Chapter 4:

Fractals I

Goals:

- To understand what fractals are.
- To see the fractal nature of some natural processes, including the period-doubling sequence of the logistic map.

A fractal is an object that looks the same when it is magnified. For example, the object in Figure 4.1 (called "the Sierpinski gasket") is a triangular shape in which there are three smaller triangles, each of which is a replica of the whole.

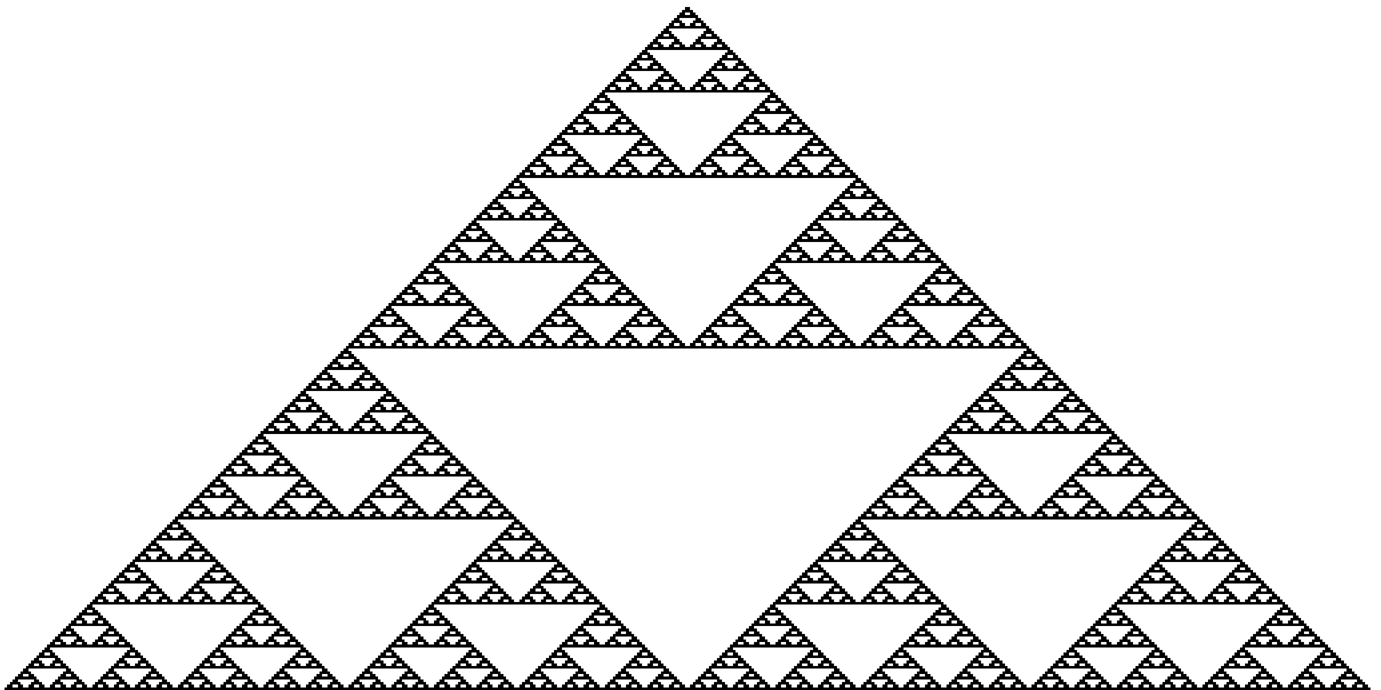


Figure 4.1 The Sierpinski gasket.

Another example of a fractal is the Cantor set, which is one of the most famous objects in modern mathematics. It is constructed in

the following way: Take the line which extends between zero and one. Erase the middle third of that line. Take the line segments which remain and erase their middle thirds as well. Do this again and again.

We will see that fractals arise naturally in the study of dynamical systems. But first we will draw a few fractals.

A. Drawing fractals.

A.1. Recursion. In Java, a method can call itself. This process is called recursion, and it is ideally suited for constructing fractals. The applet CantorRecursion constructs the Cantor set by defining a method that removes the middle third of a line segment and also calls itself recursively on the parts that are remaining. It draws the resulting line segments at each step of the process. The recursion is stopped when the line segment is one pixel long and so can't be divided any more.

```
// CantorRecursion.java
// draws Cantor set
import java.awt.*;
import java.applet.Applet;

public class CantorRecursion extends Applet{
        int length;                          //  longest segment
        int left, right;                     //  endpoints of segments
        int height;                          //  vertical size of applet
        int ypos;                            //  vertical position on applet

        final int nlevels = 6;               //  number of levels to be drawn
        final int yspace = 50;               //  vertical spacing between lines

        public void init ( )   {
                setBackground( Color.white );
                length = 1;
                for (int i = 1; i <= nlevels; i++)    {
                        length = 3*length;
                }                                    // original line segment length
                height = (nlevels + 2)*yspace;
                setSize(length + 30, height);        // make applet a convenient size
                left = 15;              // initial left and right coordinates of segment
```

