Introduction to Fractals and Fractal Generation

Kiara Gholizad[1]

[1]Department of Physics, Sharif University of Technology, Tehran, Iran

February 27, 2025

Contents

1 Introduction **2**

2 Mathematical Foundations of Fractals **2**
2.1 Definition and Key Properties 2
2.2 Fractal Dimensions and Proofs 3
2.3 Example Proof: Hausdorff Dimension of the Cantor Set 5

3 Mapping Techniques in Fractal Generation **6**

4 Fractal Generation Algorithms **9**
4.1 Iterated Function Systems (IFS) 9
4.2 Escape-Time Fractals: Mandelbrot and Julia Sets 10
4.3 L-Systems (Lindenmayer Systems) 13
4.4 Other Methods: Random Fractals and DLA 14

5 Coding Implementations **16**
5.1 Barnsley Fern 16
5.2 Heighway Dragon Curve 20
5.3 Julia Set 24
5.4 Koch Curve 26
5.5 Sierpinski Triangle 28

1 Introduction

Fractals are mathematical structures with a reputation as having a pattern that is similar in shape at every scale. They occur in nature (for instance, in coasts, fern leaves, snowflakes), and are constructed with simple repeated processes. In this paper, we give a background in fractals, describing mathematical definitions, properties (like self-similarity and dimension), methods of mapping involved in constructing fractals, as well as a variety of methods for constructing fractals. Furthermore, we have included code in Python (with accompanying mathematical explanation) that can be utilized in constructing some famous fractals.

2 Mathematical Foundations of Fractals

2.1 Definition and Key Properties

What is a fractal? There is no single universally agreed definition of a fractal, but generally a fractal is a set or shape that displays **self-similarity** and has a **fractal dimension** that exceeds its topological dimension. Benoit Mandelbrot, who coined the term "fractal" in 1975, initially defined a fractal as "*a set for which the Hausdorff–Besicovitch dimension strictly exceeds the topological dimension*". He later broadened this description to "*a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole*". In essence, fractals are **infinitely complex**—zooming into a fractal reveals new details similar in structure to the original.

Self-similarity: Most fractals exhibit self-similarity, meaning that smaller portions of the object resemble the whole. This can be **exact self-similarity** (each small piece is an exact scaled copy of the whole, as in the ideal Cantor set or the Koch curve), or **statistical self-similarity** (pieces are approximately, but not exactly, like the whole, often with some randomness). Self-similarity can also be **affine**, where pieces are copies of the whole under some linear distortion (seen in many iterated function system fractals). A classical example is the Sierpiński triangle, which is composed of three smaller copies of itself, each scaled by $\frac{1}{2}$. Another example is the Mandelbrot set, which is quasi-self-similar: certain zoomed-in regions resemble the whole set (especially around special points), but with variation.

Complexity at all scales: Fractals are often defined in contrast to ordinary geometric shapes. A polygon or smooth curve becomes simpler when viewed up close—eventually looking like a straight line. A fractal, by contrast, reveals more structure upon magnification; this infinitely detailed structure at arbitrarily small

scales is a hallmark of fractals. Moreover, many fractals are defined by recursive or iterative processes, which inherently produce detail at every scale.

Non-integer dimension: One of the most striking properties of fractals is that they often have a **fractional (non-integer) dimension**. For example, the Koch snowflake curve is a one-dimensional curve topologically, but it is so "crinkly" that it has a dimension of about 1.2619... in the fractal sense. Intuitively, fractals fill space more than a typical 1D line but less than a 2D area. Formally, their *fractal dimension* (defined in various ways, see below) is greater than their topological dimension. In other words, a fractal scales in size in a way that is not consistent with an integer dimension. For instance, if you **double** the linear size of a fractal shape, its "size" (length, area, or volume in some sense) may scale by a factor 2^D where D is not an integer. This is in contrast to ordinary shapes: doubling the side of a square multiplies its area by $2^2 = 4$ (since it has dimension 2), and doubling the radius of a sphere multiplies its volume by $2^3 = 8$ (dimension 3). But for a fractal, doubling the size could multiply its content by, say, $2^{1.2619\ldots}$ or $2^{1.585\ldots}$ etc., reflecting a non-integer dimension.

Nowhere differentiable: Many classic fractal curves are *nowhere differentiable*—they have no well-defined tangent line at any point. A famous example is the Koch curve or the Weierstrass function. These functions or curves are continuous but infinitely jagged. This property was historically surprising in analysis; before the late 19th century, such "pathological" curves were not well understood. Fractals provided natural examples of such functions arising from simple rules.

2.2 Fractal Dimensions and Proofs

Because fractals are often too irregular to be described by traditional Euclidean dimensions, mathematicians have developed several definitions of fractal dimension to quantify their complexity. Two common notions are the Hausdorff dimension and the Minkowski–Bouligand dimension (also known as the box-counting dimension). For many well-behaved fractals (especially self-similar ones), these dimensions coincide.

- **Hausdorff Dimension:** This is a rigorous definition from measure theory. The Hausdorff dimension of a set is defined by how the Hausdorff measure of the set scales with size. Formally, one defines for any $s \geq 0$ the Hausdorff s-measure by covering the set with small balls of diameter at most ϵ and summing the sth powers of the diameters. The Hausdorff dimension D_H is the threshold where this measure transitions from ∞ to 0. Intuitively, D_H is the exponent that characterizes how the number of small balls needed to cover the set grows as the ball radius shrinks. For a smooth curve, $D_H = 1$; for a smooth surface,

$D_H = 2$. For fractals, D_H can be non-integer. For example, the boundary of the Mandelbrot set is known to have Hausdorff dimension 2 (even though it's a subset of the plane). The Cantor set has Hausdorff dimension $\log 2/\log 3 \approx 0.6309$, meaning it is "less than 1-dimensional."

- **Box-Counting (Minkowski) Dimension:** This is an easier-to-compute notion in many cases. Imagine covering the fractal with a mesh of boxes (squares) of side length ε and counting how many boxes $N(\varepsilon)$ contain part of the fractal. For a true fractal, $N(\varepsilon)$ typically scales like a power of $1/\varepsilon$ as $\varepsilon \to 0$. The box-counting dimension D_B is defined as:

$$D_B = \lim_{\varepsilon \to 0} \frac{\log N(\varepsilon)}{\log(1/\varepsilon)}, \tag{1}$$

.Equivalently, if $N(\varepsilon) \approx C, \varepsilon^{-D_B}$ for small ε, then D_B is the fractal dimension. This definition often gives the same value as the Hausdorff dimension for non-pathological sets, though in general $D_B \geq D_H$. The box-counting dimension is easier to estimate from empirical data or images of fractals.

Similarity Dimension (self-similar sets): For self-similar fractals, there's a simple way to find the dimension. Consider a fractal that is exactly made up of N smaller copies of itself, each copy scaled down by a ratio r in each linear dimension. For example, the Sierpiński triangle consists of $N = 3$ copies each of linear scale $r = 1/2$; the Cantor set consists of $N = 2$ copies each of scale $r = 1/3$. In such cases, the fractal dimension D (which in these cases equals both Hausdorff and box-counting dimension) satisfies the equation:

$$N \cdot r^D = 1 \Rightarrow D = \frac{\ln N}{\ln(1/r)}. \tag{2}$$

This formula for D is sometimes called the similarity dimension.

Proof (Similarity Dimension Formula): Suppose a fractal S is exactly self-similar such that it can be decomposed into N disjoint copies of itself, each copy scaled by a factor $0 < r < 1$ in all dimensions. We want to show the Hausdorff dimension D of S satisfies $Nr^D = 1$. Cover S by N sets, each being a scaled-down version of S by factor r. If d_H is the Hausdorff dimension, by the definition of Hausdorff measure, we expect $N \cdot$(diameter of each piece)$^D \approx N \cdot r^D$(diameter of $S)^D$ to cover the whole set. But the diameter of S to the power D is related to its Hausdorff measure. For the coverage to be minimal at dimension D, we require $Nr^D = 1$. Solving this gives $D = \ln N/\ln(1/r)$. More rigorously, one can show

both Hausdorff and box-counting dimensions equal this value for self-similar sets (assuming the pieces just touch or are disjoint, etc., to avoid overlaps).

Examples: Using the formula or direct reasoning:

- **Cantor set:** $N = 2$ pieces, each of length $r = \frac{1}{3}$. So $D = \frac{\ln 2}{\ln 3^{-1}} = \frac{\ln 2}{\ln 3} \approx 0.63093$. We can verify that for $D = \ln 2 / \ln 3$, indeed $2 \cdot (1/3)^D = 1$. The Cantor set has zero length (1D Lebesgue measure) but uncountably many points, so a dimension between 0 and 1 makes sense.

