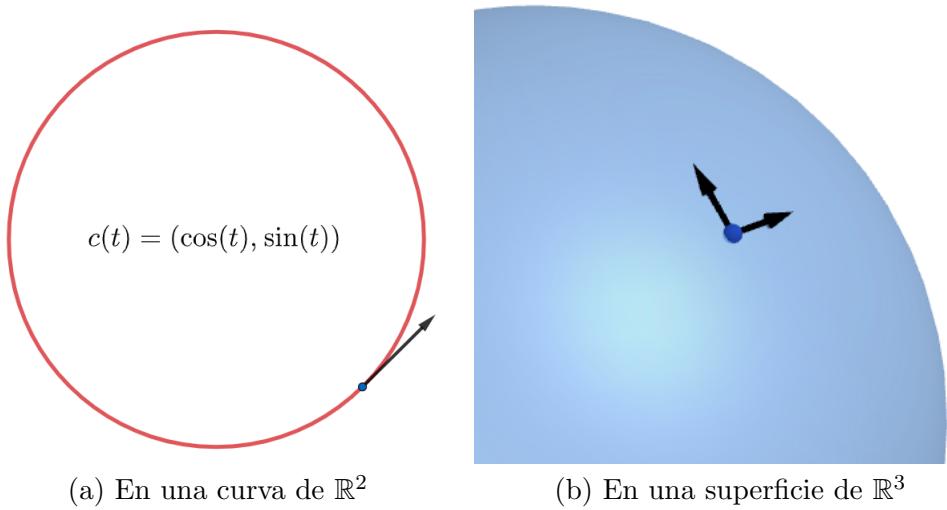


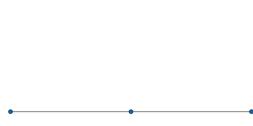
Imagen 1.9: Posible movimiento de un punto en posibles objetos de \mathbb{R}^n

Un posible enfoque para definir la dimensión puede ser el recién presentado, el cual nos sugiere pensar en el número de parámetros que definen la libertad de movimiento de un punto. Sin embargo, pensemos ahora en el triángulo de Sierpinski y en su dimensión. Comprobamos en la sección 1.1.2 que su área 2-dimensional es nula, pero en el objeto final pareciera que un punto se pudiera mover en varias direcciones. No se puede afirmar que S tenga dimensión 1 por la movilidad, pero tampoco dimensión 2 porque tiene área 0, pero sería un valor situado entre estos dos enteros.

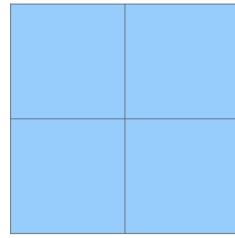
1.2.1. Dimension fractal autosimilar

Pensemos ahora en un segmento de recta, un cuadrado y un cubo, que son objetos indudablemente de 1, 2 y 3 dimensiones respectivamente. Ahora dividamos los lados de cada objeto en 2 tal y como indica la imagen 1.10. Entonces en el segmento aparecen $N(2) := 2 = 2^1$ segmentos que son imagen de una homotecia de razón $r = \frac{1}{2}$ del original, en el cuadrado se obtienen $N(2) := 4 = 2^2$ cuadrados, cada uno imagen de una homotecia de razón $\frac{1}{2}$ del cuadrado inicial y en el cubo $N(2) := 8 = 2^3$ cubos, de nuevo cada uno de ellos imagen de una homotecia de razón $\frac{1}{2}$ del cubo original.

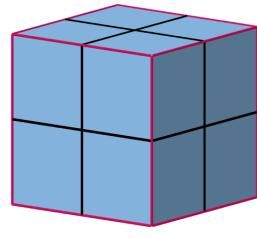
Si en lugar de 2 tomamos cualquier número natural $k \geq 1$, la razón de la homotecia sería $r = \frac{1}{k}$ y en ese caso se obtendrían $N(k) = k^1$ segmentos de recta, $N(k) = k^2$ cuadrados y $N(k) = k^3$ cubos, en todos los casos a escala $r = \frac{1}{k}$ de los



(a) Segmento dividido en 2



(b) Cuadrado dividido en 4



(c) Cubo dividido en 8

Imagen 1.10: Segmento, cuadrado y cubo divididos en objetos a razón $\frac{1}{2}$

objetos originales. Estas igualdades se pueden reescribir como:

$$\frac{N(k)}{k^1} = 1 \quad \frac{N(k)}{k^2} = 1 \quad \frac{N(k)}{k^3} = 1 \quad (1.1)$$

En este sentido, vemos que la dimensión de cada objeto es el *exponente* al que habría que elevar la razón de la homotecia k para obtener la relación (1.1). Para abreviar en el lenguaje introducimos la siguiente definición.

Definición 1.2.1 (Congruencia). Dos subconjuntos A, B de \mathbb{R}^n son **congruentes** si existe un movimiento rígido f tal que $B = f(A)$.

De manera análoga al segmento, el cuadrado y el cubo podemos tomar cualquier figura o subconjunto $X \subseteq \mathbb{R}^n$ que pueda ser subdividido en un número finito $N(k)$ de subconjuntos congruentes entre sí y cada uno de ellos imagen por una homotecia de razón $r = \frac{1}{k}$ de X . Esto es, en lenguaje formal, un conjunto autosimilar.

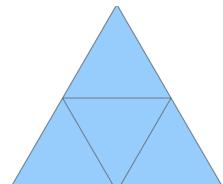
Definición 1.2.2 (Dimensión autosimilar). Dado un subconjunto X de \mathbb{R}^n que se puede dividir en $N(k)$ subconjuntos congruentes entre sí y cada uno de ellos imagen por una homotecia de razón $r = \frac{1}{k}$ de X , definimos su **dimensión autosimilar** como el único valor de d que satisface la identidad $\frac{N(k)}{k^d} = 1$, es decir:

$$d = \frac{\log N(k)}{\log(k)}$$

También se puede ver definida como *dimensión de homotecia*. Sin embargo, antes de utilizar este concepto debemos comprobar que la definición es coherente con cualquier partición y cualquier factor de escala. Es decir, que independientemente de la división del objeto que se tome la dimensión va a ser la misma.

Proposición 1.2.1. En un subconjunto X de \mathbb{R}^n autosimilar el valor de la dimensión fractal autosimilar es el mismo independientemente de la partición que se le aplique.

Demostración. TODO □



Un ejemplo distinto pero clarificador podría ser el de un triángulo equilátero, el cual podemos dividir en 4 copias congruentes entre sí, cada una a escala $r = \frac{1}{2}$ del triángulo original, de forma que tendríamos $k = 2$ y $N(2) = 4$.

Con estos valores, tenemos que $N(k) = 4 = k^d = 2^d$, por lo que necesariamente $d = 2$ y por tanto un triángulo equilátero es un objeto 2-dimensional, lo cual tampoco nos sorprende.

Observación 1.2.1. La dimensión d ya no tiene por qué ser necesariamente un número entero. A esta posibilidad de tener dimensiones no enteras es a lo que llamamos **dimensión fractal autosimilar**, y lo denotamos con \dim_A .

Retomando ahora el triángulo de Sierpinski, sabemos que este está compuesto por 3 copias de sí mismo a escala $\frac{1}{2}$ (véase sección 1.1.2), por lo que entonces su dimensión fractal autosimilar es $\dim_A(\mathbf{S}) = \frac{\log 3}{\log 2} \approx 1.58496$. Efectivamente y tal y como discutimos en el inicio de esta sección, es un valor situado entre 1 y 2.

En cuanto al conjunto de Cantor, este está formado por 2 copias de sí mismo a escala $\frac{1}{3}$ (sección 1.1.1), por lo que su dimensión fractal autosimilar sería $\dim_A(\mathbf{C}) = \frac{\log 2}{\log 3} \approx 0.63093$, que es un valor situado entre 0 y 1.

Por su parte, la curva de Koch se compone de 4 copias de sí misma, cada una a escala $\frac{1}{3}$ de la curva original (sección 1.1.4). Por tanto, su dimensión fractal autosimilar es $\dim_A(\mathbf{K}) = \frac{\log 4}{\log 3} \approx 1.26186$.

1.2.2. Dimensión por cajas

El concepto de dimensión fractal autosimilar tiene el defecto de que está limitado únicamente a objetos totalmente autosimilares, pero la mayoría de objetos no cumple esta propiedad. Para solucionar este problema, introduciremos un nuevo concepto de dimensión. Previamente definimos un término necesario en las futuras definiciones.

Definición 1.2.3. Dado un subconjunto U de \mathbb{R}^n , definimos su **diámetro** como:

$$\text{diam}(U) = \sup \{d(x, y) : x, y \in U\}.$$

Dado un conjunto $A \subset \mathbb{R}^n$ no vacío y acotado, sea $N_\delta(A)$ el mínimo número de conjuntos de diámetro a lo sumo δ necesario para recubrir A . La *dimensión por cajas superior* y la *dimensión por cajas inferior* se definen, respectivamente como

$$\begin{aligned}\underline{\dim}_B(A) &:= \liminf_{\delta \rightarrow 0} \frac{\log(N_\delta(A))}{\log(1/\delta)}, \\ \overline{\dim}_B(A) &:= \limsup_{\delta \rightarrow 0} \frac{\log(N_\delta(A))}{\log(1/\delta)}.\end{aligned}$$

En caso de coincidir, se denomina *dimensión por cajas* de A al valor

$$\dim_B(A) := \lim_{\delta \rightarrow 0} \frac{\log(N_\delta(A))}{\log(1/\delta)}. \quad (1.2)$$

También es conocida en literatura como *dimensión por conteo de cajas* o *dimensión de Minkowski-Bouligand*.

Hay varias definiciones equivalentes, generalmente más sencillas de utilizar. Por ejemplo, si tomamos en \mathbb{R}^n una cuadrícula de lado δ , *i.e.*, de la forma

$$[m_1\delta, (m_1 + 1)\delta] \times \cdots \times [m_n\delta, (m_n + 1)\delta]$$

donde m_1, \dots, m_n son enteros, tendríamos \mathbb{R}^n dividido en ‘cajas’, de diámetro $\delta\sqrt{n}$ y contando el número de cajas que recubren a A , obtenemos el mismo resultado, puede comprobarse en [3, sección 3.1].

¹Nótese la analogía con la notación de los límites superior e inferior como $\underline{\lim}$ y $\overline{\lim}$.

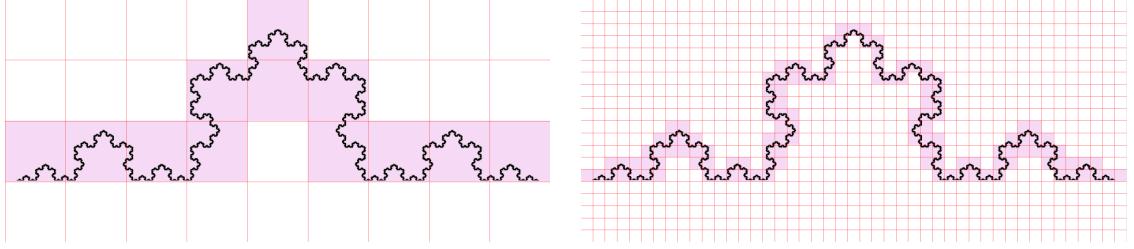


Imagen 1.11: Una forma de calcular la dimensión por cajas de la curva de Koch

1.2.3. Medida y dimensión de Hausdorff

La dimensión por cajas tiene el inconveniente de que no tenemos garantizada la existencia de límite en la ecuación (1.2) o hipotéticos problemas con conjuntos más generales, por ejemplo conjuntos densos. Así, $F = \mathbb{Q} \cap [0, 1] \subset \mathbb{R}$ es un conjunto en el que cada punto por separado tiene obviamente dimensión por cajas igual a 0, pero al ser \mathbb{Q} un conjunto denso en \mathbb{R} la dimensión por cajas de F es 1. Por tanto, no se cumple en general que para una familia $\{F_i\}$ de conjuntos $\dim_B \bigcup_{i=1}^{\infty} F_i = \sup_i \dim_B F_i$. Para solucionar estos problemas *Felix Hausdorff* publicó en 1919 un artículo que cambiaría la teoría de la medida tal y como la conocíamos [4].

Si ahora tomamos un conjunto A de \mathbb{R}^n , un valor $\varepsilon > 0$ y una familia numerable de conjuntos $\{U_i\}_{i \in \mathbb{N}}$ tales que

$$A \subseteq \bigcup_{i \in \mathbb{N}} U_i, \quad 0 \leq \text{diam}(U_i) \leq \varepsilon \quad \forall i \in \mathbb{N}$$

se dice que la familia $\{U_i\}_{i \in \mathbb{N}}$ es un ε -recubrimiento de A . Si ahora consideramos un valor $s > 0$, definimos

$$\mathcal{H}_\varepsilon^s(A) := \inf \left\{ \sum_{n \in \mathbb{N}} \text{diam}(U_i)^s : \{U_i\}_{i \in \mathbb{N}} \text{ es un } \varepsilon\text{-recubrimiento de } A \right\}$$

Si reducimos el valor de ε el número de posibles recubrimientos disminuye, por lo que $\mathcal{H}_\varepsilon^s(A)$ aumenta. Por esto nos planteamos cuál será el límite cuando ε tienda a cero, aceptando posiblemente el infinito como posible valor del límite. Definimos así

$$\mathcal{H}^s(A) := \lim_{\varepsilon \rightarrow 0} \mathcal{H}_\varepsilon^s(A)$$

En [3, Secciones 5.2 y 5.4] se comprueba que $\mathcal{H}^s(A)$ es una medida definida en la σ -álgebra de Borel que llamamos *medida s-dimensional de Hausdorff* del conjunto A .

Veamos el comportamiento de $\mathcal{H}^s(A)$ como función de s . Es claro que siempre que $\varepsilon < 1$, $\mathcal{H}_\varepsilon^s(A)$ decrece conforme s aumenta, por tanto $\mathcal{H}^s(A)$ también es decreciente. Podemos de hecho probar el siguiente resultado.

Proposición 1.2.2. Sean $A \subset \mathbb{R}^n$, $t > s$, $0 < \varepsilon < 1$ y $\{U_i\}$ un ε -recubrimiento de A . Entonces

$$\mathcal{H}_\varepsilon^t(A) \leq \varepsilon^{t-s} \mathcal{H}_\varepsilon^s(A)$$

Demostración. Sabemos que $\text{diam}(U_i)^{t-s} \leq \varepsilon^{t-s} \forall i \in \mathbb{N}$ y del hecho de que $\sum_{i \in \mathbb{N}} \text{diam}(U_i)^t = \sum_{i \in \mathbb{N}} \text{diam}(U_i)^{t-s} \text{diam}(U_i)^s$ deducimos que

$$\sum_{i \in \mathbb{N}} \text{diam}(U_i)^t \leq \varepsilon^{t-s} \sum_{i \in \mathbb{N}} \text{diam}(U_i)^s.$$

Tomando el ínfimo en la anterior desigualdad tenemos que

$$\mathcal{H}_\varepsilon^t(A) \leq \sum_{i \in \mathbb{N}} \text{diam}(U_i)^t \leq \varepsilon^{t-s} \sum_{i \in \mathbb{N}} \text{diam}(U_i)^s,$$

por lo que

$$\mathcal{H}_\varepsilon^t(A) \leq \varepsilon^{t-s} \mathcal{H}_\varepsilon^s(A)$$

□

Esta desigualdad nos será muy útil para probar el siguiente teorema que nos da la definición definitiva de lo que llamaremos dimensión de Hausdorff.