* * *

```
        right = 15 + length - 1;
        ypos = yspace;
    }

    public void paint ( Graphics g )     {
        g.setColor( Color.blue );
        cantor(left, right, ypos, g);
    }

    // Recursive definition of method cantor
    public void cantor ( int cleft, int cright, int ypos, Graphics g)     {
        int clength = cright - cleft + 1;
        g.drawLine(cleft, ypos, cright, ypos);
        if ( clength <= 1 )                // base case
            g.drawLine(cleft, ypos, cright, ypos);
        else   {
            cantor(cleft, cleft+clength/3 -1, ypos+yspace, g);     // left
            cantor(cright-clength/3 +1, cright, ypos+yspace, g);  // right
        }
    }
}
```

Program 4.1. An applet that uses recursion to draw a Cantor set.

The applet draws each 'level' in the progressive construction of the Cantor set. Level zero is a full line segment; the first level has the middle third removed, the second level removes the middle third of the two segments created at level one, and so on. The actual Cantor set is the result of the repetitive application of this process infinitely many times. Hence it consists of a set of infinitely many points. However, the applet must stop at the screen resolution and cannot display the true Cantor set.

Note the basic structure of cantor, which is typical of all recursive methods. There are two parts: the "body" of the recursion and the "terminator". The method calls itself repeatedly, decreasing the variable `clength` in each step. This is the body of the recursion. Of course we can't let this process continue indefinitely. The recursion stops when `clength <= 1` (the smallest possible length is 1 pixel). The statement `{if (clength <= 1) g.drawLine(cleft, ypos, cright, ypos);}` is therefore called the terminator.

The Cantor set just described is known as the 'middle-thirds' Cantor set, because it removes the middle third and leaves the two end

thirds of the segment at each level. One can construct other Cantor sets by leaving some other fraction 'f' at each end of the segment ($f \quad 1/2$) while removing the middle ($1-2f$) portion of the segment. The middle-thirds Cantor set has $f = 1/3$.

Exercise 4.1: Alter CanterRecursion to construct the Cantor sets with other values of f.

A.2. Iteration. Recursion is elegant, but one can construct fractals using old-fashioned iteration. The applet CantorIteration constructs the middle-thirds Cantor set iteratively as follows. First, the line [0,1] is drawn. Then the segment (1/3, 2/3) is removed. (i) This object is shrunk down to a third its size and copied into a buffer. (ii) The object in the buffer is copied onto the left and right thirds of the original line segment. Then steps (i) and (ii) are repeated until the finite resolution of the screen prevents any finer division of the results. The necessary rescaling is performed by the Graphics method drawImage:

```
public abstract boolean drawImage(Image img, int x,
   int y, int width, int height, ImageObserver observer)

Draws the specified image inside the rectangle specified
by:
          x - the x coordinate
          y - the y coordinate
          width - the width of the rectangle
          height - the height of the rectangle
```

```
//  CantorIteration.java
//  draws Cantor set
import java.awt.*;
import java.applet.Applet;

public class CantorIteration extends Applet
                              implements ActionListener    {
    int nlevels, length;
    int csheight = 1;
    int left, right;
    int nplots;                       //  number of iterations done so far
    int height = 200;                        //  vertical size of applet
    int ypos;                                //  vertical position on applet
    Image theImage, bufferImage;
    Graphics theGraphics, bufferGraphics;
    boolean firstpaint;
```

* * *

```
    public void init ( )    {
        setBackground( Color.white );
        nlevels = 5;
        nplots = 0;
        length = 1;
        for (int i = 1; i <= nlevels; i++)     {
            length = 3*length;
        }                                       //  length of line containing Cantor set
        setSize(length+30, height);             //  make applet a convenient size
        left = 15;                              //  left and right coordinates of Cantor set,
        right = 15+length-1;                    //   in pixels

        ypos = 30;                      //  initial vertical position coordinate

        //  put on Button to do next level:
        Button nextlevelButton = new Button("Plot Next Level");
        nextlevelButton.addActionListener( this );
        add (nextlevelButton);

        //  create Image object for the Cantor set picture:
        theImage = createImage(length, csheight);
        theGraphics = theImage.getGraphics();

        //  create Image object for intermediate buffer:
        int bufferwidth = length/3;
        bufferImage = createImage(bufferwidth, csheight);
        bufferGraphics = bufferImage.getGraphics();

        //  put [1,1/3] + [2/3,1] into theImage:
        theGraphics.setColor( Color.blue );
        theGraphics.fillRect(0, 0, length/3, csheight);
        theGraphics.fillRect(2*length/3+1, 0, length/3, csheight);
        theGraphics.setColor( Color.white );
        theGraphics.fillRect(length/3, 0, length/3, csheight);
        theGraphics.setColor( Color.blue );
    }                                           //  end of init

    public void actionPerformed(ActionEvent evt)
    {   if(nplots < nlevels)    {
            //  copy theImage into bufferImage, shrinking it by a factor of three:
            bufferGraphics.drawImage(theImage, 0, 0, length/3, csheight, this);

            //  shrink and copy bufferImage into left and right thirds of theImage:
            theGraphics.drawImage(bufferImage, 0, 0, length/3, csheight, this);
```

