7

Fractals

Why Study Fractals?

- **To model natural shapes**. Fractals model irregular, self-similar structures such as coastlines, mountains, river networks, trees, plants, clouds, and turbulent flows, all of which cannot be captured easily by classical Euclidean geometry.

- **To describe complex systems**. Many chaotic systems naturally produce fractal patterns. Turbulence, weather systems, and population dynamics all show self-similar structure, and fractal dimension provides a more meaningful measure of complexity than normal geometric dimensions.

- **For their utility across many application areas.** Fractal tools help analyse signals that vary with scale, including time series, heartbeat variability, seismic data, and financial market fluctuations. Many networks exhibit fractal or self-similar organisation, including neural systems, social and communication networks. Many biological structures display fractal organisation, including the lungs, blood vessel networks, and neuron branching.

- **As an artform**. Fractals generate visually compelling patterns and digital art, benefiting from the appeal of self-similar and infinitely detailed forms. They underlie the generation of terrain, textures, and natural scenery in computer graphics and simulations.

7.1 Introduction

Benoit Mandelbrot coined the word 'fractal' to describe irregular geometric shapes containing structure at arbitrarily small scales. Many fractals appear similar at those different scales, an attribute called self-similarity, quite unlike a circle for instance, where a small portion of the perimeter sufficiently magnified is almost indistinguishable from a straight line.

We learn that doubling the sides of a polygon multiplies its area by four, and if the radius of a sphere is doubled, its volume scales by eight. However, when a fractal's length is doubled, it's scaled by a power that's not necessarily an integer. Indeed, some fractals, such as the Hilbert curve (see below), completely fill 2D space even though they are topologically only 1-dimensional.

Another way to observe this difference is by measuring length, area, and volume at different scales. For a conventional shape, say a circle of unit radius, the measurement of its perimeter at small scales will always produce an answer close to 2π. However, measuring the length of a fractal, such as a von Koch curve (see below), at smaller-and-smaller scales produces ever-increasing values.

Fractal self-similarity lends itself to a recursive definition for their construction (although iterative solutions are also possible). A common approach employs initiators and generators: an initiator is a starting shape, while the generator specifies how to create scaled copies, often by utilizing affine operations (e.g. rotations and translations) and deletion.

It should be said that almost all the well-known fractals, such as the Cantor set, the von Koch curve, the Sierpinski gasket, the Menger sponge, the Julia set, and Brownian motion, have been known for a long time. Mandelbrot's brilliance lay in grouping these ideas into a recognizable field, and showing how they could be used as an organizing principle for natural phenomena. Prior to fractal geometry, nature was usually regarded as a form of *noisy* Euclidean geometry. As Mandelbrot famously wrote in *The Fractal Geometry of Nature* [**Man83**]: 'Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line.'

7.2 The von Koch Curve

The von Koch curve, named after the Swedish mathematician Helge von Koch (1870–1924), begins as a straight line segment that's divided into three (see Fig. 7.1, (a)). The middle third is discarded in (b), replaced by two sides of an equilateral triangle pointing outwards. The end result is the generator shape in (c).

Using the labeling of Fig. 7.2, the K0 curve is replaced by K1. The next level (K2) is obtained by treating the four line segments of K1 as initiators that are

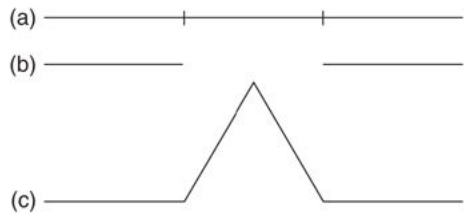

Figure 7.1. Initiator and generator for the von Koch Curve

redrawn as in Fig. 7.1 – each segment is divided into three, and the middle replaced by two outward facing lines, creating four copies of a scaled (and rotated) generator. The curve develops as this process is continued indefinitely.

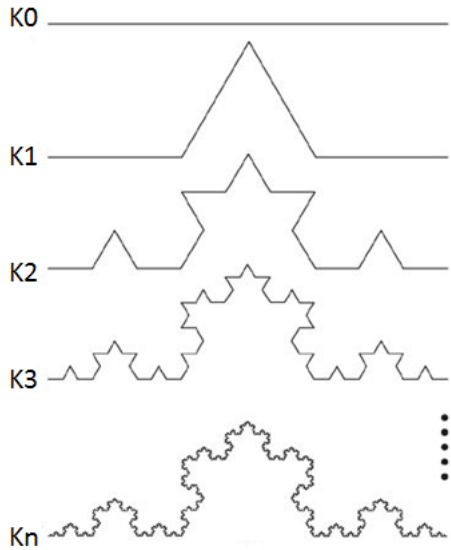

Figure 7.2. Levels of von Koch curves

The last curve in Fig. 7.2 shows the result of several iterations of the generator, and its self-similarity is quite clear.

A variant is the von Koch snowflake (Fig. 7.3), which applies the generator to the sides of an equilateral triangle.

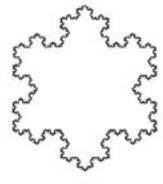

Figure 7.3. The von Koch snowflake

7.3 Fractal Dimensionality

The non-standard dimensionality of fractals becomes apparent if we consider how the length of the von Koch curve changes at each level. If the length of the initial line segment (K0 in Fig. 7.2) is l, the length L_n of the curve at the n-th level will be

$$L_n = \frac{4}{3}^n l$$

This quantity increases without bound, implying that the curve has an infinite length. However, the curve does have a bounded area even though equilateral triangles are added at each level. If one of these triangle has a side length s then its area is less than s^2 (the square containing the triangle). Hence, at step n, the area A_n "under" the curve is

$$\begin{aligned} A_n &< (\tfrac{1}{3})^2 + 3(\tfrac{1}{9})^2 + 9(\tfrac{1}{27})^2 + \cdots \\ &= \textstyle\sum_{i=1}^{n} \frac{1}{3^{i+1}} \end{aligned} \tag{7.1}$$

This is a geometric series of ratios less than one, and so converges.

