Fractals Series

Laboratory Guide

John Driscoll

With additional contributions from:
Bruce Marron, Melanie Mitchell, Vicki Niu, and Katherine Newton

"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."

—Benoit Mandelbrot

Introduction

The Fractals Series is designed to introduce fractals to a general audience. Readers will learn the concepts of iteration, scale, self-similarity, and fractal dimension, and will apply L-systems and Iterated Function Systems to explore fractals in mathematics, design and nature. Hands-on exploration (using NetLogo models) is emphasized.

The Fractal Series consists of this document and a series of NetLogo models. The NetLogo models are designed to be interactive, and each model offers direct access to the programming code that was used to create the model. The Fractal Series can be used as a stand-alone instructional module or in conjunction with the other instructional modules in the Virtual Laboratory of the Complexity Explorer website.

Figure 1: Tree and lightning, both displaying a fractal branching structure.

http://commons.wikimedia.org/wiki/File:Branches_on_a_rainy_day.jpg

http://commons.wikimedia.org/wiki/File:Blesk.jpg

To understand what fractals are, the first idea we must examine is that of self-similarity. Look at the picture of the tree (Figure 1, left) and imagine zooming in on one of the branches. You would see something that looks almost exactly like the tree itself. That is, the tree is *self-similar*. If the tree were a perfect fractal, the picture of a branch would be identical to the picture of the entire tree. Similarly, zoom in on any branch on the lightning structure and you will see something that looks very similar to the entire structure.

One consequence of this notion of self-similarity is that the spatial dimension of a fractal is typically not an integer. We are accustomed to the ordinary idea of dimension: lines are one-dimensional, squares are two-dimensional, and cubes are three-dimensional. Fractals are different; their self-similarity leads to them having *fractional* dimension.

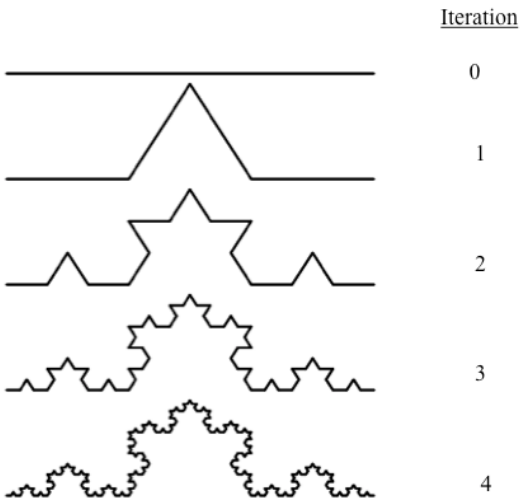


Figure 2: Iterations of the Koch Curve

To see how self-similarity can lead to fractional dimension, let's look at a simple example of a mathematical fractal, the Koch Curve, invented by Swedish mathematician Helge von Koch. To create the Koch Curve, we have a simple procedure that we repeat over and over at smaller and smaller portions of the shape. This, in the context of fractals, we call iteration.

Start with a line segment (Figure **2**, iteration 0). Now, divide the line segment in thirds, and replace the middle third with two segments that meet in an angle (Figure **2**, iteration 1). Now we have four line segments, each of which is 1/3 the length of the original line segment. Time to iterate: that is, repeat what we did before, but at a smaller scale. For each of the four line segments, divide the segment into thirds and replace the middle third by two angled segments, as before (Figure **2**, iteration 2), creating a figure with 16 segments, each 1/9 of the original segment. Now iterate again on each of the 16 segments (Figure **2**, iteration 3). If we were to keep doing this forever, we would get the Koch curve.

As you zoom in on the Koch curve, each smaller section is a copy of the larger whole. If you lived on this curve, there would be no way for you to gauge your decrease in size. You wouldn't know if you had become 9 times larger or 27 times smaller, because in both those instances, your surroundings would be identical. No matter what scale you are on, the Koch curve looks exactly the same. In short, the shape is "scale-invariant."

Now we are ready to informally define "fractal." A fractal is an object with self-similar structure at every scale. For example, at each successive iteration of the Koch curve, each part of the structure is a smaller-scale copy of the entire structure at the previous iteration. This self-similar structure can in principle go down to an infinitely small scale, if you iterate it far enough. The Koch curve is a mathematically "ideal" fractal. Real-world fractals such as trees and lightning aren't infinite or perfectly self-similar, but they can be thought of as approximating the ideal fractals of pure mathematics.

Benoit Mandelbrot famously applied the concept of fractals to the measurement of the coastline of Great Britain. If you measure Britain's very jagged coastline with increasingly smaller measuring devices, first at the level of kilometers, then with a meter stick, and even at the level of centimeters and below, you'll continue to see a self-similar jagged structure. As you use smaller and smaller measuring devices, the length you measure gets increasingly longer.

