Computational Representations of Fractals

Michael Weingert
www.michaelweingert.com
Final Project for PMATH 370: Chaos and fractals
University of Waterloo, ID: 20351333
April 24, 2014

Table of Contents

List of Figures

1.0 Introduction

Fractals are a mathematical concept that describes a curve or geometric figure; each part of which has the same characteristic of the whole. This phenomena is called self-similarity; a fractal is exactly or approximately similar to part of itself (ie. the whole has the same shape as one or more of the parts). Fractals exhibit infinite self-similarity. [1]

As a result of the infinite self-similarity of a fractal (and infinitely repeating patterns), fractals are impossible to render the entirety of a fractal as the features become too small to distinguish by a computer or human. As a result, there exist a number of approximation algorithms that aim to create good representations of a fractal. In this paper I will outline some approaches to rendering different fractals and fractal sets (Binary fractal trees, Julia sets, the Mendelbrot Set, and Iterated Function Set Fractals). I will also reference a sample project created in HTML5/Javascript to implement the various algorithms.

All of the representations can be seen at

www.michaelweingert.com/FractalHome.html

2.0 Binary Fractal Trees

A Binary Fractal tree is defined recursively by symmetric binary branching. A trunk of length one splits into two branches of variable length (call them r_1, and r_2), each making angles (θ_1 and θ_2 respectively) with the original trunk. Both of these branches then recursively split into branches of length $r_1 * r_1$, $r_1 * r_2$, $r_2 * r_1$, and $r_2 * r_2$. This pattern of branching and subdivision continues infinitely. [2]

An overview of a binary fractal tree can be seen in Figure 1 below.

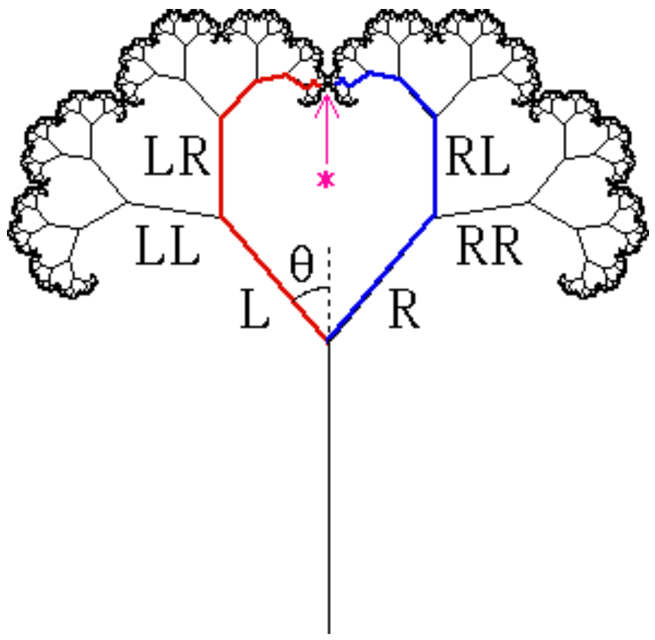

Figure 1: Overview of a Binary Fractal Tree[2]

Binary fractal trees can be efficiently evaluated without much difficulty. By treating a binary fractal tree as a series of line segments, the number of line segments is

$$\sum_{i=0}^{N} 2^{i} = 2^{N+1} - 1$$

where N is the number of iterations (each branch at the 'N-1th' iteration generates 2^N branches).

Within a 500x500 image, 11 iterations generates sufficient information about the shape and size of the fractal tree. If we assume a high radius of 0.75, with an original tree length of ~125 pixels (a quarter of the original height), we see that we generate all leaves up to a length of 125 * 0.75^11 ~ 5 pixels. On a computer screen, lines less than 5 pixels are not really visible to the human eye.

As a result, we can brute force the tree and just plot all branches directly (no approximation methods are required). 11 iterations can be performed in real time, so the tree parameters can be varied, and the tree will update itself in real time.

3.0 Julia Sets

There are several methods to produce the Julia set.

3.1 Backwards (inverse) iteration (IIM)

The first method to produce the Julia set (as outlined in the Encounters with Chaos and Fractals: Second Edition, Denny Gulick) is called Backwards (inverse) iteration (IIM).