7.4 Measuring Fractals

A simple way to measure fractals is by box-counting. The boxes can be any regular shape, but it's easiest to use squares, as in Fig. 7.4.

Smaller squares will obviously lead to a better approximation to the curve. Suppose we employ $N(r)$ squares of side length r, then $N(r) \cdot r$ approximates the curve's length, and $N(r) \cdot r^2$ its area. As a result, it's reasonable to assume that the relationship between $N(r)$ and $1/r$ is a power law, $N(r) = k(1/r)^d$, where d is termed the box-counting dimension and k is some arbitrary constant.

Taking logs of both sides:

$$\ln N(r) = d \ln \frac{1}{r} + \ln k$$

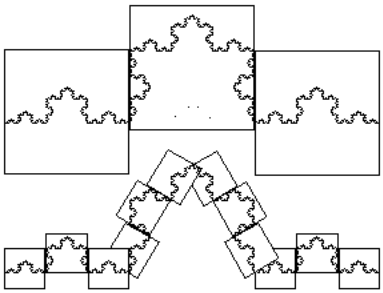

Figure 7.4. Boxes around the von Koch Curve

The expectation is that the approximation will become better for smaller r. Solving for d, and taking the limit as $r \to 0$ gives:

$$d = \lim_{r \to 0} \frac{\ln N(r) - \ln k}{\ln \frac{1}{r}}$$

Note that as $r \to 0$, $1/r \to \infty$, so $\ln 1/r \to \infty$ and $\ln k / \ln(1/r) \to 0$.

For a fractal where all of its components scale by the same factor, such as the von Koch curve and the Sierpinski gasket (see below), the equation can be simplified to

$$d = \frac{\ln N}{\ln \frac{1}{r}}$$

where N is the factor by which the number of generator copies increases at each level, and r is the scaling factor.

For the von Koch Curve, N increases by a factor of 4 at each level while the generator is scaled by 1/3 (i.e. $N = 4$ and $r = 1/3$), so we have:

$$d = \ln 4 / \ln 3 \approx 1.26186$$

The von Koch curve is more than 1-dimensional, but less than 2-dimensional. This suggests, at the very least, that the curve has a dimension not equal to the space it resides in.

Fractal dimension can be used as a measure of the complexity of a pattern, but is lacking in some respects. The number doesn't provide enough information to construct the curve, and obviously different fractals may have the same fractal dimension value.

7.5 The Sierpinski Gasket

Start with an equilateral triangle, connect the midpoints of its sides, and remove the resulting triangle, leaving three smaller, similar triangles (Fig. 7.5). The generator is recursively applied to these triangles, and after a few levels, the shape looks pleasingly complicated.

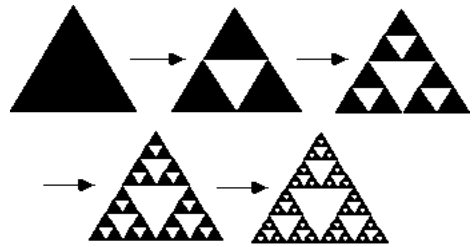

Figure 7.5. Levels of the Sierpinski gasket

The easiest way to box-count the gasket is to rotate it the left and change its initiator into a right-angled triangle. This lets the fractal be more readily covered with squares, as in Fig. 7.6.

At each level, the number of boxes, N, increases by a factor of 3 while the sides of each square are halved (i.e. $N = 3$ and $r = 1/2$), which means that:

$$d = \ln 3 / \ln 2 \approx 1.58996$$

The gasket is also more than 1-dimensional, but less than 2-dimensional, which tallies with the way that its area approaches 0 at reduced scales.

7.6 Other Ways to Define Fractals

Aside from the "initiator and generator" approach, there are several other ways to produce fractals:

- *L-systems* employ parallel string rewriting somewhat akin to the application of grammar rules. They're a popular choice for modeling plant growth, and were first used by the botanist Aristid Lindenmayer in 1968.
- *Iterated function systems* (IFSs) repeatedly apply geometric transformations to shapes, although non-linear functions, such as projective operations and circle inversion, are also possible. The popularity of IFSs is due in no small part to their use in the excellent text by Michael Barnsley, *Fractals Everywhere* [**Bar12**].

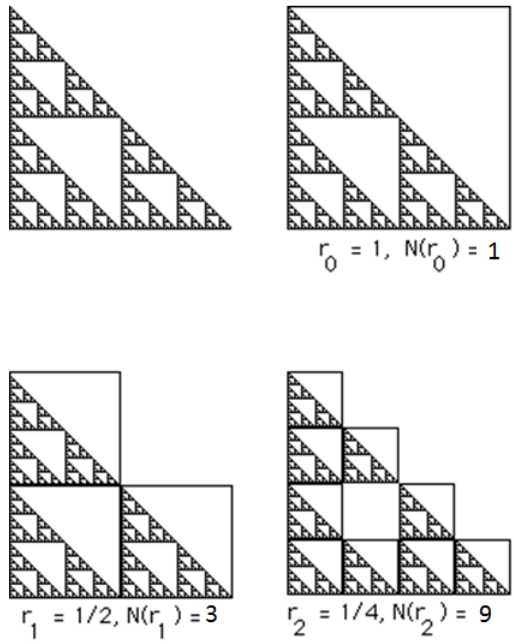

Figure 7.6. Boxing the Sierpinski Gasket

- *Escape-time fractals* apply a recurrence relation to each point in a space (usually the complex plane). After a fixed number of iterations, a test is carried out on the point to determine how it should be drawn. The Mandelbrot and Julia sets are two well-known examples.