Laboratory 1: Examples of Fractals

Netlogo Model: *ExamplesOfFractals.nlogo*

In this laboratory, we explore some well-known mathematical fractals. To start, open ExamplesOfFractals.nlogo.

Let's start with the Koch Curve. In the NetLogo model, under Examples, press "Koch Curve." At the top of the black screen ("Worldview") , you will see the first iteration of the Koch Curve—a (green) straight line segment. Click "Iterate" to go to the next level. As you keep clicking "Iterate," the Koch Curve will continue iterating through the same levels described above. When it has iterated enough that it reaches the bottom of the Worldview, each subsequent iteration replaces the bottom figure.

If you look to the right of the Worldview, you will see a series of output boxes. The first few deal with fractal dimension, which will be explained in sections to come. For now, note that the Koch Curve, while made up of one-dimensional lines and existing in a two-dimensional plane, has a fractal (Hausdorff) dimension in between 1 and 2. Then, take a look at the bottom-most output boxes, which show how the fractal's length changes as you iterate it. Watch, as you iterate, how the length continually increases, although the area bounded by the curve remains the same. Much like the coastline of Great Britain, the length of a given fractal shape depends on the scale at which you measure it, or, in this case, on the number of iterations performed.

Explore the other examples of fractals, making sure to see how the successive iterations and scaled self-similarity produce the final image.

Laboratory 1 Exercises

1. Open the NetLogo Model *ExamplesOfFractals.nlogo.* Click on the "Cantor Set" button and click "Iterate" a few times.

    a) What happens to the length of the Cantor Set fractal? Why? Come up with the equation describing L, the length of the Cantor Set fractal as a function of n, where n is the iteration level.

    b) Describe the procedure by which the Cantor Set fractal changes at each iteration (as was done in the text above for the Koch curve).

2. Repeat Exercise 1, but with the Levy Curve.

3. Click on the "Sierpinski Triangle" button in *ExamplesOfFractals.nlogo.* The Sierpinski Triangle is a so-called space-filling curve. Describe its iteration procedure and explain why its Hausdorff dimension should be 1.585 (as indeed it is).

Laboratory 2: L-Systems

Netlogo Model: *LSystems.nlogo*.

One mathematical method for creating fractals is to use L-systems. L-systems are named after the Hungarian botanist Aristid Lindemayer who developed an extensive formalization of plant taxonomies using phyllotaxis (leaf arrangement) and defined the fractal structure of plants with L-systems. L-systems can likewise be used to investigate the basic properties of fractals. In this laboratory, you will learn how the fractals from Laboratory 1 can be created using L-systems, and also how to create your own L-system fractals.

To start, open *LSystems.nlogo*.

Click on "Koch Curve" under Examples. Then press "Iterate." The basic Koch curve pattern (Iterate 1) should appear in the worldview, with four yellow arrowheads and one green arrowhead spaced along the figure. These arrowheads are called turtles[1].

Each of the yellow turtles is at the beginning of a line segment; when "Iterate" is pressed again, their job will be to replace their line segment with a smaller copy of the entire figure. Look to the right of the Worldview in the output box labeled "L-system." You should see the following (rather cryptic) string:

t fd len lt 60 t fd len rt 120 t fd len lt 60 t fd len

The string is the L-system. It tells the computer how to create the fractal. Let's first work through the commands:

t : hatch a turtle

fd len : move the turtle forward by length *len*

lt 60 : turn the turtle 60^{o} to the left

rt 60 : turn the turtle 60^{o} to the right

(For other fractals, you will see L-systems with commands *pu* and *pd*, which refer to "pen-up" and "pen-down", respectively.)

At each iteration, each yellow turtle will carry out the commands specified in the L-system string. To see this, go back and press "Koch Curve" again. You'll see a line segment, headed by a single yellow turtle. The length of this line segment, *len*, is 18 (see "initial-length" slider on the right-hand side of the interface). When you press "Iterate," *len* is set to the previous value of *len* times the scale factor. The scale-factor is the amount that the line segment shrinks at each iteration. Here it is set to 0.33 (and can be reset by the slider). Then each turtle follows the commands of the L-system in sequence. Pretend that you are the initial single turtle, and use your own pen and paper to follow these commands—you should get a figure just like the one that appears in the window at the first iteration.

---

1 In NetLogo, the word "turtle" means an agent that can perform actions (see the NetLogo User Manual under the Help tab for more information).

If you pressed “Koch Curve” and then “Iterate”, in the Worldview you will see a figure with four yellow turtles. If you click Iterate again, each of those four turtles will carry out the L-system commands, but now with *len* shrunk by 1/3. The result is the second iteration of the Koch curve, now with 16 yellow turtles.

You can use this model to design your own fractal by creating your own L-system. Set the sliders on the right side of the interface to initialize the L-system, and then type your L-system commands into the L-system box. When you’re done, click on “Setup L-system” and then Iterate L-system.