- **Koch curve:** $N = 4$ self-similar pieces (each iteration of the Koch curve replaces each line segment by 4 segments in a zigzag), each piece scaled by $r = 1/3$. Thus $D = \frac{\ln 4}{\ln 3} \approx 1.26186$. This is greater than 1 (the topological dimension of a curve), reflecting the curve's extreme twistiness. The Koch snowflake (the perimeter of the Koch curve shape) has infinite length but encloses a finite area, consistent with dimension > 1 but < 2.

- **Sierpiński triangle:** $N = 3$, $r = 1/2$. So $D = \frac{\ln 3}{\ln 2} \approx 1.5850$. The Sierpiński triangle is more than a line (which would be dimension 1) but less than a full 2D area (dimension 2). Indeed it has zero area in the plane but an infinite total boundary length.

Other dimensions: There are other notions like Lyapunov dimension, information dimension, correlation dimension, etc., used in various contexts (especially for strange attractors in chaos theory). These often coincide for simple fractals but can differ for more complex sets. In this introduction, Hausdorff and box-counting dimensions are sufficient to illustrate the concept of fractal dimension.

2.3 Example Proof: Hausdorff Dimension of the Cantor Set

Claim: The Cantor middle-thirds set C has Hausdorff dimension $D = \frac{\ln 2}{\ln 3}$.

Outline of Proof: The Cantor set is self-similar. It can be seen as two copies of itself scaled by $1/3$. We have $N = 2$ and $r = 1/3$. As reasoned above, any candidate dimension D should satisfy $2(1/3)^D = 1$. Solving yields $D = \ln 2 / \ln 3$. To rigorously show this is the Hausdorff dimension, one can show two things: (1) For any $s > \ln 2 / \ln 3$, the s-dimensional Hausdorff measure of C is 0 (the set is "too small" to have positive measure in that dimension). And (2) for any $s < \ln 2 / \ln 3$, the s-dimensional measure is infinite (the set is "too large/complex" to cover in that lower dimension without infinite mass). The critical threshold where the measure transitions is $s = \ln 2 / \ln 3$. Thus by definition, $\dim_H(C) = \ln 2 / \ln 3$. (We omit

the full ϵ-covering argument for brevity; it involves constructing coverings of C by intervals and using its self-similar structure.)This kind of argument can be adapted to many self-similar fractals. When the self-similar pieces are not all the same size (e.g., some IFS fractals), the formula generalizes to solving $\sum_{i=1}^{N} r_i^D = 1$ where r_i are the different contraction ratios. If all r_i are equal to r, this sum is $Nr^D = 1$, recovering the earlier formula. For example, a variant Cantor set that removes different lengths in each stage would yield a similar equation. In cases where pieces overlap, things get more complicated, but many classic fractals assume a no-overlap (or just-touching) condition that simplifies analysis.

3 Mapping Techniques in Fractal Generation

Fractals are generated by **iteratively applying mappings** – mathematical functions or rules that transform points, sets, or strings. Different fractal types arise from different kinds of mapping techniques. Here we outline several key mapping approaches used to create fractals, along with examples:

- **Affine Transformations (Linear maps in $\mathbb{R}^n$):** Many fractals (especially geometric ones like the Sierpiński triangle, Barnsley fern, etc.) are produced by repeatedly applying a set of affine transformations. An affine transformation in the plane has the form

$$f(x, y) = (ax + by + e, cx + dy + f), \tag{3}$$

  which can be thought of as a linear transformation (scaling/rotation/shear given by the matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ followed by a translation (e, f). Affine maps preserve lines and parallelism (hence "linear-like"), and can contract or rotate the plane. In fractal generation, we often use contractive affine maps (they shrink distances). For example, the Sierpiński triangle uses three maps: $f_1(x, y) = (\frac{x}{2}, \frac{y}{2})$, $f_2(x, y) = (\frac{x}{2} + \frac{1}{2}, \frac{y}{2})$, and $f_3(x, y) = (\frac{x}{2}, \frac{y}{2} + \frac{1}{2})$. Each of these is a similarity transformation (scale by 1/2 and translate to a corner). Starting from any initial shape (or point cloud), iteratively applying these maps will cause points to cluster onto the Sierpiński triangle. The Iterated Function System approach (see Section 4.1) formalizes this idea. Affine mappings are a core technique because they produce self-similarity – the fractal is the fixed set that reproduces itself under those mappings.

- **Complex Iterations (Conformal maps in $\mathbb{C}$):** Another class of mappings comes from complex dynamics. Here we iterate a function $f : \mathbb{C} \to \mathbb{C}$ (often a polynomial or rational function) repeatedly. The classic example is $f(z) = z^2 + c$, which gives rise to the **Julia set** and **Mandelbrot set** fractals (Section 4.2). Each iteration is a mapping of the complex plane to itself. The fractal emerges as the set of points that do not diverge under infinite iterations. These mappings are generally nonlinear. Many have the form $z \mapsto z^n + c$ or other complex functions (exponential, trigonometric, etc., can also yield fractals). Such mappings often have a property of being conformal (angle-preserving) or at least differentiable, which means small shapes are distorted but not torn or folded like in arbitrary maps. **Escape-time algorithms** use these mappings: points either escape to infinity or remain in a bounded region under iteration, and the boundary between those behaviors is fractal. Complex mapping techniques require understanding stability and divergence in dynamic systems.

- **Recursive Substitution / Grammar (Symbolic mappings):** Some fractals are generated via rewriting rules or grammars rather than directly mapping geometric points. An **L-system (Lindenmayer system)** is essentially a set of string rewriting rules applied iteratively (Section 4.3). Here the "mapping" is replacing characters (symbols) in a string with other strings according to production rules – a sort of symbolic mapping. For example, a simple L-system rule might be $F \to F + F - F - F + F$ (which, when interpreted graphically, produces a fractal called the Koch curve). The mapping technique is abstract: it's not mapping points in $\mathbb{R}^n$, but mapping strings to strings (which can then be mapped to geometry via an interpretation like "F means draw forward, + means turn," etc.). These recursive mappings create self-similar patterns through the growth of strings. Another example is a binary tree fractal generated by a rule like $X \to F[+X][-X]FX$ in an L-system, where the mapping introduces branches [...] in the string, modeling plant growth. Each rewrite is a mapping from one "generation" of the fractal to the next.

- **Stochastic / Random Mappings:** Not all fractals are deterministic. Some mapping techniques involve randomness. For instance, the **chaos game** is a method where one randomly applies one of several maps at each iteration (rather than all maps systematically). Surprisingly, this still yields the same fractal in the long run as applying all maps deterministically. Random fractals like **Brownian motion** or **fractal landscapes** involve random mappings where each step or stage involves a random choice or perturbation. For exam-

ple, the midpoint displacement algorithm for terrain generation maps a line segment to a "bumpier" line by randomly displacing the midpoint, and repeats recursively to create a fractal curve. In **diffusion-limited aggregation (DLA)**, random walks map onto a cluster by sticking when they hit it (Section 4.4) – effectively a random mapping from the space of walk paths to a growing set of occupied sites. Stochastic mappings often produce statistical self-similarity, where the fractal has self-similar qualities in a statistical sense (e.g. the roughness looks the same at different scales on average).

- **Inversion and Other Nonlinear Maps:** Some less common fractals use specialized mappings like circle inversions (used to generate Apollonian gaskets), or projective maps in the complex plane (to generate Kleinian group fractals). These are more advanced but worth mentioning: the general approach is still iterative mapping, but using transformations beyond simple affine or polynomial ones. For example, an Apollonian gasket can be generated by repeatedly inverting circles inside other circles – an inversion mapping technique.

Underpinning many of these techniques is the **contraction mapping principle** (Banach fixed-point theorem). In short, if you have a mapping (or a set of mappings) that strictly brings points closer together (a contraction), then iterating that mapping will converge to a unique fixed point or set. In the context of fractals, the set (often a Cantor-like set in the limit) is the fixed point. For an Iterated Function System (IFS) consisting of multiple contraction maps f_i, there is a unique non-empty compact set S such that

$$S = f_1(S) \cup f_2(S) \cup \cdots \cup f_N(S). \tag{4}$$

This S is the fractal (attractor) of the IFS. The operator that applies all f_i to sets (i.e. $F(A) = \bigcup_{i=1}^{N} f_i(A)$) is called the Hutchinson operator. It is a contraction on the space of compact sets (with a suitable metric), and Banach's fixed-point theorem guarantees a unique fixed set S (the fractal). This provides a rigorous foundation for why many fractal generation processes converge to a stable shape.

In summary, mapping techniques for fractals involve using functions or rules repeatedly. Whether it's moving points around the plane, iterating complex numbers, or rewriting symbols, the essence is iteration and feedback: the output of one step becomes the input for the next. Through iteration, simple rules can yield endlessly intricate structures – the hallmark of fractals.