The exercises at the end of this chapter investigate these approaches in a little more detail, with the help of code examples.

7.7 Coding Fractals

A key programming decision is how to draw the fractals. We mostly use Python's Turtle module in this section, but switch to matplotlib for plotting the Mandelbrot and Julia set coordinates (see Ex. 8) and 3D curves.

`fractals.py` lets the user choose between drawing a Sierpinski gasket, a dragon curve, a Levy curve, a von Koch snowflake, or a Minkowski island (Fig. 7.7). We'll give details on how the snowflake and gasket are generated.

7.7.1 The von Koch Snowflake. The snowflake is drawn by the following functions in `fractals.py`:

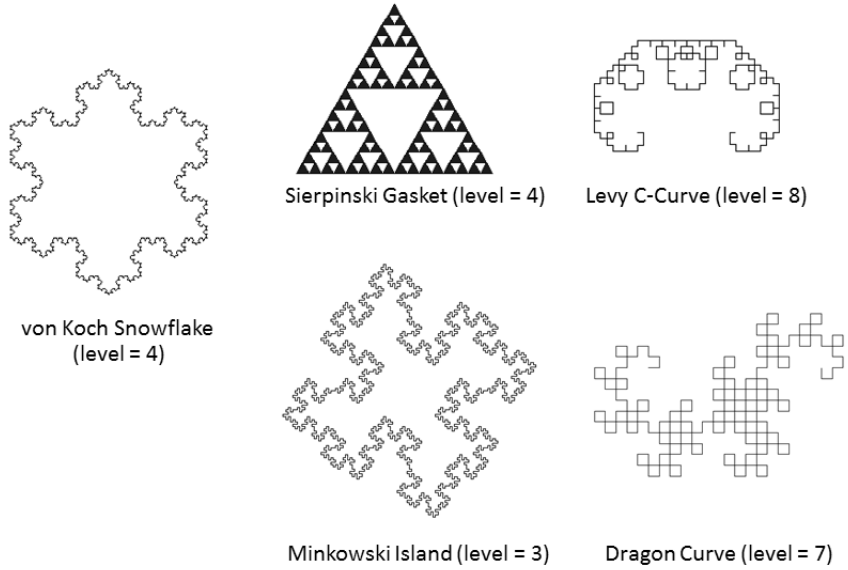

Figure 7.7. Various fractals

```
def koch(t, size, level):
  for _ in range(3):
    kochSide(t, size, level)
    t.right(120)

def kochSide(t, size, level):
  if level < 1:
    t.fd(size)
  else:
    for angle in [60, -120, 60, 0]:
      kochSide(t, size/3, level-1)
      t.left(angle)
```

Listing 7.1. Drawing the von Koch snowflake

koch() calls kochSide() three times to generate the three sides of the snowflake. When level = 1, kochSide() produces the K1 snowflake (see Fig. 7.2) using the operations illustrated in Fig. 7.8.

The curve is drawn using four forward moves and four angle changes (although the last is a turn of 0 degrees).

7.7.2 The Sierpinski Gasket. The gasket is drawn by the following function in fractals.py:

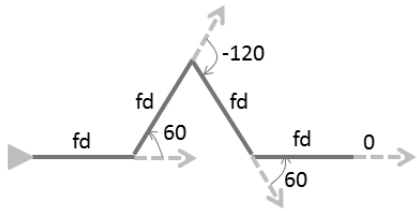

Figure 7.8. Turtle operations for Koch K1

```
def sierpinski(t, size, level):
  if level < 1:
    t.left(60)
    drawTriangle(t, size, "blue")
    t.right(60)
  else:
    sierpinski(t, size/2.0, level-1)
    t.left(60)
    t.fd(size/2.0)
    t.right(60)
    sierpinski(t, size/2.0, level-1)
    t.left(60)
    t.bk(size/2.0)
    t.right(60)
    t.fd(size/2.0)
    sierpinski(t, size/2.0, level-1)
    t.bk(size/2.0)
```

Listing 7.2. Drawing the Sierpinski gasket

drawTriangle() is imported from `TurtleUtils.py` to draw a triangle filled with a specified color.

The if-else branches in Listing 7.2 can be understood by considering Fig. 7.9.

The left-hand diagram in Fig. 7.9 illustrates the if-branch's execution when level = 0. The turtle is turned left before drawing, and returned to its starting position afterwards. The right-hand drawing deals with the more complicated `else` case when level = 1. The actions are split into three stages. Stage(a) represents the first line of the else-branch which calls sierpinski() to draw the left-most triangle. Stage (b) illustrates the next five lines:

```
t.left(60)
t.fd(size/2.0)
t.right(60)
sierpinski(t, size/2.0, level-1)
t.left(60)
```

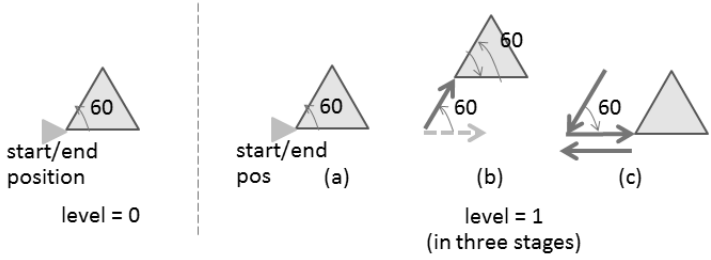

Figure 7.9. Turtle operations for two Sierpinski levels

The turtle turns left, advances to the starting point of the top triangle, and rotates right to have the correct orientation. sierpinski() draws the triangle, and the turtle is rotated left afterwards.