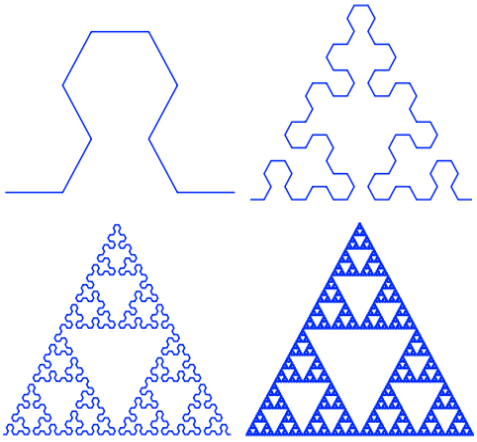
The “clear-display?” switch should be “On” if you want to clear the display after each iteration of the L-system.

Before designing your own L-system, we suggest you step through the preset examples and try modifying the initial length, the angle, and the scale-factor at first to view the different images generated.

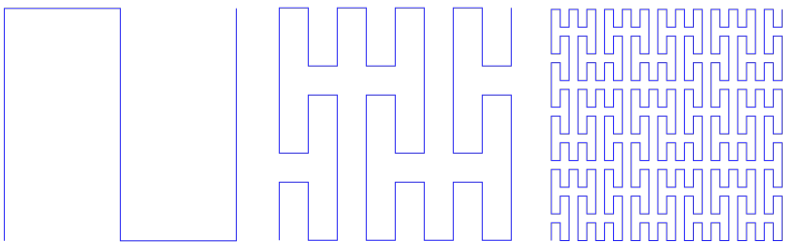
Laboratory 2 Exercises

1. Explain the L-system strings that create the Levy Curve and the Dragon Curve, as was done in the text for the Koch Curve.

2. .There is another way to implement the Sierpinski Triangle (illustrated below). Give the L-system for this version of the Sierpinski Triangle.



3. We will create another space-filling curve called the Peano Curve, shown below.



a) Describe the iteration procedure of the Peano Curve, as you did in for the Levy Curve and the Cantor Set in question 1.

b) Write the L-system string that tells the turtles how to create the Peano Curve.

4. Plants can also be modeled using L-systems. The Tree example n *LSystems.nlogo* provides a simple example of how this could be done. Find a plant or another kind of fractal appearing in nature and attempt to model it using L-systems.

Laboratory 3: Fractal Dimension: Hausdorff and Box-Counting Measures

Netlogo Models: *BoxCountingDimension.nlogo; AppliedBoxCounting.nlogo*

In this laboratory we will explore the concepts and measures of fractal dimension. As we mentioned before, fractals typically have fractional dimensions; that is, dimensions that lie between whole integer values. A fractal (fractional) dimension is more like a measure of density or a rate of growth towards infinity than a conventional description of space. For instance, the Koch curve, which was introduced in Section 4 above, has a fractal dimension of roughly 1.26. The Koch curve becomes infinitely long as the number of iterations approaches. In an intuitive sense, the Koch curve lies in a nebulous zone somewhere between a one-dimensional line and a two-dimensional plane. The Koch curve is composed of line segments, which are (topologically) one-dimensional, "embedded" in two-dimensional space (a plane). As the line segments increase in number, they increasingly "fill" the plane yet never become two dimensional; they always remain between one and two dimensions.

It turns out that there are many different ways of measuring fractal dimension. In this laboratory, we focus on two methods: the so-called "Hausdorff dimension," and the "box-counting" method. All the methods of measuring fractal dimension share the goal of quantifying something about the multi-scale roughness, irregularity, and self-similarity of fractals. In her book *Complexity*, Melanie Mitchell put it this way: "fractal dimension 'quantifies the cascade of detail' in an object. That is, it quantifies how much detail you see at all scales as you dive deeper and deeper into the infinite cascade of self-similarity."

Hausdorff Dimension

The Hausdorff dimension (named for mathematician Felix Hausdorff) applies to ideal mathematical fractals such as those we have been studying in the past two labs. When dealing with such ideally self-similar fractals, there are two important steps in each iteration. First, we take the original figure and **scale it down** by some factor: for example, the Sierpinski triangle gets scaled down by 1/2 and the Koch curve by 1/3. Then, we take those scaled-down figures and **make a number of copies** of them. Following our previous examples, we make three copies of the Sierpinski triangle, and four of the Koch curve. We then place those copies to form a new figure. Hausdorff dimension is defined in terms of the scaling factor M and the number of copies N. Note that the value of M is not the scaling fraction (1/2 or 1/3) , but rather the reciprocal of the fraction. Thus M = 2 for the Sierpinski triangle and M = 3 for the Koch curve. For the Sierpinski triangle, N = 3 (each triangle is replaced by three smaller triangles), and for the Koch curve, N = 4.

