Chapter 1

Iterated Function Systems

Insanity is doing the same thing
over and over again
and expecting different results.
— Anonymous[1]

Modern computers are remarkably powerful devices, capable of storing millions of pages of text and executing tens of billions of instructions per second, all while being small enough to fit in the palm of your hand.

The key to harnessing all this power is ***repetition***, the seemingly simple idea of performing a sequence of operations multiple times. As a simple example, suppose we want to print a million integers starting from zero. Every programmer knows how to do this: Use a loop to count from 1 to 1 000 000 and output the successive values of the loop variable `i`:

```
for (int i = 1; i <= 1_000_000; i++) {
    System.out.println(i);
}
```

This form of repetition is known as ***iteration***, and the example demonstrates its two main ingredients: Every iterative procedure repeatedly performs a sequence of operations (here the instructions inside the loop) while updating a set of variables with each repetition (here the loop variable `i`). It's the ability to update variables that distinguishes iteration from mindless repetition and allows us to do *slightly different things* over and over again.

[1] The quote is often mistakenly attributed to Albert Einstein; see [79].

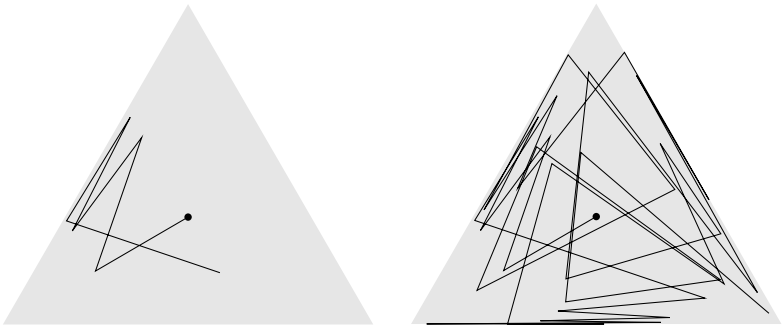


Figure 1.1 The first steps of the chaos game. The game starts with a single point in the center of the triangle and then repeatedly moves it halfway towards one of the corners chosen at random. The image on the left shows the first six points generated in one particular run of this game, and the image on the right the first 50 steps.

1.1 The Chaos Game

Iteration is so commonplace that we rarely give it the recognition it deserves — what could be more ordinary than a loop that iterates over a sequence of integers or the elements of a list? But it's not hard to find examples of simple iterative processes that show startlingly complex behavior. One such algorithm is the ***chaos game***, a simple procedure for constructing a sequence of points in the plane.

The chaos game simulates the movement of a point in the plane. We start with a point somewhere inside an equilateral triangle, for example in its center, and then repeatedly move it halfway toward a randomly chosen corner. The trajectory of the point depends on the order in which the corners are chosen. Figure 1.1 shows the first steps from one particular run of the chaos game. In the left part of the image, the point starts its journey by moving alternately towards the lower-left and upper corners and then takes a final step towards the lower-right corner. Afterward, as shown in the right part of the image, the point continues to move erratically inside the triangle.

You might expect that the points generated by the chaos game will eventually cover the entire triangle more or less evenly. This is not what happens, however. Figure 1.2 shows the point distributions after 50 000 and 200 000 iterations. As you can see, the points produced by the chaos game are distributed unevenly and converge toward a shape that looks like a triangle from which a number of successively smaller triangles have been cut out. There is a single large empty triangle in the center, which is surrounded by three smaller ones. Each of these smaller triangles is again surrounded by three smaller triangles, and the pattern continues indefinitely. The longer we let the chaos game run, the more closely the resulting set of points approximates this unexpected limiting shape.

This curious shape — known as the ***Sierpiński triangle*** — is an example of a ***fractal***. Fractals are geometric objects that share two main properties: They have a "fractured" appearance that is irregular but not random, and they can be magnified

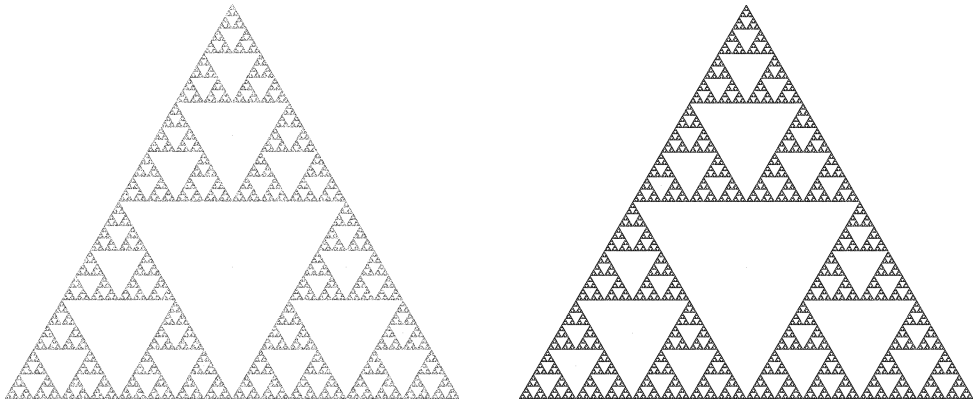


Figure 1.2 Visualization of the first 50 000 and 200 2000 points computed by the chaos game. As the number of iteration increases, the point set slowly converges toward a fractal called the Sierpiński triangle.

indefinitely, revealing new details at every level of magnification. In addition, fractals are often ***self-similar***, which means that their parts resemble the shape as a whole. The Sierpiński triangle clearly has all three properties: Its geometric structure isn't random but it's also not as regular as a simple polygon or circle, and it can be magnified indefinitely, with new copies of the Sierpiński triangle appearing at every level of magnification.

The chaos game we used to construct the Sierpiński triangle is a simple example of an ***algorithm***, a formal description of how to break down a computation into a sequence of elementary steps. To describe algorithms, we will use the format shown in Algorithm 1.1. In this case, the algorithm consists of a single function named ChaosGame that takes two inputs $\boldsymbol{s}$ and n, the starting point and the number of iterations to perform. The numbered lines that follow explain the computational steps performed by the algorithm.

Algorithm 1.1

Given a starting point $\boldsymbol{s}$, produce a sequence of n points in the plane.