Stage (c) depicts the last five lines of the else-branch:

```
t.bk(size/2.0)
t.right(60)
t.fd(size/2.0)
sierpinski(t, size/2.0, level-1)
t.bk(size/2.0)
```

The turtle moves back to the bottom left corner, rotates to the right, and advances to the starting position of the right most triangle, which is drawn by the third call to sierpinski(). The turtle then returns to its starting position.

7.8 The 2D Hilbert Curve

Our most complicated initiator+generator is for the Hilbert curve, first proposed by David Hilbert in 1891. The first three levels of the curve are shown in Fig. 7.10.

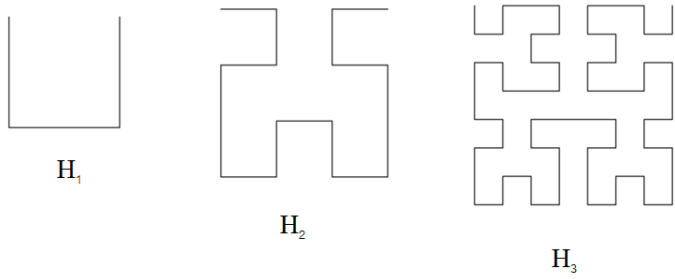

Figure 7.10. Hilbert curves (levels 1, 2, and 3)

The initiator is the ⊔ shape labeled as H_1 in Fig. 7.10. A level i curve, H_i, is obtained by the composition of four instances of H_{i-1} of half size and appropriate rotations, tying them together with three lines. Fig. 7.11 repeats the previous figure, but with those lines drawn more thickly so the sub-elements are easier to see.

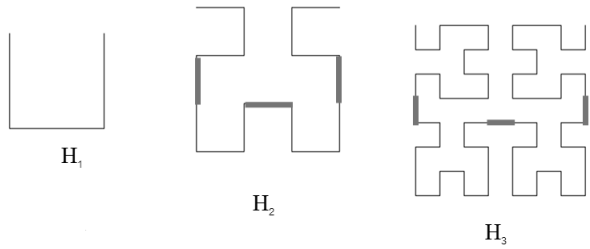

Figure 7.11. Hilbert curves (levels 1, 2, and 3) with thick connectors

The turtle code employs four recursive functions which draw the four rotations of the H_{i-1} shapes that make up H_i. We'll call these U, D, R, and L, short for "Upper", "Down", "Right", and "Left". The functions are recursive in the sense that the drawing done at level i utilizes the output of the functions at level $i-1$. The four basic drawings, denoted by U_1, D_1, R_1, and L_1, are shown in Fig. 7.12.

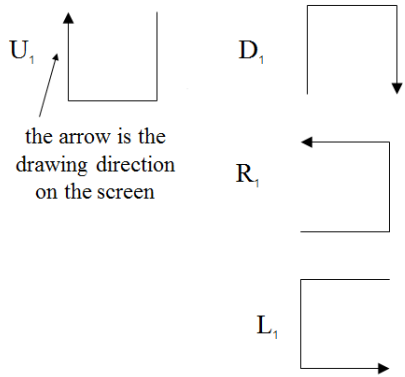

Figure 7.12. U_1, D_1, R_1, and L_1

The recursive U_i, D_i, R_i, and L_i operations follow these basic paths interspersed with calls to the other drawing functions for rendering the elements at level $i-1$ (see Fig. 7.13).

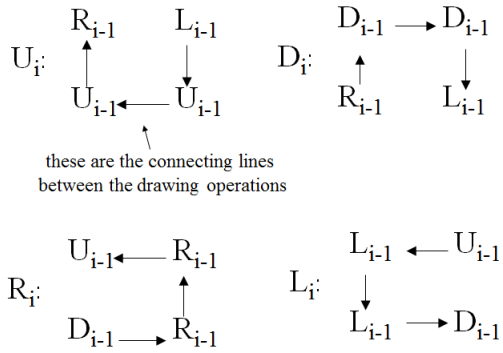

Figure 7.13. U_i, D_i, R_i, and L_i

The code for U_i (upperU()in Listing 7.3 from hilbert.py) corresponds to the U diagrams in Figs. 7.12 and 7.13. The same is true for the functions downU(), rightU(), and leftU().

```
def upperU(t, level):
  if level > 0:
    leftU(t, level-1)
    t.setheading(270)
    t.fd(step)
    upperU(t, level-1)
    t.setheading(0)
    t.fd(step)
    upperU(t, level-1)
    t.setheading(90)
    t.fd(step)
    rightU(t, level-1)
```

Listing 7.3. The U_i function

Note: setHeading() sets the turtle's orientation with 0 = east, 90 = north, 180 = west, and 270 = south

7.8.1 Dimensionality of the Hilbert Curve. The curve scales all of its components by the same factor, allowing its dimensionality to be expressed as

$$d = \frac{\ln N}{\ln \frac{1}{r}}$$

The N and r values can be obtained from the diagrams (Fig. 7.14) drawn by Hilbert on the first page of his 1891 paper (`https://terpconnect.umd.edu/~jmr/HonorsSem/Hilbert-orig.pdf`).