In general, a self-similar fractal object obeys the following relation:

$$N = M^d$$

The exponent *d* is the Hausdorff dimension[2]. You can solve this equation for *d* by taking the logarithm of both sides and re-arranging:

$$d = \log N / \log M$$

What this tells us is that the dimension of a fractal increases as the number of copies increases, and decreases as the scaling fraction decreases. (Note that *d* will be the same no matter what the base of the logarithm as long as the logs in the numerator and denominator have the same base.) This measure of dimension can be applied to objects that we are familiar with, such as lines, squares, and cubes, where the result is exactly as expected (see Melanie Mitchell's book, *Complexity: A Guided Tour*, for a more detailed discussion of this).

Let's apply the formula for Hausdorff dimension to verify that the dimension of the Koch curve is 1.26, as stated above:

$$d_{Koch} = \frac{\log(N)}{\log(M)} = \frac{\log(4)}{\log(3)} \cong 1.26$$

.

Box-Counting Dimension

You can see that if we know N and M, we can figure out the Hausdorff dimension, and we can certainly figure them out if we are actually creating the fractal on a computer. But what if we don't know N and M, or what if N and M are not clearly defined, as in physical fractals such as the lightning bolt or the tree? The box-counting dimension is an approximation of the Hausdorff dimension that can be calculated for physical fractals that have no precise values for M and N.

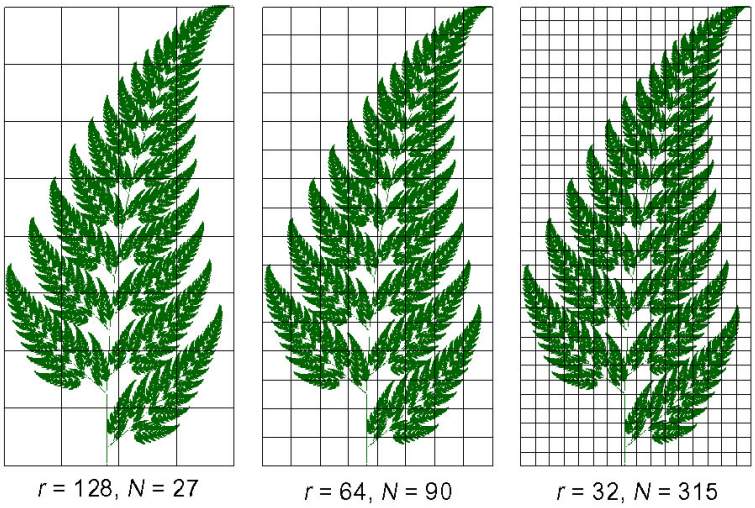


Figure 3: Illustration of box counting. (http://www.theomparticle.com/HomogeneityAndFractality.html)

[2] The formal definition of Hausdorff dimension has some additional mathematical complications beyond the scope of this guide.

The process of computing box-counting dimension starts by overlaying a grid on an image and counting how many lattice sites or 'boxes' are necessary to completely cover every part of the image. For example, consider the leftmost fern image in Figure 3. Any box containing part of the fern is counted, and any box not containing part of the fern, such as the top-left box in the grid, is not counted. The boxes are perfect squares defined by the length (or *scale*) r of a box side. Here, the scale plays the role of a ruler of a particular length. Once the boxes in the initial grid are counted, the process iterates. At each step, a new grid with a smaller scale is placed over the figure (Figure 3, middle and right), and the boxes in that grid that contain part of the figure are counted. These smaller boxes correspond to using a smaller "ruler". This process continues for a desired number of steps. Finally, for each step of the iteration, the log of the number N of boxes is plotted versus the log of $1/r$, and from these points a best-fit line is drawn. The slope of this best fit line is the box-counting dimension of the figure.

Let's go through a simple example of this process, using the grids shown in Figure 3. The values of r are given in pixels. Using a 128 x 128 pixel box, 27 boxes are required to cover all parts of the fern. With a 64 x 64 pixel box, 90 boxes are required. And with a 32 x 32 pixel box, 315 boxes are needed.

Next, plot the values of log N versus log $1/r$ and manually draw the "best-fit" line through the data (Figure 4). The slope of this line, here ~1.7, is the box-counting dimension.