Teorema 1.2.1. *Sea $A \subset \mathbb{R}^n$. Entonces existe un único valor de s para el cual $\mathcal{H}^s(A)$ no es ni 0 ni ∞ . Este valor $s = \dim_H(A)$ satisface que:*

$$\mathcal{H}^s(A) = \begin{cases} \infty & \text{si } s < D_H(A) \\ 0 & \text{si } s > D_H(A) \end{cases} \quad (1.3)$$

Demostración. Si tomamos $0 < \varepsilon < 1$ y $t > s$ dos valores reales, por la proposicion 1.2.2 tenemos que $\mathcal{H}_\varepsilon^t(A) \leq \varepsilon^{t-s} \mathcal{H}_\varepsilon^s(A)$. Por tanto, al hacer a ε tender a 0, siempre que $\mathcal{H}^s(A)$ sea finito necesariamente $\mathcal{H}^t(A) = 0$. Si aplicamos el contrarrecíproco ocurre que si $\mathcal{H}^t(A) > 0$ entonces $\mathcal{H}^s(A) = \infty$.

Por lo que necesariamente debe de existir un valor $s_0 \in [0, \infty]$ tal que $\mathcal{H}^s(A) = 0 \forall s < s_0$ y $\mathcal{H}^s(A) = \infty \forall s > s_0$. □

Definición 1.2.4 (Dimensión de Hausdorff). Llamamos **dimensión de Hausdorff** de un conjunto A al único valor $\dim_H(A)$ que satisface las condiciones del teorema 1.2.1.

1.2.4. Dimensión topológica

Por último estudiaremos un concepto de dimensión aplicable a espacios topológicos. La **dimensión topológica** se define inductivamente de la siguiente forma:

1. La dimensión del conjunto vacío es $\dim_T(\emptyset) := -1$.
2. Un espacio topológico X tiene dimensión 0 ($\dim_T(X) = 0$) si para cualquier $x \in V$ con V abierto en X existe un abierto U cuya frontera $\partial(U)$ es vacía y se verifica que $x \in U \subseteq V$.
3. Un espacio topológico X tiene dimensión menor o igual que n ($\dim_T(X) \leq n$) si para cualquier $x \in X$ y cualquier V abierto que contiene a x existe un abierto U tal que $\dim_T(\partial(U)) \leq n-1$ y se verifica que $x \in U \subseteq V$.
4. X tiene dimensión n ($\dim_T(X) = n$) si se verifica que $\dim_T(X) \leq n$ pero es falso que $\dim_T(X) \leq n-1$



Imagen 1.12: Figuras representativas de los ejemplos

Ejemplo 1.2.1. (Véase imagen 1.12 (a)) En \mathbb{R}^2 dotado de la topología usual consideramos un punto p cualquiera y el espacio topológico $X = \{p\}$. Por tanto la topología inducida por la topología usual τ en X es $\tau_X = \{X, \emptyset\}$. Sea V un abierto de \mathbb{R}^2 que contenga a p , por tanto $\tilde{V} = V \cap X = \{p\}$ es un abierto en X que contiene a p . Podemos encontrar entonces un abierto $U = \{p\}$ de X tal que su frontera $\partial(U) = \emptyset$ y además $p \in U \subseteq V$. Por tanto concluimos que X tiene dimensión 0.

Ejemplo 1.2.2. (Véase imagen 1.12 (b)) En \mathbb{R}^2 dotado de la topología usual, consideramos una recta cualquiera, llamémosla r , y el espacio topológico $X = \{r\}$. Sea $x \in X$ y V un abierto de \mathbb{R}^2 que contenga a x , de forma que $\tilde{V} = r \cap V$ es un abierto de X . Podemos entonces tomar un abierto U dentro de \tilde{V} (un segmento abierto de recta), de forma que $x \in U \subseteq \tilde{V}$. Por otro lado, $\partial(U) = \{p, q\}$, y podemos comprobar fácilmente (con ayuda del ejemplo 1.2.1) que $\dim_T(\partial(U)) = 0$. Por todo esto podemos concluir que $X = r$ es un espacio topológico de dimensión 1.

1.2.5. Relación entre los distintos tipos de dimensión fractal

La dimensión por cajas, aunque útil en la práctica, al comienzo de la sección 1.2.3 hemos comprobado que tiene algunos problemas. No obstante, para muchos fractales, y en particular para los que cumplen la *condición de conjunto abierto* (véase [7]), resulta más sencillo computacionalmente calcular su dimensión por cajas frente a su dimensión de Hausdorff.

En general, y como se puede comprobar en [3, Sección 3.1], ocurre que la dimensión por cajas acota superiormente a la dimensión de Hausdorff, esto es, dado $A \subset \mathbb{R}^n$:

$$\dim_H(A) \leq \underline{\dim}_B(A) \leq \overline{\dim}_B(A),$$

donde, insistimos, es posible que se dé la igualdad.

Sobre la dimensión topológica, se tiene que si X es un espacio topológico

separable², entonces

$$\dim_T(X) \leq \dim_H(X).$$

Este resultado fue originalmente probado por *Edward Szpilrajn*, véase [5, Capítulo VII].

Por otro lado, si nos restringimos a conjuntos totalmente autosimilares, existe un resultado que relaciona la dimensión de Hausdorff y la dimensión por cajas para conjuntos totalmente autosimilares: el teorema de Moran [6]. Este resultado nos asegura que en estos casos la dimensión por cajas y la dimensión de Hausdorff coinciden.

En conclusión, para un conjunto no vacío y acotado $A \subseteq \mathbb{R}^n$, se tiene que:

$$\dim_T(A) \leq \dim_H(A) \leq \dim_B(A) \leq n \quad (1.4)$$

A partir de esta cadena de desigualdades, en la que notamos que la dimensión fractal, entendiendo por esta la dimensión de Hausdorff que es la más general, y que siempre excede o iguala a la dimensión topológica, podemos enunciar nuestra primera definición de fractal, que fue formulada por *Benoit Mandelbrot* en [2].

Definición 1.2.5 (Fractal). Un **fractal** es un subconjunto de \mathbb{R}^n que es autosimilar y cuya dimensión fractal excede a su dimensión topológica.

²Recordamos que un espacio topológico es *separable* si contiene un conjunto denso numerable

CAPÍTULO 2

ITERACIÓN

Como hemos podido comprobar en el capítulo anterior, muchos de los fractales clásicos son generados repitiendo indefinidamente un proceso. Iteración es el proceso de repetir una y otra vez un método, ocasionalmente sobre el resultado de la aplicación anterior. Este procedimiento es muy útil en muchas disciplinas matemáticas. Por ejemplo, existen métodos numéricos basados en la iteración como el método de *Jacobi* y *Gauss-Seidel* para resolución de sistemas de ecuaciones lineales, el método de *Newton-Raphson* para encontrar soluciones de ecuaciones, el método de *Runge-Kutta* para resolución numérica de ecuaciones diferenciales, etc. Incluso en otras disciplinas como el aprendizaje automático los algoritmos de *K-Means* para “clustering” o los métodos de generación de árboles de decisión en problemas de clasificación hacen uso de procesos iterativos. Esta metodología aplicada sobre el plano complejo y sobre ciertas funciones complejas será la que nos proporcionará nuestros primeros ejemplos de imágenes fractales.

2.1. Iteración de funciones

Definición 2.1.1. Consideramos una función $f : \mathbb{C} \rightarrow \mathbb{C}$ y un punto $z \in \mathbb{C}$. La aplicación sucesiva de f a z – i.e. $z, f(z), f(f(z)), f(f(f(z))), \dots$ – produce las *iteradas* de la función f en el punto z . Al conjunto de dichas iteradas se le denomina *órbita* $O_f(z)$ de f en z .

$$O_f(z) := \{z, f(z), f^2(z), \dots, f^n(z), \dots\}.$$

donde f^n denota a $f \circ f^{n-1}$.

Lo siguiente es plantearse la posible convergencia de la sucesión $\{f^n(z)\}$. Para ello, y a partir de este momento nos ayudaremos del software **Mathematica**® en su versión 12 (concretamente la versión 12.1)¹. El comando `NestList[f,z,n]` itera una función `f`, comenzando en el punto `z` un total de `n` veces y devuelve una lista con los `n` valores.

Para saber qué ocurre a largo plazo podemos iterar un número grande de veces, fíjémonos lo que ocurre si utilizamos $f(z) := z^2$ comenzando por $z_0 = 0.9$.

¹Los códigos completos que generan cada una de las imágenes que se observan en este documento se pueden encontrar en <https://github.com/JAntonioVR/Geometria-Fractal/tree/main/Iteracion-y-Fractales>

```
In[1]:= f[z_] := z^2;
NestList[f, 0.9, 10]

Out[1]= {0.9, 0.81, 0.6561, 0.430467, 0.185302, 0.0343368,
0.00117902, 1.39008*10^-6, 1.93233*10^-12, 3.73392*10^-24,
1.39421*10^-47}
```

Como se puede observar, en cada iteración se acerca cada vez más a 0, lo cual denotamos con $\{f^n(0.9)\} \rightarrow 0$. En este caso todos los valores eran reales, pero aprovechando la correspondencia de \mathbb{C} con \mathbb{R}^2 , de forma que un número complejo $z = x + y \cdot i \in \mathbb{C}$ se corresponde con el par $(x, y) \in \mathbb{R}^2$ podemos representar en el plano y de forma visual la tendencia de dichas sucesiones. Observemos los ejemplos de las imágenes 2.1.

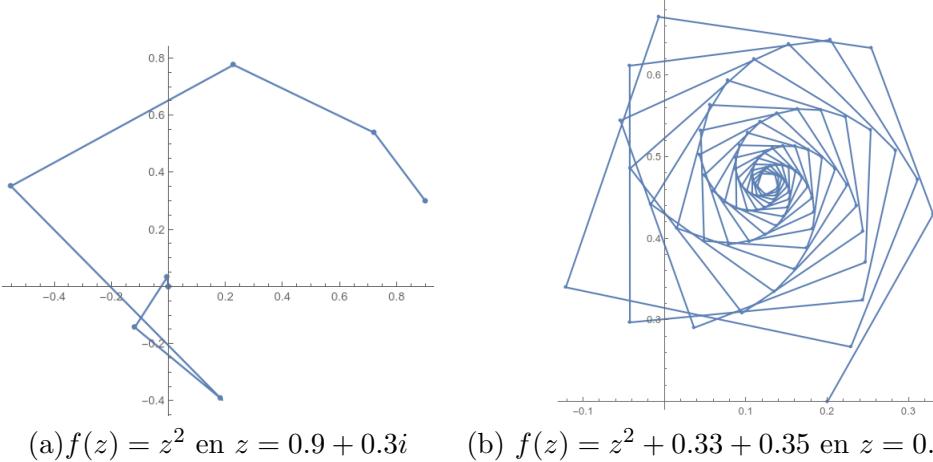


Imagen 2.1: Representación de dos órbitas en \mathbb{C}

Los ejemplos expuestos hasta ahora son todos convergentes, pero esto no es siempre así. Nuestro objetivo ahora es poder conocer el comportamiento a largo plazo de la órbita de una función y un punto dados.

2.1.1. Convergencia a un punto fijo

Fijémonos que en el ejemplo anterior si en lugar de tomar $z_0 = 0.9$ hubiésemos tomado cualquier valor con $|z_0| < 1$ la convergencia habría sido igualmente a 0, pues elevamos al cuadrado cada vez números más pequeños. Justo al contrario ocurre si $|z_0| > 1$, en cuyo caso la sucesión diverge. Por último, en caso de que $|z_0| = 1$, esto es, $z_0 \in S^1$, la sucesión $\{f^n(z_0)\}$ nunca saldrá de S^1 , siendo este el conjunto que delimita la frontera entre el conjunto de puntos cuya sucesión converge o diverge. En particular, $f^n(-1) = f^n(1) = 1 \quad \forall n \in \mathbb{N}$, por lo que $z = 1$ es un punto fijo de f , es decir, $f(z) = z$.

Nos interesa particularmente saber qué sucesiones de iteradas convergen y a qué elemento convergen. En este sentido *Stephan Banach* demostró uno de los resultados más útiles y vigentes del análisis funcional:

Teorema 2.1.1 (Punto fijo de Banach). *Sea X un espacio métrico completo y $f : X \rightarrow X$ una aplicación contractiva. Entonces f tiene un único punto fijo. Además, la sucesión de iteradas $\{f^n(x_0)\}$ converge a dicho punto fijo para cualquier $x_0 \in X$.*

Demostración. Probaremos primero la existencia:

Sea $x_0 \in X$ y sea la sucesión de iteradas $\{x_n\} = \{f^n(x_0)\}$. Por ser f contractiva, llamamos $K < 1$ a su constante de Lipschitz. Probaremos por inducción que $d(x_n, x_{n+1}) \leq K^n d(x_0, x_1) \quad \forall n \in \mathbb{N}$. El caso base es cierto pues $d(x_1, x_2) = d(f(x_0), f(x_1)) \leq Kd(x_0, x_1)$. Si suponemos que la hipótesis es cierta para cierto n , tenemos que

$$d(x_{n+1}, x_{n+2}) = d(f(x_n), f(x_{n+1})) \leq Kd(x_n, x_{n+1}) \leq K^{n+1}d(x_0, x_1).$$

Ahora, para $n, r \in \mathbb{N}$:

$$\begin{aligned} d(x_n, x_{n+r}) &\leq \sum_{j=0}^{r-1} d(x_{n+j}, x_{n+j+1}) \leq d(x_1, x_0) \sum_{j=0}^{r-1} K^{n+j} \leq K^n d(x_0, x_1) \sum_{j=0}^{\infty} K^j \\ &= \frac{d(x_0, x_1)K^n}{1-K}. \end{aligned}$$

Dado $\varepsilon > 0$, como $\{K^n\} \rightarrow 0$, existe un $m \in \mathbb{N}$ tal que para $n > m$ se tiene que $d(x_0, x_1)K^n < \varepsilon(1 - K)$. Si ahora tomamos dos naturales $p, q \geq m$ con $p < q$, aplicando la desigualdad anterior tomando $n = p$ y $r = q - p$ obtenemos que:

$$d(x_p, x_q) = d(x_n, x_{n+r}) \leq \frac{d(x_0, x_1)K^n}{1-K} < \varepsilon$$

de donde deducimos que $\{x_n\}$ es una sucesión de Cauchy, y como X es completo, tenemos que $\{x_{n+1}\} = \{f(x_n)\} = \{f^n(x_0)\} \rightarrow x \in X$. Pero f es continua, por lo que si $\{x_n\} \rightarrow x$, necesariamente $\{f(x_n)\} \rightarrow f(x)$, por lo que $f(x) = x$.

Para probar la unicidad, suponemos que $y \in X$ es otro punto fijo de f , por lo que $d(x, y) = d(f(x), f(y)) \leq Kd(x, y)$. Tomando el primero y el tercer miembro de la desigualdad deducimos que $(1 - K)d(x, y) \leq 0$, luego $d(x, y) = 0$. \square

Este teorema unido a que sabemos que \mathbb{C} , y por tanto todos sus subconjuntos cerrados, es completo, nos permite asegurar convergencia de las sucesiones $f^n(x_0)$ a un punto fijo para cualquier x_0 siempre que dicha función sea contractiva.