ChaosGame($\boldsymbol{s}, n$) =
1 **let** $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3$ = corners of an equilateral triangle
2 **var** $\boldsymbol{p} = \boldsymbol{s}$
3 **repeat** n times
4 **output** $\boldsymbol{p}$
5 **let** k = random integer between 1 and 3
6 $\boldsymbol{p} = (\boldsymbol{p} + \boldsymbol{c}_k)/2$

Let's take a closer look at the steps of Algorithm 1.1. Line 1 defines three variables $\boldsymbol{c}_1$ to $\boldsymbol{c}_3$ that represent the corners of an equilateral triangle; we assume that these corners are already known or that they are computed using a method

not specified in the algorithm. We use the **let** keyword to define read-only variables that don't change over the course of the algorithm. Line 2 defines a new variable $\boldsymbol{p}$ for the current position and initializes it to the starting point $\boldsymbol{s}$. This variable is defined using the **var** keyword, which indicates that the value of $\boldsymbol{p}$ is modified as the algorithm progresses. The **repeat** loop on the following line repeats the indented lines in its body n times. It starts with an **output** statement that outputs the current value of $\boldsymbol{p}$. Line 5 then sets the constant k to a random integer between 1 and 3, and line 6 moves $\boldsymbol{p}$ toward the kth corner. The loop is then repeated $n-1$ more times. Since **output** is executed n times, the algorithm produces n points.

> Mathematicians usually distinguish between ***points*** and ***vectors***: Points represent *locations* in space, whereas a vectors represent *directions*. However, for simple geometric problems in the plane, the two concepts are closely related: Every point P with coordinates (x, y) can be represented by the ***position vector*** $\boldsymbol{p} = \binom{x}{y}$ that points from the origin to P, and vice versa.
>
> For the geometric problems discussed in this book, we will keep things simple and use points and positions vectors interchangeably. Boldface symbols like $\boldsymbol{p}$ and $\boldsymbol{q}$ are always vectors, and if we say something like "the point $\boldsymbol{s}$," what we really mean is "the position vector $\boldsymbol{s}$."

Implementing the Chaos Game

We can now turn to the fun part: implementing the chaos game so that it runs on a real computer and produces real images. In most cases, turning an algorithm into a working program is more complicated than translating it step by step. For instance, we may have to make changes based on the chosen programming language, flesh out details that weren't specified precisely, or integrate the algorithm with other parts of a larger program. Here are some of the technical details we have to consider when implementing the ChaosGame algorithm:

1. How do we store points and move them?
2. How do we compute the three corners of the outer triangle?
3. How do we create random numbers between 1 and 3?
4. What does it mean to **output** a point?

Let's address these questions one by one.

We start by defining a custom data type called `Point` that allows us to store and compute with points in the plane. What kinds of operations do we require in `Point`? Obviously, we need a way to construct points at certain positions in the plane, so that we can specify the starting point of the chaos game and the three corners that enclose the fractal. To store a point's position we use ***Cartesian coordinates*** (x, y), which measure the point's position along the horizontal and vertical axis. The `Point` type defined in Listing 1.1 stores these coordinates in two fields `x` and `y`.

`Point` **1.1**

Listing 1.1 ch1/Point

```
1  // A point in 2D.
2  public record Point(double x, double y) {
3      // ...
4  }
```

To implement the chaos game, we also need a way to move the current point halfway toward one of the triangle's corners, that is, we need to compute the ***midpoint*** of two points. The easiest way to do this is to average their coordinates, as shown here for the points p and q:

```
Point midPoint = new Point(
        0.5 * (p.x() + q.x()), 0.5 * (p.y() + q.y()));
```

This is a special case of a more general operation that computes points along the line segment between two arbitrary points $\boldsymbol{p}$ and $\boldsymbol{q}$. In computer graphics, the point that lies some fraction t between $\boldsymbol{q}$ and $\boldsymbol{q}$ is commonly denoted by $\text{lerp}(\boldsymbol{p}, \boldsymbol{q}, t)$.[2] Three special cases of this ***lerp function*** are worth noting: For $t = 0$ we obtain the first end point $\boldsymbol{p}$, for $t = 1$ the second end point $\boldsymbol{q}$, and for $t = 1/2$ their midpoint. The function is illustrated in Fig. 1.3.

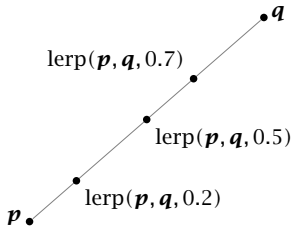


Figure 1.3 The lerp function computes a point that lies on the line segment between two end points.

There are two equivalent formulas for defining the lerp function. The first is obtained by starting at the first point $\boldsymbol{p}$ and then advancing along the line to $\boldsymbol{q}$ by adding the difference vector $\boldsymbol{q} - \boldsymbol{p}$ scaled by t:

$$\text{lerp}(\boldsymbol{p}, \boldsymbol{q}, t) = \boldsymbol{p} + t(\boldsymbol{q} - \boldsymbol{p}).$$

The second formula has a more algebraic flavor and expresses the lerp function as the weighted average of $\boldsymbol{p}$ and $\boldsymbol{q}$:

$$\text{lerp}(\boldsymbol{p}, \boldsymbol{q}, t) = (1 - t)\boldsymbol{p} + t\boldsymbol{q}. \tag{1.1}$$

Both definitions can be transformed into each other by reorganizing the terms on the right-hand side. Even though the first definition is slightly more intuitive, we

Point 1.1
Point() 1.1
x() 1.1
y() 1.1

[2] The name "lerp" is a contraction of "linear interpolation."

will use the second one since it is more symmetrical and slightly easier to generalize to other forms of interpolation; the implementation is shown in Listing 1.2.

Listing 1.2 ch1/Point

```
// Compute the linear interpolation of two points.
public static Point lerp(Point p, Point q, double fraction) {
    return new Point((1 - fraction) * p.x + fraction * q.x,
            (1 - fraction) * p.y + fraction * q.y);
}
```

It's now easy to implement the chaos game itself. The `chaosGame()` function in Listing 1.3 takes three arguments — a list of corners, a starting point, and the number of iterations that should be performed — and returns the sequence of points it visits as a list of `Points`. The function first creates a random number generator `rng` that is used to select one of the corners in each iteration and then allocates a data structure for the return value, in this case an `ArrayList` of `Points`.[3] The actual algorithm is implemented by the `for` loop that follows, which stores the current point in `points` and then moves it towards a randomly chosen corner.