The IIM theory states that, for any point z_0 that is not a periodic point of g_c, the backwards iterates will converge to the Julia set. At a high level, this works as, because z_0 is not in the Julia set, the backward iterate z_1 (where $z_0 = (z_1^2) + c$) must also not be in the Julia set. However, $z_1 < z_0 < c$. As the Julia set is the set of repelling fixed points, if we take reverse iterates, the set will acts as an attractor. And thus we note that $g'(z_0) < z_0$ where g' is the notation for the backwards iterate.

The algorithm is as follows:

1. Pick a point z_0 that is not a periodic point of g_c.
2. Randomly choose one of the two pre-images of z_0 of g_c and title this point z_1: $\sqrt{z_0 - c} \; or - \sqrt{z_0 - c}$
3. Continue the process with z_1, z_2, ... [3]

However, there are several problems that arise when using this method to draw the Julia set.

First of all, certain parts of the Julia set are quite difficult to access with the reverse Julia algorithm.

For instance, the following plot was generated using random iteration with 1000000 iterations, and $c = -0.75 + 0.11i$:

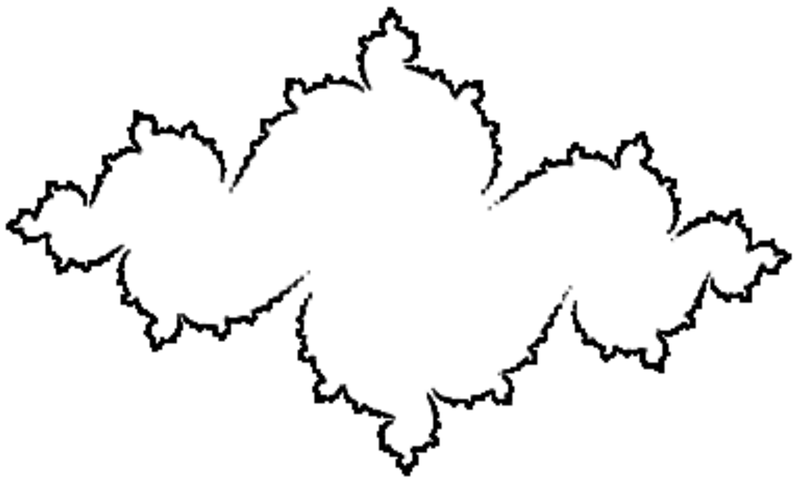

Figure 2: Julia Set for c=-0.75 + 0.11i rendered with IIM

However, if we examine the plot drawn by other means, we can see much more detail internally about the Julia set.

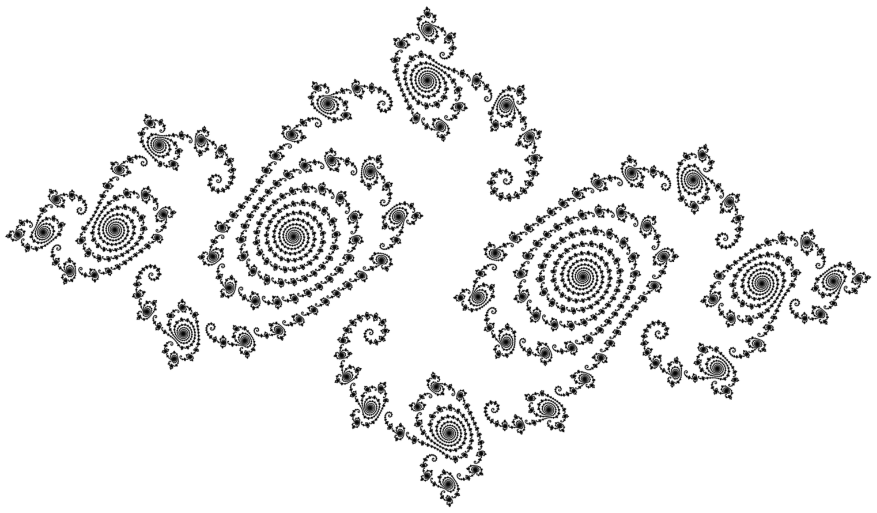

Figure 3: Julia Set for c = -0.75 + 0.11i rendered with Distance Estimation Metric[4]

We can see that, even though we were able to gain information about the external shape of the Julia set, much of the internal detail was not visible.

