Monte-Carlo Methods for computing the dimension of fractals

Benjamin Fattori - December 2018

Introduction

The purpose of thisproject is to explore the method of computing the box-counting dimension of a fractal using Monte-Carlo random sampling. I chose this project as I have an interest in dynamical systems. The topic of computing fractal dimension using Monte-Carlo methods sounded very interesting to me and I have created my own method/algorithm for estimating the dimension of some fractals.

Mathematical Background

This project will involve computing the dimension of 2 different types of fractals. In theory, fractal dimension is defined by its **Hausdorff Dimension**. This is a fine definition for theory, however, we cannot really use this to compute the dimensions of any sets nicely. In practice, we use what is called the **Box-Counting Dimension**.

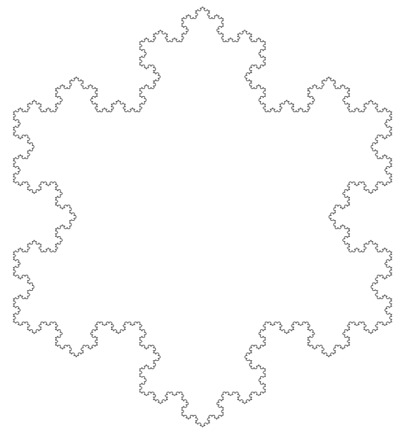
Definition. Given a set, S, the Box-Counting dimension is defined as:

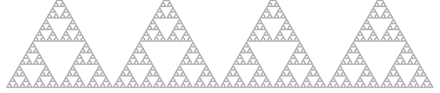
$$\dim(S) = \lim_{\epsilon \to 0} \frac{log(N(\epsilon))}{log(1/\epsilon)}$$

Where $N(\epsilon)$ is the number of boxes of side length ϵ required to cover the set.

The box-counting definition is a bit easier to work with computationally and for the fractals we will be working with, the dimensions will coincide

The first type of fractals we will be calculating the dimension for will be **Iterated Function Systems**: roughly, these are a way of generating self-similar shapes by starting with a base set, then repeatedly scaling down the set and piecing it together. We will calculate the dimension of 2 iterated function systems, the Koch Snowflake and the Sierpinski Triangle:





The next kind of fractals we will compute the dimension for are **Time-Escape Fractals**. These fractals are defined to be sets of points that do not escape to infinity when repeatedly iterated by their defining function. We will deal with the **Julia Set**, $J(c)$ which is defined by:

$$J(c) = \{z \in \mathbb{C} : |f_c^n(z)| \leq 2 \, \forall n \in N\} \text{ and } f_c(z) = z^2 + c$$

The Julia Set we will be analyzing is the Dendrite Julia set, it is the Julia set, $J(i)$:

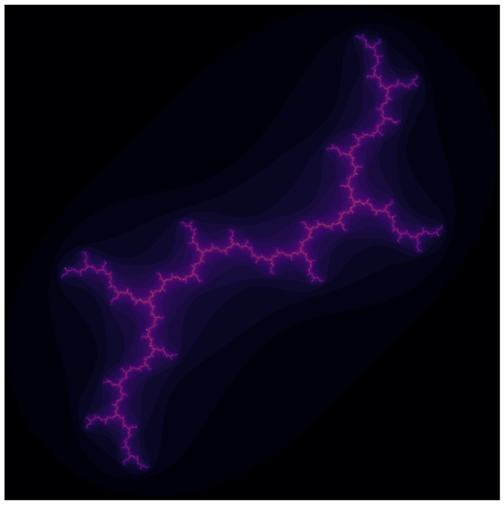


Figure 1: The Dendrite Julia Set

Methods and Pseudo-code

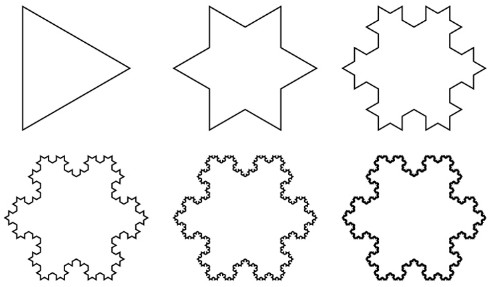
Both types of fractals had the same basic algorithm for computing their dimensions. The method worked as follows:

- Import our fractal into Python as a 2-d array
- Choose a set number of points to randomly choose. Call this value `N` and choose a size for our box, call this value, `k`.
- Using `np.random.randint()` twice, select `x,y` coordinates for the center of our box.
- Compare this box we have created to our list of previous centers to ensure none of the boxes overlap. We use a method that ensures that any 2 midpoints for boxes of side length, L are not closer than $\frac{2L}{\sqrt{2}}$.
- From the center of the box, look at the fractal array components, `[x-k:x+k,y-k:y+k]` and compute the average of the values in this box using `np.mean`.
- Using our pre-determined conditions for the acceptable averages for a box lying inside our fractal, decide whether the chosen box is inside our fractal.
- If the box is inside our fractal, add 1 to the sum of all boxes covering our fractal. As well, add this box center to our array of box centers.
- Calculate the total sum and return the dimension, `np.log(sum)/np.log(1/epsilon)`.

For the Koch-Snowflake, we will do extra explaining with pictures to illustrate the method.

The Koch Snowflake

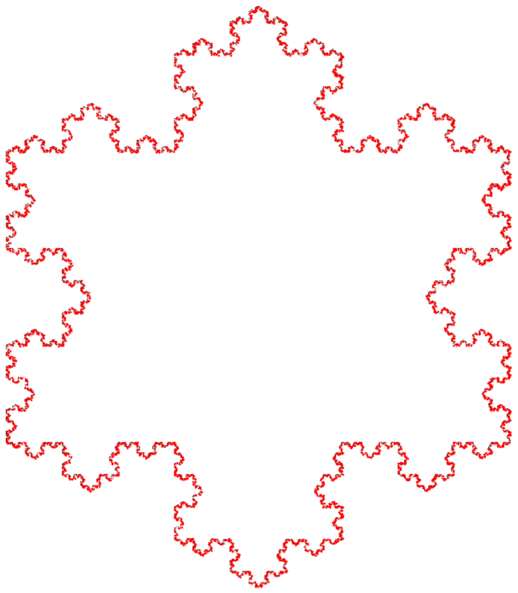
The Koch-Snowflake is created via the following method on an equilateral triangle:

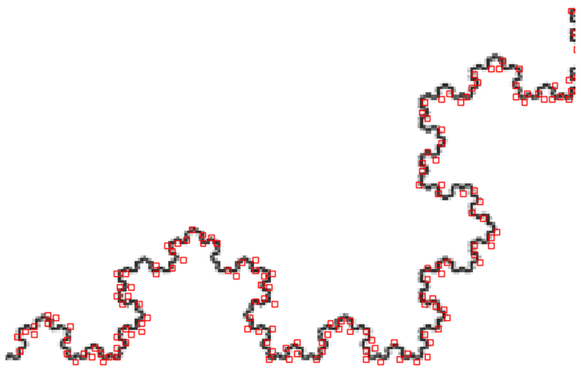


The calculated Hausdorff Dimension of the Koch Snowflake is [1]

$$\frac{2\ln 2}{\ln 3} \simeq 1.26185$$

We will be working with a 1717×1717 pixel image of the 10th iteration (see the image of the Koch-Snowflake above). [1] When importing this image into python, we are treating an average value of 1 as being totally inside our outside of our fractal. After running for 500000 iterations, we get the following covering:



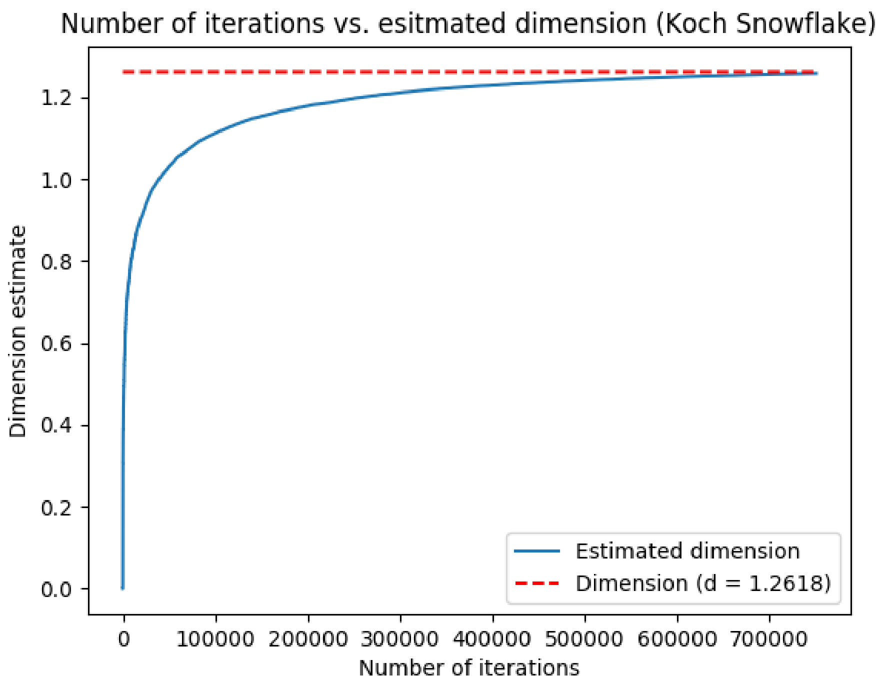


This looks to be a decent covering of the set and illustrates how the method works. Running the code for different N values gives the following:

N **Value**	**Dimension Estimate**
50000	1.0379008485077195
100000	1.1155621331284706
500000	1.240290044525778
750000	1.258359579040857

[1]We have created the image for this fractal and the Sierpinski Triangle using a website cited in our references

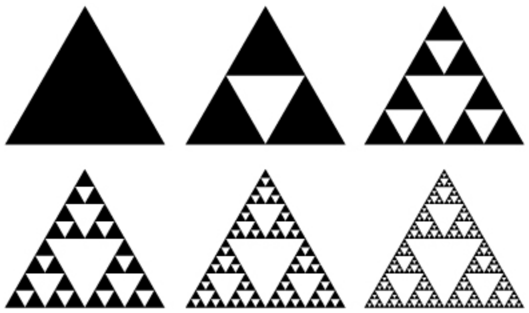
Here is the plot for the computation of dimension over the iterations:



This shows a very definite convergence towards the correct value for the dimension. This is a very accurate result using only 25% of the total points!

The Sierpinski Triangle

The Sierpinski Triangle is another iterated function system created using the following method on an equilateral triangle:



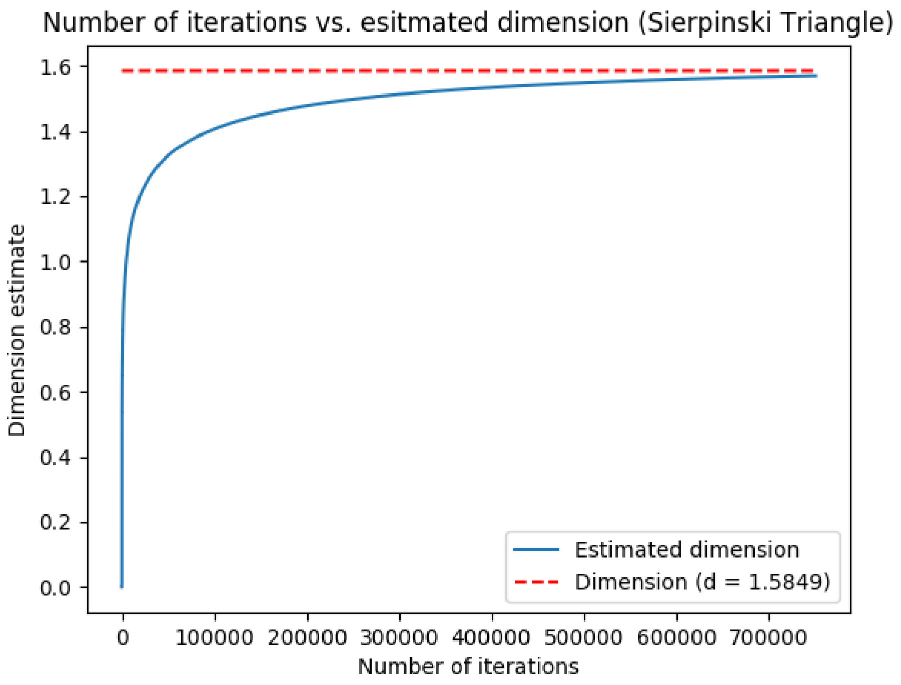
The calculated Hausdorff Dimension of the Sierpinski Triangle is[2]:

$$\frac{\ln 3}{\ln 2} \simeq 1.5849625$$

Similar to the Koch Snowflake, we will be working with a 2000×1717 pixel image. Once again, we are treating the average value of 1 as being totally inside our outside our fractal. We can run the code for different N values to get the following:

N Value	Dimension Estimate
50000	1.3203470723177708
100000	1.4059682914008882
500000	1.548739140003689
750000	1.579931848762905

Here is the plot for the computation of dimension over the iterations:



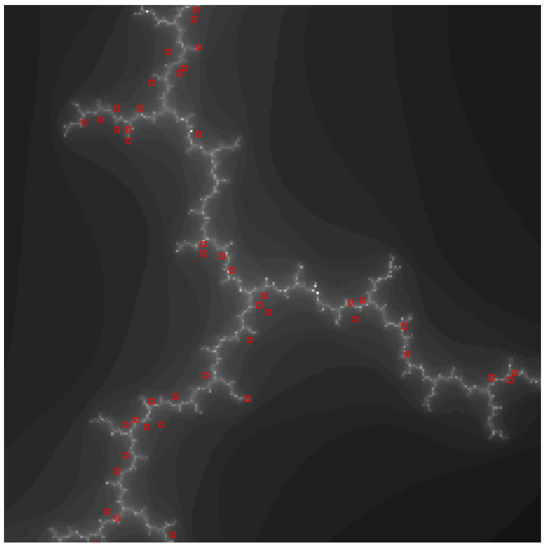
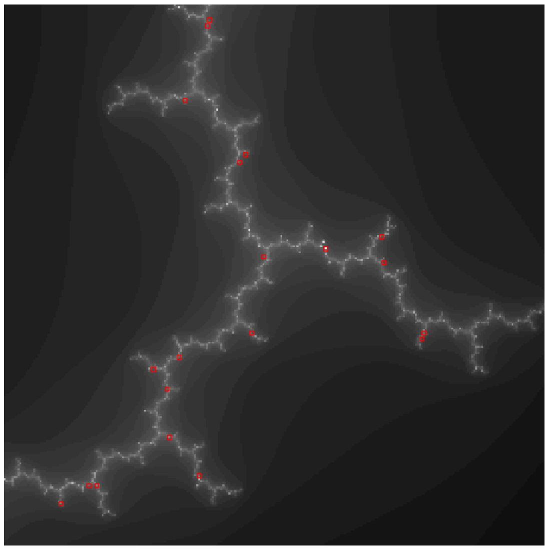
We can see a clear convergence to the correct dimension. Once again, this is a very accurate result for using less that 25% of the total points.