2.1.2. Velocidad de convergencia

Si recordamos el teorema del punto fijo de Banach (teorema 2.1.1), que nos decía que las funciones contractivas en un espacio métrico completo admiten un único punto fijo, el cual se puede hallar iterando la función. Recordemos que dada una función $f : \mathbb{C} \rightarrow \mathbb{C}$ contractiva, esta cumple que $|f(z_1) - f(z_2)| \leq K|z_1 - z_2| \quad \forall z_1, z_2 \in \mathbb{C}$, siendo $0 < K < 1$ la constante de Lipschitz de f , nombrada en este caso la constante de contractividad. Fijémonos que, si iteramos n veces la función f

$$\begin{aligned} |f^n(z_0) - f^{n-1}(z_0)| &\leq K|f^{n-1}(z_0) - f^{n-2}(z_0)| \\ &\leq K^2|f^{n-2}(z_0) - f^{n-3}(z_0)| \\ &\leq \dots \leq K^{n-1}|f(z_0) - z_0| \end{aligned}$$

por lo que en realidad la constante K define de forma genérica la velocidad de convergencia en términos de la función. Cuanto más cercana a 0, más rápida será la convergencia al punto fijo. Sin embargo, vemos que realmente también depende

de la condición inicial fijada, y es este el aspecto que explotaremos para obtener imágenes fractales.

En los casos que tengamos asegurada la convergencia, es interesante saber cuántas iteraciones son necesarias en cada punto para saber si hemos alcanzado el valor al que converge la sucesión. Este alcance se toma como aproximado, pues generalmente nunca se llega a alcanzar el punto fijo, tan solo podemos reducir la diferencia tanto como queramos. En este sentido, *Mathematica* tiene dos comandos útiles:

- `FixedPointList[f,expr]` genera una lista de valores resultantes de aplicar `f` a `expr` repetidamente hasta que los dos últimos valores no cambian. Para parametrizar la precisión en la proximidad de los valores podemos utilizar el argumento opcional `SameTest`².
- `FixedPoint[f,expr]` hace lo mismo pero produce como salida únicamente el valor último producido.

El siguiente código muestra un ejemplo de uso:

```
In[2]:= f[z_] = z/2 + 1/z;
          FixedPointList[f, 0.75 + 0.1 I, SameTest -> (Abs[#1 - #2] < 10^-4 &)]
          FixedPoint[f, 0.75 + 0.1 I]

Out[2]= {0.75 + 0.1 I, 1.68504 - 0.124672 I, 1.43275 - 0.0186668 I,
          1.41421 - 0.000241452 I, 1.41421 - 2.45537*10^-10 I,
          1.41421 + 3.57849*10^-18 I}

Out[3]= 1.41421 + 0. I
```

Y a partir de la longitud de la lista que nos devuelve `FixedPointList`, es decir, con el sencillo comando `Length[FixedPoint[f,x]]` podemos medir la velocidad de convergencia de cada punto.

2.2. El método de Newton y cuencas de atracción

Gracias a la iteración y al estudio de la convergencia de las sucesiones que definen las órbitas, podemos definir muchos métodos numéricos que nos ayuden a resolver numéricamente problemas que de forma teórica o analítica serían difíciles de abordar. Entre estos está el conocido **método de Newton**. Este es un procedimiento iterativo que nos permite adivinar con cierta precisión donde se anula una función derivable (al menos en un entorno de la raíz), pudiendo así utilizarlo como herramienta para la resolución de ecuaciones. Existen varias y distintas versiones del método de Newton, siendo la más sencilla la que busca las raíces de una función real de variable real $f : \mathbb{R} \rightarrow \mathbb{R}$, conociéndose también en este caso como *método de Newton-Raphson*. Además, hay una generalización para aplicaciones $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$, la cual se apoya en la inversa del operador diferencial. Sin embargo, nosotros estamos trabajando con funciones complejas $f : \mathbb{C} \rightarrow \mathbb{C}$, que aunque \mathbb{C} y \mathbb{R}^2 sean isomorfos, la dinámica es distinta, pues si la función compleja es suficientemente buena, podemos usar su derivada $f'(z)$ y no el operador diferencial (véase [11]).

²Más información en la documentación oficial <https://reference.wolfram.com/language/ref/FixedPointList.html>

Recordemos que dada una ecuación $f(z) = 0$ para cierta función compleja y analítica $f : \mathbb{C} \rightarrow \mathbb{C}$, el método de Newton itera la función

$$N_f(z) = z - \frac{f(z)}{f'(z)} \quad (2.1)$$

comenzando por un punto z_0 cercano a la raíz. Es decir, calcula la sucesión $\{z_n\}$ definida como $z_{n+1} = N_f(z_n) \forall n \in \mathbb{N}$, la cual aplicando el teorema 2.1.1 podemos deducir que converge a un punto $a \in \mathbb{C}$ que verifica $f(a) = 0$ siempre que $f'(a) \neq 0$. Para mayor detalle acerca de la convergencia local ver [9, Capítulo 7] y [10, Sección 5.4].

Sin embargo, en muchas ecuaciones, comenzando por las polinómicas de grado mayor que 1, existen varias soluciones distintas, pero el método de Newton converge sólo a una de ellas, dependiendo de qué z_0 fijemos. Nuestro objetivo ahora se sitúa en discernir, para cada punto z_0 del plano, a qué solución de la ecuación $f(z) = 0$ converge la sucesión $\{z_n\}$ dada por el método de Newton utilizando a z_0 como semilla. En las siguientes secciones veremos algunos ejemplos de esta distinción y su utilidad, llegando a las primeras imágenes fractales generadas por iteración.

Ejemplo 2.2.1. Consideramos la función compleja $f : \mathbb{C} \rightarrow \mathbb{C}$ dada por $f(z) = z^2 - 1$, la cual tiene dos raíces: 1 y -1 , las dos raíces cuadradas de 1. Una forma sencilla de comprobar a qué raíz converge cada sucesión utilizando como semilla cada $z_0 \in \mathbb{C}$ es asociando un color a cada punto del plano dependiendo de la raíz a la que converja y pidiendo a *Mathematica* que coloree el plano complejo siguiendo este criterio.

```
In[4]:= iteracionN = #2 - #1[#2]/Derivative[1][#1][#2] &;
newtonArgumento =
Compile[{{z, _Complex}},
Arg[FixedPoint[iteracionNR[f, #] &, z, 50]]];
DensityPlot[newtonArgumento[x + I*y], {x, -2, 2}, {y, -2, 2},
PlotPoints -> 100, Mesh -> False, ColorFunction -> "Rainbow"]
```

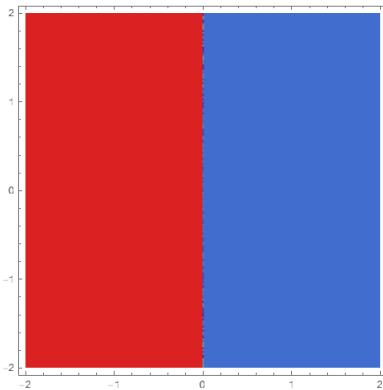


Imagen 2.2: Cuenca de atracción de $f(z) = z^2 - 1$.

Producido como resultado la imagen 2.2. La forma de deducir a qué raíz converge cada sucesión consiste en evaluar los argumentos de los valores a los que se acerca la misma. Como podemos ver, y teniendo en cuenta que la imagen solo

grafica el intervalo $[-2, 2] \times [-2, 2]$ y un número finito de puntos³ la sucesión cuya semilla es un punto perteneciente al semiplano abierto de la derecha converge a la raíz 1. Por otro lado, si la sucesión comienza con un complejo del semiplano abierto de la izquierda, entonces esta converge a la raíz -1 . Apoyándonos en este ejemplo, definimos el siguiente concepto.

Definición 2.2.1 (Cuenca de atracción). Definimos como *cuenca de atracción* de una raíz $a \in \mathbb{C}$ de una función compleja $f : \mathbb{C} \rightarrow \mathbb{C}$ (i.e. $f(a) = 0$), y denotamos como $A(a)$ al conjunto de puntos $z_0 \in \mathbb{C}$ tales que la sucesión $\{z_n\}$ dada por $z_{n+1} = N_f(z_n)$ converge a a utilizando a z_0 como primer valor de la sucesión.

En muchas de las funciones que trataremos ocurre que existen distintas cuencas de atracción y la distinción sobre a cuál de ellas pertenece cada punto del plano complejo es la que nos permitirá graficar imágenes fractales. Si aplicamos el mismo procedimiento antes descrito en el ejemplo 2.2.1 con otras funciones encontramos las imágenes 2.3.

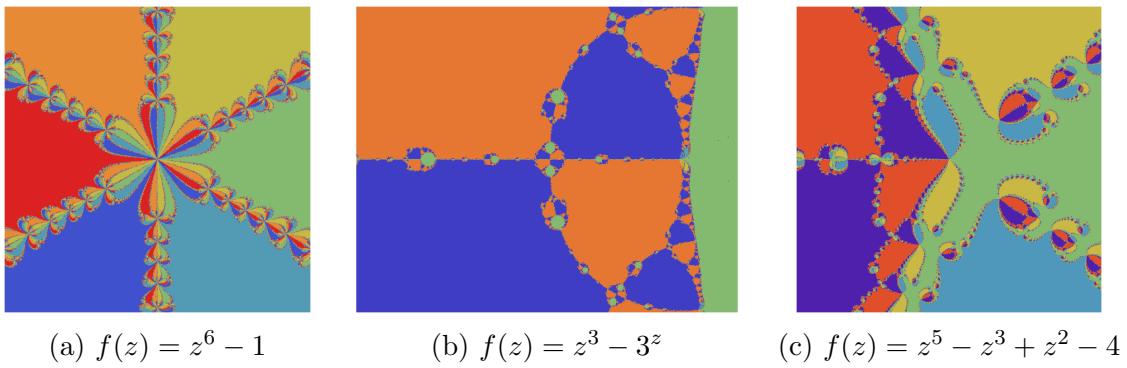


Imagen 2.3: Cuencas de atracción de distintas funciones coloreadas.

En estas imágenes podemos ahora sí ver los primeros ejemplos de estructuras fractales producidos por la iteración. Observemos que las sucesiones de dos puntos muy próximos pueden converger a raíces distintas, por lo que este es un ejemplo de *caos matemático*: pequeñas variaciones en las condiciones iniciales conducen a comportamientos muy diferentes. Además, en lugar de colorear el plano según la raíz a la que converja la sucesión también podemos fijarnos en la velocidad de convergencia. Esto es, asociar a cada punto del plano un color dependiendo del número de iteraciones necesarias para converger a la raíz. En este sentido, podemos revisitar las imágenes 2.3 y en función de la velocidad de convergencia de las sucesiones utilizar un color distinto. El código necesario es el siguiente y los resultados se pueden observar en las imágenes 2.4:

```
In[5]:= newtonVelocidad = Compile[{{z, _Complex}},
Length[FixedPointList[iteracionN[f, #] &, z, 50]]];

DensityPlot[newtonVelocidad[x + I*y], {x, -2, 2}, {y, -2, 2},
PlotPoints -> 200, Mesh -> False, ColorFunction -> GrayLevel]
```

donde se puede cambiar el valor de f , la región del plano $\{x, -2, 2\}, \{y, -2, 2\}$ o el valor del argumento `ColorFunction` para obtener distintas imágenes.

³El número de puntos que se grafica se puede parametrizar con el argumento opcional “`PlotPoints`”

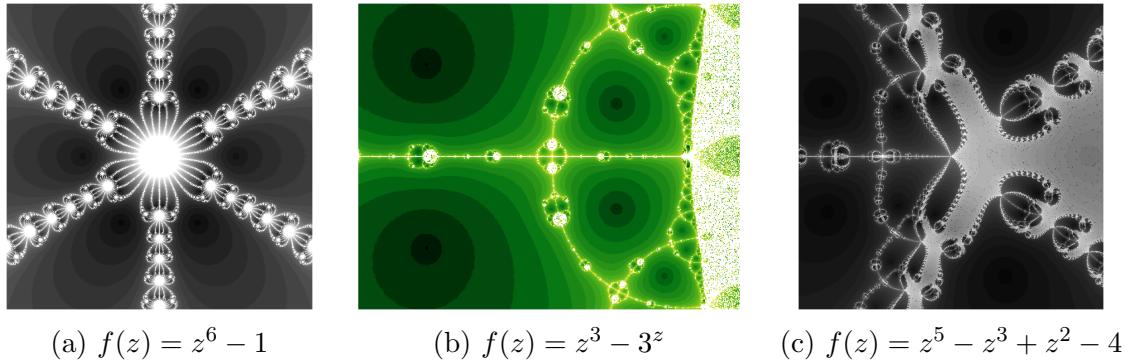


Imagen 2.4: Evaluación de la velocidad de convergencia en cada punto

Podemos jugar con estos parámetros como son la función a representar f , la región del plano que queremos visualizar $\{x, -2, 2\}$, $\{y, -2, 2\}$ o el valor del argumento `ColorFunction` para obtener distintas imágenes de diferentes colores.

2.2.1. Autosimilaridad

Consideramos ahora la representación de las cuencas de atracción de la función compleja $f(z) = z^3 - 1$, que podemos ver en la imagen 2.5

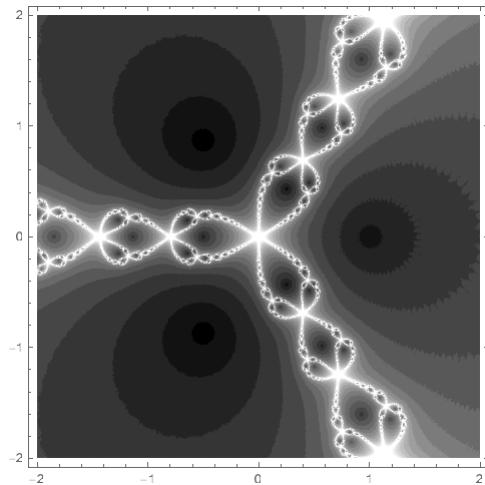


Imagen 2.5: Cuencas de atracción de $f(z) = z^3 - 1$

de las funciones “Plot”. A mayor valor mayor calidad de imagen y resolución, pero mayor tiempo de ejecución.

Fijémonos además que si hacemos *zoom* en ciertas partes de la imagen, es decir, representamos una región más pequeña, encontramos estructuras que son iguales independientemente del zoom que se aplique. En la figura 2.6 podemos ver distintos detalles de la figura 2.5, cada una es una ampliación de la anterior, y podemos ver como esa estructura se repite en todas las escalas.

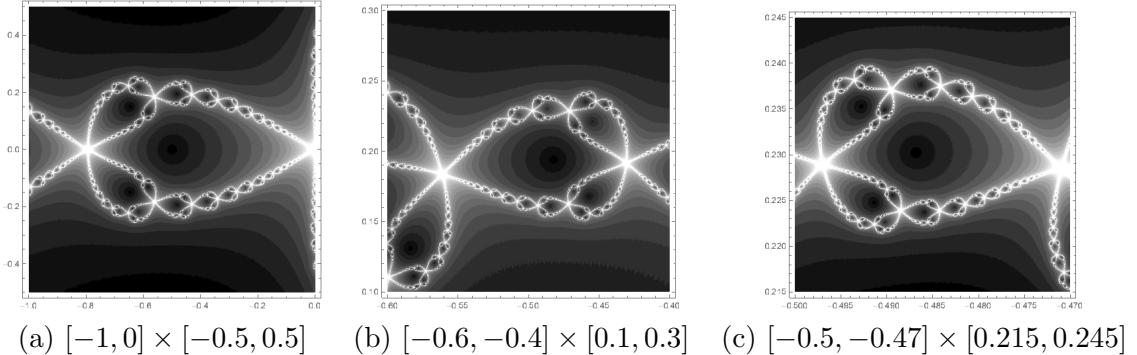


Imagen 2.6: Diferentes regiones ampliadas de la figura 2.5

Este aspecto es el que realmente nos define la naturaleza fractal de las cuencas de atracción, pues aunque las imágenes no son objetos totalmente autosimilares en el sentido de la definición 1.0.1 si que contienen regiones que sí lo son, como es el caso que acabamos de describir.