* * *

```
            theGraphics.drawImage(bufferImage, 2*length/3, 0, length/3,
                                                        csheight, this);
            repaint();
            nplots ++;
            ypos += 20;
        }
    }                                   //  end of actionPerformed

    public void update (Graphics g)  {
        paint( g );
    }

    public void paint ( Graphics g )  {
        g.drawImage(theImage, left, ypos, this);
        if (nplots >= nlevels) {
            showStatus("at screen resolution: can't divide any more");
        }
    }
}
```

Program 4.2. An applet that constructs the Cantor set using iteration.

Exercise 4.2: Please go through, understand, and run this program. Then alter the program to construct the Cantor sets which leave a fraction f of the segment at each end in each step (as in Exercise 4.1).

Problem 4.1: Write a program to construct the Sierpinski gasket, the fractal set shown in Figure 4.1.

Menu Project. Using a turtle to draw fractals. This project consists of two parts. The first is to create a "turtle" graphics system (much like LOGO). The turtle graphics system has a turtle at some position pointing in a certain direction. The turtle can move forward, leaving a trail of ink behind (drawing a line), and it can rotate. It can also turn the ink on and off. So your class has to support the following commands:

```
public class Turtle {
   public Turtle(Graphics g, int height, int width, int xpos, int ypos, int angle);
   // constructor.  Draws into the graphics context g, which has height 'height' and
   // width 'width.'  Starts turtle at initial position (xpos, ypos) and angle 'angle.'
```

* * *

```
        public void right(int angle);   // rotates turtle right by angle degrees
        public void left(int angle);     //  rotates turtle left by angle degrees
        public void pendown();          //  puts turtle's pen down (turns ink on)
        public void penup();   //  puts the turtle's pen up (turns ink off)
        public void forward(int distance);    // moves forward distance steps.
                // if pen is up, doesn't draw anything; if pen is down, draws line
        public void forward(double distance); // optional.  Use double precision
                // variables for extra accuracy
}
```

(See also problems 5.21 and 9.29 in Deitel and Deitel.)

The second part of the project is to use your turtle to draw fractals. In the appendix, we describe turtle commands needed to draw one particular fractal, the Koch curve. Have your turtle make the Koch curve and/or other fractals of your choosing.

B. Fractal dimension. One fundamental property of a set is its dimension. A point is a zero-dimensional set, a line is one-dimensional, and a circle is two-dimensional. Fractals can have non-integral dimension. There are many ways to define the dimension d of a set, all of which reduce to the expected results for points, lines, and spheres. Here we will present what is called the capacity, or box-counting, dimension.

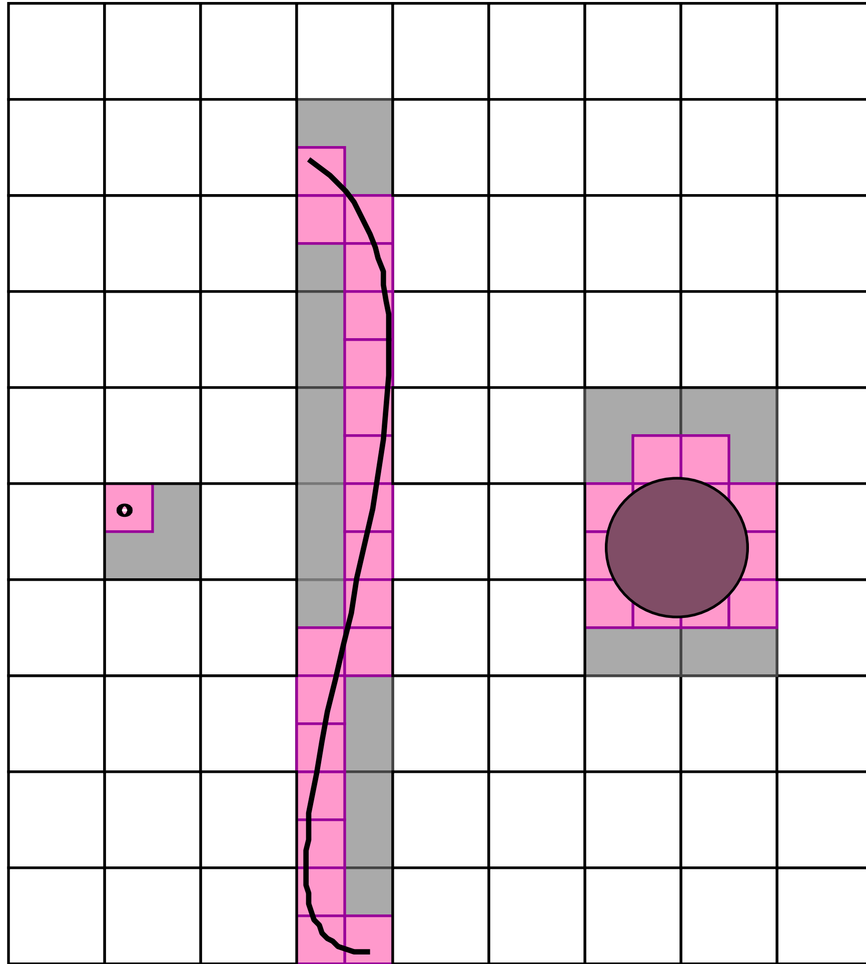
Assume the set in question is embedded in an N-dimensional space. (For example, figure 4.2 shows three sets embedded in a 2-dimensional space.) We imagine covering the space by a grid of N-dimensional boxes of side length , and then we count the number of boxes N( ) needed to cover the set. We do this for smaller and smaller values of , and then define the box-counting dimension d as the limit (if it exists):[1]