The Dendrite Julia Set

As mentioned in the introduction, we are calculating the dimension for the Julia set corresponding to the value $c = i$. This fractal was very easy to compute and plot using Python, so I have coded it in. To speed up the computations, we have pre-rendered and saved a .csv file of a 3000×3000 pixel dataset and are using the 300th iterate of the set (all points that have absolute value less than or equal to 2 after iterating 300 times). The numbers for determining when a box is in the set is a bit more involved. This is because the colouring for this set is gradient like (this is just due to the nature of the fractal). We need an average that will not overestimate or underestimate the values:

- For example, using the formula that if a box has average greater than or equal to 12, then it is in our set is too weak as it will take too many points right outside the set. This will result in a large overestimation when we use a large number of samples.
- Using the formula that if a box has average greater than or equal to 15.5, seems to be a good fit as the estimated dimensions approached the correct value for large samples and did not overestimate it by a significant amount.

Note the difference between where points are registering as being inside our set in the 2 averages:

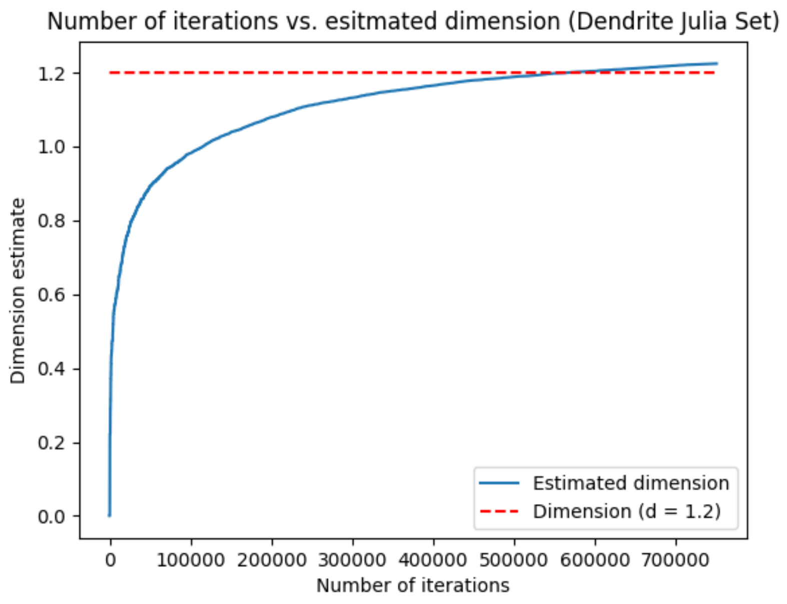



Note that in the plot for average = 12 case, there are many boxes that lie right outside the set, these are all errors that will add up over time. Raising the acceptance average to 16, reduces this significantly (however it is still not perfect).

Using the formula where we accept boxes with averages greater than 15.5, we get the following:

N **Value**	**Dimension Estimate**
50000	0.8845558944720685
100000	0.9562629342042503
500000	1.1616033695595138
750000	1.2142726096135412

We get the following plot for the convergence over time:



As we can see, running for 750000 iterations slightly overestimates our dimension [3]. This is due to the gradient averages as discussed above. If I had more time, I would implement a smarter method for calculating the averages (maybe involving making the interior colour of the Julia set much brighter than the gradient colours to increase the distinction between average points on the inside and outside of the Julia set). Overall though, this is an accurate estimation of the fractal dimension for this Julia set and we have done it using a small percentage (8.3%) of the total points.

Overview of Errors

The main source of error is the following:

> Since we are dealing with finite pixel pictures, we are limited in how small we can make our boxes.

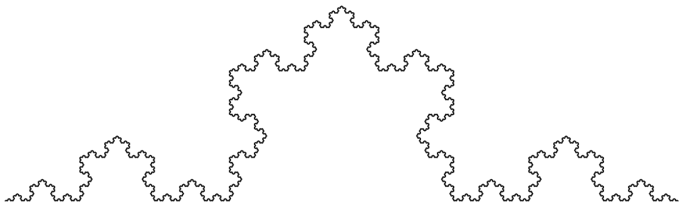
Recall that the actual box-counting dimension is a limit as the side length of the boxes go to 0. To attempt to reduce this error as much as possible, we use the smallest boxes we can make (using the midpoint method). Even still, this method will very slightly overestimate or underestimate the dimension of the fractal (due to the way the boxes we generate can lie over our set). In theory, we could have this error tend to 0 as we made our image plots arbitrarily large (and thus the side length of our boxes arbitrarily small). This would come at the cost of increased computation time however.