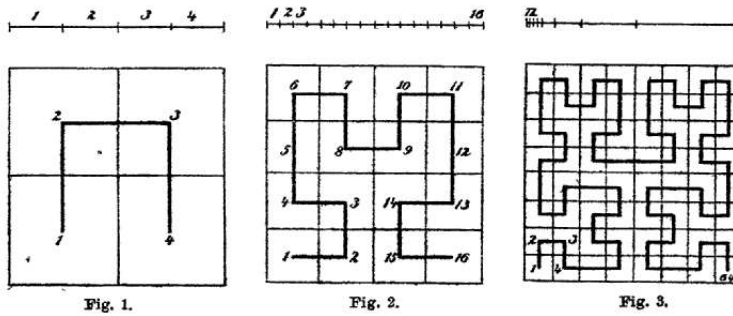

Figure 7.14. Hilbert's H1, H2, and H3

At each level, the number of squares increase by a factor of 4, and the sides of each square are halved. So $N = 4$ and $r = 1/2$, which means that

$$d = \ln 4 / \ln 2 = 2$$

The curve has the same dimensionality as the area of a standard geometric shape, which is reassuring since the curve does indeed cover its area as the number of levels increase. For example, consider H_6 in Fig. 7.15. This space-filling property has meant that the curve has found many uses related to mapping between 1D and 2D spaces.

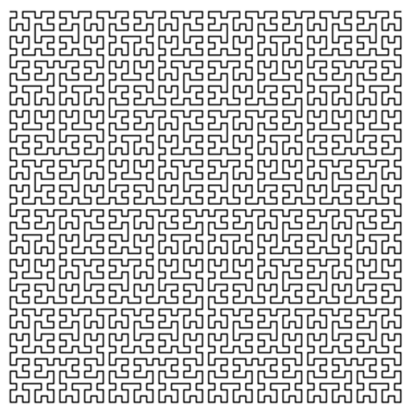

Figure 7.15. Hilbert curve (level = 6)

7.9 The 3D Hilbert Curve

It's possible to have the Hilbert curve cover 3D space. In hilbert3d.py the coordinates for the curve are generated recursively, then passed to matplotlib (Fig. 7.16). A pleasing aspect of its 3D plots is that it's possible to drag and rotate the curve to view it from various angles.

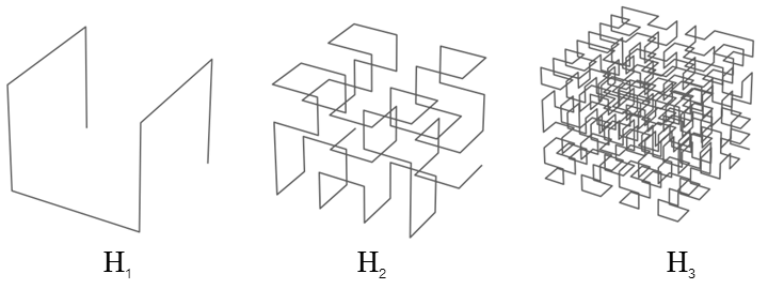

Figure 7.16. Hilbert 3D curves (levels = 1, 2, and 3)

The curve is rendered based on the ordering of the eight vertices used to define H1 (see Fig. 7.16) which are highlighted in Fig. 7.17.

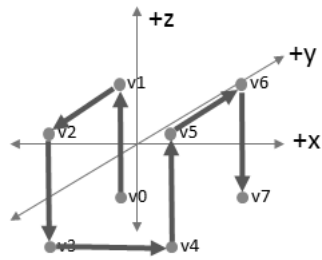

Figure 7.17. Hilbert 3D points (level = 1)

The eight points are stored in a list:

```
verts = [
    (center[0]-w/2, center[1]+w/2, center[2]-w/2),
    (center[0]-w/2, center[1]+w/2, center[2]+w/2),
    (center[0]-w/2, center[1]-w/2, center[2]+w/2),
    (center[0]-w/2, center[1]-w/2, center[2]-w/2),
    (center[0]+w/2, center[1]-w/2, center[2]-w/2),
    (center[0]+w/2, center[1]-w/2, center[2]+w/2),
    (center[0]+w/2, center[1]+w/2, center[2]+w/2),
```

```
    (center[0]+w/2, center[1]+w/2, center[2]-w/2)
]
```

Listing 7.4. Coordinates used for H1

It's challenging to see how H2 is derived from H1, so we'll focus on the four versions of H1 used to start building H2, which are circled in Fig. 7.18. The shape labeled 'call1' begins H2, followed by two copies of the same shape ('calls 2 and 3'), and then 'call4'. All of these shapes could be created by rotating and translating H1.

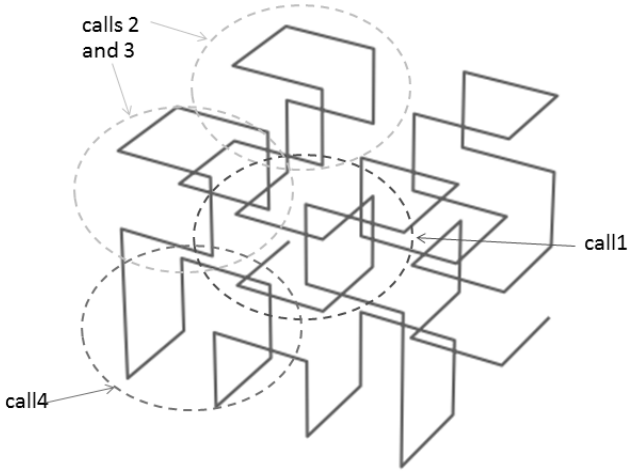

Figure 7.18. Drawing H2 Using H1s

Rather than use affine transformations, we've taken a simpler approach and defined the H1 versions by *reordering* the eight vertices in the original H1. Each reordering defines the drawing path for a variant, as shown in Fig. 7.19.