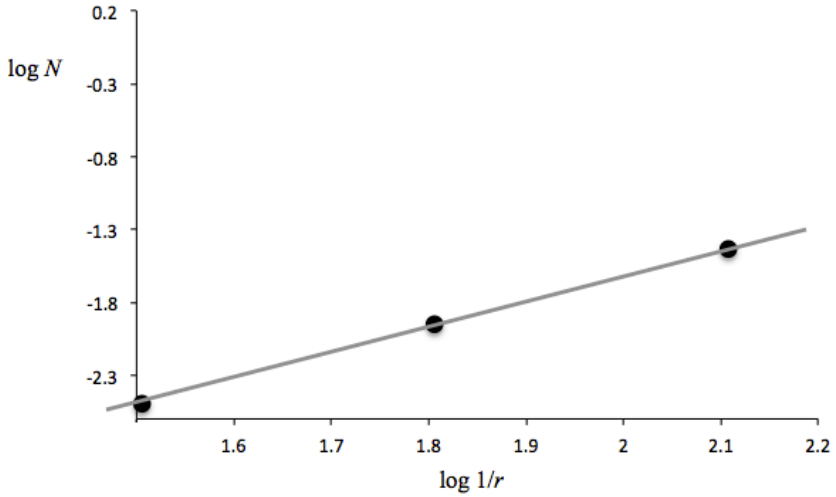


Figure 4: Log N versus log r, where N is the number of boxes and r is the box scale at a particular iteration of the box-counting process. Here we use base 10 log, but the slope of the line will be the same no matter what base is used.

Note that this procedure can result in a poor approximation to Hausdorff dimension if there are too few iterations.

Running BoxCountingDimension.nlogo

In this section we describe the NetLogo model called BoxCountingDimension.nlogo. In this model, boxes are implemented as turtles designed to detect and cover patches of defined colors. The box-counting algorithm first exploits the local environment with a focused search and then explores the environment in an unfocused search looking for additional non-continuous patches that have not been covered.

The interface consists of three blocks: Fractal Examples, Box-Counting Controls and Plots/Results. Start by selecting one of the fractal examples, e.g., "Koch Curve". Then click "Iterate". Notice at the first iteration the Hausdorff dimension (1.262 for the Koch Curve) appears in a monitor on the right. Click "Iterate" a few more times to get a developed fractal. Now you are ready to apply box-counting to this fractal.

Under Box-Counting Controls, set "initial box length" (i.e., length of side of each box) and "increment" (i.e., amount that box length increases per iteration of box counting). Press "Box Counting Setup" and then "Box Counting Go" and you should see small red boxes covering the fractal. After the fractal has been completely covered, the log of number of boxes will be plotted versus the log of 1/box-length on the plot on the right. These iterations will continue until you press "Box Counting Go" again to stop the process. At that point you can press "linear-regression" on the right side (under the plot) to fit a line to the plotted points. The slope of this line is the box-counting dimension measured by your run; it appears in the monitor on the right, and can be compared with the Hausdorff dimension.

As you experiment with different settings and examples, notice that the box-counting dimension becomes a more accurate approximation of Hausdorff dimension as the number of iterations increase. The accuracy also can depend on the initial settings for box-length and increment (you should experiment with all of these parameters). However, you have probably noticed the iterations taking longer and longer to generate for each successive iteration. This is because the number of line segments is increasing exponentially. The Koch curve, for instance, begins with four line segments but has over a million line segments after ten iterations.

Applied Box-Counting

Open the *AppliedBoxCounting.nlogo* model. With this model we will see how the box-counting dimension, unlike the Hausdorff dimension, can be used in the real world. Click on the drop-down menu under “image,” select the “coastline” option, and click “Image Setup.” You should see a black and white image of the coastline of Great Britain (a famous example employed by Beniot Mandelbrot, discussed in his book *The Fractal Geometry of Nature*). Y ou can set the initial box size using the “initial-box-length” slider. The box length is measured in pixels. Now click on “Box Counting: Setup.” You should see the first (red) box appear on the coastline. Click “Box Counting: Go.” The program will start applying the red boxes over the entire coastline. You can set the increment (i.e., how much the box size increases at each step) using the “increment” slider. Each time the program covers the coastline in boxes, it counts the number of boxes N and plots $\log(N)$ versus $\log(1/r)$ (r is the size of the box). After the program has run for several iterations, click “Box Counting: Go” again to stop the program’s iterations (or go to the Tools menu and click “Halt”). Then click “Linear Regression.” The program will draw the line it fit to the points, and output the box-counting dimension, best fit equation, and the R^2 value (the correlation coefficient that indicates how well the data matches up with the best-fit line).

In addition to the two images provided, you can upload your own images on which to run box counting. Your image file needs to be in .png format, have square dimension, and be called “custom-image.png”. You also need to identify the color of the pixels that you want box-counting to measure, and find the corresponding color value from the Netlogo Color Swatches in the Tools menu. The easiest way to do this is to make the color of interest black, and set color-value to 0. Make sure that the image is square and has no black borders; the only black in the image should be part of the figure itself. Note that the larger the image, the longer it will take to run Image Setup.

Laboratory 3 Exercises

1. Experiment with running box-counting on the Koch Curve fractal, using *BoxCountingDimension.nlogo*. How does changing the initial box length, increment, and number of iterations affect the box-counting dimension calculated by the program? Does increasing number of iterations increase the accuracy of the approximation to the Hausdorff dimension?