$$d = \lim_{\ 0} \frac{\ln N(\ \ )}{\ln(1/\ \ )} \quad . \qquad (4.1)$$

Figure 4.2 demonstrates that this definition of d gives the expected results for points, curves, and two-dimensional figures. The point is covered by a single box for any , so it has d = 0. The number of boxes needed to cover a line is proportional to 1/ , so it has d = 1.

---

[1]The limit in equation (4.1) does exist for the fractals that we discuss in this chapter.



Finally, the circle (including its interior) requires a number of boxes proportional to (1/)^2, so it has d = 2.

Figure 4.2 Covering procedure used to calculate box-counting dimension of a set.

Now we show that the middle-thirds Cantor set described above has a fractional box-counting dimension d. To do this, we use one-dimensional 'boxes' (line segments of length $_n$) with the particularly convenient sequence of lengths $_n = \frac{1}{3}^{n}$, and calculate the number of boxes needed to cover the set for each value of n.

For n=0, a single box of length 1 covers the set. When n=1, then 2 boxes are needed to cover the set (one for [0,1/3] and the other for [2/3,1]). When n=2, 4 boxes (covering [0,1/9], [2/9,1/3], [2/3,7/9],

and [8/9,1]) can cover the set. Similarly, for any n, the number of boxes needed to cover the set is 2^n. Therefore, using the definition of d in equation (4.1), we find for this set:

$$d = \lim_{n} \frac{\ln\left(2^n\right)}{\ln\left(3^n\right)} = \frac{\ln 2}{\ln 3} \quad 0.63 \qquad (4.2)$$

In applying equation (4.1), we changed variables from to n, replacing the limit from $\quad 0$ with the equivalent limit n . This is often convenient when computing the box-counting dimension of an object. As you might have guessed, the middle-thirds Cantor set has a dimension between 0 and 1; in dimensionality it lies somewhere between a point and a line segment.

Exercise 4.3. Find the box-counting dimension of the Cantor set constructed by leaving a fraction ‘f’ on the end of each line segment at each level.

Problem 4.2: Find analytically the box-counting dimension of the Sierpinski gasket. (Hint: Use triangle shaped ‘boxes.’)

C. Fractals in dynamical systems. Fractals arise naturally in the study of dynamical systems. In fact, the Cantor set itself comes up when one studies the "tent map," a one-dimensional map defined by:

$$x_{i+1} = f(x_i) = \begin{cases} A\,x_i & x_i \quad \frac{1}{2} \\ A - A\,x_i & x_i \quad \frac{1}{2} \end{cases} \qquad (4.3)$$

Problem 4.3. The tent map and the Cantor set. Investigate the map defined in equation (4.3). Show analytically that when A=3, the set of x points whose orbits are bounded (those having $|f^n(x)| <$ for all n) is the middle-thirds Cantor set. (Hint: first show that points outside the interval [0,1] always escape to infinity. Then consider which points inside the unit interval get mapped to places outside of it.) Other values of A generate other Cantor sets. What is the relationship between A and the fractal dimension of the set of points with bounded orbits? Assume A>2.

* * *

Menu Project. Julia sets. In this project you will investigate the map

$$f(z) = z^2 + c,$$

where z and c are both complex. We are again interested in the set of z whose orbits are bounded, meaning that $|f^n(z)| <$ for all n. See what this set looks like when the parameter value c = -0.5 + 0.5i. What happens when the parameter c is changed? You may find chapter 13 of Peitgen, Jurgens, Saupe, Chaos and Fractals: New Frontiers of Science, Springer-Verlag (1992) useful for this project.

C.1. Scaling in the Logistic Map. Now let's consider the period-doubling sequence of our old friend the logistic map:

$$x_{i+1} = f_r(x_i) = rx_i(1 - x_i). \tag{4.4}$$

Looking at the bifurcation plot you calculated in Required Project I, it is clear that the r values at which the period-doubling bifurcations occur get closer and closer together as the period length increases. Now we wish to examine the period-doubling sequence in detail. We will find that there is a fractal here, and moreover the fractal has properties (such as the fractal dimension) that are are not special to the logistic map they are identical for a huge variety of dynamical systems (they are "universal"). We will gain some insight into this universality in the next chapter.

