Recreating the Mandelbrot Set in JAVA

Yash Gharat

Doherty

Introduction

Getting inspired for this investigation was one of the most laborious tasks I had ever come across. The easy way out seemed to be a simple regression model on a set of data and finding correlation and using T-tests. However, statistics has never been my forte and neither has it particularly piqued my interest ever. So I really wanted to delve into some form of calculus but I knew my limits in being able to derive long equations from theory. But I realized, after much research on areas of math, that sequences and series are a part, albeit a small one, of calculus. It was my mom who gave me the true inspiration. As any artist, she often doodles and she recently showed me her drawing of the Serpinski triangles and it was then that fractals came to mind.

Fractals are a go-to when delving into the depths of iterative and recursive functions. They not only demonstrate sound logic when programming, but they also show the perfect intersection between math and computers. The Mandelbrot set is such an example. It first fascinated me when I came across it while practicing for an upcoming programming competition. The problem was simply to recreate a picture of the Mandelbrot set using an iterative function. At first glance, I believed it would be an easy task because I had recreated the Serpinski Triangles already before. However, it took me more than an hour, so I moved to the next problem.

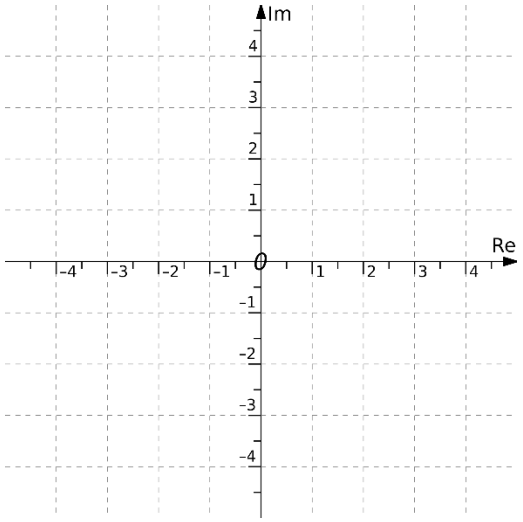
From my prior programming experience, I knew that graphics would be harder in other languages and I knew Java the best. This way, time would not be spent trying to learn a new language but instead be focused on understanding the mathematical problem at hand and finding an efficient solution for it that can be understood by many. The other benefit of using Java to construct this program is how user-friendly it is. If used correctly, Java looks a lot like regular English and thus can be conveyed to people who do not understand coding. My main goal in this investigation is to show how mathematics can be applied to other fields and be used in the real world.

Recreating the Mandelbrot Set in JAVA

Starting the Solution

The first thing when starting to create the program was to understand what the Mandelbrot set really was. It is defined as "a fractal that when plotted on a computer screen roughly resembles a series of heart-shaped disks to which smaller disks are attached and that consists of a connected set of all points c in the complex plane for which the recursive expression $z_{n+1} = z_n^2 + c$ for n = 0, 1, 2, 3, … with the starting value $z_o = 0$ remains bounded as n approaches infinity."[1] To anyone who has never encountered fractals, there are three major parts to this definition that will help understand it.

The first part to understand is how it is "a connected set of all points c in the complex plane". The first question I asked myself when reading this definition was "What is a complex plane?" The complex plane is analogous to a normal Cartesian plane except the horizontal axis represents all real numbers and the vertical axis represents imaginary numbers.



so the complex number -2 + 3i would be represented on the plane as two units to the left and 3 units up.

The next part to understand about the definition of the Mandelbrot set is "the recursive expression $z_{n+1} = z_n^2 + c$ for n = 0, 1, 2, 3, … with the starting value $z_0 = 0$". So, the equation, $z_{n+1} = z_n^2 + c$ starts with some given value of c, in this case c is a number on the complex plane.

[1] "Mandelbrot Set." Merriam-Webster, Merriam-Webster, www.merriam-webster.com/dictionary/Mandelbrot%20set.