2. Using *BoxCountingDimension.nlogo,* which of the fractal examples has a box-counting dimension most different from its Hausdorff dimension? Speculate on what causes this difference.

3. Various researchers have published the fractal dimension they measured for various coastlines. Do an online search to find estimates for the fractal dimension of the coast of Great Britain, and compare these estimates to the value you find using *AppliedBoxCountingDimension.nlogo* with the "coastline" image. How well does the NetLogo model's value on this simple image agree with published values?

4. Try putting some of the images you create using *LSystems.nlogo* into the *AppliedBoxCounting.nlogo.* Since you created these fractals, you also know *N* and *M.* How does the Hausdorff dimension of, say, the Peano Curve compare to its box-counting dimension?

Laboratory 4: Iterated Function Systems

Netlogo Models: *AffineTransformations.nlogo; IteratedFunctionSystems.nlogo*

Iterated function systems provide an alternative framework for creating fractals. To explain how an iterated function system (IFS) works, we'll first cover the topic of linear affine transformations. These include transformations such as rotation, scaling, and translation. These transformations can be expressed in terms of matrix multiplication and vector addition. If you have never worked with matrices before, you might find it helpful to complete the "Matrix Algebra" tutorial on the Complexity Explorer website.

Affine Transformations

The formal definition of an affine transformation is fairly technical, and we won't go into the details. For the purposes of this model, an affine transformation will be considered a linear transformation. (There is a difference between them, but for our purposes it won't be too important). The distinguishing quality of linear affine transformations is that they preserve the relations of points on the line with respect to one another. In other words, lines always remain straight, and are never bent or twisted.

Let's first define the notions of *vector* and *matrix*. A vector, as we use the term here, will be an arrow pointing from the origin to a point the Cartesian plane. A matrix is simply a two-dimensional array of numbers. Note that there is an equivalence between points and vectors: when I talk about a point in the Cartesian plane, I can equivalently talk about a vector pointing from the origin to that point. There is also an equivalence between linear transformations and matrices. We usually say that a matrix is a representation of a linear transformation.

Assume that we have a point in two dimensions, described by coordinates (x,y). On that point, or equivalently, that vector, we can perform three different kinds of linear affine transformations:

1. Translation
2. Scaling
3. Rotation

That is, we can take the point (x,y) and translate it a distance d in the x direction and a distance h in the y direction as follows: $(x,y) + (d,h) = (x+d,y+h)$. We can scale the point by multiplying it by a number c, as in $c(x,y)=(cx,cy)$. In essence, scaling consists of taking the point and stretching it by a distance c away from the origin. (There is another, more complicated kind of scaling that we will describe shortly.) Finally we can rotate the point, as we'll describe below.

Suppose now that you want to stretch the x coordinate by some amount and the y coordinate by a different amount. Then multiplying (x,y) by c won't be good enough. We will need to multiply (x,y) by some object so that we will get something that looks like (ax,by). This operation will be referred to as "dilation.

The dilation operation is written as a matrix multiplication:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} * \begin{pmatrix} x_t \\ y_t \end{pmatrix} = \begin{pmatrix} x_{t+1} \\ y_{t+1} \end{pmatrix}$$

Here, a represents the horizontal dilation and b the vertical dilation.

Rotation is expressed as yet another matrix multiplication:

$$\begin{pmatrix} \cos(\theta_x) & \sin(\theta_y) \\ -\sin(\theta_x) & \cos(\theta_y) \end{pmatrix} * \begin{pmatrix} x_t \\ y_t \end{pmatrix} = \begin{pmatrix} x_{t+1} \\ y_{t+1} \end{pmatrix}$$

In this case, θ_x represents the rotation of horizontal lines, and θ_y the rotation of vertical lines. Some of you may note that the translation, dilation and rotation operations can be combined into once step, making the general equation for an iteration of the linear transformation the following:

$$\begin{pmatrix} a * \cos(\theta_x) & b * \sin(\theta_y) \\ -a * \sin(\theta_x) & b * \cos(\theta_y) \end{pmatrix} * \begin{pmatrix} x_t \\ y_t \end{pmatrix} + \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = \begin{pmatrix} x_{t+1} \\ y_{t+1} \end{pmatrix}$$

Now, open the NetLogo model labeled *AffineTransformations.nlogo* to explore linear affine transformations.

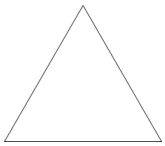
Here, you can investigate how linear affine transformations operate on points, and the concept of iterative transformations. Select a random point with the x- and y-coordinates between −10 and 10. You will see the point appear, along with a line from the origin to that point: this line represents the position vector.

Having selected your initial point, choose a degree value (0-360°) for the rotation along x and y, a scalar (any real number) for the scale-factor along x and y, and some offset for the translation. If you press the "Transform" button, you will see this new point appear, along with its predecessor If you look in the window labeled "Transformation Operations", you will see the matrix multiplication and vector addition that led to the creation of this new point.