4 Fractal Generation Algorithms

In this section, we delve into specific algorithms for generating fractals, linking each to the mapping techniques above. We discuss the theory and some mathematical background for each method, then in Section 5 we will show code implementations.

4.1 Iterated Function Systems (IFS)

Theory: An Iterated Function System is a set of contraction mappings $f_1, f_2, \ldots, f_N$ on a complete metric space (typically $\mathbb{R}^2$ for planar fractals). The classic result by Hutchinson (1981) states that there is a unique non-empty compact set S that is invariant under all these mappings:

$$S = f_1(S) \cup f_2(S) \cup \cdots \cup f_N(S). \tag{5}$$

S is called the attractor of the IFS, and it is what we recognize as the fractal. For example, for the Sierpiński triangle IFS mentioned earlier, the attractor S is the Sierpiński triangle itself. The existence and uniqueness of S come from the contraction mapping (Banach) theorem: one can define an operator F on sets by $F(A) = \bigcup_{i=1}^{N} f_i(A)$. If each f_i is a strict contraction (shrinks distances by a factor at most $c < 1$), then F is a contraction on the space of compact sets (with the Hausdorff metric). Thus, starting from any initial compact set A_0, the iterative sequence $A_{n+1} = F(A_n)$ converges to S. In practice, repeatedly applying all f_i to an initial shape (like a triangle for Sierpiński or a line segment for the Cantor set) and taking the union will eventually approximate the fractal attractor.

Construction methods: There are two primary ways to generate points in S:

- **Deterministic:** Start with an initial set (like a filled shape) and at each step, replace it with $f_1(A) \cup \cdots \cup f_N(A)$. This yields a sequence of approximations $A_1, A_2, \ldots$ converging to S. For instance, begin with a solid triangle and at each step, apply the three Sierpiński mappings (scale and translate) to get three smaller triangles; remove the middle part. Repeating this gives closer approximations to the Sierpiński gasket.

- **Random (Chaos Game):** Start with a point p_0 (say within the attractor's bounds). Then iterate $p_{n+1} = f_{i_n}(p_n)$ where at each step i_n is chosen randomly from $1, \ldots, N$ with certain probabilities. Surprisingly, after many iterations, the points p_n will be distributed according to the fractal S (ignoring some initial transient). This chaos game method is computationally efficient for drawing fractals: you randomly hop around, always landing on the fractal. It's

essentially because S is an attractor: random or not, points get "attracted" to S. One must choose the probabilities carefully (often proportional to something like the contraction ratio's effect on area, or one can simply choose uniformly if not concerned with exact measures).

Examples of IFS Fractals:

- **Sierpiński Triangle:** IFS with 3 maps as described (each $f_i(x, y) = \frac{1}{2}(x, y)+t_i$ where t_i is translation to one of the three corners of an initial triangle).
- **Barnsley Fern:** A famous IFS with 4 affine maps on the plane, which produces a shape resembling a natural fern (see coding section for formulas). It uses one main map with probability 85 % that scales and rotates points slightly (forming the dense leafy part), and three other maps with smaller probabilities that form the stem and smaller leaves. Despite the randomness, the resulting set of points is an orderly fern shape – a testament to the IFS attractor principle. (Barnsley's fern IFS code is given in Section **4.1.**) Koch Curve (as IFS): Although commonly generated by a deterministic substitution, the Koch curve can also be seen as an IFS: 4 maps, each scaling by 1/3 and appropriate rotation/translation. The attractor is the Koch curve (a continuous 1D fractal).
- **Cantor set:** IFS with 2 maps: $x \to \frac{x}{3}$ (left piece) and $x \to \frac{x}{3} + \frac{2}{3}$ (right piece) on the unit interval. The attractor is the Cantor middle-thirds set.

Dimensions: For an IFS of self-similar non-overlapping pieces, the Hausdorff dimension D satisfies $\sum_{i=1}^{N} r_i^D = 1$ as discussed. If all maps have the same scaling factor r (similitudes), this simplifies to $Nr^D = 1$. For the Barnsley fern, the maps have different scales and rotations, but one can still in principle solve $\sum r_i^D = 1$ (though because some pieces overlap slightly, the formula is more an upper bound). In practice, often one just empirically measures box-counting dimension for complicated IFS fractals. Many IFS fractals also carry a natural probability measure (each map contributes a certain proportion of points), and one can discuss dimensions of that measure (information dimension, etc.), but that's beyond our scope here.

4.2 Escape-Time Fractals: Mandelbrot and Julia Sets

Complex Iteration and Julia Sets: Let $f_c(z) = z^2 + c$ be a complex quadratic map (where c is a complex parameter). For a given fixed c, one can look at what points z_0 in the complex plane do under iteration $z_{n+1} = f_c(z_n)$. Typically, some

initial points will have their $|z_n|$ tend to infinity (they escape to infinity), while others remain bounded (their iterates stay within some radius forever). The filled Julia set K_c is defined as the set of all initial points z_0 whose iterates remain bounded. The Julia set J_c is the boundary of K_c. For most values of c, the Julia set is a fractal curve (if disconnected, it can be a dust of points; if connected, it's a complicated curve). Each c gives a different Julia set. Some famous ones:

- For $c = 0$, $f_0(z) = z^2$. The Julia set J_0 is the unit circle (not fractal in that case, just a circle).
- For $c = -1$, the Julia set is known as the dragon curve fractal (connected and quasi-self-similar).
- For $c = -0.123 + 0.745i$ (just as an example), J_c is a dendrite-like fractal.
- For c outside the Mandelbrot set (explained below), J_c is disconnected (a Cantor dust of points). For c inside, J_c is connected.

The way to generate a Julia set is via an **escape-time algorithm**: For a grid of points z_0 in the complex plane, iterate each point until either:

- $|z_n|$ exceeds some escape radius (usually we can prove a radius like 2 is enough for $z^2 + c$), in which case the point will go to infinity and is not in the filled Julia set.
- Or we reach a maximum number of iterations without escaping, in which case we assume the point is in K_c (likely in the Julia set or interior if interior exists).

Plotting the points that do not escape (or coloring points by how quickly they escape) produces an image of the Julia set. The mathematical foundation for using an "escape radius" is that for $f_c(z) = z^2 + c$, if $|z| > 2$ at any iteration, the sequence will diverge to infinity. Proof Sketch: If $|z_n| > 2$, then $|z_{n+1}| = |z_n^2 + c| \geq |z_n|^2 - |c|$. If $|z_n| > 2$, note $|z_n|^2 > 4$. For any fixed c, beyond some point this inequality will ensure growth. In fact, one can show by induction that once outside the circle of radius 2, the sequence escapes to infinity. Thus 2 is a convenient universal escape radius for all c (in fact, for any quadratic polynomial the escape radius can be set to 2). So the algorithm is: if at any time $|z_n| > 2$, declare the starting point as escaping.

Julia sets are often **nowhere dense** (having empty interior) and are typically fractal curves. Many Julia sets have Hausdorff dimension between 1 and 2.

The Mandelbrot Set: The Mandelbrot set M is a fractal in the parameter space of c. It is defined as the set of all complex parameters c for which the orbit of 0 under $f_c(z) = z^2 + c$ stays bounded.

Equivalently, $M = c \in \mathbb{C} : |z_n| \not\to \infty$ for $z_0 = 0, z_{n+1} = z_n^2 + c$. Remarkably, this set M is connected and has a very intricate boundary which contains miniature copies of M itself (self-similarity). It can be shown that $c \in M$ if and only if the corresponding Julia set J_c is connected. So the Mandelbrot set acts like an "index" of Julia sets: inside M you have connected (and typically chaotic) Julia sets; outside M you get Cantor-set-like dusts.

To generate the Mandelbrot set, one also uses an escape-time algorithm: For each c in some region of the complex plane (usually we view c in, say, $[-2, 1]\times[-1, 1]$ which covers the heart of M), iterate $z_{n+1} = z_n^2 + c$ starting with $z_0 = 0$. If $|z_n|$ exceeds 2 at any point (escape), we color c as outside M (often color it by the iteration count at escape). If we reach the max iterations without escape, we color c as inside M (usually black). This produces the well-known picture of the Mandelbrot set.

Properties: The Mandelbrot set has a cardioid (heart-shaped) main body with an infinite cascade of circular bumps, and an infinite number of satellite copies of itself attached by filaments. Its boundary is a fractal curve of Hausdorff dimension 2 (it is very "fuzzy" and actually area-filling in a sense). Some points on the boundary are especially interesting: for example, $c = -0.75$ is on the boundary and corresponds to a Julia set that is known as the dendrite with a single cusp. The structure of M encodes all the dynamics of $z^2 + c$.