But first, we will explore the properties of the period-doubling sequence of the logistic map. Specifically, we want to calculate a sequence of r values, r_0, r_1, . . ., r_n, which give the 2^n cycles. We now discuss how to perform this calculation.

From a computational point of view, it is inconvenient to try to calculate the bifurcation values of r. It is much easier to find the r values for which the cycles are superstable. (Recall that for superstable cycles of length N, the derivative of $f^N(x)$ vanishes.) Why? Because for any superstable 2^n-cycle, the point x = 0.5 must be an element of the cycle. This fact follows from the chain rule for the derivative of $f^N(x)$ and the fact that the derivative of f(x) vanishes at x=0.5 (see if you can prove it). This simplifies things

because now we only need to find r-values and not x-values: x = 0.5 is always an element of the cycle we are looking for.

So one can find a superstable 2^n-cycle by applying Newton's method to the function

$$\mathrm{F}(r) = \mathrm{f}_r^N(x = 0.5) - 0.5 \qquad (4.5)$$

where N 2^n. The r dependence of this equation is hidden inside f (hence the subscript). So to find the r-value, r_n, of the 2^n cycle, you just apply Newton's method to the variable r in equation (4.5), since whenever $F(r_n)$ is zero, we have a superstable 2^n cycle. To use Newton's method, we need the derivative of F with respect to r, which can be calculated directly using the chain rule.

The remaining detail is what to use for the initial r value in applying Newton's method to equation (4.5). Well, we already know that r_0 (the r for which the 1-cycle is superstable) is equal to 2 (the Floquet multiplier vanishes at this value   see Exercise 3.7). It is also true that r_1 must fall somewhere between 3 and 3.57 (since the 2-cycle is born at r = 3, and, as we shall see, r   3.57 is the maximum r value for period doubling). So a good guess for r_1 might be 3.2. For higher period doublings, it is important that we begin Newton's method at an r value that is a pretty good guess for the actual r value (otherwise we might end up with the r value of the previous cycle). To do this, suppose that we have calculated r_0, r_1, . . . , r_{n-1}, and we are now ready to use Newton's method to calculate r_n. Then the best choice for the initial guess of r_n is:

$$r_n \quad r_{n-1} + (r_{n-1} - r_{n-2}) / 4.7 \qquad (4.6)$$

You will see shortly where the number 4.7 comes from.

In Required Project II, we ask you to write an applet to compute a sequence of r values for the superstable 2^n-cycles of the logistic map using Newton's method. You should be able to get as far as n = 10. You should see that the r values are converging to something close to 3.57.

Why are we so interested in this sequence of r-values? Well, it turns out that they are converging to an accumulation value, r , at a geometric rate. That is,

$$r_n \quad r \quad - \quad ^{-n} \qquad (4.7)$$

* * *

where   is a number that we want to calculate. We can rewrite equation (4.7) as[2]

$$\frac{r_{n-1} - r_{n-2}}{r_n - r_{n-1}} \qquad (4.8)$$

Once you compute the r_n's, it is a simple matter to plug them into equation (4.8) to get an estimate of . We ask you to do this in Required Project II.

Exercise 4.4. Scaling and self-similarity. The geometric convergence of the r-values means that near r a plot of the values of r_n will look the same when it is magnified. Be sure that you understand why this is so. (You might find it helpful first to consider the sequence $r_n = (1/2)^n$, which has = 2 and r = 0, and to show that a plot of these points near r = 0 looks exactly the same if the scale of r is changed by a factor of two.)

C.2. Scaling in state space . We just discussed how the change in the parameter value r to go from a 2^n to a 2^{n+1} superstable cycle is characterized by a scaling factor . This is a scaling in "parameter space," since r is a parameter of the map. In Required Project II you will also demonstrate that there is also a scaling in x (which we can call "state space," since x describes the state of the system).

You will compute the $y_n = x_{2^{n-1}} = f_{r_n}^{2^{n-1}}(x = 1/2)$ for n = 2 through 10. You should find that the y_n's also converge geometrically with n:

$$\frac{1}{2} - y_n \quad (- \ )^{-n}, \qquad (4.9)$$

where

$$- \qquad \frac{y_{n-1} - y_{n-2}}{y_n - y_{n-1}}, \qquad (4.10)$$

and you will find .

---

[2] Equation (4.8) is obtained by taking equation (4.7) for the values n and n-1, and subtracting them, giving $r_n - r_{n-1} = \ ^{-(n-1)} - \ ^{-n} = \ ^{-n}(\ -1)$. Do the same thing for n-1 and n-2: $r_{n-1} - r_{n-2} = \ ^{-(n-1)}(\ -1)$. The ratio of these two equations is $\frac{r_{n-1} - r_{n-2}}{r_n - r_{n-1}} = \frac{^{-(n-1)}(\ -1)}{^{-n}(\ -1)} = \ $, which is equation (4.8).