CAPÍTULO 3

CONJUNTOS DE JULIA Y MANDELBROT

Terminamos el capítulo 2 fijándonos en la autosimilaridad de las imágenes que nos proporcionaba la aplicación del método de Newton en ciertas funciones complejas para deducir las cuencas de atracción de las distintas funciones. Pudimos comprobar que a pesar de ser imágenes que no son totalmente autosimilares, por lo que no son fractales en el sentido estricto, contienen fragmentos que sí lo son. Este mismo hecho se produce en los conjuntos de Julia y en el conjunto de Mandelbrot. Podemos observar este hecho en las imágenes 3.1, que son nuestros primeros dos ejemplos de conjuntos de Julia.

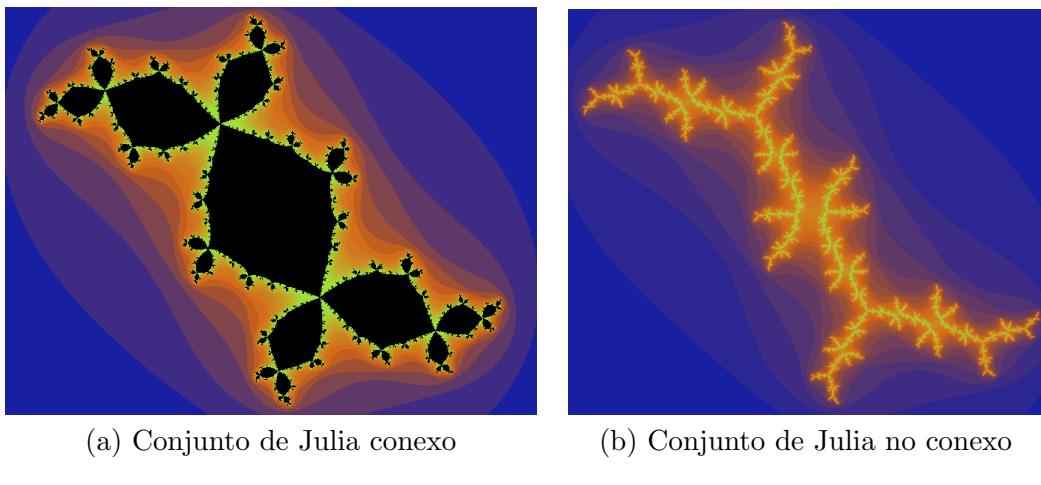
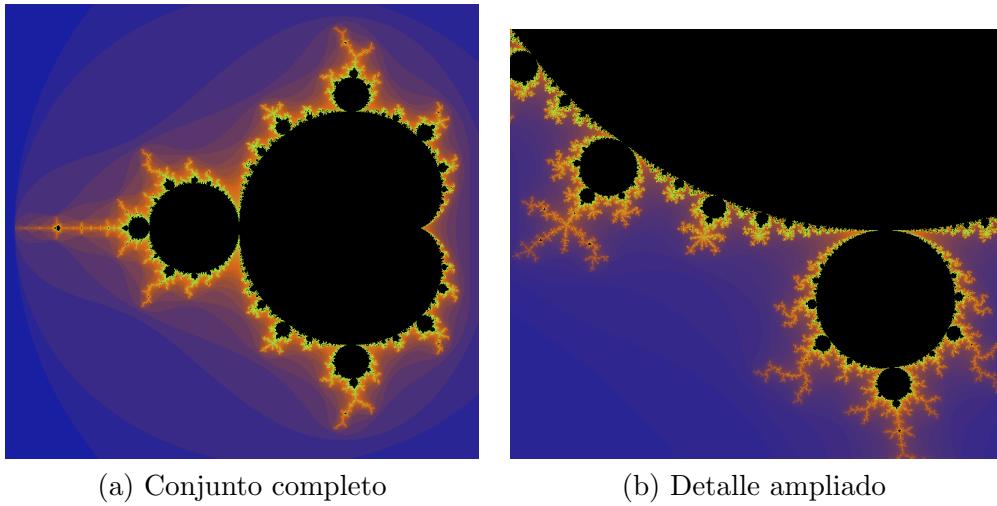


Imagen 3.1: Primeras imágenes de conjuntos de Julia

Una notable diferencia entre las imágenes 3.1 se encuentra en que mientras en la imagen (a) se puede percibir cierta conexión en el conjunto, esta desaparece en el caso de la imagen (b). Precisamente en esa distinción reside la génesis del conocido *conjunto de Mandelbrot*, que podemos también ver por primera vez, junto con algunas de sus autosimilaridades, en las imágenes 3.2.

En este capítulo aprenderemos qué elementos componen estos conjuntos, y cómo llegar a visualizar estas imágenes tan llamativas.



(a) Conjunto completo

(b) Detalle ampliado

Imagen 3.2: Primeras imágenes del conjunto de Mandelbrot

3.1. Iteración convergente y no convergente

Recordamos que en el capítulo 2, a partir de una función analítica $f : \mathbb{C} \rightarrow \mathbb{C}$, se le aplicaba una transformación $N_f(z)$ de forma que en muchos casos la iteración de dicha función era convergente independientemente del término $z_0 \in \mathbb{C}$ inicial. Sin embargo recordemos que la iteración de una función cualquiera f no siempre es convergente, como pudimos comprobar en el ejemplo situado al comienzo de la sección 2.1.1, en el que comprobamos que las iteradas de la función $f(z) = z^2$ divergen siempre que $|z_0| > 1$, convergen a 0 si $|z_0| < 1$ y quedan encerradas en S^1 en caso de que $|z_0| = 1$. Otro posible comportamiento es el cíclico, como el que tienen las iteradas de la función $g(z) = z \cdot i$ en $z_0 = 1$, que si nos fijamos, $O_g(1) = \{1, i, -1, -i, 1, \dots\}$.

Un último caso de posible comportamiento de una órbita son las órbitas caóticas, en las cuales no se percibe ningún patrón y además es muy sensible a las condiciones iniciales, fíjemonos en lo que ocurre en el caso de la función $h(z) = z^2 - 1.9$ si miramos sus órbitas en $z_0 = 0, 0.1$:

```
In[6]:= h[z_] := z^2 - 1.9;
NestList[h, 0.1, 20]
NestList[h, 0.0, 20]

Out[6]= {0.1, -1.89, 1.6721, 0.895918, -1.09733, -0.695866,
-1.41577, 0.104404, -1.8891, 1.6687, 0.884552, -1.11757,
-0.651043, -1.47614, 0.279, -1.82216, 1.42026, 0.117148,
-1.88628, 1.65804, 0.849092}

Out[7]= {0., -1.9, 1.71, 1.0241, -0.851219, -1.17543, -0.518374,
-1.63129, 0.761102, -1.32072, -0.155688, -1.87576, 1.61848,
0.719477, -1.38235, 0.0109007, -1.89988, 1.70955, 1.02256,
-0.854379, -1.17004}
```

No se observa ningún patrón de convergencia y además, a pesar de ser semillas muy cercanas, las órbitas son muy diferentes.

La dicotomía existente entre qué z_0 iniciales hacen que las iteradas de una función converga o no, restringida a cierta familia de funciones es la que define a los

distintos conjuntos de Julia. Presentamos por tanto, para cada $c \in \mathbb{C}$, la familia de funciones

$$P_c(z) = z^2 + c \quad \forall z \in \mathbb{C} \quad (3.1)$$

Nuestro objetivo es entonces clasificar para qué $z_0 \in \mathbb{C}$, las iteradas $\{P_c^n(z_0)\}$ convergen, divergen, ciclan, o tienen posiblemente un comportamiento caótico.

3.2. Conjuntos de Julia

Sin dejar de tener en cuenta la familia de funciones $\{P_c(z)\}_{c \in \mathbb{C}}$ introducimos la siguiente definición.

Definición 3.2.1. Dado un número complejo $c \in \mathbb{C}$ fijo consideramos $P_c(z) = z^2 + c$. Entonces:

- Se denomina **conjunto de puntos de escape**, y denotamos como E_c al conjunto de puntos cuyas iteradas divergen, es decir:

$$E_c = \{z_0 \in \mathbb{C} : \{|P_c^n(z_0)|\} \rightarrow \infty\}$$

- Se denomina **conjunto de puntos prisioneros**, y denotamos como P_c al conjunto de puntos cuyas iteradas no divergen, por lo que es el complemento de E_c .

A partir de estas dos definiciones, que insisten en clasificar los puntos del plano complejo entre de escape o prisioneros según su órbita, podemos introducir la definición que esperábamos.

He buscado información acerca de la duda que le planteé y ya tendría material para completar formalmente este apartado. Sin embargo, creo que es un poco técnica y realmente no termino de entenderla bien. La realidad es que se define el conjunto de Julia como el conjunto de puntos en los que la dinámica es caótica y por una serie de resultados acaba siendo equivalente a estas definiciones más sencillas. Pero para llegar a eso hay que meter artillería un poco pesada y quizás un poco fuera de contexto. Podemos verlo juntos y en función decidir.

Definición 3.2.2 (Conjunto de Julia). Dado un número $c \in \mathbb{C}$, se define su **conjunto de Julia**, y se denota como \mathcal{J}_c a la frontera de E_c . Es en este conjunto donde las iteradas tienen un comportamiento caótico. Se denomina **conjunto de Fatou** al complemento del conjunto de Julia.

Ejemplo 3.2.1. En el caso $c = 0$, es decir, $P_0(z) = z^2$, sabemos ya que $\mathcal{J}_c = S^1$, pues precisamente es S^1 la frontera entre los puntos cuyas iteradas divergen o convergen a 0.

Observación 3.2.1. Fijémonos por tanto que hay tantos conjuntos de Julia como números complejos, al poder asociar a cada número complejo un conjunto de puntos prisioneros, de escape, y por tanto un conjunto de Julia.

3.2.1. Representación gráfica de los conjuntos de Julia

Tenemos entonces una definición de los conjuntos de Julia, pero aparentemente está muy alejada de las imágenes 3.1 que presentamos en la introducción. La forma de llegar a ellas es similar a la que utilizamos para graficar las imágenes

que utilizaban la velocidad de convergencia en el capítulo 2. Sin embargo, y al contrario que al utilizar el método de Newton, ahora no tenemos ningún tipo de convergencia asegurada, por lo que el método de aplicar iteradas hasta encontrar un patrón no es el más correcto. Debemos por tanto encontrar una manera eficiente de clasificar cada punto del plano como prisionero o de escape.

Para ello podemos fijarnos en que la operación de elevar z_n al cuadrado prima sobre la de sumar una constante c siempre que el módulo $|z_n|$ sea ‘suficientemente grande’. Procedemos entonces a enunciar el siguiente resultado:

Teorema 3.2.1. *Dado un $c \in \mathbb{C}$, consideramos la función $P_c(z) = z^2 + c$. Si un número $z_0 \in \mathbb{C}$ verifica que $|z_0| > \max\{|c|, 2\}$, entonces z_0 es un punto de escape. Al número $e_c = \max\{|c|, 2\}$ se le denomina número de escape.*

*Demuestra*ión. Supongamos que $|z_0| > e_c = \max\{|c|, 2\}$, por tanto necesariamente debe existir un número $\varepsilon > 0$ tal que $|z_0| = 2 + \varepsilon$. Aplicamos entonces la cadena de desigualdades siguiente:

$$\begin{aligned} |P_c(z_0)| = |z_0^2 + c| &\geq |z_0^2| - |c| \quad (\text{Propiedades del m\'odulo}) \\ &= |z_0|^2 - |c| \\ &\geq |z_0|^2 - |z_0| \quad (\text{Porque } |z_0| > |c|) \\ &= (|z_0| - 1)|z_0| \\ &= (1 + \varepsilon)|z_0| \end{aligned}$$

Por tanto tenemos que $|P_c(z_0)| \geq (1 + \varepsilon)|z_0|$, por lo que en cada iteración el módulo aumenta al menos $1 + \varepsilon$ unidades, que es mayor que uno, es decir, $|P_c^k(z_0)| \geq (1 + \varepsilon)^k|z_0|$, por lo que la sucesión diverge y z_0 es un punto de escape. \square

Podemos aplicar este teorema para programar un algoritmo que grafique conjuntos de Julia. Para ello, fijamos un número $M \in \mathbb{N}$ que será el máximo de iteraciones que se aplicarán en cada punto antes de decidir si el punto es prisionero o de escape. Si pasadas esas M iteraciones el módulo de z_0 no ha alcanzado el número de escape e_c entonces consideramos que z_0 es un punto prisionero. En caso contrario, en el momento que se alcance el número de escape cesarán las iteraciones y se etiquetará el punto como prisionero. El valor de M puede ser alto si queremos aumentar la precisión a cambio de mayor tiempo de c\'omputo, y viceversa. Es posible que algunos de los puntos sean de escape pero alcancen e_c despu\'es de las M iteraciones, pero tomando un valor suficientemente alto el resultado es prácticamente el mismo.

Para graficar un conjunto de Julia podemos asignar un color fijo a los puntos prisioneros y a los puntos de escape asignarle otro en función de las iteraciones necesarias antes de alcanzar el número de escape. Por ejemplo, si queremos representar en *Mathematica* el conjunto de la figura 3.1 (a), que era $\mathcal{J}_{-0.12+0.75i}$ utilizaríamos el siguiente código:

```
In[8]:= M = 50;
Julia[z_, c_] := Length[FixedPointList[#^2 + c &, z, M,
SameTest -> (Abs[#] > Max[2.0, Abs[c]] &)];
DensityPlot[
Julia[x + I y, -0.12 + 0.75 I], {x, -1.5, 1.5},
{y, -1.25, 1.25}, PlotPoints->200, AspectRatio->Automatic,
ColorFunction -> (If[# >= 1, RGBColor[0, 0, 0], Hue[#]] &)]
```

Fijémonos en que hemos utilizado el argumento `SameTest` para especificar cuando dejar de iterar, especificando que se haga cuando se alcance e_c . Hemos utilizado también $M = 50$ como máximo número de iteraciones. Podemos variar el segundo argumento de la llamada a la función `Julia`, el rango de valores, el argumento `PlotPoints` o el número máximo de iteraciones M para graficar distintos conjuntos y detalles de los mismos. Algunos ejemplos de resultados de estos códigos se encuentran en la imagen 3.3.