3.2 Escape Time Approximation (ETA)

Escape Time Approximation is another method to produce Julia sets that utilizes the definition of the Julia set.

The Julia set is defined as the set of repelling fixed points for a given g_c function. Thus, we can utilize the definition of a fixed point and test whether every pixel in an image is a fixed point or not.

The algorithm is as follows: for a given g_c

1. Choose a value (pixel) on the imaginary plane (a+bi).
2. Iterate the given value many times.
3. If the iterates are not bounded (ie. if any iterate has magnitude greater than $|c| + 1$) then the point is not in the Julia set.
    a. If the iterate is not bounded, record the number of iterations it took to become unbounded. This is one metric of 'distance' from the Julia set.
    b. If the iterate is bounded, the point is computationally in the Julia set.
4. Repeat steps 1-3 with all points (pixels) in the image.

Once we have values for each pixel (indicating the number of iterations it took for that point to become unbounded, or infinite if it was a fixed point), we can map the values onto a colour map (I have utilized a linear map for my renderings, but a logarithmic or other map could also be utilized), and render the resulting colour at each pixel.

In this manner we figure out the approximate distance from each pixel to the Julia set (using the number of iterations before the point became unbounded) , and map the distance onto a colour map, and render it accordingly.

Utilizing this method, the following plot is created (note that rendering took approximately 1-2 seconds through HTML5/Javascript)