Generalizations: Escape-time fractals are not limited to z^2+c. One can iterate other functions: e.g. $z_{n+1} = z_n^3 + c$ gives a different Mandelbrot-like set in the c-plane (with a similar shape but different symmetry). Newton's method fractals are generated by iterating Newton's method for root-finding in the complex plane—those fill the plane with basins of attraction, whose boundaries are fractal. One can even iterate matrices, quaternions, or other systems to get fractals, but the 2D complex plane is the most common for visualization.

Mathematical intricacy: The theory of Julia and Mandelbrot sets is deep complex dynamics. For example, it is known that the Mandelbrot set is connected (Douady and Hubbard). Many open problems remain: it is conjectured that the Mandelbrot set is locally connected (the "MLC" conjecture), which would imply things about the structure of Julia sets. These fractals are not just pretty pictures; they connect to chaos theory, number theory, and the topology of dynamic systems.

4.3 L-Systems (Lindenmayer Systems)

Theory: An L-system is a formal grammar system introduced by Aristid Lindenmayer in 1968 to model the growth of plants. It consists of an alphabet of symbols, a set of production rules that tell how to replace symbols with strings of symbols, an axiom (starting string), and usually an interpretation scheme to turn the final strings into geometric structures (like drawings).

The L-system rewriting is done in **parallel**: at each iteration, every symbol in the string is replaced simultaneously according to the rules. For example, consider a simple L-system:

- Alphabet: $F, +, -$ (where F means "draw forward", and $+$, $-$ mean "turn left/right" by some angle).
- Axiom (start): F.
- Production rule: $F \to F + F - F - F + F$ (this is the Koch curve rule).
- Angle: 90° for + and -.

Iteration $n = 0$ (initial string): F (just a line).
Iteration $n = 1$: apply rule to F -> F+F-F-F+F.
Iteration $n = 2$: replace each F in the string with F+F-F-F+F again, yielding a much longer string: F+F-F-F+F+F+F-F-F+F-F+F-F-F+F-F+F-F-F+F+F+F-F-F+F (which corresponds to the Koch curve after two iterations, as in the example from Wikipedia).

If you interpret this string as drawing instructions (say start drawing a line of a certain length, turning at the symbols), the resulting figure after each iteration is a more detailed approximation to the Koch fractal. In the limit of infinite iterations (with line length shrinking accordingly), you'd get the Koch curve.

Key point: L-systems generate fractals by recursion at the level of symbols. The self-similarity is encoded in the production rules. For the Koch curve, the rule $F \to F+F-F-F+F$ essentially says: one line segment F is replaced by 4 segments arranged in a specific pattern (a scaled copy of the original line with a kink). This is the self-similar structure in symbolic form.

Many fractals and plant-like structures can be described by L-systems:

- **Fractal Curves:** Koch snowflake, Sierpiński triangle (which can be generated by an L-system with two variables F, G and rules $F \to F - G + F + G - F$, $G \to GG$), the dragon curve, Hilbert curve, etc., all have L-system descriptions.

- **Fractal Plants:** By adding branching symbols (like [and] to push and pop the drawing state, enabling branches), L-systems can model trees, ferns, and other organic forms. For example, a simple branching L-system might use X as a variable that expands into say F[+X][-X]FX (meaning: draw forward, branch left (with + turn) recursively X, branch right (- turn) recursively X, then draw forward, then end of recursion for this branch). With an angle of, say, 25°, repeated application produces a tree-like fractal.

- **Cantor set:** Even the Cantor set can be done with an L-system: alphabet A, B, axiom A (interpreted as a filled segment), rules $A \to ABA$, $B \to BBB$ (where B is interpreted as a gap). After iterations and interpreting A as drawing and B as moving without drawing, you'd get the Cantor set pattern.

Formal grammar perspective: L-systems are a type of rewrite system (specifically a parallel rewriting system). This is in contrast to other fractal algorithms which are more geometric or analytic. L-systems have a close connection to turtle graphics for interpretation. A "turtle" (from the Logo programming language metaphor) moves and turns according to the symbols and draws lines, creating the fractal image.

Recursion and self-similarity in L-systems: You can often deduce the fractal dimension of the resulting curve by analyzing the production. For Koch's $F \to F + F - F - F + F$, one F produces 4 F's in the next iteration, each 1/3 the length of the original (assuming we scale lengths down to keep total size constant). That matches the $N = 4, r = 1/3$ rule, giving dimension $\ln 4 / \ln 3$. For a binary fractal tree L-system, if each branch splits into 2 sub-branches of, say, 3/4 the length, one might solve $2(3/4)^D = 1$ giving a dimension around 1.7095 (though tree fractals can be more complex if they have random variation).

Note: L-systems are very flexible; by adjusting rules, one can interpolate between fractal forms, add randomness (stochastic L-systems choose among multiple rules for a symbol at random), or make context-sensitive rules. This goes beyond pure fractals into realistic modeling, but at heart, an L-system with deterministic rules defines a particular fractal structure.

4.4 Other Methods: Random Fractals and DLA

Finally, beyond the above, there are some other notable fractal generation methods:

- **Random (Statistical) Fractals:** These fractals are generated by stochastic processes rather than exact deterministic rules. One example is fractional

Brownian motion (fBm), which generalizes the notion of a random walk to have fractal-like trajectories characterized by a Hurst exponent H. The fractal (Hausdorff) dimension of an fBm curve in 2D is $D = 2 - H$ (for $0 < H < 1$). So by tuning H, one can get a curve of any dimension between 1 and 2. Such fractal paths are used to model natural phenomena like coastlines (Mandelbrot famously analyzed coastlines as fractals), stock market fluctuations, or terrains.

- **Fractal terrain generation:** A popular algorithm is the midpoint displacement or diamond-square algorithm. Starting with a large triangle or square and initial corner heights, one randomly displaces the midpoint of edges or squares and repeats subdivision. The result is a landscape with fractal statistics (mountains and valleys at multiple scales). The roughness can be controlled by a parameter (roughness $\rightarrow$ fractal dimension of the terrain surface). While these algorithms don't produce an exact self-similar set, they produce a surface that is statistically self-similar (the distribution of height differences follows a power-law over scale). The Plasma fractal and Perlin noise are related techniques to create natural-looking fractal textures.

- **Diffusion-Limited Aggregation (DLA):** DLA is a process that simulates particles undergoing random motion and sticking together to form aggregates. It was introduced by T. Witten and L. Sander in 1981. The algorithm: start with a seed particle (e.g., at the origin). Then for each new particle, start it at a random far-away position and let it perform a random walk (diffuse) until it either wanders far (and we reset it) or comes in contact with the existing cluster, at which point it sticks. Repeat with many particles. The cluster that forms grows in a branching, tree-like way. These clusters are called **Brownian trees** and are fractal: in 2D, simulations estimate the fractal dimension ≈ 1.71. The resulting shape looks like a branching lightning bolt or dielectric breakdown pattern. In fact, DLA has been used to model things like electrodeposition, mineral dendrites, and even urban growth patterns.DLA is inherently random. No two runs will produce exactly the same aggregate, but they will be similar statistically. The fractal nature comes from the random walk's tendency to explore more area and the sticking that causes an irregular structure with gaps. One interesting fact: If you shine light through a DLA cluster, the diffraction pattern is fractal (because the cluster has a random fractal Fourier transform).

- **Cellular Automata and Fractals:** Some cellular automata, like certain rules in Conway's Game of Life or other simpler 1D automata (e.g. Rule 90) produce fractal patterns. For example, Rule 90 (an elementary cellular automaton)

starting from a single "on" cell produces the Sierpiński triangle pattern. The mapping here is the rule that maps a neighborhood of cells to a new state (which is like a parallel map on the grid each generation). After many steps, a fractal emerges in the pattern of live cells.

- **Fractals in Higher Dimensions:** One can generate 3D fractals via similar iterative methods. E.g. the Menger sponge (3D analog of Cantor set) is an IFS on a cube (remove the middle of each face and the core). The Mandelbulb and Mandelbox are 3D escape-time fractals (iterating a 3D map inspired by complex power). L-systems can generate 3D trees. DLA can be done in 3D (it produces a cluster of dimension 2.5). While visualization is harder, the algorithms are conceptually similar to their 2D counterparts.

- **Escape-time in other systems:** You can iterate functions on other domains for fractals. For example, iterating $z \mapsto \lambda z(1-z)$ in the real interval [0,1] (the logistic map) and plotting points in parameter space that remain stable yields a 2D fractal (the Feigenbaum bifurcation diagram has self-similar structure). While not a traditional fractal set in the plane, it is a fractal attractor in a different sense.