Repeat the transformation a few times, noting how the point generated by the first transformation serves as the input for the second transformation, and so on. Experiment with different rotations, scales, and translations to get a feel for the transformations. You can even try to calculate the new point yourself, and compare your results with the model's output.

Using Iterated Function Systems to Generate Fractals

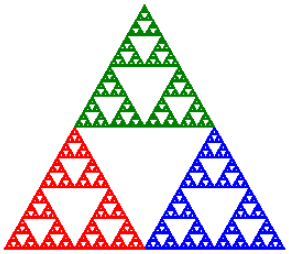
Now we will look into how to apply iterated function systems to the task of generating fractals. First, let us explore how we have traditionally thought of creating fractals. Consider the Sierpinski triangle, a familiar example. We start with a single triangle:



Then we take that triangle and create three copies, each side being 1/2 the size of the original. One of the copies remains stationary, the other is translated halfway across the bottom side, and the last translated a quarter of the way across the bottom side and halfway up the height of the triangle.

If we repeat this process of dilation, copying, and translation once more, we get the next iteration.

If this process continues *ad infinitum*, the image becomes something close to the following:



Thus, the concept of iteration arises: taking a particular operation and applying it to an initial object, and subsequently to the results of the first operation.

You may have noted that we can express the steps taken to obtain the last image in terms of an affine linear transformation. With the Sierpinski triangle, we would express these steps in terms of three different linear transformations:

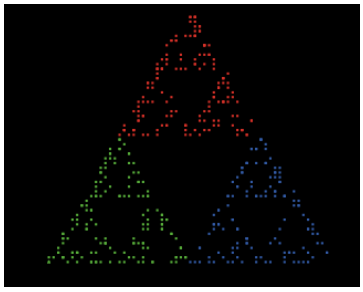
Dilate by ½ (to get the red image)

Dilate by ½, translate right by $\frac{1}{2}$ (to get the blue image)

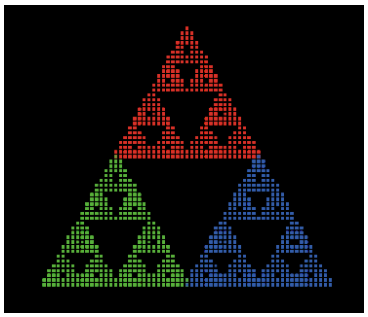
Dilate by ½, translate right $\frac{1}{4}$, translate up $\frac{\sqrt{3}}{2}$ (to get the green image)

With iterated function systems, the process starts with choosing a random point and transforming it over a series of iterations, each time with one of the three linear transformations listed above. The points corresponding to each transformation are drawn, here in one of three colors, depending on which transformation was used to create the point.

The first several hundred iterations will render an image which does not quite resemble the intended fractal:

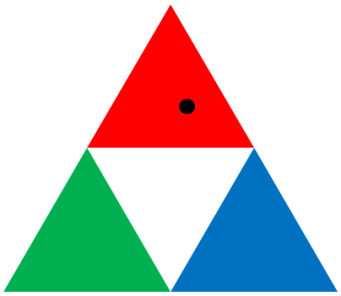


However, as the number of iterations approaches a few thousand, an image remarkably similar to the desired endpoint arises:

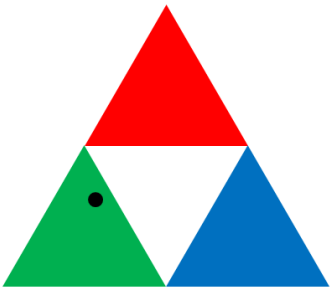


Let's go into a little more detail. We start with a random point. At each iteration, we probabilistically choose a transformation (out of our three options) to apply to it. The probability of choosing a particular transformation is equal to the determinant of that transformation's associated matrix divided by the sum of the determinants of all the transformation matrices. This transformation process, like in the Linear Affine Transformations model, is iterative, meaning that the first transformation's result becomes the second transformation's input. In the image above, the three different colors correspond to the three possible transformations.

It may seem mystifying as to why the colors correspond to the triangles in the way they do. To get some intuition on this, let us start by assuming that the first random point lies inside of the large triangle, like so:

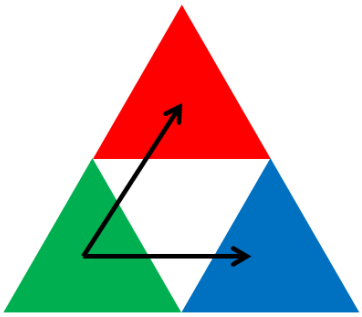


Here, the differently colored sub-triangles represent the corollaries to the regions in the IFS-generated Sierpinski triangle. If the black dot were the original point, we can assume that, regardless of which transformation it undergoes, it will be scaled by ½. So let's first scale it by ½. It may be a bit difficult to see this without axes, but this takes the black dot inside of the green sub-triangle, as shown:



Note that any point inside of the large triangle, when scaled by ½, will move inside of the green sub-triangle. Why is this? If we scale down the larger triangle by ½, we will get the green sub-triangle. This means that all of the points inside of the larger triangle, when scaled by ½, become points in the green triangle. That is why all points that have had the first transformation performed to them lie in the same area.