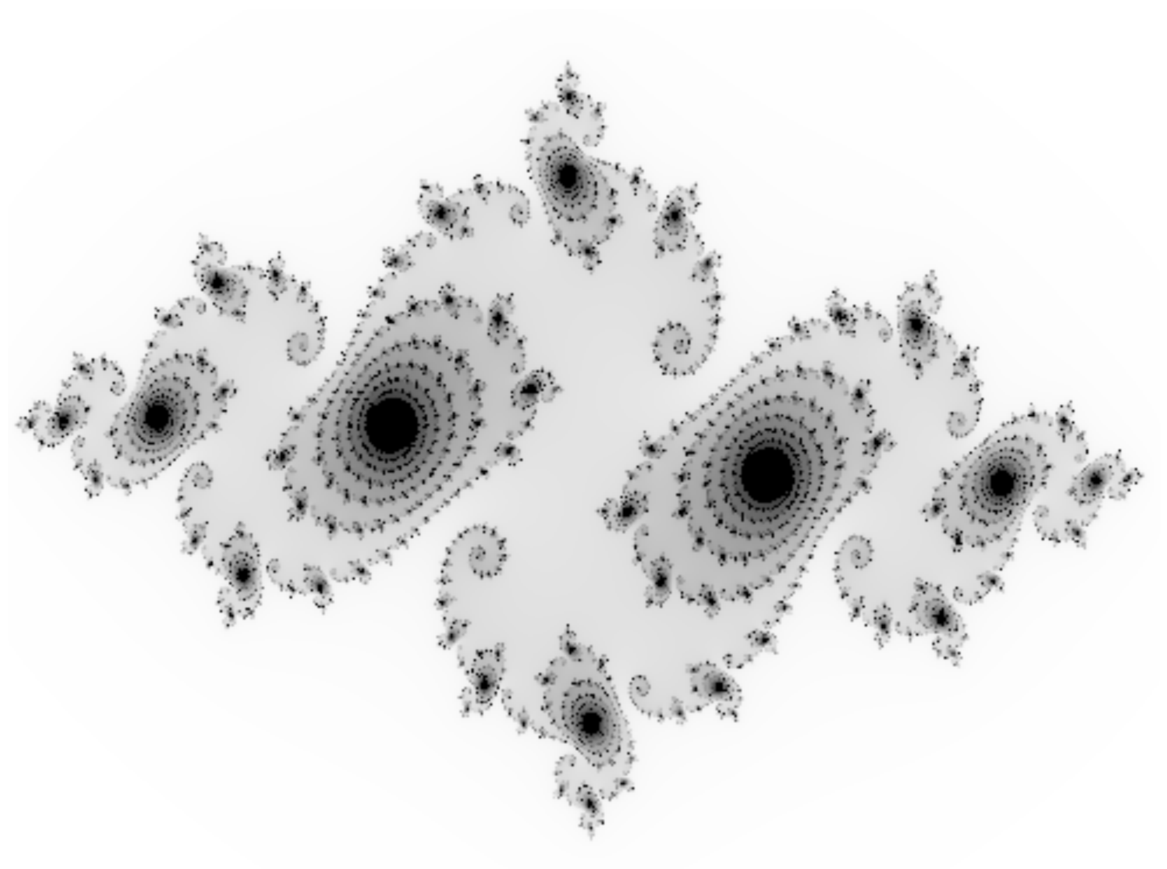

Figure 4: Julia Set for c = -0.75 + 0.11i rendered with Escape Time Approximation

3.3 Different Distance Metrics

There are other alternate distance metrics including exterior distance metric which involves calculation of the derivative at each pixel, to figure out how quickly the pixel is converging or diverging from the Julia set. This method was not pursued as determining a suitable colour map was proving to be difficult (a linear map produced unsuitable results, so a logarithmic / Gaussian map would likely have to be utilized). [5]

4.0 Rendering the Mendelbrot Set

The Mendelbrot set was rendered utilizing the same techniques as the Julia set. Random iteration was not a choice for this method as the Mendelbrot set as the set is effectively a combination of Julia sets that differ at every pixel. Encounters with

Chaos and Fractals (Gulick, 2012) lists essentially a 'binary' classification for determining the mendelbrot set.

Algorithm presented in Chaos and Fractals (Gulick, 2012) :

1. Choose every pixel (or as small an increment as possible)
2. Set c equal to the value at the pixel
3. Iterate 0 for 80 iterates
4. If the first 80 iterates are still bounded, then that pixel is deemed to be in the Mendelbrot set. [3]

The problem with this algorithm is that, should an iterate of 0 become unbounded on the 79th iteration, it will be not treated as part of the Mendelbrot set. Although this is true, we are not including important information that shows us that that point is very close to being in the Mendelbrot set, and is very close to points that are in the Mendelbrot set. Utilizing the ETA algorithm, we can map that point onto a colour map and show that is very close to being in the Mendelbrot set.

The Mendelbrot set using the Binary algorithm can be seen in Figure 5 below.