Listing 1.3 ch1/ChaosGame

```
public static List<Point> chaosGame(List<Point> corners,
        Point start, int iterations) {
    var rng = new Random();
    var points = new ArrayList<Point>(iterations);
    Point p = start;
    for (int i = 0; i < iterations; i++) {
        points.add(p);
        Point c = corners.get(rng.nextInt(corners.size()));
        p = Point.lerp(p, c, 0.5);
    }
    return points;
}
```

If we want to create an image of the Sierpiński triangle like the one in Fig. 1.2, we have to simulate a few thousand iterations of the chaos game and then draw the resulting points. One way to do this is shown in Listing 1.4. We first construct a large equilateral triangle with a side length of 1000 and then call `chaosGame()` to compute 50 000 points inside this triangle, using the center of the triangle as the starting point. Finally, we draw each of the resulting points as a small, filled circle. The `Graphics2D` type used in this listing is the standard interface provided by Java for producing 2D graphics; Exercise 1.7 on page 19 asks you to complete this program by setting up a suitable instance of `Graphics2D`.

ArrayList →
ChaosGame
  chaosGame() 1.3
  drawSierpinski() **1.4**
Graphics2D →
Point **1.1**
  Point() **1.1**
  lerp() 1.2
Random →

You may be wondering why `drawSierpinski()` removes the first 50 elements of `points` before drawing them. The reason is that for some starting points the

[3] In Java, an `ArrayList` is a special type of `List`, so the code is correct even though the return type of `chaosGame()` doesn't match the type of `points` exactly.

Listing 1.4 ch1/ChaosGame

```
public static void drawSierpinski(Graphics2D g) {
    double sideLength = 1000;
    var corners = List.of(
            new Point(0, 0),
            new Point(sideLength, 0),
            new Point(sideLength / 2, sideLength * Math.sqrt(3) / 2));
    Point start = new Point(
            sideLength / 2, sideLength / Math.sqrt(3) / 2);
    List<Point> points = chaosGame(corners, start, 50_000);

    float radius = 0.01f;
    for (Point p : points.subList(50, points.size()))
        g.fill(new Ellipse2D.Double(p.x(), p.y(), radius, radius));
}
```

chaos game may require a few iterations to "warm up" and produce points that are sufficiently close to the actual fractal. In the present example, for instance, we start at the center of the triangle, which isn't part of the Sierpiński triangle. But with every iteration, the point moves closer to the final fractal until it is imperceptibly close. Discarding the first few points produced by the chaos game is enough to get rid of the obvious outliers and obtain clean-looking images.

Exercises

Exercise 1.1. Given two points $\boldsymbol{p}$ and $\boldsymbol{q}$, what is the geometric interpretation of $\operatorname{lerp}(\boldsymbol{p}, \boldsymbol{q}, -1)$?

Exercise 1.2. Extend the `Point` class with a method `distance()` that computes the distance between two points.

Exercise 1.3. What happens if you place the starting point of the chaos game outside of the triangle defined by the three corners? What happens if you move the corners of the triangle so that it is no longer equilateral?

1.2 Generalizing the Chaos Game

The chaos game as we have discussed it in the previous section can be generalized to a more powerful algorithm for generating fractals. As before, we simulate the movement of a point in the plane, but instead of moving the point toward a randomly chosen corner of a triangle, we instead transform it using a randomly chosen function. For reasons we will explain in a moment, we usually associate with each function f_k a probability p_k, which determines how frequently the chaos game chooses this function relative to the others. We call the combination of functions and their probabilities an ***iterated function system*** or ***IFS***.

ChaosGame
  chaosGame() **1.3**
  drawSierpinski() 1.4
Ellipse2D →
Graphics2D →
List →
Math →
Point **1.1**
  Point() **1.1**

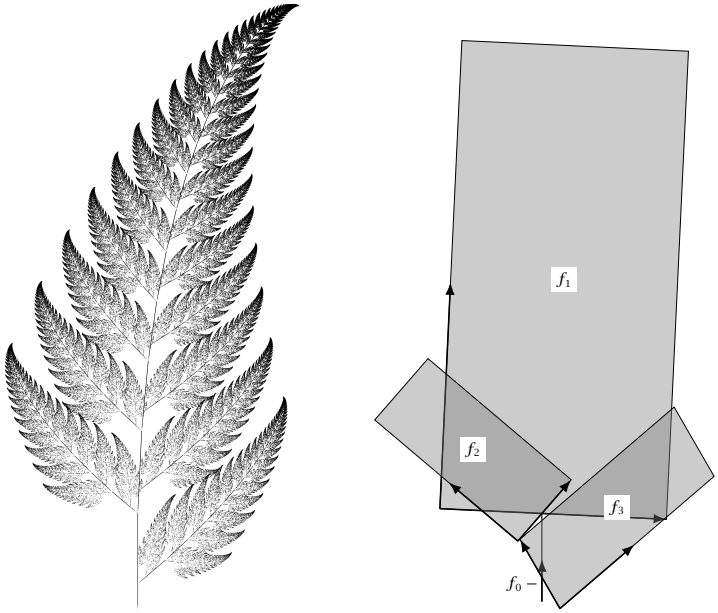


Figure 1.4 The Barnsley fern is a plant-like fractal that consists of countless smaller copies of itself (left). It is generated by four affine transformations, each of which corresponds to one part of the fractal (right).

It's easy to see that this is a proper generalization: If $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3$ are the corners of a triangle, we can define three functions that move a given point $\boldsymbol{p}$ halfway toward each of the corners:

$$f_1(\boldsymbol{p}) = (\boldsymbol{p} + \boldsymbol{c}_1)/2, \qquad f_2(\boldsymbol{p}) = (\boldsymbol{p} + \boldsymbol{c}_2)/2, \qquad f_3(\boldsymbol{p}) = (\boldsymbol{p} + \boldsymbol{c}_3)/2.$$

If we start with a point inside the triangle and repeatedly transform it using a randomly chosen function from the set $\{f_1, f_2, f_3\}$, we end up with a random process that is equivalent to the chaos game from the previous section. In the case of the Sierpiński triangle, all three corners are chosen with the same probability, so we set $p_k = 1/3$. When visualizing fractals that are less symmetric, these probabilities can be used to control the distribution of the points produced by the chaos game.

The Barnsley Fern

A famous example of a fractal based on iterated function systems is the ***Barnsley fern*** shown in Fig. 1.4. The fern consists of a stem that bends to the right and an infinite number of branches and leaves that are attached to the left and the right of the stem. Each branch is itself a scaled and rotated version of the Barnsley fern, as are the leaves of each branch: If you zoom into any part of the Barnsley fern, you find an endless number of smaller copies of the whole shape.