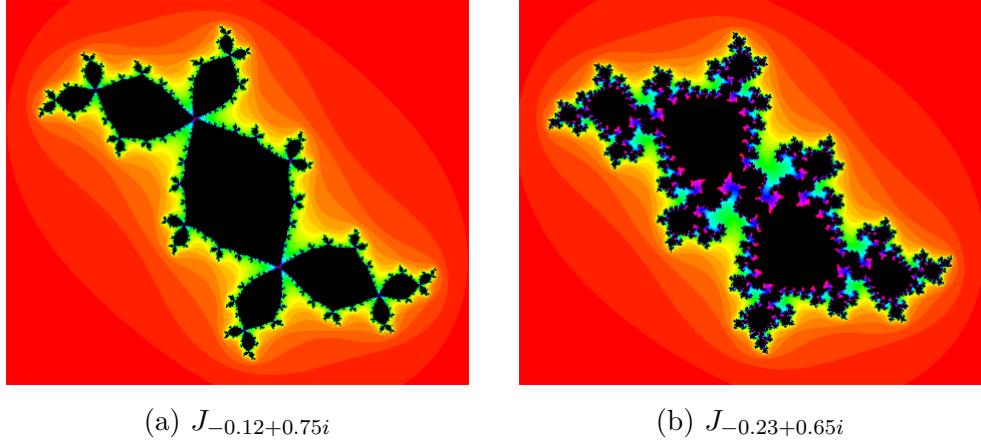


Imagen 3.3: Conjuntos de Julia graficados con Mathematica

Por sí solo Mathematica incluye una función `JuliaSetPlot[c]` que muestra una imagen del conjunto \mathcal{J}_c . Esta función permite modificar los mismos parámetros que el método que acabamos de programar por nuestra cuenta¹, véanse las imágenes 3.4.

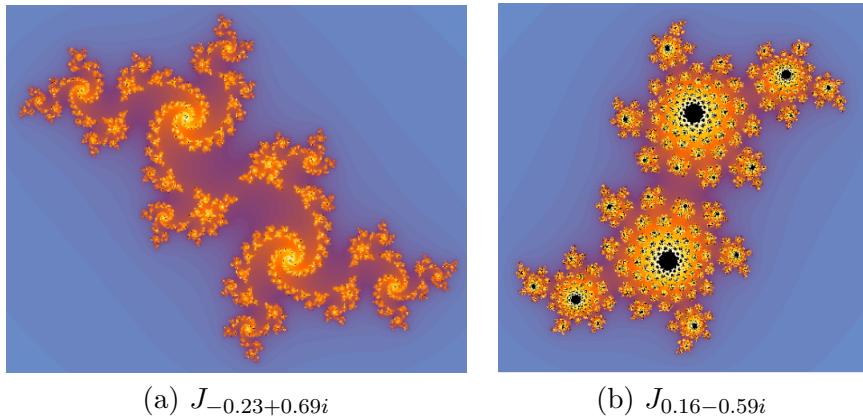


Imagen 3.4: Resultados de la orden ‘JuliaSetPlot’

3.3. Distinción entre conjuntos de Julia conexos y polvaredas

Ya en la introducción de este capítulo y tras presentar las imágenes 3.1 prestamos atención a un detalle que, ahora que tenemos más ejemplos e imágenes

¹Más información en la documentación oficial: <https://reference.wolfram.com/language/ref/JuliaSetPlot.html?q=JuliaSetPlot>

de distintos conjuntos de Julia, se vuelve más evidente. Mientras algunos conjuntos de Julia se presentan conexos, como los de las imágenes 3.3, otros parecen no ser conexos y estar formados por trozos más pequeños, como los de las imágenes 3.4. Cabe recordar que en realidad los conjuntos de Julia son la frontera entre los puntos que se representan en negro y los que se representan de colores.

Para saber qué conjuntos de Julia son conexos y cuales no *Gaston Julia y Pierre Fatou* demostraron en 1918 el siguiente teorema, cuya prueba se puede encontrar en [12, Theorem 9.5].

Teorema 3.3.1 (Teorema de Fatou-Julia). *Si \mathbb{P}_c contiene todos los puntos críticos de $P_c(z)$, entonces $\mathcal{J}_c = \partial\mathbb{P}_c$ es conexo. Si al menos un punto crítico de $P_c(z)$ pertenece a \mathbb{E}_c , entonces \mathcal{J}_c es isomorfo al conjunto de Cantor, es decir, tiene un conjunto no numerable de componentes conexas. En este último caso se conoce como «polvo de Fatou».*

En realidad el teorema en su versión original está enunciado para polinomios de grado mayor o igual que 2, entendiendo las extensiones naturales del conjunto de puntos de escape y prisioneros. Si nos restringimos entonces a la familia $\{P_c(z)\}_{c \in \mathbb{C}}$, es muy sencillo caracterizar qué conjuntos de Julia son conexos y cuales son polvaredas.

Corolario 3.3.1. *Dado $c \in \mathbb{C}$, el conjunto de Julia \mathcal{J}_c es conexo (resp. polvareda) si, y solo si, la sucesión de iteradas $\{P_c^n(0)\}$ no diverge (resp. diverge), es decir, si $0 \in \mathbb{P}_c$ (resp. $0 \in \mathbb{E}_c$).*

Demostración. Es claro que $P'_c(z) = (z^2 + c)' = 2z$, por lo que los puntos críticos de $P_c(z)$ se alcanzan cuando $z = 0$, por lo que aplicando el teorema 3.3.1 tenemos el resultado. \square

Sobre la dicotomía entre qué conjuntos de Julia son conexos y cuales son polvareda surge el conocido **conjunto de Mandelbrot**, el cual adelantamos que está formado por los números complejos c tales que su conjunto de Julia \mathcal{J}_c es conexo.

3.4. El conjunto de Mandelbrot

Como ya veníamos anunciando al final de la sección 3.3, el conjunto de Mandelbrot está formado por los $c \in \mathbb{C}$ tales que \mathcal{J}_c es conexo. La idea inicial de *Benoit Mandelbrot* para graficar el conjunto que denotaremos a partir de ahora como \mathcal{M} fue pintar de negro los puntos del plano cuyo conjunto de Julia fuese conexo y de blanco el resto. De entrada parece una tarea titánica graficar, para cada punto del plano (aunque realmente sería solo un subconjunto suficientemente representativo), su conjunto de Julia y decidir si éste es o no conexo, pues en algunos casos la decisión se torna muy complicada.

Afortunadamente, gracias al teorema 3.3.1 y al corolario 3.3.1 esta tarea se vuelve mucho más sencilla, tan solo habría que tomar las iteradas en el origen, es decir $\{P_c^n(0)\} = \{c, c^2 + c, (c^2 + c)^2 + c, \dots\}$ y decidir si la sucesión diverge o no mediante algún método similar al utilizado para graficar conjuntos de Julia.

Por resumir, de momento sabemos que:

$$\begin{aligned}\mathcal{M} &= \{c \in \mathbb{C} : \mathcal{J}_c \text{ es conexo}\} \\ &= \{c \in \mathbb{C} : 0 \in \mathbb{P}_c\} \\ &= \{c \in \mathbb{C} : \{P_c^n(0)\} \not\rightarrow \infty\}\end{aligned}$$

3.4.1. Representación gráfica del conjunto de Mandelbrot

Buscamos ahora obtener alguna figura similar a la imagen 3.2 a partir del conocimiento que tenemos de \mathcal{M} . Como ya viene siendo costumbre, la idea es dividir el plano en una cantidad finita de píxeles, asignando a cada uno un número complejo c y evaluar en cada uno si la órbita en $z = 0$ converge o diverge. Para tomar esta decisión nos podemos apoyar en el teorema 3.2.1 para probar este resultado.

Proposición 3.4.1. Dado un número complejo $c \in \mathbb{C}$, si $|c| > 2$ entonces la sucesión de iteradas $\{P_c^n(0)\}$ es divergente.

*Demuestra*ación. Consideramos la sucesión de iteradas $z_0 = 0, z_n = P_c^n(0)$. Partimos de que $|c| > 2$, por lo que buscamos encontrar un $m \in \mathbb{N}$ tal que $|z_m| > e_c = \max\{|c|, 2\} = |c|$.

Tenemos que $z_1 = P_c(0) = c$, pero $|z_1| = |c| \not> |c|$.

Si iteramos una vez más, $z_2 = P_c^2(0) = P_c(c) = c^2 + c$. Ayudándonos de una propiedad del módulo tenemos que:

$$\begin{aligned} |c^2 + c| &\geq ||c^2| - |c|| \\ &= ||c|^2 - |c|| \\ &= |c|^2 - |c| \quad (\text{Pues al ser } |c| > 2, |c|^2 > |c|) \\ &= (|c| - 1)|c| \end{aligned}$$

Y como $|c| > 2 \Leftrightarrow |c| - 1 > 1$, concluimos que

$$|z_2| = |c^2 + c| > |c| = \max\{|c|, 2\} = e_c$$

Por lo que aplicando el teorema 3.2.1, la sucesión $\{z_n\}$ es divergente. \square

Este resultado nos facilita mucho la elaboración de un algoritmo que grafique a \mathcal{M} , pues de entrada nos afirma que todo número complejo cuyo módulo sea superior a 2 no pertenece a \mathcal{M} . O dicho de otra forma,

$$\mathcal{M} \subseteq \{c \in \mathbb{C} : |c| \leq 2\} = \bar{D}(0, 2),$$

donde $D(z, r)$ denota el disco abierto de centro z y radio r mientras que $\bar{D}(z, r)$ denota el cierre del disco abierto, es decir, el disco cerrado de centro z y radio r .

Además, en el momento que una iterada se sitúe fuera de $\bar{D}(0, 2)$, podemos asegurar que esa sucesión va a diverger, por lo que podemos dejar de iterar. Para concluir que la sucesión no diverge, al igual que para graficar conjuntos de Julia, fijamos un número máximo de iteraciones $M \in \mathbb{N}$, de forma que si al calcular la M -ésima iterada el módulo del elemento $P_c^M(0)$ no ha excedido a 2, podemos considerar que el número c pertenece a \mathcal{M} . En conclusión, el algoritmo consiste en, para cada píxel identificado con un punto del plano complejo $c \in \mathbb{C}$, calcular sus iteradas hasta un máximo de M iteraciones, en caso de exceder en módulo a 2 guardamos el mínimo valor $m \in \mathbb{N}$ tal que $|P_c^m(0)| > 2$ y asignamos un color en función; en caso de no exceder a 2 asignar un color fijo.

El código en *Mathematica* utilizado es por tanto el siguiente, muy similar al utilizado para graficar conjuntos de Julia:

```
In[9]:= M = 100;
Mandelbrot = Compile[{{c, _Complex}},
```

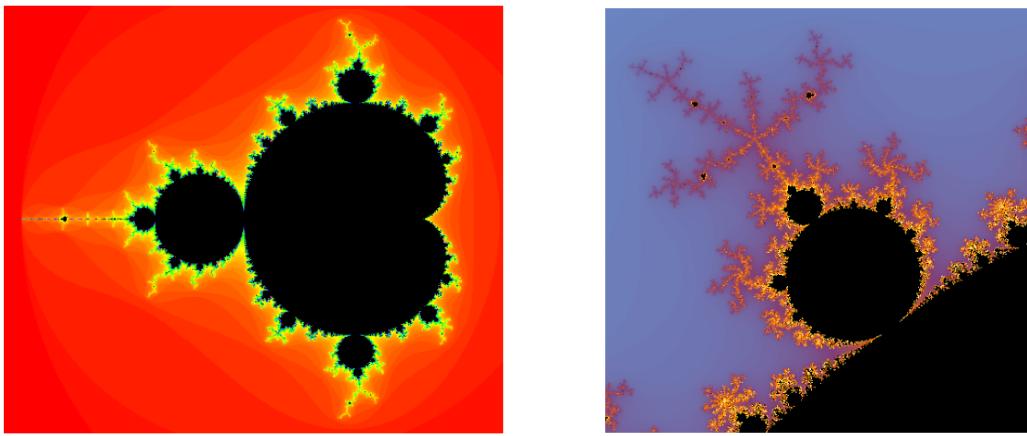
```

Length[FixedPointList[#^2 + c &, 0, M,
SameTest -> (Abs[#] > 2.0 &)]]];

DensityPlot[Mandelbrot[x + I y], {x, -2.1, .7}, {y, -1.2, 1.2},
Mesh -> False, Frame -> False, PlotPoints -> 200,
AspectRatio -> Automatic,
ColorFunction -> (If[# >= 1, Hue[0, 0, 0], Hue[#]] &)]

```

Al igual que en el caso de los conjuntos de Julia, *Mathematica* tiene una función preprogramada que grafica el conjunto de Mandelbrot, llamada `MandelbrotSetPlot`, la cual admite varios argumentos opcionales, como la región a graficar, el máximo de iteraciones, resolución, etc.².



(a) Salida del código *Mathematica* (b) Salida de la orden ‘`MandelbrotSetPlot`’

Imagen 3.5: Representación de \mathcal{M} y detalle en $[-0.65, -0.4] \times [0.47, 0.72]$

El conjunto de Mandelbrot es considerado por muchos como el objeto más complejo de la matemática. Además de la belleza que muestra por sí solo, los detalles de \mathcal{M} esconden bonitas estructuras: bulbos, valles, antenas, copias reducidas del propio \mathcal{M} ...

3.5. Autosimilaridad de los conjuntos de Julia y Mandelbrot

Ya en el comienzo de este capítulo mencionamos, y en las propias imágenes presentadas se puede comprobar, que los conjuntos de Julia y el conjunto de Mandelbrot no son objetos autosimilares, no al menos en el sentido de la definición 1.0.1. Sin embargo, sí contienen regiones y detalles que son autosimilares. En esta sección presentaremos algunas de las mismas.

3.5.1. Autosimilaridad en conjuntos de Julia

Recordemos el conjunto de Julia $\mathcal{J}_{-0.23+0.65}$, que podemos ver en la imagen 3.3 (b). En las imágenes 3.6 podemos ver distintos detalles, de forma que la primera es una ampliación de la original y las siguientes son cada una un ‘zoom’ de la anterior.

²Más información en la documentación oficial <https://reference.wolfram.com/language/ref/MandelbrotSetPlot.html>

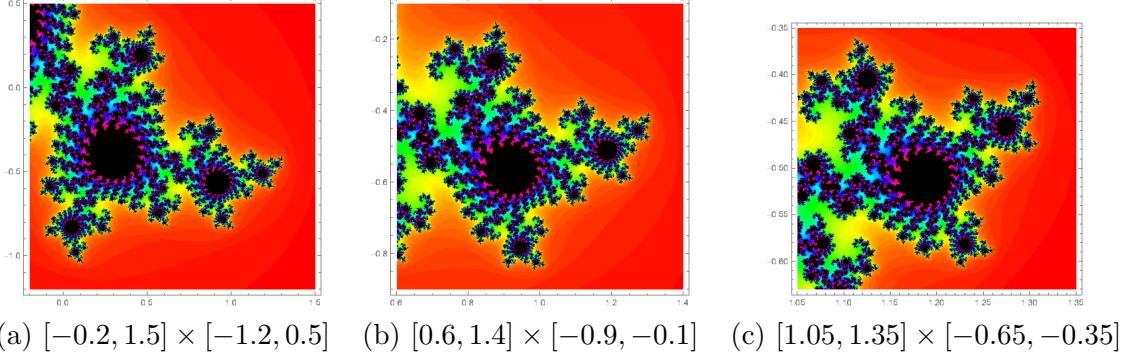


Imagen 3.6: Diferentes regiones ampliadas de la figura 3.3 (b)

Obsérvese cómo efectivamente las imágenes, salvo giro, son prácticamente iguales, pudiendo ver así una de las regiones autosimilares de $\mathcal{J}_{-0.23+0.65}$. En las imágenes 3.7 podemos ver algunas regiones ampliadas de ciertos conjuntos de Julia, pudiendo observar regiones autosimilares.