Each method has its own mathematical underpinnings and applications. The ones discussed in detail (IFS, complex iteration, L-systems, random fractals) form a good baseline for fractal generation techniques.

5 Coding Implementations

In this section, various fractals are drawn using different methods, as explained in the previous sections. One or two algorithms have been proposed for each fractal. Please note that these are not the only available algorithms for drawing these fractals, and other algorithms exist, but not all of them have been included here. Python was chosen as the programming language due to its ease and simplicity. The results of each algorithm are presented in the corresponding figures below each algorithm, and the related code is available on GitHub.

5.1 Barnsley Fern

The Barnsley fern is defined by four affine transformations in $\mathbb{R}^2$, applied with given probabilities. We'll implement the chaos game method for the Barnsley fern: start

with a point (0,0), then at each iteration, randomly choose one of the 4 transformations according to their probabilities and apply it to the current point. We will generate a large number of points that form the fern shape.The affine transformations $f_i(x, y) = (x', y')$ for Barnsley's fern (from Barnsley's book) are:

- f_1 (85% probability): $x' = 0.85x + 0.04y, \quad y' = -0.04x + 0.85y + 1.6.$
- f_2 (7% probability): $x' = 0.20x - 0.26y, \quad y' = 0.23x + 0.22y + 1.6.$
- f_3 (7% probability): $x' = -0.15x + 0.28y, \quad y' = 0.26x + 0.24y + 0.44.$
- f_4 (1% probability): $x' = 0, \quad y' = 0.16y.$

These were chosen by Barnsley to mimic the shape of a fern leaf.
The algorithm is presented in Table 1.

Barnsley Fern Algorithm

Algorithm 1 Generate Barnsley Fern

Initialize starting point $(x, y) = (0.0, 0.0)$
Define four transformation functions:
    $f_1(x, y) = (0.85x + 0.04y, -0.04x + 0.85y + 1.6)$
    $f_2(x, y) = (0.20x - 0.26y, 0.23x + 0.22y + 1.6)$
    $f_3(x, y) = (-0.15x + 0.28y, 0.26x + 0.24y + 0.44)$
    $f_4(x, y) = (0, 0.16y)$
Define probabilities:
    $p_1 = 0.85, p_2 = 0.07, p_3 = 0.07, p_4 = 0.01$
Initialize an empty list *points*
for $i = 1$ to *n_points* **do**
    Generate random number $r \in [0, 1]$
    if $r < p_1$ **then**
        Apply transformation $f_1(x, y)$
    else if $r < p_1 + p_2$ **then**
        Apply transformation $f_2(x, y)$
    else if $r < p_1 + p_2 + p_3$ **then**
        Apply transformation $f_3(x, y)$
    else
        Apply transformation $f_4(x, y)$
    end if
    Append (x, y) to *points*
end for
Extract x and y values from *points*
Plot the points (x, y) using a scatter plot with green color

Table 1: Barnsley Fern Algorithm

(a) $n = 10000$ (b) $n = 100000$ (c) $n = 1000000$

Figure 1: Barnsley Fern for different values of n.

Explanation: We maintained one moving point (x,y). At each step, we apply one of the four linear formulas to get a new (x,y). Over time, due to the contraction nature of the transformations, the point tends to the attractor (fern shape). The probability biases ensure that the denser parts of the fern (the smaller leaflets) which use f_1 are drawn more frequently, whereas the stem (f_4) gets drawn rarely (1% of the time). This matches the relative area those parts take in the final shape. This algorithm is an implementation of the chaos game approach for IFS.

5.2 Heighway Dragon Curve

The **Heighway Dragon Curve**, or *Dragon Curve*, is a self-similar fractal resulting from an iterative folding and rotating of line segments. The Heighway Dragon Curve, first discovered by John Heighway, is characterized by extreme symmetry and complexity, considering its set of simple rules. The Heighway Dragon Curve is widely recognized for its appearances in mathematical art, in computer graphics, and in pop culture, for instance, in *Jurassic Park*'s logo.

The following is an algorithm based on a GIF on Wikipedia and is presented in Table 2.

Description: This code attempts to generate Heighway Dragon Curve by adding successively a piece of a segment, which is rotated, to a base piece. The `rotate_point` operation is a 90-degree pivot about a pivot, which is necessary for this fractal's self-similarity. The `plot_hywei_dragon(n)` procedure forms the fractal in a cumulative, piece-by-piece, process, maintaining a list of piecewise line segments and rotating it based on a pivot selected dynamically. The pivot is an interesting, although nonstandard, selection, producing a variation of the Dragon Curve. The **color difference (blue and red)** is a nice touch, separating old and new pieces.