The Barnsley fern is generated by an iterated function system that consists of four functions and four probabilities:

$$f_0(\boldsymbol{p}) = \begin{pmatrix} 0 & 0 \\ 0 & 0.16 \end{pmatrix} \boldsymbol{p} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \qquad p_0 = 0.01$$

$$f_1(\boldsymbol{p}) = \begin{pmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{pmatrix} \boldsymbol{p} + \begin{pmatrix} 0 \\ 1.6 \end{pmatrix} \qquad p_1 = 0.85$$

$$f_2(\boldsymbol{p}) = \begin{pmatrix} 0.2 & -0.26 \\ 0.23 & 0.22 \end{pmatrix} \boldsymbol{p} + \begin{pmatrix} 0 \\ 1.6 \end{pmatrix} \qquad p_2 = 0.07$$

$$f_3(\boldsymbol{p}) = \begin{pmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{pmatrix} \boldsymbol{p} + \begin{pmatrix} 0 \\ 0.44 \end{pmatrix} \qquad p_3 = 0.07$$

All four functions have the same form: They first multiply the vector $\boldsymbol{p}$ by a 2×2 matrix and then translate the result by adding a vector. Functions of this form are called ***affine transformations***, and they describe a wide range of geometric transformations such as rotation, scaling, shearing, and translation; see also Box 1.1 on the next page.

The right part of Fig. 1.4 illustrates how these four affine transformations determine the shape and structure of the Barnsley fern. Imagine a rectangle that encloses the Barnsley fern and then transform this bounding box using each of the four transformations. The first transformation f_0 flattens the box into a short vertical line that forms the stem of the fern; f_1 moves a slightly scaled and rotated version of the bounding box to the end of the stem and generates the upper part of the fern; and the last two transformations f_2 and f_3 form the first leaf on the left and right of the fern, respectively. Notice that f_3 also flips the coordinate system and introduces a slight shear, so the leaves on the right side of the fern bend to the left and are slightly skewed. Smaller details of the fern are obtained by applying multiple functions in sequence. For instance, applying f_2 followed by f_0 produces the stem of the left branch, and applying f_1 followed by f_3 the second branch on the right. Every part of the fractal corresponds to a particular combination of the four functions in its iterated function system; you can think of the four functions as the fractal's geometric blueprint.

This geometric blueprint is filled with a random distribution of points using the chaos game. The probabilities $p_1, \ldots, p_3$ control *how* the points are distributed over the different parts of the fractal: In the case of the Barnsley fern, the probabilities are 0.01, 0.85, 0.07, and 0.07, which means that approximately 1% of all points will be placed in the fern's stem (or one of its many copies), 85% in its body, and 7% in the left and right leaves each. The four probabilities sum to 1, which reflects the fact that every point is allocated to one of the available transformations.

It is instructive to consider what happens if we choose other values for the probabilities. If we set all probabilities to $1/4$, for instance, each part of the fractal receives the same number of points, and we obtain the rendering of the Barnsley fern shown in the left part of Fig. 1.5. With this choice of probabilities, most points end up in or close to the stem of the Barnsley fern and fewer points in its leaves. If, on the other hand, we set the four probabilities to 0.01, 0.94, 0.025, and 0.025,

Box 1.1: Linear and Affine Transformations

Many important geometric transformations can be described mathematically using matrix operations. For example, scaling vectors by a constant factor α

$$\boldsymbol{p} \quad \mapsto \quad \alpha \boldsymbol{p}$$

can also be written as the matrix product

$$\boldsymbol{p} \quad \mapsto \quad \begin{pmatrix} \alpha & 0 \\ 0 & \alpha \end{pmatrix} \boldsymbol{p},$$

and similar matrix versions exist for other transformations such as *rotation* around the origin, *shearing*, and *reflection*. In general, a mapping that can be described by a matrix multiplication is called a *linear transformation*:

$$\boldsymbol{p} \quad \mapsto \quad \boldsymbol{A}\boldsymbol{p} \qquad \text{or} \qquad \begin{pmatrix} x \\ y \end{pmatrix} \quad \mapsto \quad \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}.$$

An important generalization of linear transformations are *affine transformations*, which first perform a matrix multiplication and then add a fixed vector $\boldsymbol{t}$:

$$\boldsymbol{p} \quad \mapsto \quad \boldsymbol{A}\boldsymbol{p} + \boldsymbol{t} \qquad \text{or} \qquad \begin{pmatrix} x \\ y \end{pmatrix} \quad \mapsto \quad \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} = \begin{pmatrix} ax + by + t_x \\ cx + dy + t_y \end{pmatrix}.$$

Affine transformations let us rotate, scale, and shear as before, but they can also be used to move geometric objects in the plane. We will discuss affine transformations in more detail in Section 3.2.

most points are allocated to the function f_1, which increases the density of points in the tips of the branches, as illustrated in the right part of Fig. 1.5.

In practice, suitable values for the probabilities must usually be determined by trial and error, especially if the primary goal is to produce pleasing images. But it is possible to compute reasonable starting values so that more points end up in parts of the fractal that cover a large surface area; we will discuss one such an approach in Exercise 1.5.

Affine Iterated Function Systems

An affine transformation is a function that consists of a linear transformation — usually a combination of rotation, scaling, and shearing — followed by a translation. We usually write affine transformations in the following form:

$$\boldsymbol{p} \quad \mapsto \quad \begin{pmatrix} a & b \\ c & d \end{pmatrix} \boldsymbol{p} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}. \tag{1.2}$$

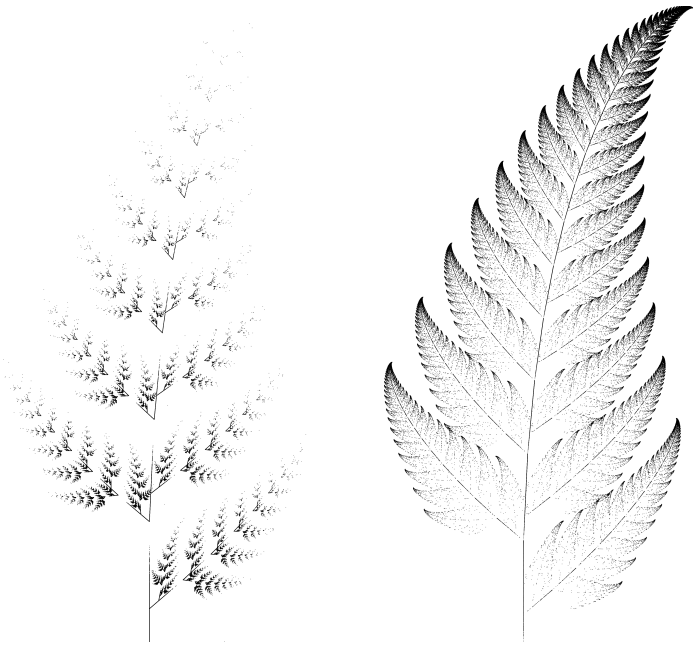


Figure 1.5 The influence of IFS probabilities. The probabilities associated with an iterated function system determine the spatial distribution of points produced by the chaos game. Decreasing the probability of the stem places more points in the leaves (left) whereas increasing it emphasizes the stem but places fewer points in the leaves (right).

where $\boldsymbol{p}$ is the point being transformed. The transform is completely defined by six numbers: the four coefficients a, b, c, d of the 2×2 matrix and the two coefficients t_x, t_y of the translation vector. To implement affine transformations, we define a record called `Affine` that stores these six numbers as floating-point numbers (Listing 1.5).