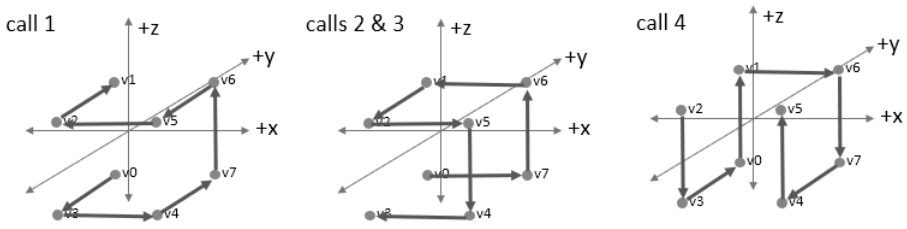

Figure 7.19. The H1 versions used to start drawing H2

The corresponding code in hilbert3d.py is quite succinct. The order of the vertices passed to each hilbert3D() call match their path order. This can be checked by comparing the path orders in Fig. 7.19 with the first four calls in the snippet in Listing 7.5 (hilbert3D.py).

```
coords = \
  hilbert3D(gray[0],w/2,d-1, v0,v3,v4,v7,v6,v5,v2,v1) + \
  hilbert3D(gray[1],w/2,d-1, v0,v7,v6,v1,v2,v5,v4,v3) + \
  hilbert3D(gray[2],w/2,d-1, v0,v7,v6,v1,v2,v5,v4,v3) + \
  hilbert3D(gray[3],w/2,d-1, v2,v3,v0,v1,v6,v7,v4,v5) + \
  hilbert3D(gray[4],w/2,d-1, v2,v3,v0,v1,v6,v7,v4,v5) + \
  hilbert3D(gray[5],w/2,d-1, v4,v3,v2,v5,v6,v1,v0,v7) + \
  hilbert3D(gray[6],w/2,d-1, v4,v3,v2,v5,v6,v1,v0,v7) + \
  hilbert3D(gray[7],w/2,d-1, v6,v5,v2,v1,v0,v3,v4,v7)
return coords
```

Listing 7.5. The coordinates for H2

Listing 7.5 indicates how the coordinates for H_i are constructed by eight calls to H_{i-1} with the w/2 argument reducing the length of each path edge by 1/2. This means that the curve's dimensionality uses $N = 8$ and $r = 1/2$:

$$d = \ln 8 / \ln 2 = 3$$

In other words, the curve has the same dimensionality as the *volume* of a standard geometric shape, which corresponds to how the curve fills 3D space.

Exercises

(1) The initiator for a Cantor set is a line segment, and its generator removes the middle third.

Figure 7.20. The first five levels of the Cantor Set

Evaluate the set to several levels, and determine its fractal dimensionality (*hint*: it's approximately 0.63093). This value may seem rather surprising since the set appears to be on it way to completely disappearing. In fact, despite the deletions there will be as many points left behind as there were at the start, implying that the set is uncountable.

Code for generating the set is in cantorset.py. Modify it to remove a different size segment or segments. How do these changes affect the set's dimension?

Note that if the generator produces parts with different sizes then we can no longer use the measure:

$$d = \frac{\ln N}{\ln \frac{1}{r}}$$

A good explanation of how to deal with more complex dimensionalities can be found in Section 2.D at the Yale "Fractal Geometry" site (`https://users.math.yale.edu/public_html/People/frame/Fractals/`), and the associated textbook [**FU16**].

Multi-dimensional Cantor sets are often called Cantor dust, such as the one shown in Fig. 7.21 which was drawn by `cantorCheese.py`; compare its code with `cantorset.py`.

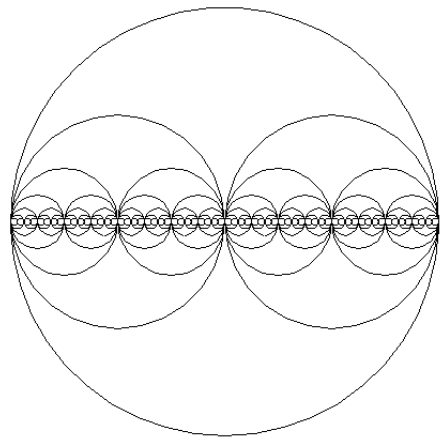

Figure 7.21. Cantor cheese

(2) Mandelbrot's "How Long Is the Coast of Britain?" [**Man67**] considers the coastline measurement problem: the fact that the measured length of a stretch of coastline depends on the size of the measuring instrument. For example, if the coastline of Great Britain is measured in units of 100 km, then the length comes to approximately 2,800 km. But with 50 km units, the total is nearer 3,400 km.

This is worth trying for yourself: obtain a map of some favorite coastline (e.g from `https://free-map.org/`) and measure its length at a variety of scales. Plot your data as a log-log graph, and ask yourself what the slope of the resulting line represents.

Figure 7.22. Measuring a coastline

A good description of this problem can be found in Section 2.H at the Yale "Fractal Geometry" site (`https://users.math.yale.edu/public_html/People/frame/Fractals/`) in the "Coastlines" subsection.

(3) The Sierpinski carpet starts with a square which the generator divides into 8 smaller squares with the middle one removed. Evaluate the set to several levels, and determine its fractal dimensionality (*hint*: it's approximately 0.63093). Code for drawing the carpet in Fig. 7.23 can be found in sierCarpet.py.