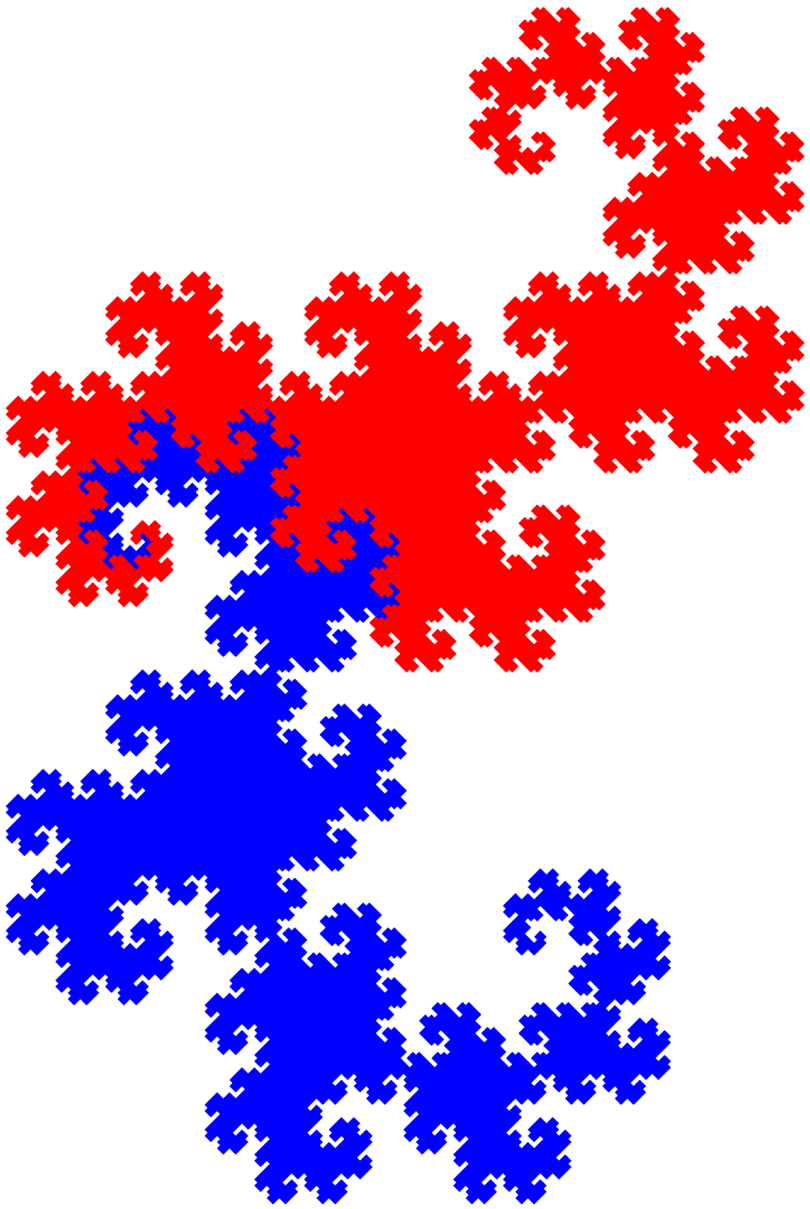


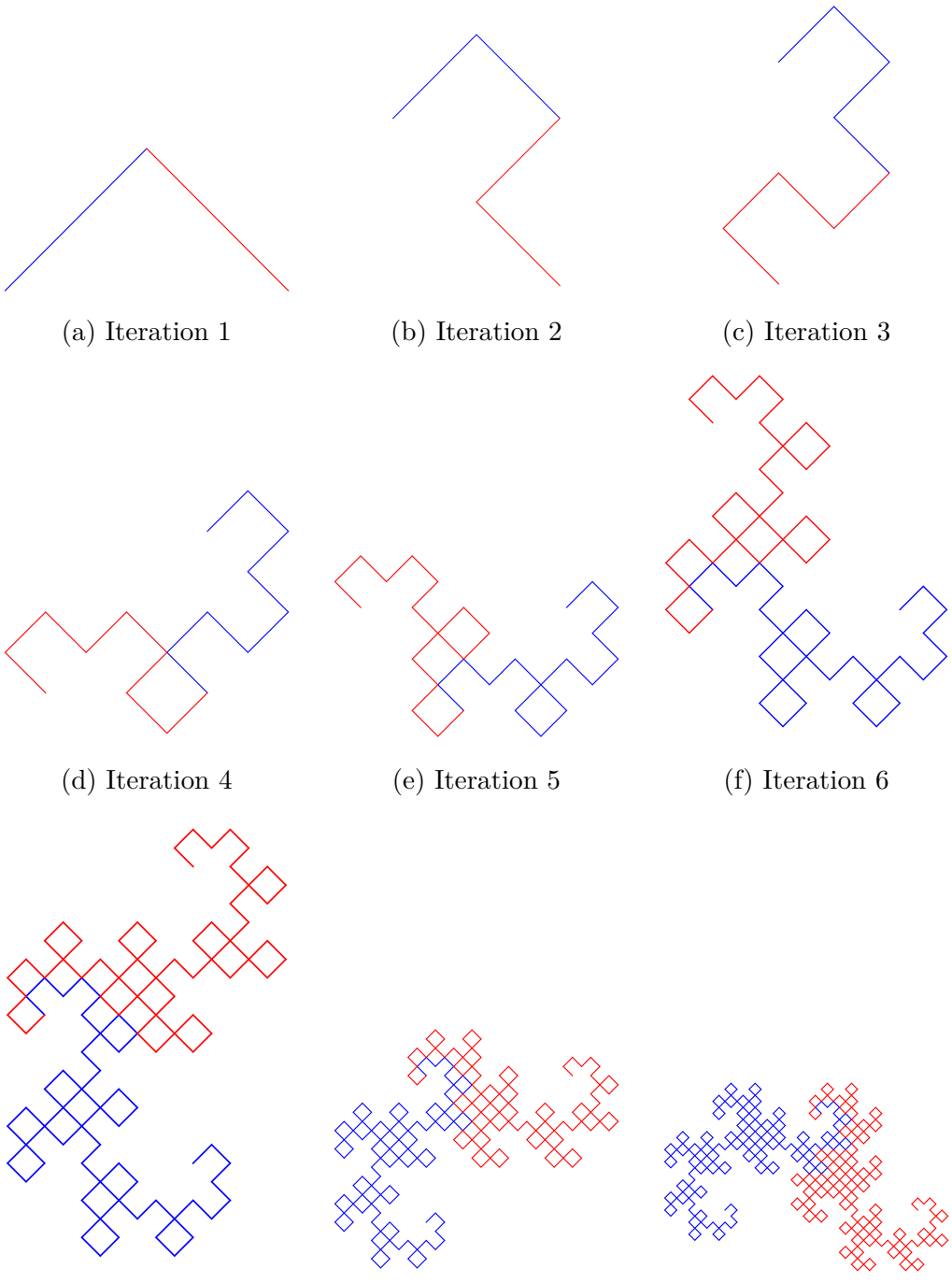
Figure 2: Heighway Dragon Curve at Iteration 15

Heighway Dragon Curve Algorithm

Algorithm 2 Generate and Plot Heighway Dragon Curve

Function: Rotate a point around a pivot by 90° counterclockwise
Input: Point (x, y) and Pivot (px, py)
Compute new coordinates:
    $new\_x = px - (y - py)$
    $new\_y = py + (x - px)$
Return (new_x, new_y)
Function: Generate Heighway Dragon Curve of order n
Initialize base segment: $segments = [((0, 0), (1, 1), blue)]$
for $i = 1$ to n **do**
    Create a new list $new\_segments$
    Extract all points from current $segments$
    if $i > 1$ **then**
        Choose final pivot as the first point of segment at index $2^{(i-1)}$
    else
        Choose final pivot as the point with maximum x and minimum y
    end if
    for each segment $(start, end, color)$ in $segments$ **do**
        Rotate $start$ around pivot
        Rotate end around pivot
        Append rotated segment with red color to $new\_segments$
    end for
    Append $new\_segments$ to $segments$
end for
Initialize plot with equal aspect ratio
for each segment $(start, end, color)$ in $segments$ **do**
    Plot segment with specified color
end for
Hide axis and display the plot

Table 2: Heighway Dragon Curve Algorithm



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3

(d) Iteration 4 (e) Iteration 5 (f) Iteration 6

(g) Iteration 7 (h) Iteration 8 (i) Iteration 9

Figure 3: Evolution of the Heighway Dragon Curve from iteration 1 to 9.

5.3 Julia Set

The **Julia Set** is a complex fractal resulting from iterating a simple function in the complex plane. Named for French mathematician *Gaston Julia*, this fractal is closely related to the *Mandelbrot Set* and is distinguished by its complex and self-similar patterns. The Julia Set is greatly influenced by the complex number c, which determines whether the resulting fractal is disjoint or connected. Such lovely representations can be viewed in mathematical graphics, physics, and chaos theory, which illustrate the sensitivity of dynamic systems toward initial parameters.

Julia Set Algorithm

Algorithm 3 Generate Julia Set for a Given Complex Number c

```
Initialize image grid with dimensions (width, height)
Define complex grid Z = X + iY where X, Y are ranges from −2 to 2
for each point (x, y) in grid do
    Initialize z = Z[x, y]
    for iteration n = 1 to max_iter do
        if |z| > 2 then
            Assign img[x, y] = n                ▷ Exit the loop if z escapes
            break
        end if
        Update z = z^2 + c                      ▷ Iterate using Julia set formula
    end for
end for
```

Table 3: Julia Set Algorithm for a Single Complex Value c

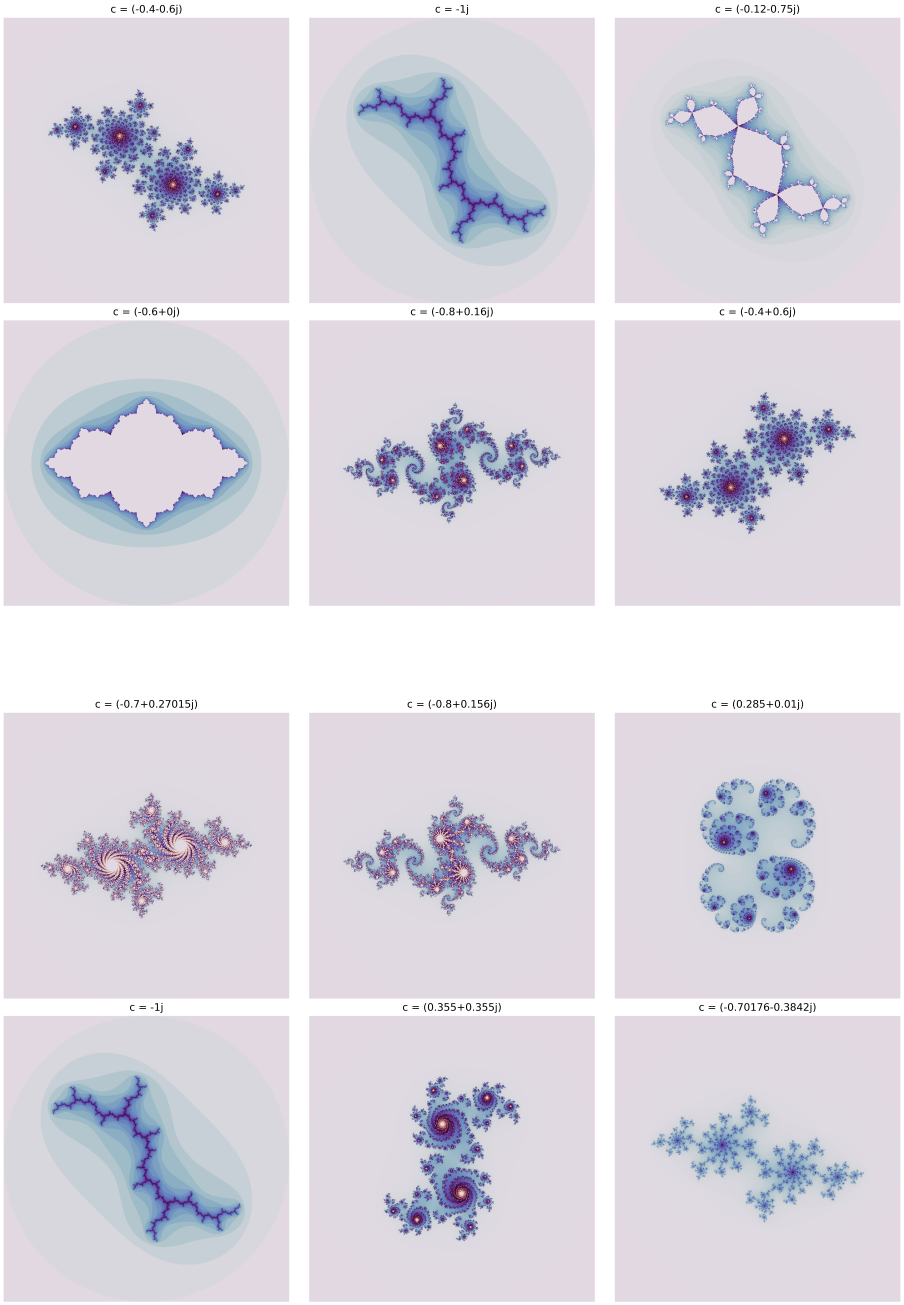


Figure 4: Comparison of Julia Sets for different values of c.