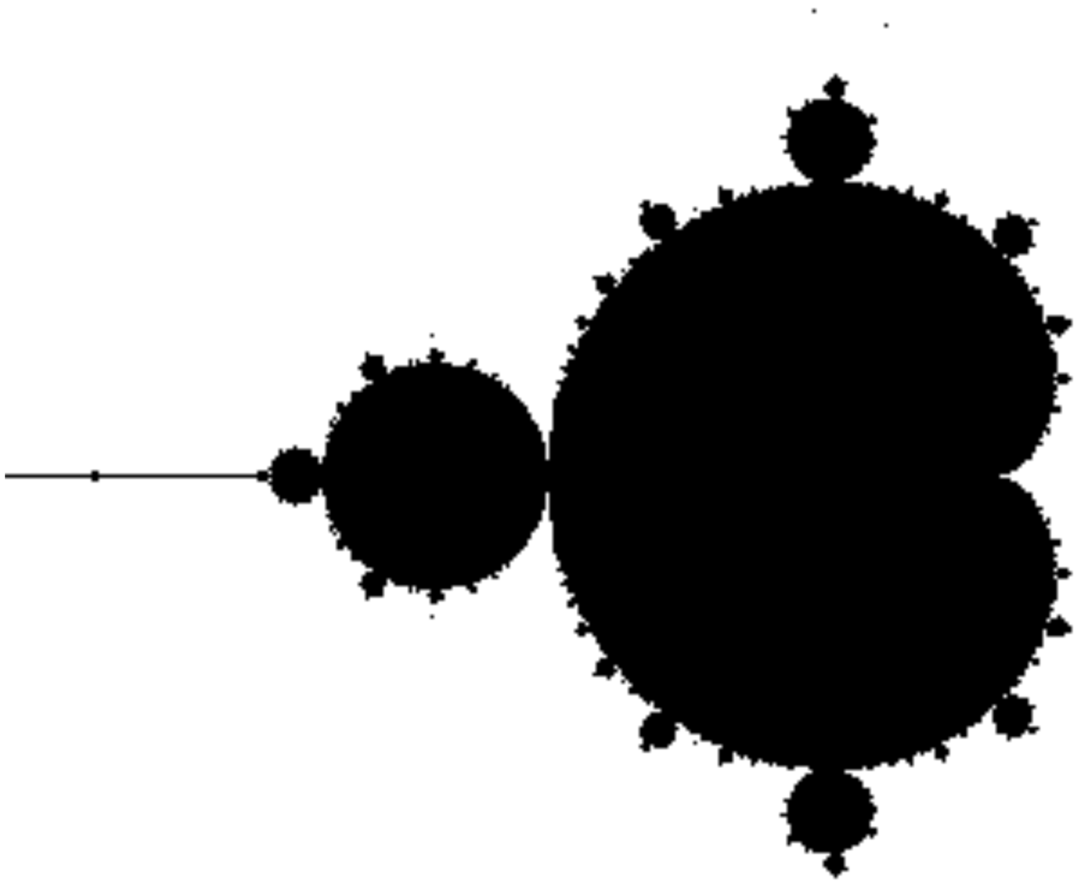

Figure 5: Mendelbrot Set rendered with Binary Classification Algorithm

Rendering the Mendelbrot set using the ETA algorithm produces Figure 6 below:

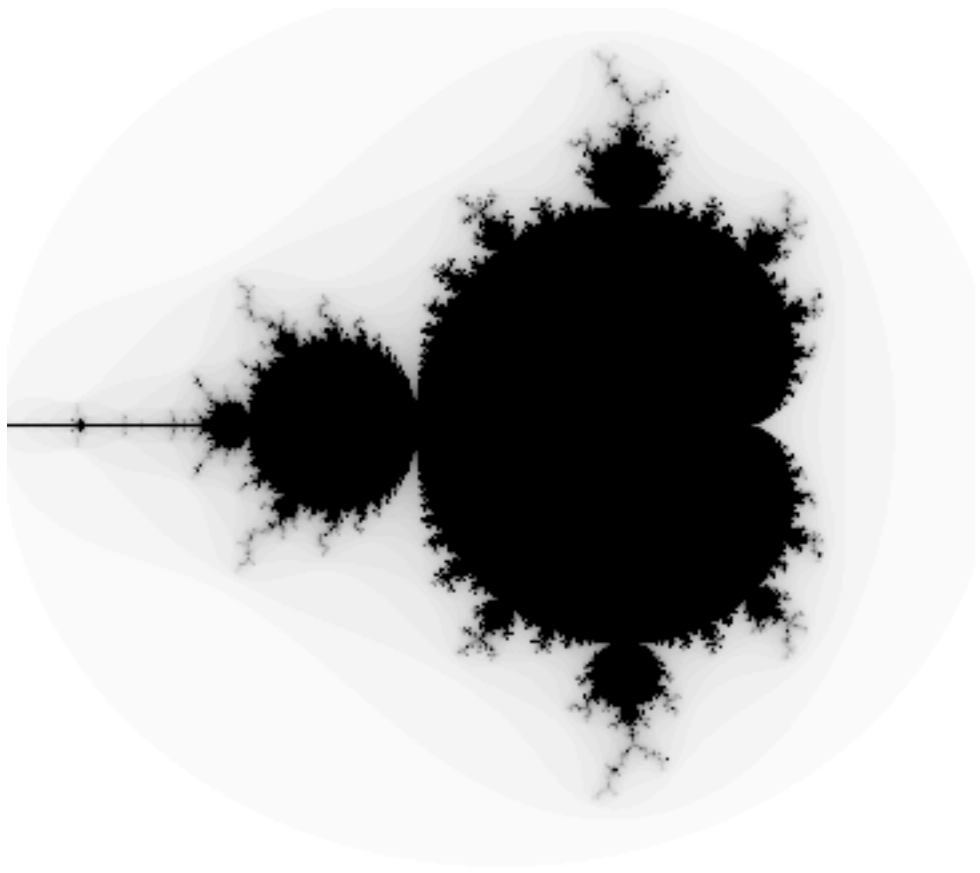

Figure 6: Mendelbrot Set rendered with Escape Time Algorithm

We can see that there are several 'antennae' that although they are not in the Mendelbrot set, they are very close to the Mendelbrot set. We can zoom in on those areas of interest and examine points that are in the Mendelbrot set.

Using a binary classification, we have missed out on important information about the shape and size of the Mendelbrot set.

5.0 Runtime of the Julia Set / Mendelbrot Set Algorithms

For each of the Julia Set / Mendelbrot Set algorithms, it is important to consider the runtime, as well as the accuracy when creating an interactive / real time rendering application. The random iteration for the Julia set has the best runtime (as we can choose a small number of iterations and still get a reasonably good shape of the set). However, as noted before, a lot of information is lost, or not easily accessible within a small number of iterations.

The approximate flops is on the order of # of iterations (in this case we used 1,000,000 iterations). We note that image size has no effect on the runtime.

Thus the runtime is on the order of 1,000,000 flops.

With the ETA or other distance metrics, the **average** flops is:
image width x image height x number of iterations / 2 (treat average as half of the points in an image in the Julia/Mendelbrot set, and half of them not in the set).

For a 500x500 image, with 1000 iterations at each point, this is

~500 x 500 x 500 flops = 125,000,000 flops

We can see that this is order of magnitudes more operations than the random iteration algorithm (and is thus slower). However, the quality of image is much superior.

We note that, in a browser, the approximate render time for the ETA algorithm on the order of 1-5 seconds (depending on the zoom level and how much iterations is required for an image). However, on a computer, this algorithm could be parallelized across a GPU/CPU.

Bearing in mind the trade off between algorithmic complexity and accuracy of the resultant image, the ETA algorithm was used for both the Julia set and Mendelbrot set in the HTML5/Javascript implementation.

6.0 Iterated Function Systems (IFS)

Rendering Iterated Function Systems through brute force is not practically feasible. A brute force approach would require taking an initial shape and rotating/shrinking/transforming it through tens or even hundreds of iterations. The amount of computation that would be required to transform the shape (matrix multiplication) is very costly, and as a result this method would take a long time to render.

However, the approach suggested in Encounters with Chaos and Fractals (Gulick, 2012) is to take random points and iterate the points individually until the resultant shape approaches the attractor for the IFS. The algorithm is as follows:

1. Choose a starting point in your image ([0,0] works well).

2. Iterate that point by applying one of the functions in the IFS system (each function will have a probability associated with it).
3. Iterate the resultant point through ‘n’ iterations. [3]

For high number of iterations (I utilize 200,000 iterations to render the barnsley fern in near real-time with HTML5/Javascript in browser), iterating a point with random functions approaches the attractor. This creates a good representation of the attractor, and is also able to be rendered in near real time.

7.0 References

[1] Wikipedia.org. Self-Similarity. (2014). Retrieved from: http://en.wikipedia.org/wiki/Self-similarity

[2] Frame, Mendelbrot, and Neger. Dimensions of Fractal Trees. (No date). Retrieved from: http://classes.yale.edu/fractals/FracTrees/SelfContact/SelfContact.html

[3] Gulick, Danny. Encounters with Chaos and Fractals: Second Edition. (2012).

[4] Wikipedia.org. File:Julia dem.png. (2011). Retrieved from: http://en.wikipedia.org/wiki/File:Julia_dem.png

[5] Wikibooks.org. Fractals/Iterations in the complex plane/Julia set. (2014). Retrieved from: http://en.wikibooks.org/wiki/Fractals/Iterations_in_the_complex_plane/Julia_set#How_to_use_distance