Listing 1.5 ch1/Affine

```
// An affine transformation in 2D.
public record Affine(
        double a, double b,
        double c, double d,
        double tx, double ty) {
    // ...
}
```

If we set $\boldsymbol{p} = \begin{pmatrix} x \\ y \end{pmatrix}$ and expand the matrix product in Eq. (1.2), we obtain the coordinates x' and y' of the transformed point:

$$\begin{aligned} x' &= ax + by + t_x, \\ y' &= cx + dy + t_y. \end{aligned} \tag{1.3}$$

Affine 1.5
Affine() 1.5

The `transformPoint()` method in Listing 1.6 uses this formula to apply an affine transformation to a given `Point`, producing a new `Point`. Since we will often work with lists of points, we also define a second method `transformPoints()` that transforms multiple points and returns the result as a new list. That's all the functionality we need for now, although we will extend `Affine` with few additional methods in Chapter 3.

Listing 1.6 ch1 / Affine

```
// Transform a single point.
public Point transformPoint(Point p) {
    return new Point(a * p.x() + b * p.y() + tx,
            c * p.x() + d * p.y() + ty);
}

// Transform a list of points.
public List<Point> transformPoints(List<Point> points) {
    var result = new ArrayList<Point>(points.size());
    for (Point p : points)
        result.add(transformPoint(p));
    return result;
}
```

We can now represent an affine iterated function system as an array of affine transformations of type `Affine` together with an array of probabilities of type `double` (Listing 1.7). The constructor of `AffineIFS` initializes `transforms` using the argument of the same name, but for `probabilities` it performs an additional normalization step to ensure that its entries sum to 1. Normalizing the probabilities ensures that exactly one of the affine transformations is chosen by the chaos game in each step. The second argument of `AffineIFS()` is therefore simply an array of *frequencies* that indicate how often each affine transformation is chosen relative to the others, which is then converted into a table of *probabilities* by dividing each entry by `total`, the sum of all frequencies.

Listing 1.8 demonstrates how to construct an instance of `AffineIFS` that represents the Barnsley fern. In this case, the probabilities in the second array are already normalized.

Affine **1.5**
  transformPoint() 1.6
  transformPoints() 1.6
AffineIFS **1.7**
  AffineIFS() **1.7**
  probabilities **1.7**
  transforms **1.7**
ArrayList →
List →
Point **1.1**

Listing 1.7 ch1/AffineIFS

```
1  // Representation of an affine iterated function system.
2  public class AffineIFS {
3      public final Affine[] transforms;
4      public final double[] probabilities;
5
6      public AffineIFS(Affine[] transforms, double[] frequencies) {
7          this.transforms = transforms;
8          this.probabilities = new double[frequencies.length];
9          double total = 0.0;
10         for (double freq : frequencies)
11             total += freq;
12         for (int i = 0; i < frequencies.length; i++)
13             this.probabilities[i] = frequencies[i] / total;
14     }
15
16     // ...
17 }
```

Listing 1.8 ch1/AffineIFS

```
1  public static AffineIFS BARNSLEY_FERN = new AffineIFS(
2          new Affine[]{
3                  new Affine(0, 0, 0, 0.16, 0, 0),
4                  new Affine(0.85, 0.04, -0.04, 0.85, 0, 1.6),
5                  new Affine(0.2, -0.26, 0.23, 0.22, 0, 1.6),
6                  new Affine(-0.15, 0.28, 0.26, 0.24, 0, 0.44),
7          },
8          new double[]{0.01, 0.85, 0.07, 0.07}
9  );
```

The Extended Chaos Game

We can now turn to the problem of implementing the generalized version of the chaos game that lets us visualize the fractals described by affine iterated function systems. The formal description of this algorithm is shown in Algorithm 1.2. The main difference compared to the original chaos game in Algorithm 1.1 is that the algorithm now uses a fixed set of functions $f_1, \ldots, f_K$ and associated probabilities $p_1, \ldots, p_K$ to move the current point in each step. In Line 4, we choose one of the functions by calling a function ChooseIndex, which returns an integer k between 1 and K, and in Line 5, we move the current point $\boldsymbol{p}$ to its new position by transforming it using the kth function f_k.

The job of the ChooseIndex function referenced in Algorithm 1.2 is to take a sequence of probabilities $p_1, \ldots, p_K$ and generate a random index between 1 and K so that the integer k occurs with probability p_k (in other words, it returns the number 1 with probability p_1, the number 2 with probability p_2, and so on). How can we implement ChooseIndex?

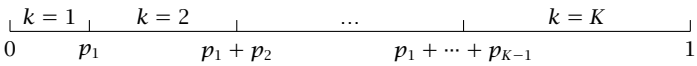
Algorithm 1.2

Given a starting point $\boldsymbol{s}$, produce a sequence of n points in the plane. The variables $f_1, f_2, \ldots f_K$ are affine transformations and $p_1, p_2, \ldots p_K$ the associated probabilities.

EXTENDEDCHAOSGAME($\boldsymbol{s}, n$) =

```
1   var p = s
2   repeat n times
3       output p
4       let k = CHOOSEINDEX(p_1, ..., p_K)
5       p = f_k(p)
```

One solution is to approach the problem geometrically. If the probabilities are normalized, we have $p_1 + \cdots + p_K = 1$, which we can visualize by subdividing the interval $[0, 1]$ into K sub-intervals of length $p_1, p_2, \ldots$:



The kth interval starts at $p_1 + \cdots + p_{k-1}$ and ends at $p_1 + \cdots + p_k$ and therefore has length p_k. If we now pick a random real number u between 0 and 1, the probability that u lands in any particular sub-interval is obviously equal to the length of that interval, and the probability that it lands in the kth interval is p_k. The index of the interval that contains u is therefore an integer between 1 and K with the desired probability distribution.