Having explained the location of the green points, it is simple extrapolation to get to the blue and the red. The only difference in the three transformations is the translations: the green has none, the blue a horizontal translation, and the red a diagonal translation. Thus, if all points when scaled by ½ become the green triangle, it follows that all points when scaled by ½ and translated left land in the blue triangle, and so on for the red triangle.



Now, what happens if the randomly chosen point lies outside of the original (large) triangle? Here, it is important to note that the leftmost vertex of the green triangle will always appear at (0, 0), regardless of the starting position. Second, the first few hundred iterations of an IFS generator are generally discarded.

Assume that the translations are as follows:

$(green)\ \begin{bmatrix}0\\0\end{bmatrix}$

$(blue)\ \begin{bmatrix}30\\0\end{bmatrix}$

$(red)\ \begin{bmatrix}15\\27\end{bmatrix}$

All of the transformation matrices are:

$$\begin{bmatrix}0.5 & 0\\ 0 & 0.5\end{bmatrix}$$

This will generate a Sierpinski triangle with side length 60. If we start with a point outside this triangle's bounds, how do we end up inside? Each transformation will halve the x- and y-coordinates, and as we continue to do this repeatedly, the x- and y-coordinates will both approach 0. There is also a 2/3 probability that we will translate the point, increasing its x- and y-coordinates. However, for these translations to increase the value of the point's coordinates, the point must already lie within the triangle (Challenge: calculate the inequalities that prove this!). Thus, no matter where we begin, applying these transformations iteratively will always land us inside the Sierpinski triangle. From there, we have already shown why the different transformations correlate to the different regions of the triangle.

The Sierpinski triangle is an easier example to understand in this way than some other fractals, as there is no rotation involved, but similar rationales apply for all IFS fractals. Now, it's time to explore some of these other fractals.

The NetLogo Iterated Function Systems Model

Open the NetLogo model labeled *IteratedFunctionSystems.nlogo* to explore iterated function systems.

Let's begin by generating one of the Fractal Examples, starting with the familiar Sierpinski triangle. Press the *Sierpinski Triangle* button under "Fractal Examples", and then *Go forever*, under "Procedures". This will continually iterate the model (after first discarding some initial number of iterations). You should see points fill in on the screen, gradually creating the fractal. To view the process step-wise, hit the *Go 100 iterations* button, which will advance by 100 iterations. On the right-hand side, there are various input boxes. The column labeled "Rotation" allows you to input different values for the rotation of the horizontal and vertical lines for the transformation. The different rows, separated by gaps, are the different linear transformations the point undergoes. The next column is labeled "Scale", and allows you to input the horizontal and vertical dilations. Next to that column is the "Translation" column, where you can input the translation vectors for the various transformations. The output boxes on the far right represent the transformation matrix, which incorporates the information from the rotation and scaling inputs.

Laboratory 4 Exercises

1. Practice creating linear transformations: Make up a linear transformation (involving at least rotation and dilation) and an initial point. Compute the effect of the linear transformation on the initial point by hand. Check your work by implementing the transformation in *AffineTransformations.nlogo*.

2. Take a vector pointing from the origin in the Cartesian plane to a point *(x,y).* We wish to rotate the vector by an angle θ. Call the rotated vector (*x',y').* Find *x'* and *y'* in terms of *x, y*, and θ. [Equivalent alternative: take the axes *x* and *y*. Draw rotated (by θ) axes *x'* and *y'*. Write *x'* and *y'* in terms of *x, y*, and θ.] Show that you get the same thing as you would if you multiplied the vector (*x,y*) by the matrix discussed in the section on rotations.

3. In the text above, we discussed how, in the IFS for the Sierpinski triangle, at each iteration, each of the linear transformations has a specific probability of being chosen. Write down each transformation matrix for the three Sierpinski triangle transformations. Calculate their determinants. Find the probability of each matrix being chosen, according to the rule discussed in the text: the probability of a transformation being chosen is equal to the determinant of that transformation matrix divided by the sum of the determinants of all the transformation matrices. Does this rule make sense? Why? [Hint: what does the determinant represent?]

4. In the interface, create three inputs for the probabilities for each of the three linear transformations, to be used instead of the probabilities based on the matrix determinants described above. (Hint: look under the "globals" section in the code to find the right variables). For a given fractal in Fractal Examples, how does the fractal change when you input particular probabilities?