* * *

Exercise 4.6 . Verify that the bifurcation diagram of the logistic map looks the same when it is magnified about the point (x = 1/2, r = r) by a factor - in the x-direction and in the r-direction.

C.3. Universality. The quantities and are of interest because these numbers turn out to be universal in the sense that they do not depend on the particular choice of mapping. Almost all maps that look roughly like the logistic map (i.e. having a single hump in the middle) will have the same values of and even though their values of r_n are different.[3]

In Required Project II you will produce some empirical evidence that universality holds by computing the scaling properties of the period-doubling sequence for other functions with quadratic maxima. You should find that and are the same as they are for the logistic map.

Appendix: Turtle fractals

In this appendix we describe how to construct a particular fractal, the Koch curve, using the turtle commands described in one of the Menu Projects. Recall that the turtle is constructed to obey the following commands:

F: move a specified distance forward (forward(DISTANCE))

R: turn right by a specified angle (right(ANGLE))

L: turn left by a specified angle (left(ANGLE))

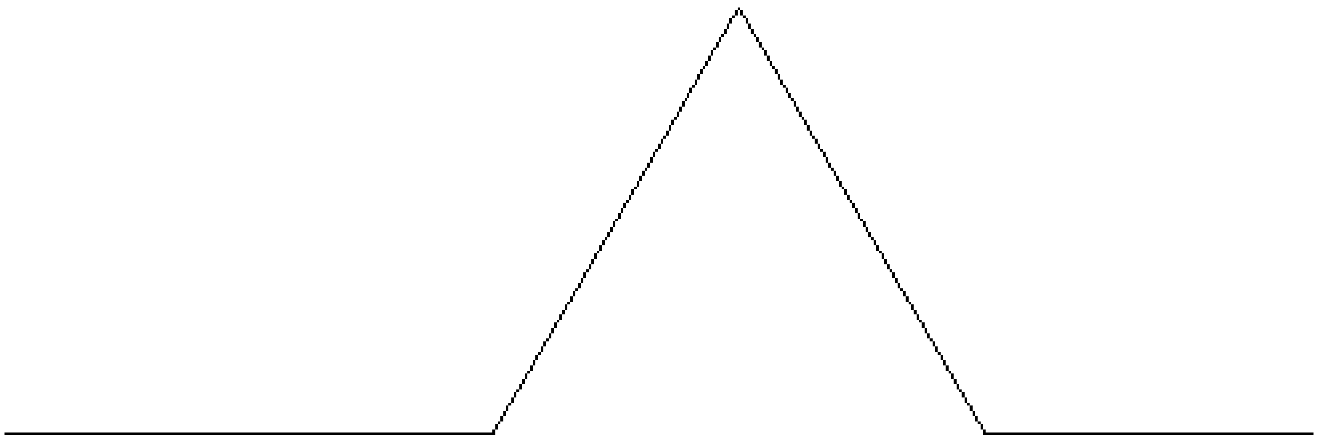
The Koch curve is defined in stages. Stage 0 is the base stage and consists of the command "F". To go to stage 1, replace the "F" by the sequence of commands "FLFRRFLF." If we are in stage n, we go to stage (n+1) by taking each "F" command in stage n and replacing it by the sequence of commands "FLFRRFLF." For example, the stage 2 command sequence is "FLFRRFLF L FLFRRFLF RR FLFRRFLF L FLFRRFLF."

The resulting curves for each stage are shown below.