Recreating the Mandelbrot Set in JAVA

Recursive functions are those where the solution of one input is used as the input for the next solution. For example, in this instance,

$$z_0 = 0$$

$$z_1 = 0^2 + c \rightarrow z_1 = c$$

$$z_2 = c^2 + c$$

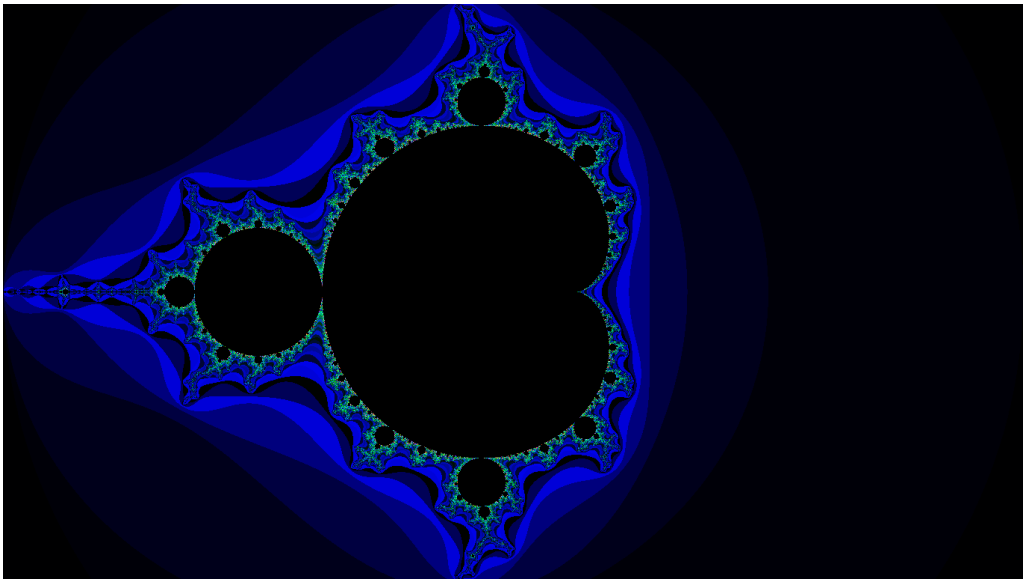
$$z_3 = (c^2 + c)^2 + c$$

the value c for the Mandelbrot set is every point on the previously mentioned complex plane. The value of c also iterates to test every coordinate.

The third and final part to understand about this definition is "consists of a connected set of all points c… bounded as n approaches infinity." This is where computers come in. Each point on the complex plane is tested with the recursive function, $z_{n+1} = z_n^2 + c$, with c being the current coordinate. The computer tests if, when using that starting value, the function is bounded. It does this by running through a lot of iterations and checking the value. If it does not converge by the end of the iterations, it is considered divergent. The more the iterations, the better. Even though true infinity can never be achieved, one can get close enough to conclude divergence or convergence in this case. The Mandelbrot set only includes the values that converge as n approaches infinity and the rest are discarded.

The Solution

There are usually two ways that the Mandelbrot set is represented, bi-color with only black and white, or with multiple colors. For the purpose of truly showing the set, I decided to represent it with colors so that the rings of iterations can be shown. The result was this:



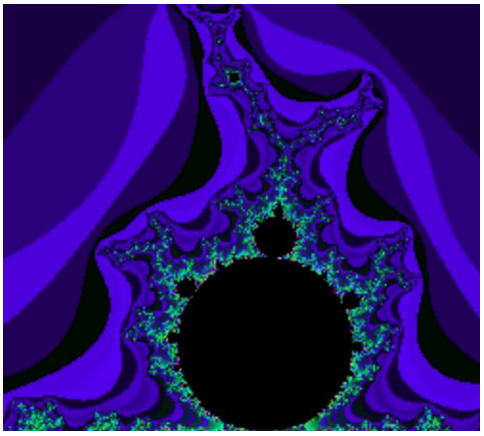
In this photo, the shape of the set is clear. It is often called a cardioid because of the appearance of the sideways heart attached to a circle. This set represents about 100,000 iterations for each coordinate. Theoretically, the shape's area should keep reducing as complex numbers get cut off at every iteration. The color scheme is based on what iteration the coordinate got cut off at. If the coordinate is not a part of the set, it is colored black. If it is part of the set, the pixel is colored in using an array of colors depending on how many iterations the coordinate takes to converge. Thus, clear rings can be shown.