To find the interval that contains u we can simply try them one after another. If $u < p_1$, the number lies in the leftmost interval with index $k = 1$; if $u < p_0 + p_1$, it lies in the second interval with index $k = 1$; otherwise, we continue in this manner until we find the first index k such that $u < p_0 + p_1 + \cdots + p_k$. The resulting algorithm is shown in Algorithm 1.3. The variable s accumulates the probabilities so it always contains the upper limit of the current interval, and the smallest index for which $u < s$ is the desired random index.

The generalized version of the chaos game can now be implemented as shown in Listing 1.9. The `extendedChaosGame()` method simulates the movement of the point `start` over the specified number of `iterations` and returns a list of visited points. In each iteration, the function appends the current position to `points` and then advances to the next point applying one of the affine transformations chosen at random. The `chooseIndex()` method implements the CHOOSEINDEX algorithm: It uses the given random number generator to produce one floating point number between 0 and 1 and then searches for the interval that contains this number.

AffineIFS **1.7**
chooseIndex() **1.9**
extendedChaosGame() **1.9**

Algorithm 1.3

Generate a random integer with a given distribution. The return value is an integer between 1 and K, and the number k is returned with probability p_k.

CHOOSEINDEX$(p_1, \ldots, p_K)$ =

```
1   let u = random number between 0 and 1
2   var s = 0
3   for k = 1 to K
4       s = s + p_k
5       if u < s
6           return k
7   return K
```

Listing 1.9 ch1/AffineIFS

```
1  public List<Point> extendedChaosGame(
2          Point start, int iterations, Random rng) {
3      Point p = start;
4      var points = new ArrayList<Point>(iterations);
5      for (int i = 0; i < iterations; i++) {
6          points.add(p);
7          p = transforms[chooseIndex(rng)].transformPoint(p);
8      }
9      return points;
10 }
11
12 public int chooseIndex(Random rng) {
13     double u = rng.nextDouble();
14     double sum = 0.0;
15     for (int i = 0; i < probabilities.length - 1; i++) {
16         sum += probabilities[i];
17         if (u < sum)
18             return i;
19     }
20     return probabilities.length - 1;
21 }
```

Affine **1.5**
  transformPoint() **1.6**
AffineIFS **1.7**
  chooseIndex() 1.9
  extendedChaosGame() 1.9
ArrayList →
List →
Point **1.1**

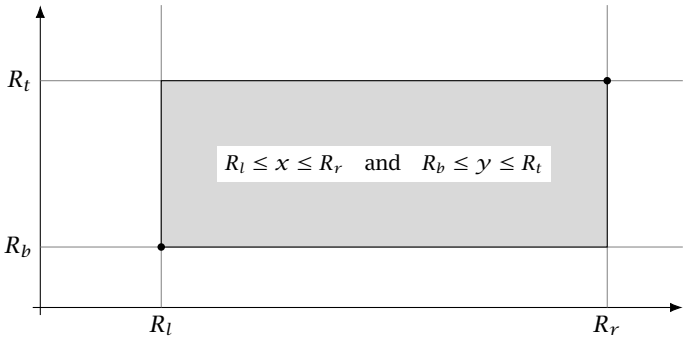


Figure 1.6 An axis-aligned rectangle.

1.3 Problems

Exercise 1.4. A simple geometric shape that is useful in many applications is the ***axis-aligned rectangle***, a rectangle whose sides are parallel to the coordinate axes. As shown in Fig. 1.6, each such a rectangle R is uniquely defined by the position of its four edges: the left and right edges R_l and R_r and the top and bottom edges R_t and R_b. The interior of such a rectangle consists of all points (x, y) that satisfy

$$R_l \le x \le R_r \qquad \text{and} \qquad R_b \le y \le R_t.$$

The rectangle is empty if there are no points that satisfy these relations; this is the case if either $R_l > R_r$ or $R_b > R_t$.

a) Given a set of points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$, how do you compute the smallest axis-aligned rectangle that contains all points?. (This rectangle is also referred to as the point set's ***bounding box***).

b) Given two axis-aligned rectangles A and B, how do you compute the smallest axis-aligned rectangle that encloses them both?

c) If two axis-aligned rectangles overlap, their intersection is also an axis-aligned rectangle. How do you compute its coordinates? The left (top) edge of the intersection must be the maximum of the two left (top) edges of the original rectangles, and the right (bottom) edge is the minimum of the two right (bottom) edges. These expressions also handles the special case in which the two rectangles don't overlap; in this case, the returned rectangle has a negative width or height and is therefore considered empty.

d) Listing 1.10 shows one possible way of defining a data type `Rectangle` for working with axis-aligned rectangles. Implement the missing methods.

`Rectangle` **C.2**

```
public record Rectangle(Point bottomLeft, Point topRight) {
    // The width and height of the rectangle.
    public int width();
    public int height();

    // Is the rectangle empty?
    public boolean isEmpty();

    // Determine whether 'p' lies inside the rectangle.
    public boolean contains(Point p);

    // Return intersection of 'this' and 'r'.
    public Rectangle intersection(Rectangle r);

    // The smallest rectangle that includes all points in the list.
    public static Rectangle boundingBox(List<Point> points);
}
```

Listing 1.10 Outline of a data type for representing axis-aligned rectangles.

Variations of the Chaos Game

Exercise 1.5. A simple heuristic for computing the probabilities associated with an affine IFS is based on ***determinants***. For a 2×2 matrix $\boldsymbol{A}$, the determinant $\det \boldsymbol{A}$ can be computed as follows:

$$\det \boldsymbol{A} = \det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ac - bd.$$

A fundamental result from linear algebra is that a linear transformation of the form

$$\boldsymbol{p} \mapsto \boldsymbol{A}\boldsymbol{p} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \boldsymbol{p}$$

scales the area of any shape by the factor $\det \boldsymbol{A}$. In the case of the Barnsley fern, for example, the areas of the four parallelograms shown in Fig. 1.4 are proportional to the determinants of the matrix parts of the corresponding functions. (If the determinant is negative, the transformation mirrors one of the coordinate axes; in this case, the *absolute value* of the determinant must be used.)

a) Compute the determinants of the four matrices that are used in the definition of the Barnsley fern. What do you observe?

b) Suggest a way to turn the values you obtain into probabilities that measure the footprint of each function.