Conclusions

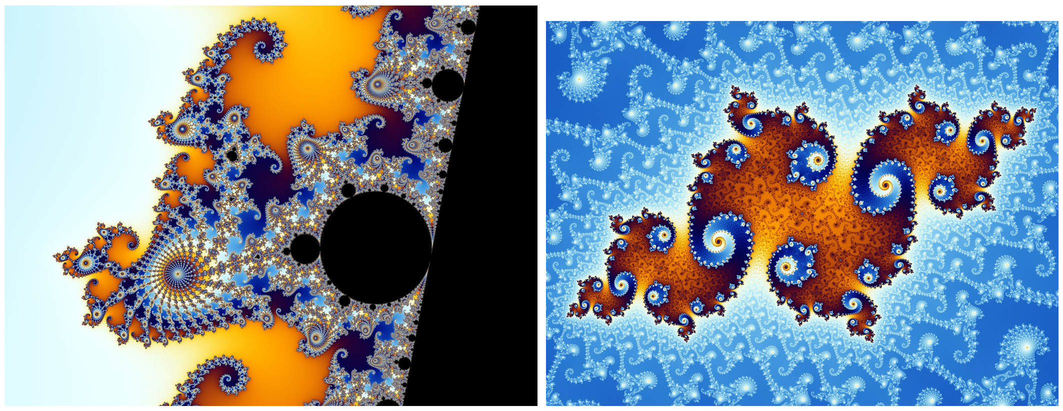
While this method was very successful on fractals that had quite a lot of self-similarity, the method was very poor at calculating the fractal dimension for fractals such as the Mandelbrot set and the Julia set (for other c values). After a lot of experimenting and time spent trying to get accurate results, I came to the following conclusion:

> My method does not work well with fractals that do not display high levels of self-similarity.

My reason for this is that in all the fractals that we have discussed in this document, they have all been highly self-similar. If we zoomed in on any of them, we would see a very similar pattern to the one we can view at the top level. What our algorithm is doing is taking a big picture estimate of the set, which only works if the fractal looks similar on a much smaller level. If we consider the original Mandelbrot set and try to apply the box counting algorithm to the boundary, it will always vastly undercalculate it (running my algorithm on it, the highest result I was able to calculate was a dimension of $d = 1.23524432324$, this is very inaccurate compared to the true dimension of $d = 2$)[4]. Compare zooming into the Koch Snowflake versus zooming into the Mandelbrot set:



Note that this pattern for the zoom in of the Koch Snowflake is indistinguishable from the full shape we have started with.



Comparing this with the Mandelbrot set, we have very different geometry depending on where we zoom in! Because of this, any covering of boxes, will always miss a large amount of the detail of the set. The second set is not even visible if we look at the full picture of the Mandelbrot set.

From these results shown, we can conclude that a Monte-Carlo style algorithm for determining the box counting dimension is efficient when applied to iterated function systems:

- It can calculate accurate results using a small percentages of total points.
- It is a relatively quick method to obtain good estimates on the fractal dimension for iterated function systems.

References

[1] Casselman, B:Lindenmayer Fractals - Fractal Dimension- Koch Snowflake
`http://www.math.ubc.ca/ cass/courses/m308-03b/projects-03b/skinner/ex-dimension-koch`$_{s}nowflake.htm$

[2] NA: Fractal Foundation,
`http://fractalfoundation.org/OFC/OFC-10-3.html`

[3] Branner, B *Chaos and Fractals: The Mathematics behind the Computer Graphics.* Providence, RI: Amer. Math. Soc., pp. 1989

[4] Shishikura, M *The Hausdorff Dimension of the Boundary of the Mandelbrot set and Julia Sets.* https://arxiv.org/abs/math/9201282v1

[5] Huo: Fractal Generator,
`http://codinglab.huostravelblog.com/math/fractal-generator/index.php?fractal=step=10size=300color=00`