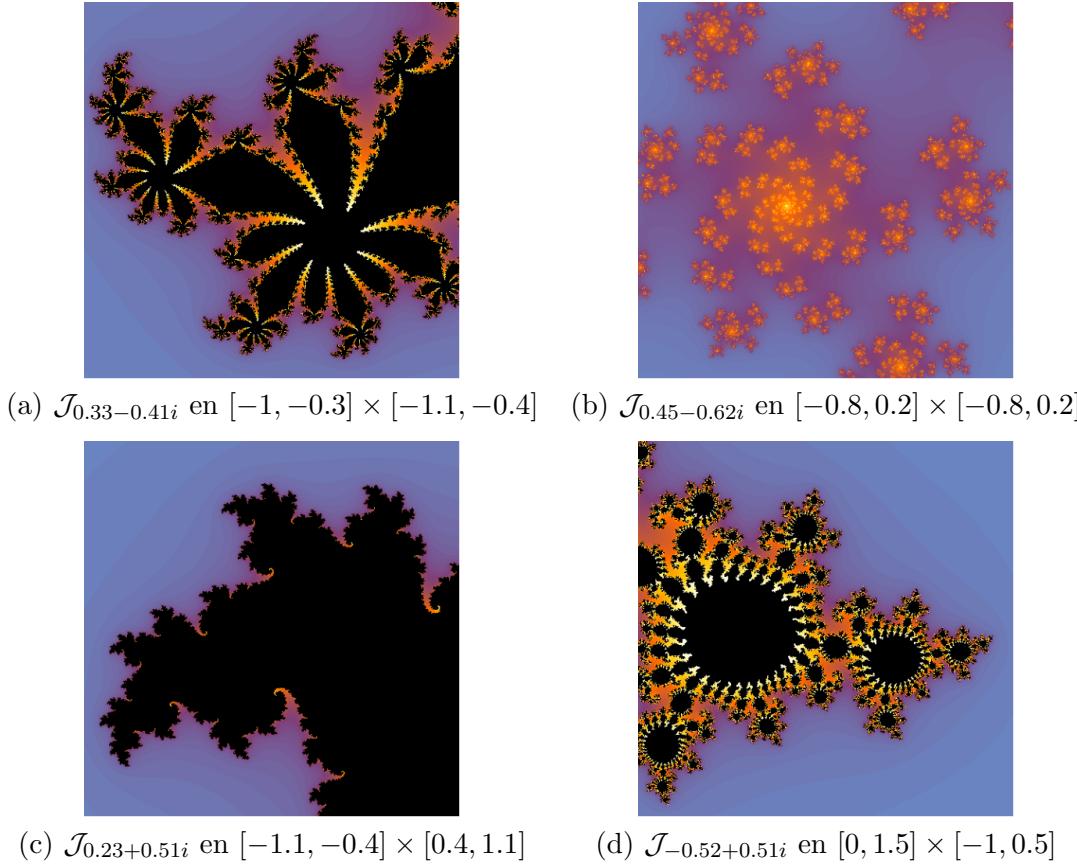


Imagen 3.7: Detalles autosimilares de algunos conjuntos de Julia

3.5.2. Autosimilaridad en el conjunto de Mandelbrot

El conjunto de Mandelbrot, el cual ya conocemos, está compuesto fundamentalmente de un cuerpo principal con forma de cardioide con gran cantidad de bulbos adosados al mismo (imagen 3.8 (a), (b) y (d)), siendo cada uno de estos autosimilar y terminando en una antena que se bifurca (imagen 3.8

(b) y (c)). Además, en el propio \mathcal{M} hay minúsculas copias de sí mismo, en las cuales se vuelve a repetir su estructura (imagen 3.8 (c)).

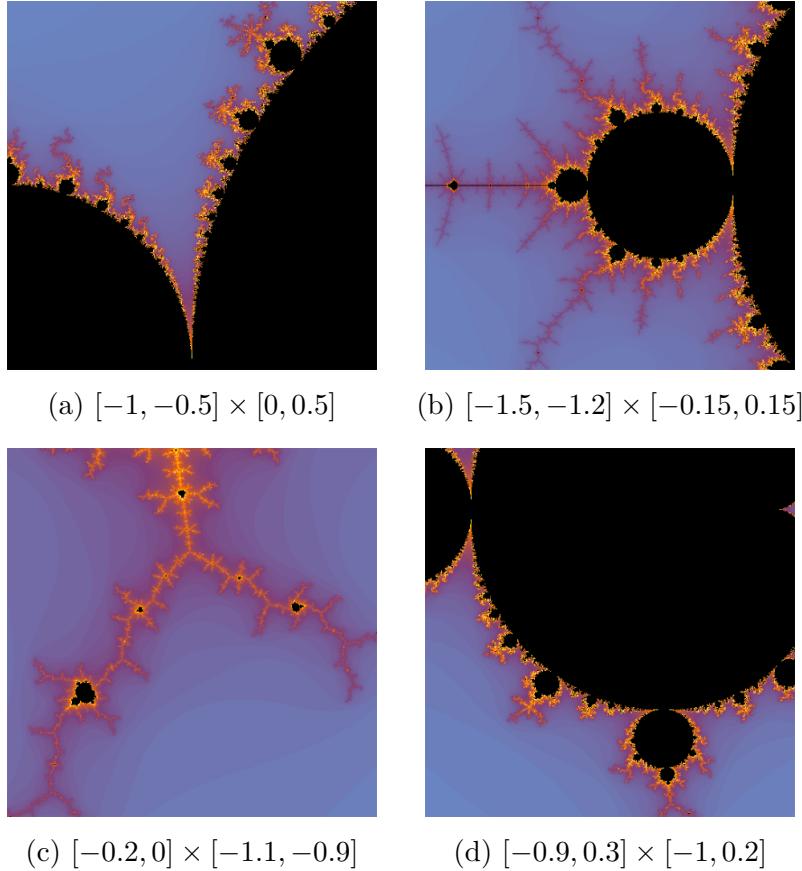


Imagen 3.8: Detalles autosimilares de \mathcal{M}

En la imagen 3.8 (b) podemos ver una representación del que se conoce como ‘bulbo principal’, pues es el más grande de todos los que están pegados al cuerpo principal. Podemos comprobar, observando los detalles que ofrecen las imágenes 3.9, que dicho bulbo muestra autosimilaridad.

3.6. Conjuntos de Julia y Mandelbrot generalizados

Hasta el momento todo lo explicado y todas las imágenes obtenidas se han basado en la familia de funciones $\{P_c(z)\}_{c \in \mathbb{C}} = \{z^2 + c\}_{c \in \mathbb{C}}$. Sin embargo, y como cabe esperar, la definición de los conjuntos de Julia como conjuntos frontera entre el conjunto de puntos prisioneros y de escape es completamente válido para las iteradas de cualquier función, o mejor dicho, para cualquier familia de funciones. En esta sección trataremos de extender los conocimientos obtenidos hasta ahora a otras funciones.

3.6.1. Familia $\{z^N + c\}_{c \in \mathbb{C}}$

La extensión más natural y la que nos proporcionará resultados más llamativos consiste en cambiar el exponente de la función polinómica $P_c(z)$, a la cual ahora denotaremos como $P_{c,N}(z) = z^N + c$, con $N \geq 2$ natural para enfatizar el exponente. Es fácil comprobar que el teorema 3.2.1 es igual de válido con esta

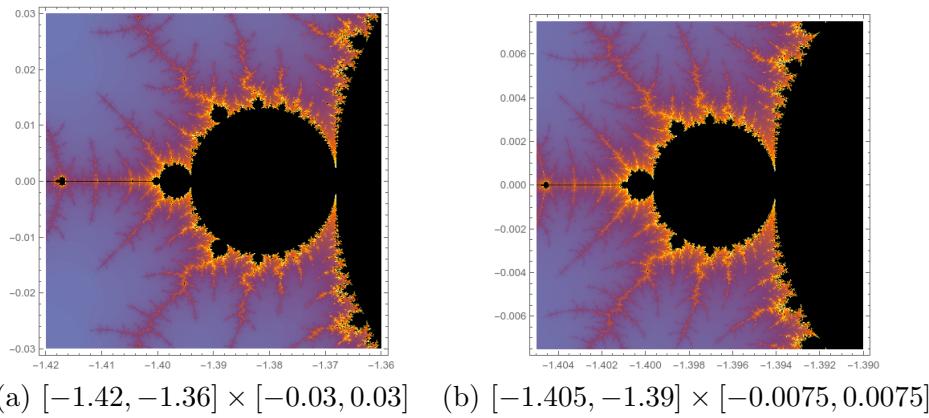


Imagen 3.9: Ampliaciones del bulbo principal de \mathcal{M}

familia de funciones independientemente del exponente, por lo que el algoritmo sigue siendo el mismo, solo que en la función `Julia` debemos cambiar el exponente. Además, la orden `JuliaSetPlot` ya presentada en la sección 3.2.1 admite la posibilidad de indicarle una función arbitraria. Es esta la forma en la que hemos graficado las imágenes 3.10.

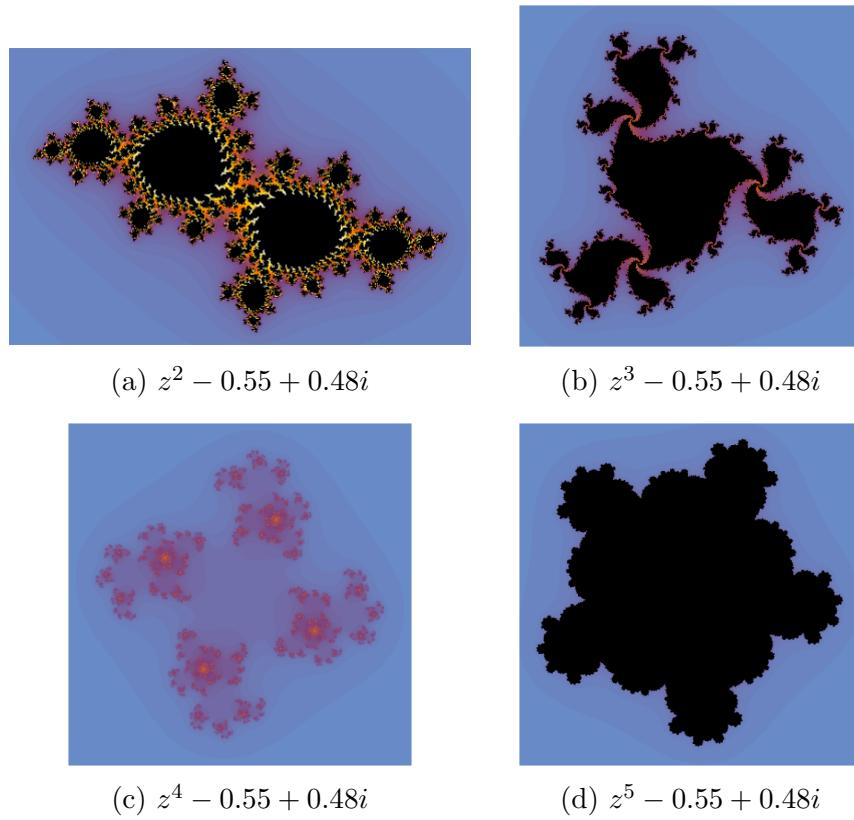


Imagen 3.10: Conjuntos de Julia con $P_{-0.55+0.48i,N}$ para distintos valores de N .

Fijémonos en que hemos fijado $c = -0.55 + 0.48i$ y hemos variado el exponente de la función polinómica. Llama la atención lo distintos que son los conjuntos entre sí cuando realmente el c fijado es el mismo en todos los casos.

Como ya comentamos en la sección 3.3, el teorema 3.3.1 (teorema de la dicotomía de Fatou-Julia) es válido para cualquier polinomio de grado mayor o

igual que 2, en particular es válido para la familia de funciones $\{P_{c,N}(z)\}_{c \in \mathbb{C}}$. Tenemos por tanto que de la misma forma es válido el corolario, por lo que la distinción entre conjuntos de Julia conexos y polvaredas se vuelve a basar en buscar la convergencia o divergencia de la sucesión $\{P_{c,N}^n(0)\}$.

De esta forma, podemos hablar por tanto de conjuntos de Mandelbrot generalizados como representación gráfica de qué puntos tienen un conjunto de Julia conexo o polvareda utilizando la función $P_{c,N}(z)$, el cual denotamos como \mathcal{M}_N . De la misma forma que el resultado 3.2.1 es válido en la generalización, la proposición 3.4.1 también lo es, por lo que podemos también reutilizar el código que empleamos para representar el conjunto de Mandelbrot con tan solo variar el exponente, ver imágenes 3.11. Por su parte, la función de *Mathematica* `MandelbrotSetPlot` admite un argumento `n` que representa el exponente de $P_{c,N}(z)$, a la hora de iterar.