Exercise 1.6. In the simplest version of the chaos game, which we discussed at the beginning of this chapter, the current point is always moved halfway toward a

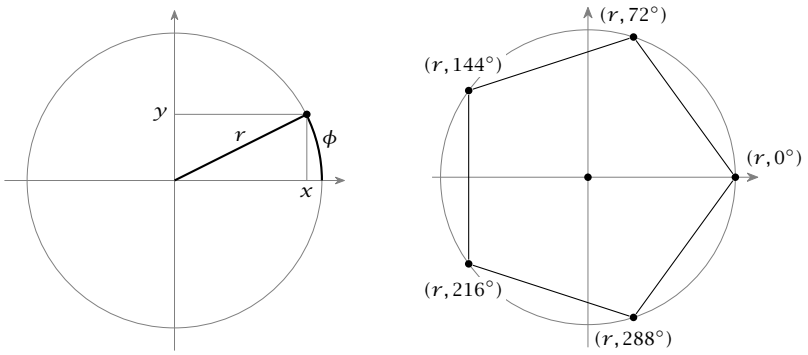
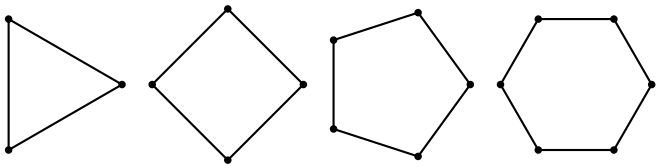


Figure 1.7 (Left) Points in the plane can be specified either using Cartesian coordinates x and y or polar coordinates, which indicate the distance from the origin r and the angle ϕ relative to the x axis. (Right) Polar coordinates make it easy to specify the corners of regular polygons.

random corner of a triangle. In this exercise we discuss a few interesting variations of this simple procedure.

The first variation is to replace the triangle with another regular polygon:



The easiest way to construct these polygons is to use ***polar coordinates***, which specify the position of a point using its distance r from the origin and the angle ϕ it makes with the x-axis, as illustrated in the left part of Fig. 1.7. Polar coordinates (r, ϕ) are related to the conventional Cartesian coordinates (x, y) by elementary trigonometry:

$$x = r\cos\phi, \qquad y = r\sin\phi. \tag{1.4}$$

a) Extend the `Point` type with a factory method called `polar()` that constructs a `Point` from its polar coordinates.

b) Explain how to perform the opposite conversion, from Cartesian to polar coordinates and implement two methods `Point.radius()` and `Point.angle()` that return a `Point`'s polar coordinates. *Hint:* to find the formula for ϕ, consider the ratio y/x.

The right part of Fig. 1.7 illustrates how polar coordinates can be used to compute the corners of a regular n-gon that is centered on the origin. The n corners of the polygon all have the same radius, whereas the angles are $360°/n$ degrees apart.

`Point` **1.1**
`angle()` **C.4**
`radius()` **C.4**

c) Implement a version of the chaos game that uses a regular n-gon as its base shape. Again, every step of the game should move the current point halfway towards a randomly chosen corners.

Another simple variation is to use a different rule for moving the current point. Instead of moving it *halfway* toward a corner, we can move it by a by a certain fraction α:

$$\boldsymbol{p} \quad \mapsto \quad \operatorname{lerp}(\boldsymbol{p}, \boldsymbol{c}_i, \alpha).$$

d) Experiment with different n-gons and different values of α and draw the resulting point sets. For which values of α does this version of the chaos game produce fractals?

Exercise 1.7. In Listing 1.4 we showed a simplified program for drawing the point sets produced by the chaos game. To complete this program, we need to create a suitable instance of `Graphics2D`, which is usually either a window on the screen or a bitmap image of type `BufferedImage`, and we need to scale and move the coordinates of the points so that they fit inside the window or image we are drawing into.

a) Write a program takes point sets generated by the chaos game and draws them, properly scaled, either to the screen or to a bitmap image.

b) Table 1.1 shows the parameters of four affine iterated function systems. Use the chaos game and the program developed in part (a) to generate images of the corresponding fractals.

Blending Iterated Function Systems

Exercise 1.8. Figure 1.8 shows a few frames from an animation that smoothly transforms the Barnsley fern on the left into the tree-shaped fractal on the right. The intermediate fractals are obtained by mixing the iterated function systems of the first and last fractal and are rendered using the standard chaos game. In this exercise we discuss how this is done.

Let's start with two arbitrary affine transformations of the form $f(\boldsymbol{p}) = \boldsymbol{F}\boldsymbol{p} + \boldsymbol{f}$ and $g(\boldsymbol{p}) = \boldsymbol{G}\boldsymbol{p} + \boldsymbol{g}$ and define a new function h that combines f and g using linear interpolation:

$$h(\boldsymbol{p}; t) = \operatorname{lerp}(f(\boldsymbol{p}), g(\boldsymbol{p}), t)$$

The parameter t is assumed to be a fixed number between 0 and 1. This new function is equal to f for $t = 0$, it is equal to g for $t = 1$, and it is a mixture of the two for every other value of t.

a) Show that h is also an affine transformation and can be written as

$$h(\boldsymbol{p}; t) = \operatorname{lerp}(\boldsymbol{F}, \boldsymbol{G}, t)\boldsymbol{p} + \operatorname{lerp}(\boldsymbol{f}, \boldsymbol{g}, t) \tag{1.5}$$

Here, the notation $\operatorname{lerp}(\boldsymbol{F}, \boldsymbol{G}, t)$ stands for the interpolation of the two matrices $\boldsymbol{F}$ and $\boldsymbol{G}$, which is obtained by interpolating its coefficients.

BufferedImage →

Table 1.1
Parameters of a few affine iterated function systems!examples.

Name	a	b	c	d	t_x	t_y	Probability
Tree	0.01	0	0	0.45	0	0	1/4
	0.42	-0.42	0.42	0.42	0	0.4	1/4
	0.42	0.42	-0.42	0.42	0	0.4	1/4
	-0.01	0	0	-0.45	0	0.4	1/4
Dragon curve	1/2	1/2	-1/2	1/2	0	0	1/2
	-1/2	1/2	-1/2	-1/2	1	0	1/2
Letters	0	-0.2	5/8	0	1/8	0	1/9
	3/8	0	0	0.2	3/16	1/2	1/9
	3/8	0	0	0.2	3/16	0	1/9
	0	-0.2	3/8	0	5/16	1/8	1/9
	3/8	0	0	0.2	5/8	1/2	1/9
	3/8	0	0	0.2	5/8	1/4	1/9
	3/8	0	0	0.2	5/8	0	1/9
	0	-0.2	1/8	0	3/4	3/8	1/9
	0	-0.2	1/8	0	3/4	1/8	1/9
Crystal	.255	0	0	.255	.3726	.6714	0.2
	.255	0	0	.255	.1146	.2232	0.15
	.255	0	0	.255	.6306	.2232	0.15
	.370	-.642	.642	.370	.6356	-.0061	0.75

Figure 1.8 Blending IFS fractals. The Barnsley fern on the left can be smoothly transformed into the tree-shaped fractal on the right. The intermediate fractals are obtained by interpolating the underlying iterated function systems.