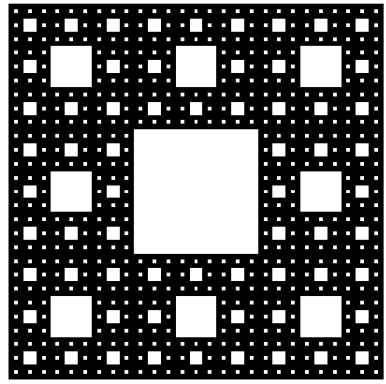

Figure 7.23. A Sierpinski Carpet

(4) The **L-system** approach to coding fractals can be tried out by running lsys.py, which can load rules for a variety of fractals stored as text files in `http://coe.psu.ac.th/~ad/explore/code/Fractals/lsys/`. Examine the

code, try out a few examples, and note the code's features and limitations. For instance, consider the rules for generating a tree in Listing 7.6 (tree.txt).

```
# axiom
F

# rule
FF+[+F-F-F]-[-F+F+F]

# angle
22.5
```

Listing 7.6. L-system tree definition

This definitions are loaded and executed like so:

```
> python lsys.py
fnm=? tree
axiom: F
rule: FF+[+F-F-F]-[-F+F+F]
angle: 22.5

level? 3
```

The user entered the filename and `level` value, and Fig. 7.24 was drawn.

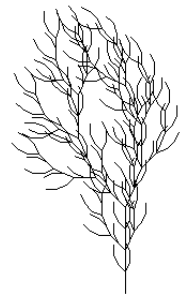

Figure 7.24. An L-system drawing of a tree

Hints: our L-system implementation offers stack pushing and popping which is necessary for state storage and retrieval (as utilized in bracketed L-systems), but only supports a single rule. This makes it impossible to implement most of the fractals listed on the L-systems wikipedia page (`https://en.wikipedia.org/wiki/L-system`). It would be a fun project to remedy that deficit by supporting multiple rules in lsys.py.

(5) The **IFS** approach to coding fractals can be tested by running our ifs.py, which can load parameters for drawing fractals stored in `http://coe.psu.ac.th/~ad/explore/code/Fractals/ifs/`. Examine the code, and note its features and limitations. For instance, consider the data for generating a Barnsley fern in Listing 7.7 (barnsley.txt).

```
4
   0.01   0.85   0.07   0.07

4 3
   0.00   0.00   0.500
   0.85   0.04   0.075
   0.20  -0.26   0.400
  -0.15   0.28   0.575

4 3
   0.00   0.16   0.000
  -0.04   0.85   0.180
   0.23   0.22   0.045
   0.26   0.24  -0.086
```

Listing 7.7. IFS definition for a Barnsley fern

This data is loaded and executed like so:

```
> python ifs.py
fnm=? barnsley
probs: [0.01, 0.85, 0.07, 0.07]

cx: [0.0, 0.0, 0.5] [0.85, 0.04, 0.075]
    [0.2, -0.26, 0.4] [-0.15, 0.28, 0.575]

cy: [0.0, 0.16, 0.0] [-0.04, 0.85, 0.18]
    [0.23, 0.22, 0.045] [0.26, 0.24, -0.086]
```

The user entered the filename and Fig. 7.25 was drawn.

The underlying mechanism used by ifs.py is a matrix-based affine transformation of an (x,y) point:

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

The a, b, c, d, e, and f parameters are read from the text file passed to ifs.py, and utilized in:

```
x0 = cx[r][0]*x + cx[r][1]*y + cx[r][2]
y0 = cy[r][0]*x + cy[r][1]*y + cy[r][2]
```

Figure 7.25. An IFS drawing of a Barnsley fern

Chapter 10 of *Fractals Everywhere* [**Bar12**] uses IFSs to generate deterministic fractals. We've added a simple notion of randomness, which you should try to locate in the code.

Also, have a look at the IFS for the Sierpinski gasket (sierpinski.txt) and understand its relationship to the Sierpinski initiator+generator in Listing 7.2.

(6) It can be argued that none of our initiator+generator code really matches our description of how that mechanism works. For example, the most natural way of describing the generator for the Sierpinski gasket is as the *removal* of the middle triangle of the initiator. However, our code implements this by turtle operations involving moves and turns.

The Turtle module doesn't support set operations on images (e.g. union, intersection, and (crucially) subtraction). Pillow is one library which does have these features (`https://python-pillow.org/`), and we've included a small example of how to subtract one image from another in subtract.py. It loads an existing image (`http://coe.psu.ac.th/~ad/explore/code/Fractals/triangle.png`) and subtracts the inner triangle, as depicted in Fig. 7.26.

Install the Pillow library, and try out the code; it could become the basis of a removal-based generator for the Sierpinski gasket.

(7) Our code examples include mandel.py and julia.py which generate the Mandelbrot and Julia sets respectively. The Mandelbrot set contains the complex numbers c for which the function $f_c(z) = z^2 + c$ does *not* diverge to infinity when iterated from $z = 0$. The corresponding pixel in the image (Fig. 7.27) is painted black if the sequence doesn't run away to infinity, otherwise it's color is based on how quickly it went to infinity.

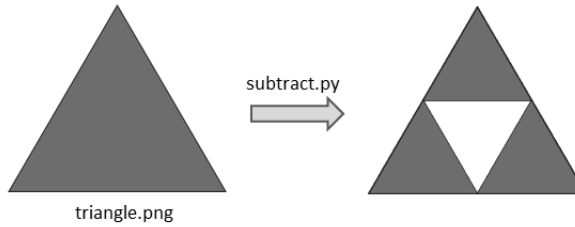

Figure 7.26. Using the Pillow module to subtract a triangle

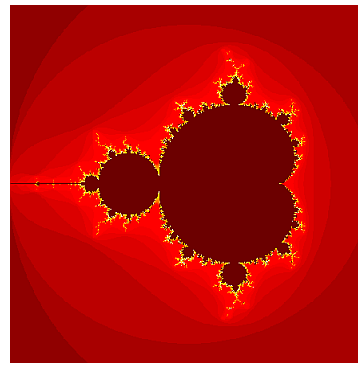

Figure 7.27. The Mandelbrot Set

One of the surprises is that the edges of the Mandelbrot set are filled with tiny copies of the entire set, each of which is surrounded by its own halo of still tinier copies, and so on, without end. All of these copies are attached to the main body, so every part of the image is connected.