5.4 Koch Curve

The **Koch Curve** is a famous example of a fractal, which was first discovered in 1904 by Swedish mathematician *Helge von Koch*. The Koch Curve is made by successively subdividing each of an equilateral triangle's sides into smaller pieces, adding a new triangular "bump" on each of these sides. This results in a jagged, infinitely complex boundary which is a self-replicating pattern each time it is made. The Koch Curve is a perfect example of fractal geometry, which shows just how complicated and beautiful patterns can be made using a few geometric transformations. The Koch Curve is traditionally referred to as a "snowflake curve" when it is made in a large number of steps.

Here, two algorithms are used: one for drawing the Koch curve on a line and the other for generating the Koch snowflake, which starts from a triangle.

Koch Snowflake Algorithm

Algorithm 4 Generate Koch Snowflake

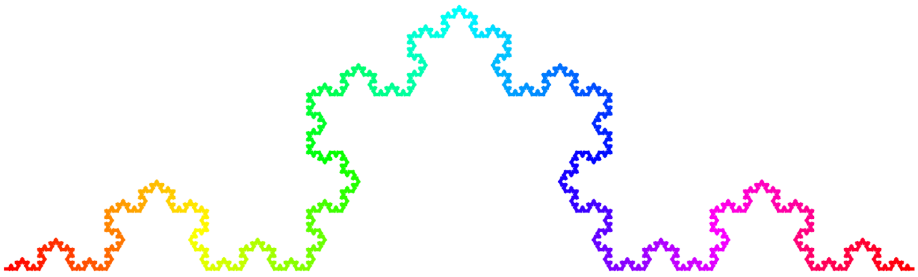
1: Initialize equilateral triangle with vertices (p_1, p_2, p_3)
2: **for** $n = 1$ to order **do**
3: Create empty list *new_points*
4: **for** each edge (p_i, p_{i+1}) in current shape **do**
5: Compute $p_A = \frac{2}{3}p_i + \frac{1}{3}p_{i+1}$
6: Compute $p_B = \frac{1}{3}p_i + \frac{2}{3}p_{i+1}$
7: Compute peak p_C by rotating $(p_B - p_A)$ by $60°$
8: Append p_i, p_A, p_C, p_B to *new_points*
9: **end for**
10: Update points list with *new_points*
11: **end for**
12: Return final set of points

Koch Curve Algorithm

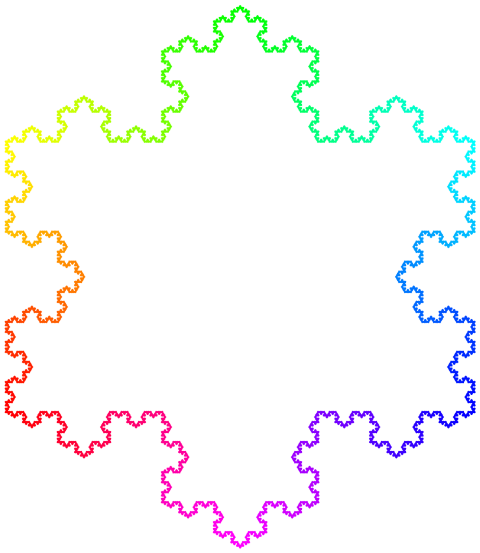
Algorithm 5 Generate Koch Curve

1: Initialize line segment (p_1, p_2)
2: **for** $n = 1$ to order **do**
3: Create empty list *new_points*
4: **for** each segment (p_i, p_{i+1}) **do**
5: Compute $p_A = \frac{2}{3}p_i + \frac{1}{3}p_{i+1}$
6: Compute $p_B = \frac{1}{3}p_i + \frac{2}{3}p_{i+1}$
7: Compute peak p_C by rotating $(p_B - p_A)$ by $60°$
8: Append p_i, p_A, p_C, p_B to *new_points*
9: **end for**
10: Update points list with *new_points*
11: **end for**
12: Return final set of points

Table 4: Comparison of Koch Snowflake and Koch Curve Algorithms



(a) Koch Curve



(b) Koch Snowflake

Figure 5: The Koch Curve and Koch Snowflake.

5.5 Sierpinski Triangle

The Sierpinski triangle is one of the best-known of all fractals, made by recursively subdividing an equilateral triangle into smaller triangles. The process starts off in a triangle, and in each operation, the middle triangle is discarded, leaving a set of three smaller triangles. This is repeated indefinitely, resulting in a complex, self-similar pattern on all scale levels. The Sierpinski triangle has been made using three algorithms, each of which emphasizes different aspects of fractal geometry.

Explanation of the Algorithms and the Fractal

The Sierpinski triangle can be constructed using a simple recursive procedure, in which a triangle is divided into three smaller triangles, and this procedure is iterated recursively a number of steps based on a desired depth. This procedure is shown in **Table 5**.

The second procedure makes use of the Chaos Game, in which randomly chosen points are mapped using given functions which pull them toward vertices of a triangle. This process is illustrated in **Table 6**.

The third process employs Pascal's Triangle in forming the Sierpinski Triangle, with mathematical properties regarding binomial coefficients utilized in its formation of its self-similarity. The process is outlined in **Table 7**

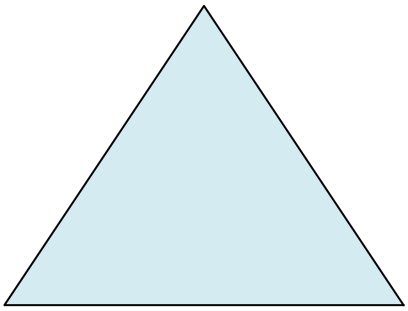
Each algorithm is a unique way in which a fractal is developed, a demonstration of the varied mathematical and computational methods that have been developed in order to create these immensely complex structures. The resulting image is a beautiful, captivating fractal with endless depth and symmetry, a beautiful illustration of mathematical recursive self-similarity.

Sierpinski Triangle Algorithm

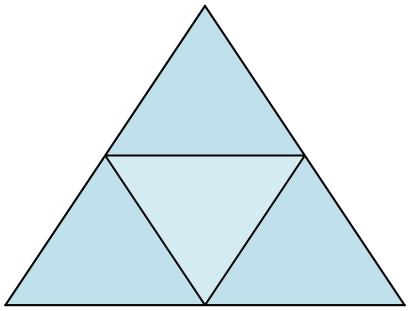
Algorithm 6 Generate and Plot Sierpinski Triangle

Function: Draw Triangle
Input: A list of three points $points = [(x_1, y_1), (x_2, y_2), (x_3, y_3)]$
Plot the triangle formed by the points
Fill the triangle with a light blue color and set transparency to 0.5
Function: Get Midpoint between two points
Input: Points $p_1(x_1, y_1)$ and $p_2(x_2, y_2)$
Output: Midpoint M with coordinates $M = \left(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}\right)$
Function: Generate Sierpinski Triangle of order n
Input: Points $points = [(x_1, y_1), (x_2, y_2), (x_3, y_3)]$ and degree n
Draw the triangle using *points*
if $n > 0$ **then**
    Compute midpoints for each edge of the triangle
    Recursively call *sierpinski* for each of the three sub-triangles formed by the midpoints, with degree $n - 1$
end if
Call *sierpinski* function with initial points and degree n
Display the plot