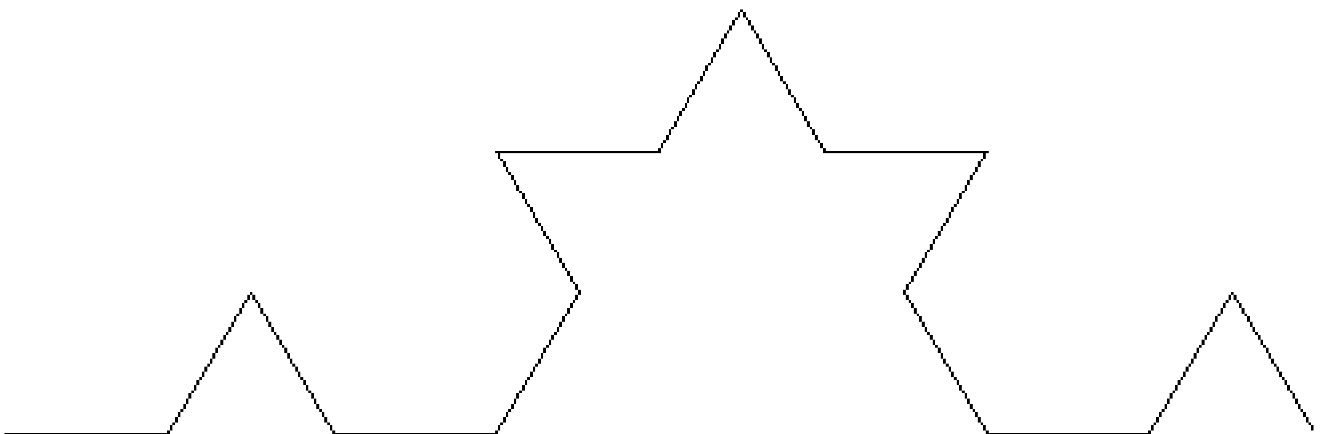
---

[3]The values of and will be the same as long as the second derivative of f(x) evaluated at the maximum point is nonzero. This is the general (or technically, the generic) case: "almost any" function with a hump has a quadratic hump; only very specially chosen functions have non-quadratic humps.

Here is the Stage 0 curve. The command sequence is F.

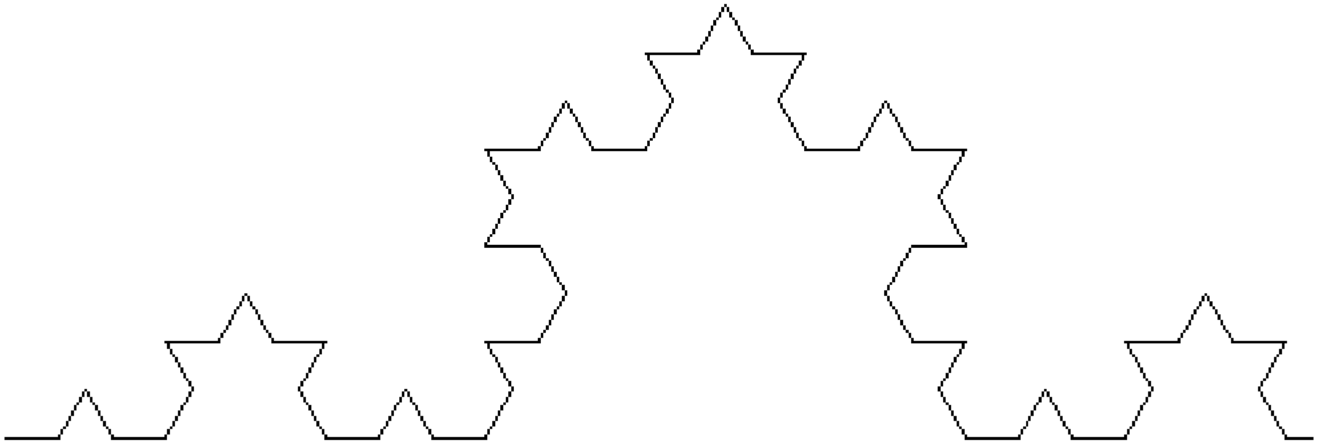


Here is the Stage 1 curve. The command sequence is FLFRRFLF

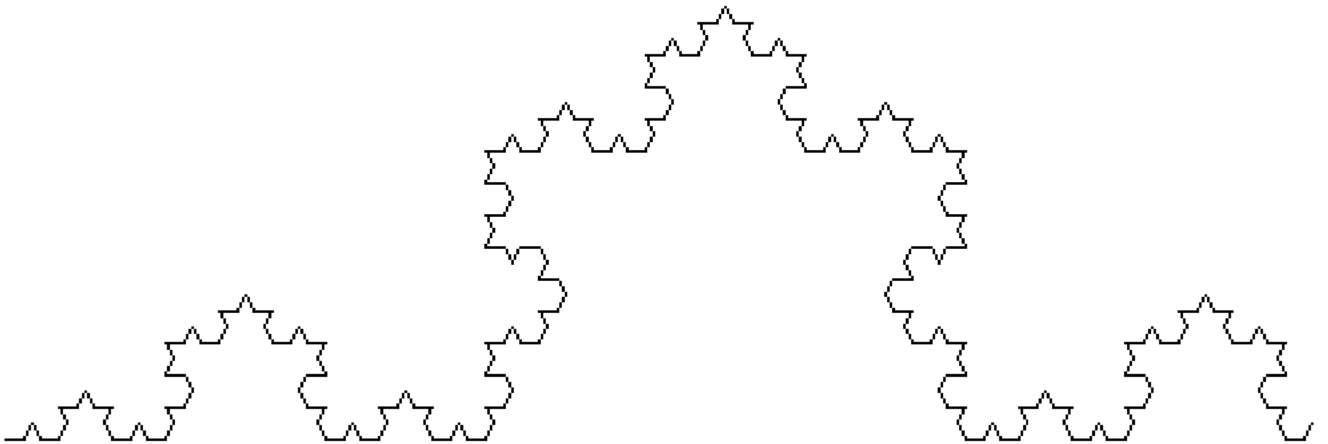


Here is the Stage 2 curve. The command sequence is FLFRRFLF LFRR FLFRRFLF L FLFRRFLF