Not only can we blend individual affine transformations, we can also blend *systems* of affine transformations. Assume that $F = (f_1, \ldots, f_n)$ and $G = (g_1, \ldots, g_n)$ are two iterated function systems of the same size n. We can now define a blended function system $H_t = (h_1, \ldots, h_n)$ linearly interpolating each pair of functions:

$$h_k(\boldsymbol{p}; t) = \operatorname{lerp}(f_k(\boldsymbol{p}), g_k(\boldsymbol{p}), t) \tag{1.6}$$

As we saw in part (a), each h_k is an affine transformation, so the system H is another affine IFS that can be rendered using the chaos game of this chapter. The parameter t indicates the time that has passed since the start of the animation: We start with the original function system $F = H_0$ at $t = 0$ and end with the target function system $G = H_1$ at $t = 1$. For intermediate values of t, the fractals generated by H_t are a mixture of those generated by F and G.

b) Write a program that uses Eq. (1.6) to generate animations of iterated function systems similar to the one shown in Fig. 1.8. (Refer to Table 1.1 for the parameters of fractals you can use for testing.) How do you blend the tables of probabilities and how do you handle the case where the two function systems don't contain the same number of functions?

1.4 Chapter Notes

Iterated function systems are a popular means of constructing fractal shapes: They are easy to describe, easy to program, and can be used to model infinitely detailed geometric shapes using just a few parameters. See Paul Bourke's website (`https://paulbourke.net/fractals/ifs/`) for an overview of shapes that can be generated in this way.

It is possible to extend the simple IFSs discussed in this chapter to produce still images and animations that are even more striking. The best-known example is the so-called ***fractal flame algorithm***, which adds new function types, colorization, as well as additional transformation and post-processing steps to produce and endless variety of images and animations such as the one shown in Fig. 1.9 [31]. The *Electric Sheep* screen saver (`https://electricsheep.org/`) is based on this algorithm.

There is a deep and beautiful mathematical theory behind iterated function systems that concerns itself with questions such as:

- Under which conditions does the chaos game converge?
- How are the functions in the IFS related to the resulting fractal?
- Why does the chaos game always produce the same fractal even if we change the starting point or make different random choices during the process?

For details, see the books by Barnsley [10] and Peitgen et al. [76].

To give you a taste of this theory, let us briefly discuss one of its main results: The chaos game converges only if the functions are ***contractive***, which means

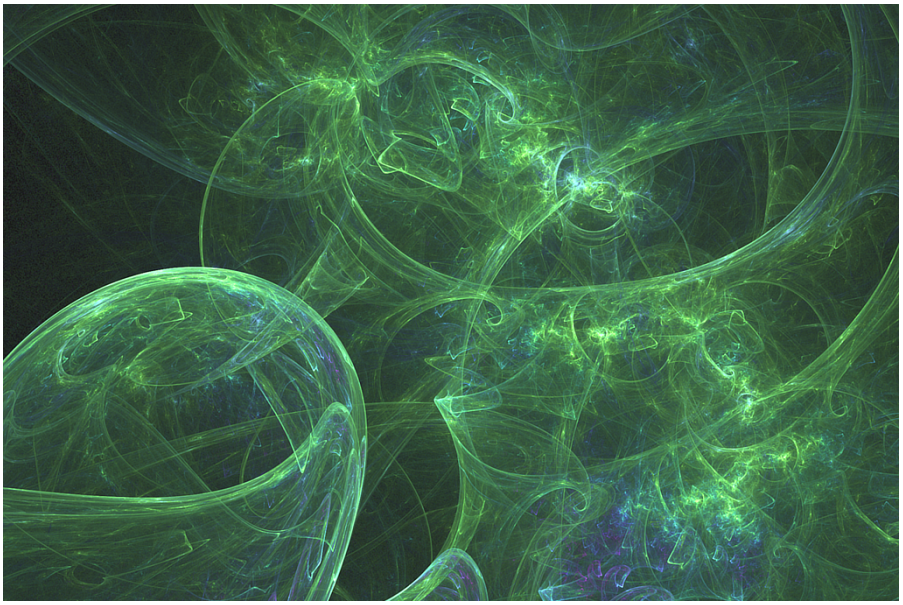


Figure 1.9 An image generated using the fractal flame algorithm.

that they make everything smaller. In mathematical terms, a function f is called contractive if the points $f(\boldsymbol{p})$ and $f(\boldsymbol{q})$ are closer together than $\boldsymbol{p}$ and $\boldsymbol{q}$ for every possible choice of $\boldsymbol{p}$ and $\boldsymbol{q}$:

$$|f(\boldsymbol{p}) - f(\boldsymbol{q})| < |\boldsymbol{p} - \boldsymbol{q}| \qquad \text{for all } \boldsymbol{p} \text{ and } \boldsymbol{q}.$$

The upper bound of the ratio $|f(\boldsymbol{p}) - f(\boldsymbol{q})|/|\boldsymbol{p} - \boldsymbol{q}|$ is called the ***contraction factor*** of the function f. A function is therefore contractive if its contraction factor is less than 1.

Consider the function $\boldsymbol{p} \mapsto (\boldsymbol{p} + \boldsymbol{c})/2$ that we used at the beginning of this chapter to move points halfway towards the corner $\boldsymbol{c}$. It's easy to show that this function is contractive with a contraction factor of $1/2$:

$$|f(\boldsymbol{p}) - f(\boldsymbol{q})| = |(\boldsymbol{p} + \boldsymbol{c})/2 - (\boldsymbol{q} - \boldsymbol{c})/2| = |\boldsymbol{p} - \boldsymbol{q}|/2.$$

Therefore, any set of functions that move their argument halfway toward a fixed set of points forms a convergent IFS.

It is also possible to derive the contraction factors of arbitrary affine transformations, but a few concepts from linear algebra are required. The main result is that the contraction factor of

$$f(\boldsymbol{p}) = \boldsymbol{A}\boldsymbol{p} + \boldsymbol{v},$$

is the square root of the largest eigenvalue of $\boldsymbol{A}^t\boldsymbol{A}$, where $\boldsymbol{A}^t$ is the transpose of the matrix $\boldsymbol{A}$. For instance, in the case of the Barnsley fern, the contraction factors of the functions $f_0, \ldots, f_3$ are 0.16, 0.85, 0.32, and 0.34.