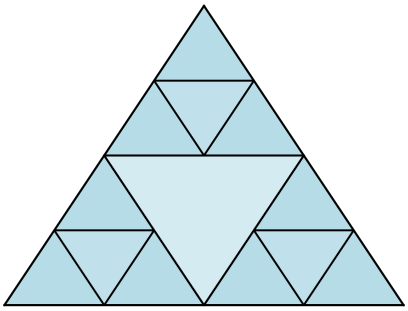
Table 5: Sierpinski Triangle Algorithm



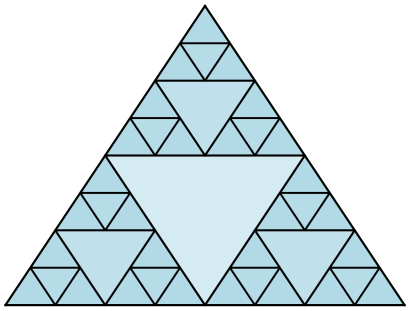
(a) Iteration 0



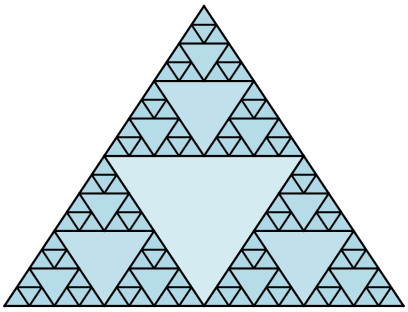
(b) Iteration 1



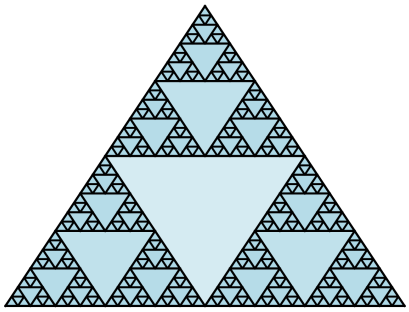
(c) Iteration 2



(d) Iteration 3



(e) Iteration 4



(f) Iteration 5

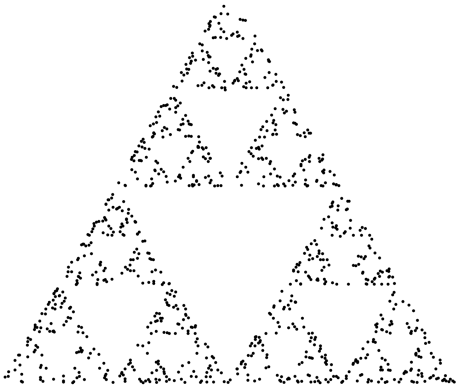
Figure 6: Sierpinski Triangle - Different Iterations

Sierpinski Triangle Algorithm (Chaos Game + IFS)

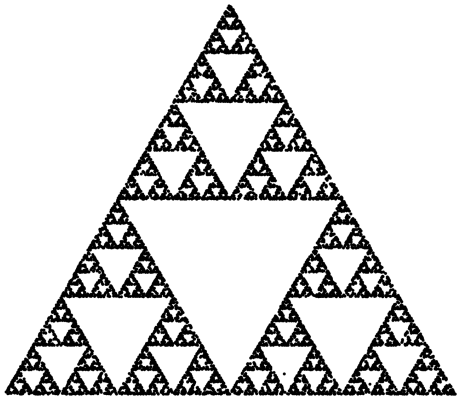
Algorithm 7 Generate Sierpinski Triangle using Chaos Game and IFS

Define three transformation functions:
    $f_1(x, y) = \left(\frac{x}{2}, \frac{y}{2}\right)$ ▷ Towards vertex (0, 0)
    $f_2(x, y) = \left(\frac{x+1}{2}, \frac{y}{2}\right)$ ▷ Towards vertex (1, 0)
    $f_3(x, y) = \left(\frac{x+0.5}{2}, \frac{y+\frac{\sqrt{3}}{2}}{2}\right)$ ▷ Towards vertex (0.5, $\sqrt{3}/2$)
Initialize the list of transformations $transformations = [f_1, f_2, f_3]$
Initialize random starting point (x, y) where $x, y \in [0, 1]$
Initialize an empty list *points* and add initial point (x, y) to *points*
for $i = 1$ to *iterations* **do**
    Randomly choose a transformation $transform \in transformations$
    Apply the transformation: $(x, y) = transform(x, y)$
    Append the new point (x, y) to *points*
end for
Unzip the points list into x_vals, y_vals
Plot the points (x_vals, y_vals) using a scatter plot with black color

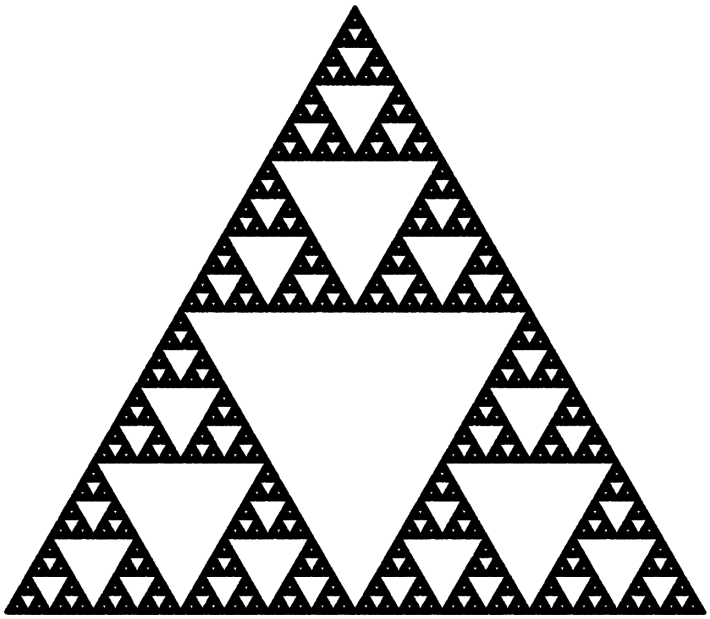
Table 6: Sierpinski Triangle Algorithm using Chaos Game and IFS



(a) Iteration 1000



(b) Iteration 10000



(c) Iteration 100000

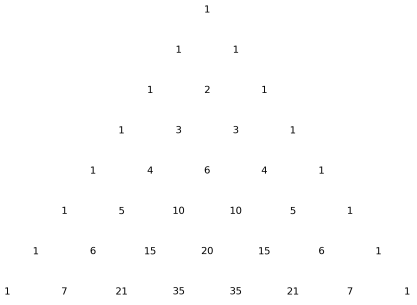
Figure 7: Sierpinski Triangle - Different Iterations using Chaos Game and IFS

Sierpinski Triangle Algorithm using Pascal's Triangle

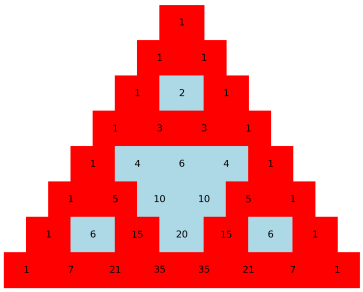
Algorithm 8 Generate Sierpinski Triangle using Pascal's Triangle

Function: Generate Pascal's Triangle Numbers
Input: Number of rows n
Initialize an empty list *triangle*
for $i = 0$ to $n - 1$ **do**
    Create a row of size $i + 1$ filled with 1's
    for $j = 1$ to $i - 1$ **do**
        Set $row[j] = triangle[i - 1][j - 1] + triangle[i - 1][j]$
    end for
    Append the row to *triangle*
end for
Output: *triangle*
Function: Plot Pascal's Triangle
Input: Pascal's triangle list *pascal_triangle*, flag to show numbers *show_numbers*, save path *save_path*, DPI *dpi*, flag to draw rectangles *draw_rectangles*
Create a new figure and axes using Matplotlib
Hide the axis
Calculate the number of rows and columns in the Pascal's triangle
for each row in *pascal_triangle* **do**
    for each value in the row **do**
        Calculate the x and y positions for the rectangle to represent the value
        if *draw_rectangles* is True **then**
            Draw a rectangle with a color based on the value (red for odd, light blue for even)
        end if
        if *show_numbers* is True **then**
            Display the Pascal number in the center of the rectangle
        end if
    end for
end for
Show the plot

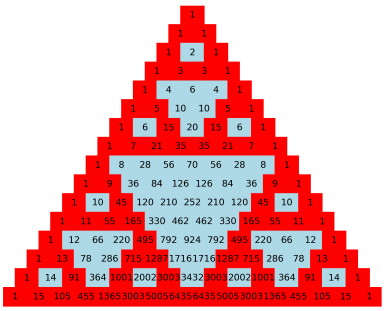
Table 7: Sierpinski Triangle Algorithm using Pascal's Triangle



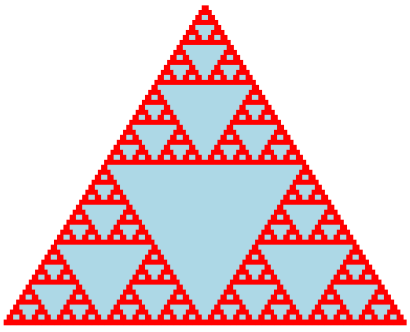
(a) Pascal's Triangle



(b) Connecting the odd numbers generates the Sierpinski triangle



(c) The Sierpinski triangle inside a larger Pascal's triangle



(d) Painted on Pascal's triangle of size 64

Figure 8: Sierpinski Triangle - Different Iterations using Pascal's Triangle