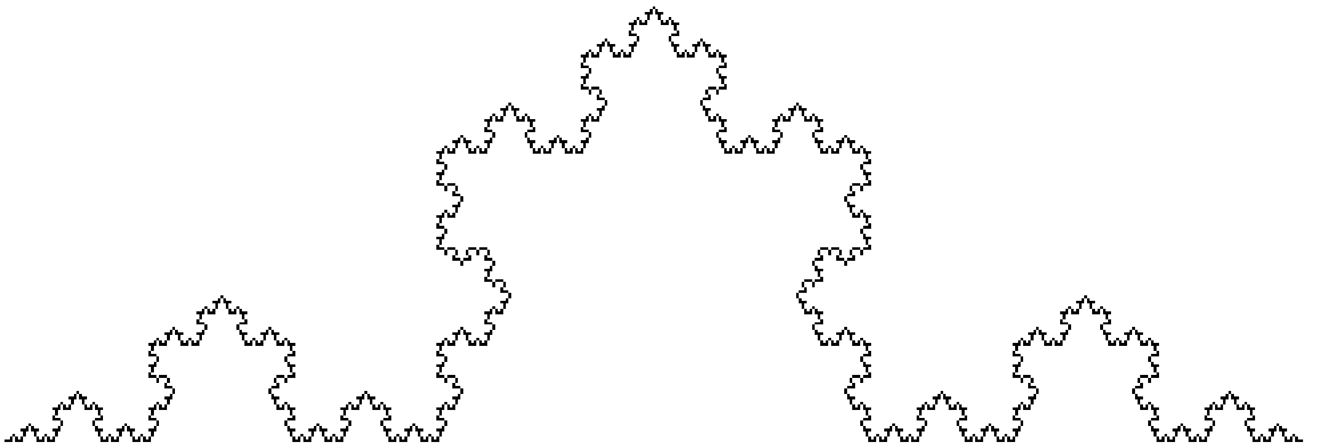
* * *



Here is the Stage 3 curve.

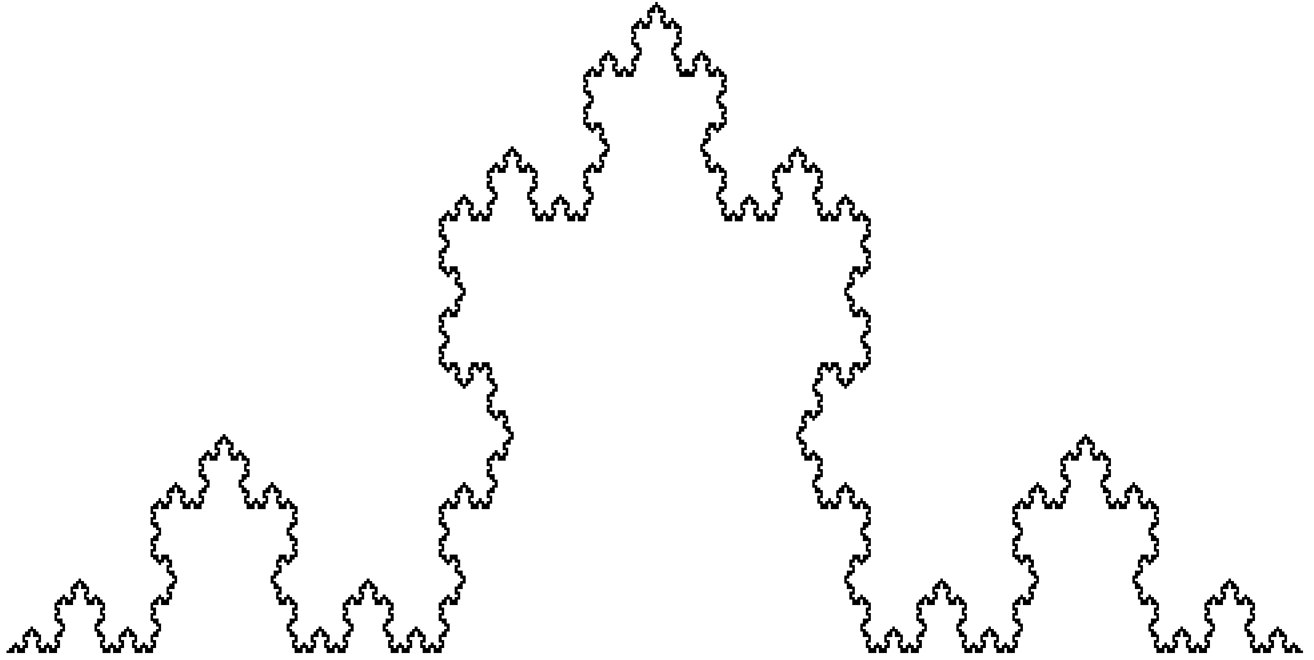


Here is the Stage 4 curve.



Here is the Stage 5 curve.

* * *



Here is the Stage 6 curve.

* * *