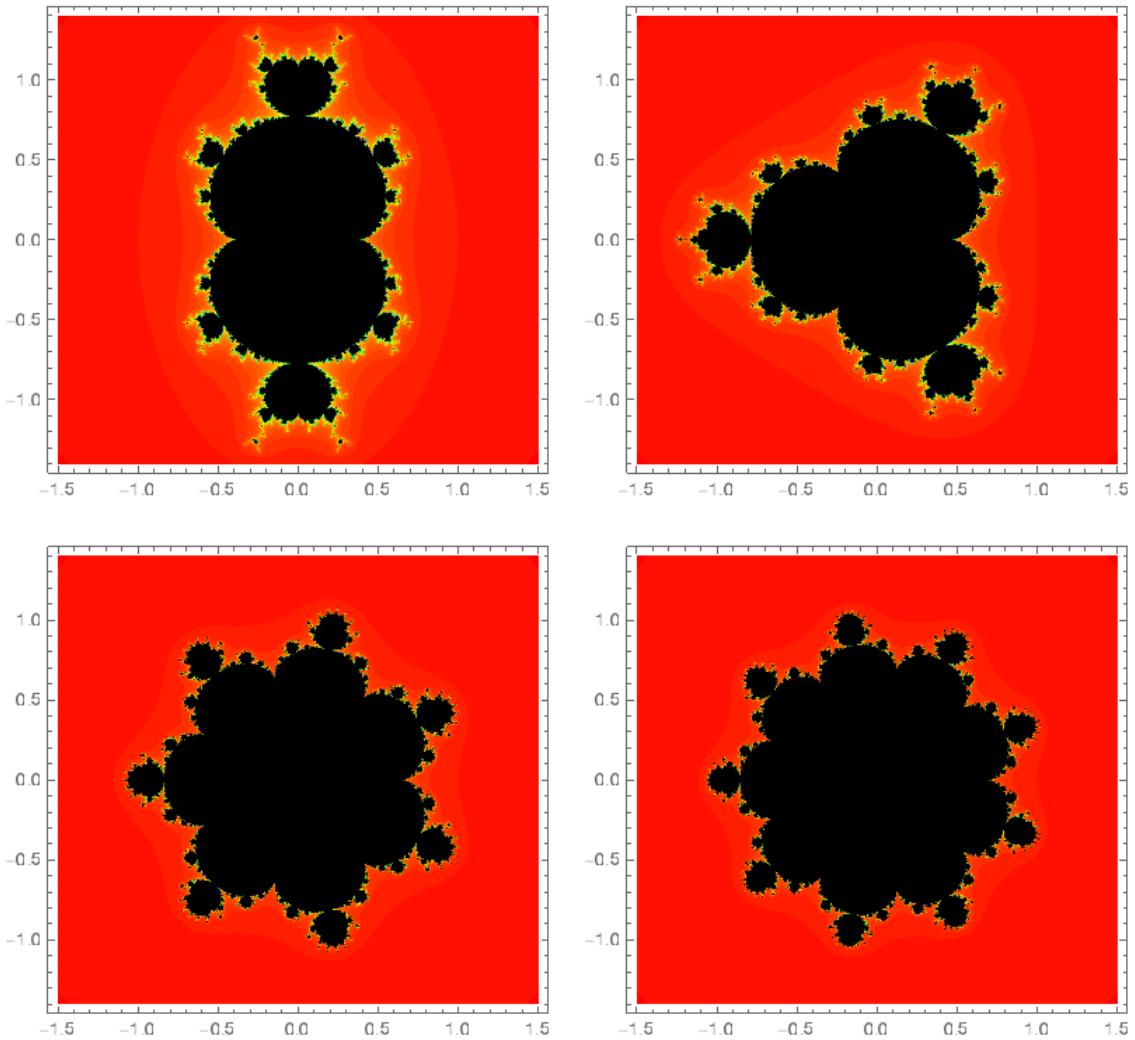


Imagen 3.11: Conjuntos de Mandelbrot \mathcal{M}_N para $N = 3, 4, 8, 10$

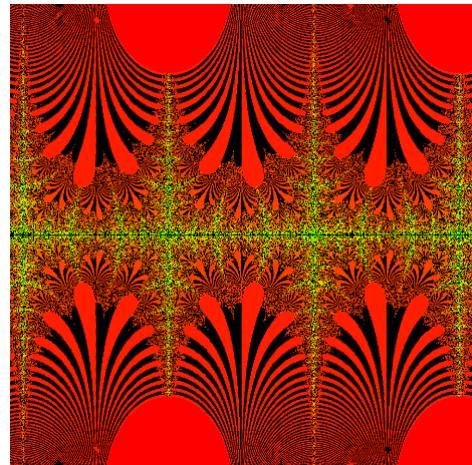
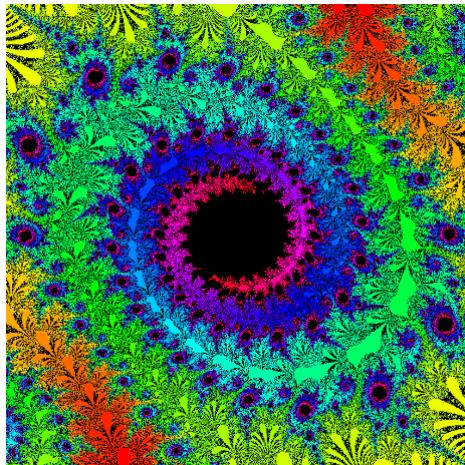
Obsérvese que a pesar de encontrar cierta similaridad en las formas de los distintos conjuntos de Mandelbrot generalizados, podemos encontrar puntos que para unos exponentes se encuentran dentro y en otros casos fuera de su respectivo \mathcal{M}_N . Este hecho explica lo ocurrido con $c = -0.55 + 0.48i$, que para ciertos exponentes el conjunto de Julia es conexo y para otros es polvareda.

Finalmente, invitamos al lector a visitar una web interactiva en la que poder visualizar tantos conjuntos de Julia y Mandelbrot estándar y generalizados como desee. La web forma parte del proyecto y su url es [https://jantoniov.github.io/Geometria-Fractal/\(WIP\)](https://jantoniov.github.io/Geometria-Fractal/(WIP)).

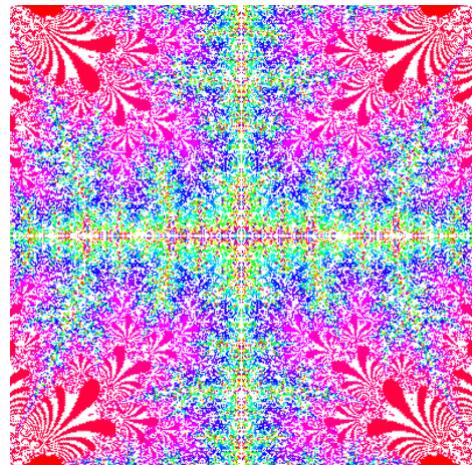
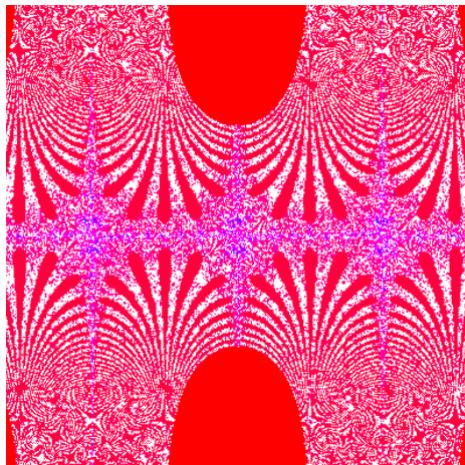
3.6.2. Conjuntos de Julia con funciones no polinómicas

Además de todos los conjuntos de Julia ya tratados y visualizados, utilizando otras funciones no polinómicas como pueden ser las trigonométricas y las exponenciales (complejas) también se pueden graficar algunos conjuntos con formas distintas a las ya vistas y con hermosas propiedades. A continuación mostraremos tan solo algunos ejemplos:

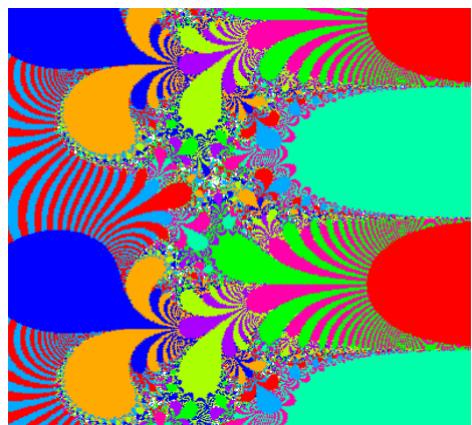
- Con la familia $f_c(z) = c \cdot \sin(z)$:



- Con la familia $f_c(z) = c \cdot \cos(z)$:



- Con la familia $f_c(z) = c \cdot e^z$:



CAPÍTULO 4

INTRODUCCIÓN A LAS HERRAMIENTAS DE RENDERIZACIÓN

Durante los capítulos anteriores hemos tratado los fractales desde un punto de vista teórico y matemático, ayudándonos del software *Mathematica* para visualizar imágenes de naturaleza fractal, desde los primeros ejemplos clásicos como el triángulo de Sierpinski o el copo de Koch (imágenes 1.3 y 1.7), representaciones de cuencas de atracción en el capítulo 2, conjuntos de Julia y Mandelbrot en el capítulo 3 y atractores de sistemas de funciones iteradas.

Queda claro que *Mathematica* es un software de cálculo muy útil, pero también es lento, ya que realmente no está orientado a el renderizado de imágenes. En este sentido, obsérvese que sólo hemos utilizado *Mathematica* para visualizar fractales 2D, que se presentan como subconjuntos del plano euclídeo \mathbb{R}^2 o coloreando el plano complejo \mathbb{C} . Esto se debe no solo a la simplicidad algorítmica que nos proporciona limitar los razonamientos a 2 dimensiones, sino a que el software es realmente lento en 3 dimensiones.

Sin embargo, y como cabría esperar, existen herramientas que nos ayudan a visualizar las directivas geométricas que nosotros mismos queramos, y además podremos hacerlo a tiempo real. En nuestro caso, utilizaremos la librería **WebGL**, que es una API de JavaScript para el renderizado de gráficos 2D y 3D dentro de cualquier navegador web. Precisamente este último aspecto es el que nos permitirá desplegar nuestro proyecto en una página web interactiva y accesible desde cualquier dispositivo cuya GPU se lo permita¹.

WebGL permite que las aplicaciones web utilicen una API basada en OpenGL ES 2.0 para renderizar imágenes 3D en un elemento `<canvas>` de HTML en los navegadores y sistemas que lo soporten sin necesidad de plug-ins. El principal defecto de herramientas como WebGL es la dificultad a la hora de querer comenzar a visualizar imágenes, pues tiene varios componentes complejos de enlazar en un principio. A continuación explicaremos el flujo de trabajo que utiliza WebGL y cómo podemos comenzar a utilizar la herramienta para visualizar fractales.

¹Esto se puede comprobar gracias a páginas dedicadas a ello como esta página de testeo WebGL o WebGL Report

4.1. Componentes de WebGL

A grandes rasgos, los programas que utilizan WebGL se componen de código de JavaScript que interactúa con la propia biblioteca junto con código GLSL que se ejecuta en la GPU. Para dar nuestros primeros pasos con WebGL debemos, en nuestro documento HTML, utilizar un elemento `<canvas>` al cual especificamos sus dimensiones en píxeles. En el siguiente ejemplo, el canvas sería cuadrado de 720×720 píxeles. Evidentemente, a mayor número de píxeles mayor resolución, pero también mayor tiempo de cómputo.

```
1 <main>
2   <canvas id="glCanvas" width="720" height="720"></canvas>
3 </main>
```

4.1.1. Contexto de WebGL

Por su parte, en JavaScript podemos acceder mediante el DOM al elemento `<canvas>` y extraer un objeto que será lo que a partir de ahora denominemos **contexto de WebGL** (`WebGLRenderingContext`). Este objeto nos proporciona una interfaz a un contexto de OpenGL ES 2.0 para dibujar en la superficie del canvas, de forma que se pueden invocar muchas de las funciones utilizadas en OpenGL. La forma de extraer este contexto es mediante la función `getContext`:

```
1 const canvas = document.querySelector("#glCanvas");
2 // Initialize the GL context
3 const gl = canvas.getContext("webgl2");
4
5 // Only continue if WebGL is available and working
6 if (gl === null) {
7   alert("Unable to initialize WebGL. Your browser or machine
8       may not support it.");
9 }
```

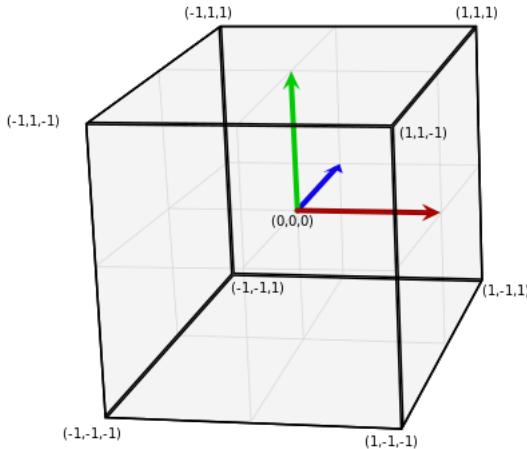
En caso de éxito, que se tiene en la mayoría de las ocasiones, debemos conservar este objeto, pues será necesario para utilizar la gran mayoría de directivas que implementa WebGL. Los siguientes elementos son los auténticos encargados de los objetos que se renderizan en el canvas.

4.1.2. El programa *Shader*

El ‘*shader*’ es un programa escrito en el llamado OpenGL ES Shading Language, más comúnmente conocido como **GLSL**, que a partir de la información sobre los vértices que forman una figura genera un color para cada píxel, para así dibujar dicha figura en el canvas. Hay dos tipos de *shader*: el **vertex shader** y el **fragment shader**. Ambos se escriben en GLSL, de forma que se le especifica el código GLSL a WebGL y este se ejecuta en la GPU. A continuación se explicarán las principales diferencias entre estos dos componentes.

El *vertex shader* se ejecuta una vez por cada vértice de la figura, su misión es transformar la coordenada de mundo de dicho vértice en coordenadas normalizadas en el intervalo $[-1, 1]$, rango utilizado por WebGL en su clip space. Tras realizar estos cálculos y ajustes, se almacena el valor de salida en la variable `gl_Position`. También podemos utilizar el vertex shader para otros cometidos como calcular la coordenada de textura de un objeto, calcular la normal a un objeto en dicho

vértice para posteriormente aplicar algún modelo de iluminación, o cualquier otro procesado que podamos hacer en un vértice con la idea de posteriormente pasarlo dicho valor al *fragment shader*.



Clipspace

Imagen 4.1: *Clip space de WebGL*

Por su parte, el *fragment shader*, que es en el que nos centraremos mayormente, es un programa cuyo código se ejecuta una vez por cada píxel y siempre después de que se ejecute el *vertex shader*. Su objetivo es determinar el color del píxel en cuestión en función de la escena que tengamos dibujada, posiblemente aplicando un modelo de iluminación a los objetos que la componen.

El conjunto formado por el *vertex shader* y el *fragment shader* es conocido como **shader program**, que comúnmente se refiere al mismo únicamente como *shader*. A partir del código fuente de ambos *shaders*, cada uno se crea y compila por separado, para seguidamente unirse en único programa. En el siguiente código se puede ver este procedimiento, siendo el objeto `shaderProgram` el objeto que representa el *programa shader*.

```

1 // vsSource: Vertex Shader Source Code
2 // fsSource: Fragment Shader Source Code
3
4 const vertexShader = loadShader(gl, gl.VERTEX_SHADER,
5                               vsSource);
6 const fragmentShader = loadShader(gl, gl.FRAGMENT_SHADER,
7                                 fsSource);
8
9 // Create the shader program
10
11 const shaderProgram = gl.createProgram();
12 gl.attachShader(shaderProgram, vertexShader);
13 gl.attachShader(shaderProgram, fragmentShader);
14 gl.linkProgram(shaderProgram);

```

Además, GLSL utiliza tres tipos especiales de variables además de las propias variables locales que se definen en el programa, cada una con su cometido, que procedemos a explicar.

- **Variables ‘attribute’:** Sólo están disponibles en el *vertex shader* y en

el código de JavaScript, desde el cual se les da valor. Se suelen utilizar para almacenar información de color, coordenadas de textura, o en general cualquier información que deba ser compartida entre el código de JavaScript y el *vertex shader*.

- **Variables ‘varying’:** Son declaradas por el *vertex shader* y son utilizadas para enviar información desde el *vertex* para el *fragment shader*, de manera que la información se interpola. Por ejemplo, si el vertex shader asocia color negro a un vértice en una variable **varying** y blanco a otro vértice en la misma variable, entonces los píxeles situados entre estos dos vértices tendrán en esa variable un tono de gris.
- **Variables ‘uniform’:** Estas variables se definen por el código de JavaScript y se puede acceder a ellas tanto en el *vertex* como en el *fragment shader*. Se usan para especificarle a los shaders valores que no cambian independientemente del vértice o del píxel que se esté ejecutando. Por ejemplo, el color de un material o el zoom que se está aplicando a la escena.

Vemos por tanto que mediante estas variables podemos intercambiar información entre el código GLSL que se ejecuta en GPU y el código de JavaScript que comanda el uso de WebGL. Sin embargo, claro está que debe de haber alguna forma de transferir esa información desde JavaScript hasta la GPU. De eso mismo se encargan las estructuras conocidas como *buffers*, que procedemos a explicar.