Have a look at `mandel.py` to see how this is implemented. Sadly, there's no interaction, such as panning or zooming.

The Julia set contains the complex numbers z for which the $f_c(z) = z^2 + c$ doesn't diverge to infinity. Fig. 7.28 shows the image generated with c = (-0.3, 0.71j).

`julia.py` is coded in a similar way to `mandel.py`, but compare them to see how they differ. Try varying the c starting values; a few interesting ones are included as comments at the start of the program

The link between the two sets is that the Mandelbrot contains those c values for which the Julia set forms a single connected image. More visually, a particular Julia set corresponds to a particular c coordinate in the Mandelbrot image. Fig. 7.29 shows a series of different Julia sets around its edge, to

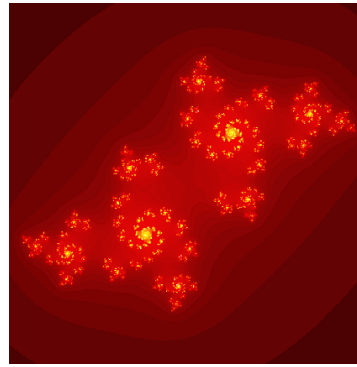

Figure 7.28. The Julia Set for c = (-0.3, 0.71j)

which we've added lines that link to their corresponding c coordinate in the Mandelbrot.

(8) sierpinskiCA.py generates the Sierpinski Gasket using a **Cellular Automaton** (CA). The output begins as shown in Fig. 7.30.

A CA is defined by four attributes:

(a) The state space represented as a collection of cells.
(b) The neighborhood of a cell – those cells whose current states affect the next state of the cell.
(c) A collection of the different states that a cell can assume.
(d) A rule (or rules) which specify how the states of the cells in the neighborhood determine the next state of a cell.

sierpinskiCA.py implements a 1-dimensional CA where a cell only has a left and right neighbor, and can be in one of two states ("*" meaning True; " " for False). The rule states that the status of the current cell is set to True if one of its two neighbors at the previous step was True, otherwise it's set to False. Rather amazingly, this rule causes the automaton to generate a Sierpinski gasket. You should check that you understand why this occurs.

Despite their simplicity, CAs have proved useful for modeling a wide range of physical, chemical, and biological problems.

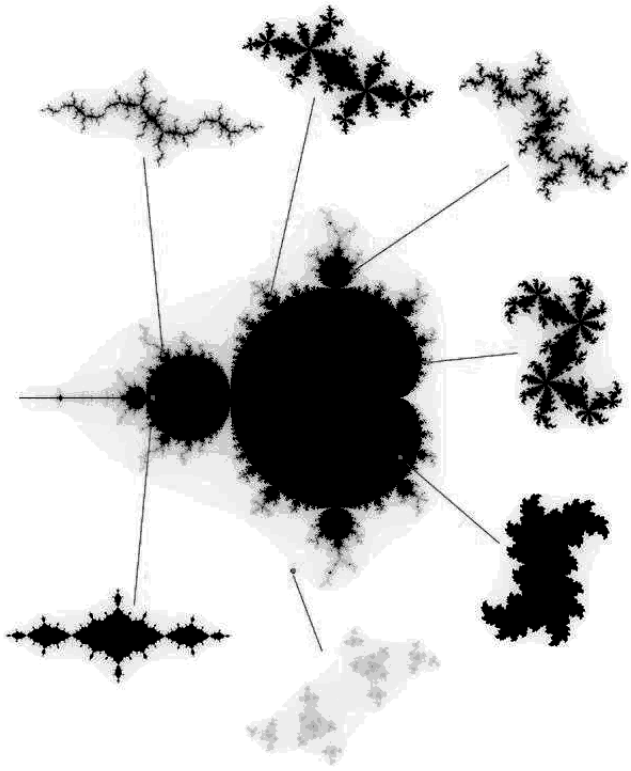

Figure 7.29. Linking the Mandelbrot and Julia Sets

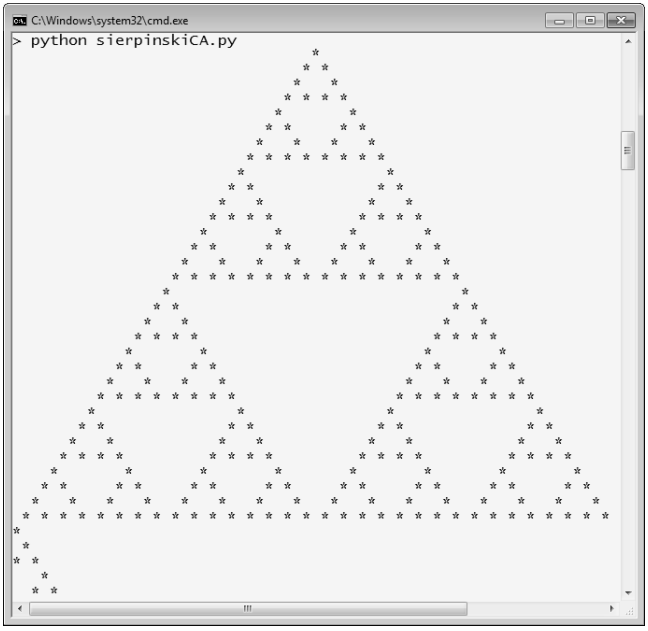

Figure 7.30. The Sierpinski Gasket using a cellular automaton