4.1.3. Los *Buffer*

En el sentido más general de la palabra, un *buffer* es una memoria de almacenamiento temporal de información que permite transferir los datos entre unidades funcionales con características de transferencia diferentes. En nuestro contexto, existen estructuras que almacenan las variables definidas en JavaScript y se transfieren como variables **attribute** o **uniform** al programa *shader*. Por ejemplo, en el siguiente código creamos, a partir de un array de JavaScript que almacena 4 tripletas, cada una representando un color para un vértice, un buffer que las almacena.

```

1 const colors = [
2   1.0, 1.0, 1.0, 1.0,      // white
3   1.0, 0.0, 0.0, 1.0,      // red
4   0.0, 1.0, 0.0, 1.0,      // green
5   0.0, 0.0, 1.0, 1.0,      // blue
6 ];
7 // Buffer creation
8 const colorBuffer = gl.createBuffer();
9 // We select the buffer for the future buffer operations
10 gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
11 // We assign the 'colors' array data to the buffer
12 gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors),
               gl.STATIC_DRAW);

```

Y una vez creados los buffers y el *shader* por separado, debemos especificarle al buffer que transfiera la información que contiene a las posiciones de memoria que tiene el *shader* reservadas para las variables que espera. Supongamos que hemos creado en un *vertex shader* una variable **attribute vec4 aVertexColor;** de forma que queremos asociar los colores del código anterior a esta variable

por cada uno de los 4 vértices. Mostramos a continuación la forma de hacerlo, pudiendo encontrar más información clarificadora en la documentación de la clase WebGLRenderingContext.

```

1 const location = gl.getAttribLocation(shaderProgram, 'aVertexColor');
2 const numComponents = 4;           // Number of vertex
3 const type = gl.FLOAT;            // GLSL type of the vars
4 const normalize = false;          // Do not normalize
5 const stride = 0;                // Stride
6 const offset = 0;                // offset
7
8 // Select the buffer
9 gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
10 gl.vertexAttribPointer(
11     location,
12     numComponents,
13     type,
14     normalize,
15     stride,
16     offset);
17 gl.enableVertexAttribArray(location);

```

De forma muy similar, pero con las funciones correspondientes, se procede desde JavaScript para darle valor a las variables tipo `uniform`.

Tras crear y compilar el programa `shader` y dotarlo de valores para sus variables `attribute` y `uniform`, procedemos a renderizar la escena completa con la función `drawArrays`.

```

1 const offset = 0;
2 const vertexCount = 4;
3 gl.drawArrays(gl.TRIANGLE_STRIP, offset, vertexCount);

```

4.2. Primera imagen renderizada

Una vez conocemos los componentes principales de WebGL, es momento de utilizarlos para crear alguna imagen.

A modo de ejemplo, y aprovechando las situaciones concretas que se han explicado a lo largo de esta sección, supongamos que queremos dibujar en el canvas un cuadrado de colores. Para ello, necesitamos 4 vértices, y para cada vértice su posición y un color, de forma que necesitamos variables `attribute` en el *vertex shader* que representen posición y color. Además, dicho color queremos que se interpole en cada píxel a partir del color de los vértices, para ello usaremos una variable `varying` a la que le asociaremos el color del vértice en el *vertex shader* y recibirá el color del pixel en el *fragment shader* (es decir, recibirá del vertex shader precisamente lo que tiene que devolver).

```

1 attribute vec4 aVertexPosition;
2 attribute vec4 aVertexColor;
3
4 varying lowp vec4 vColor;

```

Crearemos un buffer para almacenar las posiciones de los cuatro vértices y otro para sus colores. Éste último es el que se ha mostrado en el código de ejemplo de la sección 4.1.3. Servimos al shader de valores a partir de los buffer y el resultado es el mostrado en la imagen 4.2.

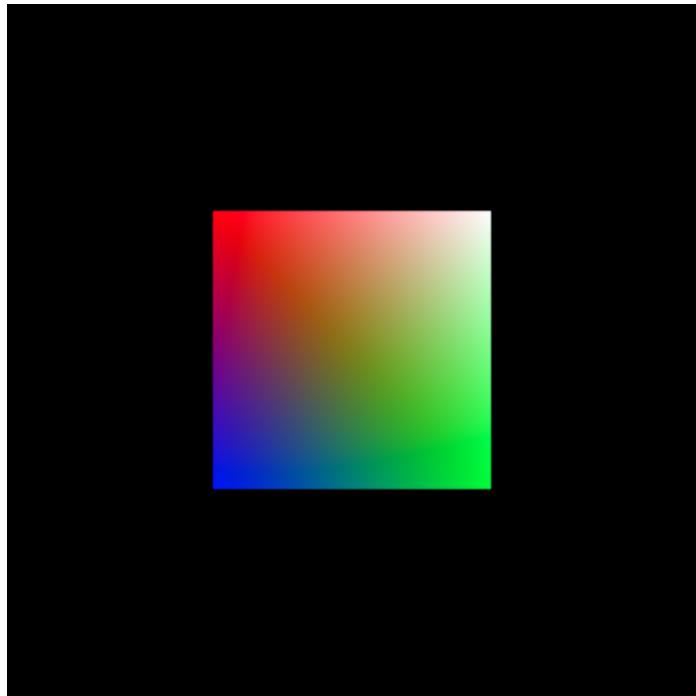


Imagen 4.2: Cuadrado de colores renderizado con WebGL

Fíjese cómo hay un vértice blanco, uno rojo, uno verde y uno azul, de forma que en los píxeles intermedios hay colores intermedios entre estos, fruto de la interpolación que se realiza entre *vertex shader* y *fragment shader*.

Este ejemplo se corresponde al expuesto en [13], cuya adaptación podemos encontrar en el repositorio de este proyecto, concretamente en <https://github.com/JAntonioVR/Geometria-Fractal/blob/main/static/js/canvas.js>. Podemos observar en la imagen 4.3 un esquema de los componentes de WebGL y cómo se relacionan entre ellos hasta finalmente visualizar la imagen deseada.

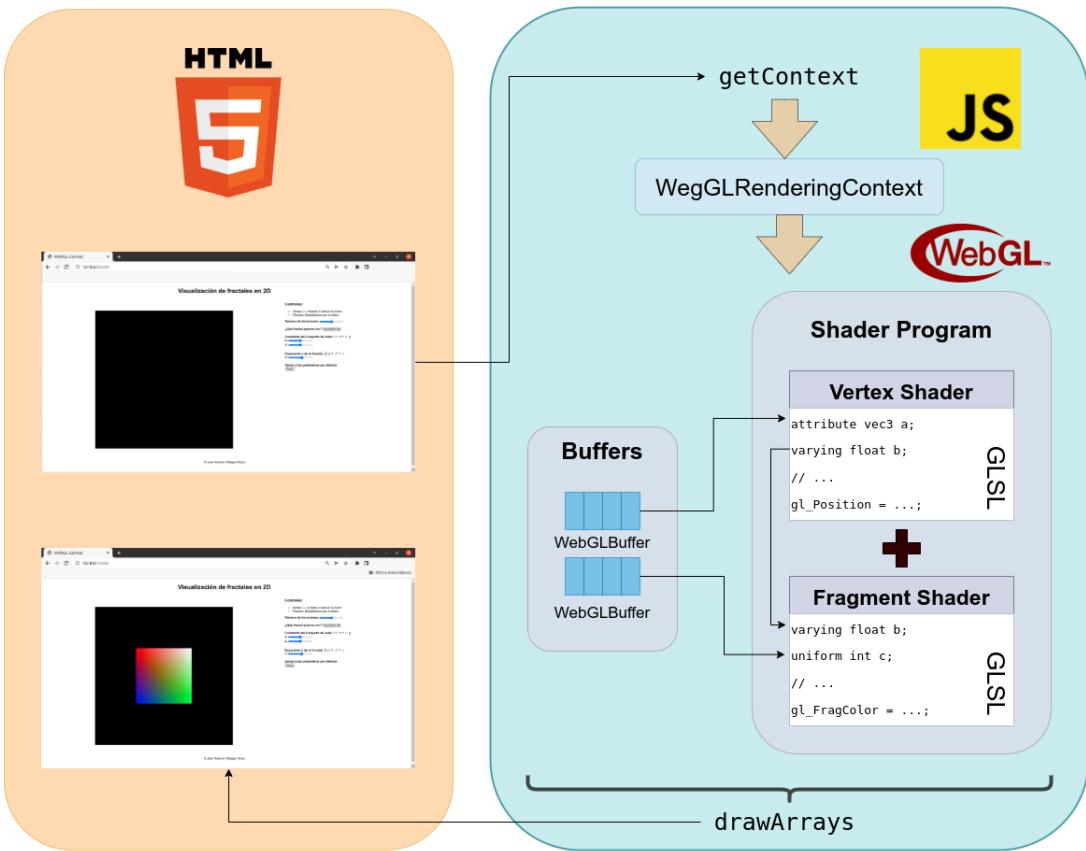


Imagen 4.3: Componentes de WebGL e interacciones entre ellos

CAPÍTULO 5

VISUALIZACIÓN DE FRACTALES EN 2D

En el capítulo 4 introdujimos el uso de WebGL como herramienta de renderizado de imágenes y estudiamos sus componentes, sin embargo, no olvidemos que nuestro objetivo es la visualización de fractales, donde la característica principal de los mismos es que no se pueden expresar a partir de un conjunto de vértices o líneas, sino que son curvas o superficies totalmente irregulares. Por tanto, a efectos prácticos, nuestro vertex shader tomará como entrada los vértices $(-1, -1)$, $(1, -1)$, $(1, 1)$ y $(-1, 1)$ y no aplicará ninguna transformación, pues ya están normalizados en el clip space (teniendo en cuenta que estaríamos visualizando un fragmento del plano $z = 0$). Cabe en este momento aclarar que en el ámbito de herramientas de renderizado se utiliza el convenio de utilizar la coordenada Y para la altura y la coordenada Z para la profundidad.

A partir de estos cuatro vértices, en el canvas se visualizarán dos triángulos que completarán la superficie completa del mismo. El vertex shader a partir de ahora será totalmente trivial, pues solo devolverá en la variable `gl_Position` la misma posición que obtiene del buffer de posición.

```
1 attribute vec2 a_Position;
2 void main() {
3     gl_Position = vec4(a_Position.x, a_Position.y, 0.0, 1.0);
4 }
```

Mientras que, por su parte, el fragment shader podrá acceder a la posición (en coordenadas de dispositivo) del píxel que se está ejecutando mediante la variable `gl_FragCoord` y a partir de estas coordenadas devolver un color en la variable `gl_FragColor`. Es decir, estamos dibujando una escena completa, próximamente un fractal, en dos triángulos. Por ejemplo, las imágenes 3.1 y 3.2 son el resultado de esta metodología.

5.1. Objetivo

Procedemos a explicar el objetivo principal de este objetivo y para el cual programaremos cada línea de código: Queremos desarrollar una página web interactiva, que cuente con un canvas donde se renderice el fractal que deseemos y, además, haya una serie de parámetros que se puedan controlar dinámicamente, de forma que conforme se cambia un parámetro el canvas modifica la imagen que está renderizando.

Queremos visualizar conjuntos de Julia J_c para distintos $c \in \mathbb{C}$, el conjunto de Mandelbrot, y las generalizaciones de los conjuntos de Julia y Mandelbrot ocasionadas si se itera la función $P_{c,N}(z) = z^N + c$ para distintos valores de $N \in \mathbb{N}$. Además, si revisitamos el algoritmo que utilizamos en la sección 3.2.1 para graficar en *Mathematica* conjuntos de Julia y el que utilizamos en la sección 3.4.1 para visualizar el conjunto de Mandelbrot, podremos recordar que para aproximar qué puntos del plano complejo eran prisioneros o de escape fijábamos un número máximo de iteraciones M , tras las cuales se consideraba que un número $z_0 \in \mathbb{C}$ era prisionero si la sucesión de los módulos de sus iteradas $\{P_{c,N}^n(z_0)\}$ no superaba el número de escape $e_c = \max\{2, |c|\}$. Este valor M también podría ser un parámetro modificable, para así poder ver dinámicamente cómo cambia la resolución cuando se cambia el número máximo de iteraciones.

5.2. Estructurando el código

Podemos usar como base el código utilizado para visualizar el cuadrado de colores, ya que nos puede venir bien su estructura para adaptar la misma a la renderización de fractales. Sin embargo, tiene una estructura muy procedural. Podemos mantener la misma arquitectura de forma que cambiando los elementos que sean necesarios y el código de los shaders podamos ver los fractales que deseemos, pero en ese caso la depuración se complicaría, el código es más difícil de leer y cuesta mucho añadir interactividad. Por este motivo, adaptaremos el código a un paradigma orientado a objetos, modularizando los distintos componentes, creando abstracciones de las herramientas que proporciona WebGL y siguiendo los principios SOLID.

Esta separación se ha hecho siguiendo

BIBLIOGRAFÍA

- [1] Edgar, G.A. (2008). *Measure, Topology, and Fractal Geometry* (2nd ed. 2008.). Springer New York. <https://doi.org/10.1007/978-0-387-74749-1>
- [2] Mandelbrot, B. (1983). *The Fractal geometry of nature*. Freeman.
- [3] Falconer, K. (1990). *Fractal geometry: mathematical foundations and applications*. John Wiley.
- [4] Hausdorff, F. *Dimension und ausseres Mass*. Mathematische Annalen 79 (1919): 157-179. <http://eudml.org/doc/158784>.
- [5] Hurewicz, W.; Wallman, H. (1948). *Dimension Theory*. Princeton University Press.
- [6] Moran, P.A. (1946). *Additive functions of intervals and Hausdorff measure*. In *Mathematical Proceedings of the Cambridge Philosophical Society* (Vol. 42, No. 1, pp. 15-23). Cambridge University Press.
- [7] Wagon, S. (2010). *Mathematica in Action*. Springer Publishing. p. 124
- [8] Payá, R (2008). *Apuntes de Análisis Matemático I...*
- [9] Ostrowski, A.M. (1973). *Solution of equations in Euclidean and Banach spaces*. (3rd ed.). Academic Press.
- [10] Atkinson, K; Han, W. (2009). *Theoretical Numerical Analysis: A Functional Analysis Framework* (3rd ed. 2009). Springer New York. <https://doi.org/10.1007/978-1-4419-0458-4>
- [11] Dubeau, F.; Gnang, C. (2018). Fixed Point and Newton's Methods in the Complex Plane. *Journal of Complex Analysis*, 2018, 1–11. <https://doi.org/10.1155/2018/7289092>
- [12] Milnor, J. (2006). *Dynamics in one complex variable*. (3rd ed.). Princeton University Press. <https://doi.org/10.1515/9781400835539>
- [13] *Adding 2D content to a WebGL context - Web APIs — MDN*. (2022, 24 abril). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Adding_2D_content_to_a_WebGL_context

- [14] colaboradores de Wikipedia. (2022, 16 febrero). *WebGL*. Wikipedia, la enciclopedia libre. <https://es.wikipedia.org/wiki/WebGL>
- [15] *Data in WebGL - Web APIs* — MDN. (2022, 14 marzo). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Data
- [16] *Getting started with WebGL - Web APIs* — MDN. (2022, 24 abril). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Getting_started_with_WebGL
- [17] *LearnOpenGL - OpenGL*. (s. f.). OpenGL. <https://learnopengl.com/Getting-started/OpenGL>
- [18] *OpenGL ES - The Standard for Embedded Accelerated 3D Graphics*. (2011, 19 julio). The Khronos Group. <https://www.khronos.org/api/opengles>
- [19] *WebGL: 2D and 3D graphics for the web - Web APIs* — MDN. (2022, 27 abril). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API
- [20] *WebGL Fundamentals - HTML5 Rocks*. (s. f.). HTML5 Rocks - A Resource for Open Web HTML5 Developers. https://www.html5rocks.com/en/tutorials/webgl/webgl_fundamentals/
- [21] *WebGL model view projection - Web APIs* — MDN. (2022, 26 abril). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/WebGL_model_view_projection#clip_space
- [22] *WebGLRenderingContext - Web APIs* — MDN. (2022, 20 enero). WebGLRenderingContext. <https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext>

APÉNDICE A

APPENDIX TITLE

APÉNDICE B _____

ANOTHER APPENDIX