The set starts with a circle of radius 2 because everything past that diverges. With the next iteration, it is moved to another shape inside and many of the numbers in the circle are found to converge. So the actual cardioid in the middle and black regions surrounding that are not part of the set. The colors at the edge are what are contained in the set. The construction of the logic is as follows. If the current coordinate iterates for more than 100,000 times, then it is considered a black pixel, otherwise it is colored depending on the iteration it converges at.

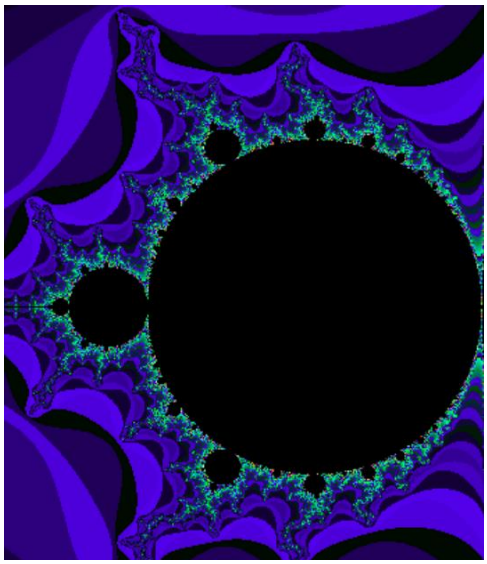
Analysis of the Solution

Overall, the program is simple. It constructs a complex plane in an image and walks through it, treating each pixel as a coordinate. If the iterative output of the value assigned to the pixel does not converge, it becomes black. The maximum number of iterations can be adjusted. Although I have set it at 100,000 iterations, it is somewhat unnecessary to have that many because the differences between 1,000 and 100,000 iterations are hardly noticeable. However, the closer to infinity it is, the more accurate the solution.

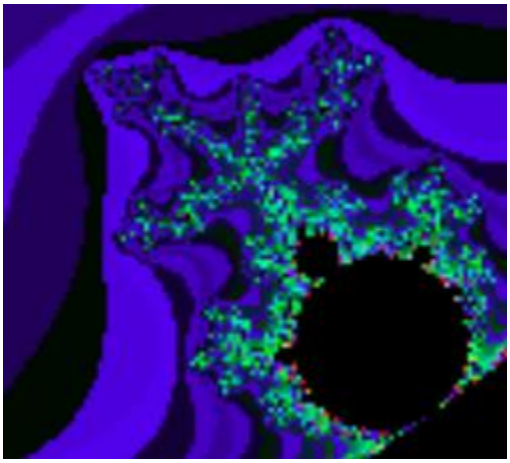
The most intriguing part of the Mandelbrot set is when one zooms into the edges of the cardioid formed.



This is taken from the top of the cardioid and it is a shape that is nearly identical to the tail that can be found at the edge of the center cardioid, as seen below.



However, the first image is a smaller and rotated version of it with smaller “entrails” hanging off of it. This repetitive geometry can be found throughout the image. When zoomed in further, more patterns are revealed.



For example, this is a smaller tail found on the cardioid.

Works Cited

- "Mandelbrot Set." *From Wolfram MathWorld*, mathworld.wolfram.com/MandelbrotSet.html.
- "Mandelbrot Set." *Merriam-Webster*, Merriam-Webster, www.merriam-webster.com/dictionary/Mandelbrot%20set.

Appendix

```
import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;

public class Mandelbrot_set_mine {
	private static int max = 1000;
	public static void main(String args[]) throws IOException {
		int width = 1920;
		int height = 1080;
		BufferedImage img = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);
	int black = 0;
	int[] colors = new int[max];
	for (int i = 0; i<max; i++) {
	   //int base = i+4915330;
	   colors[i] = i*i*i;//base*1022;
	}
		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				double re = (j - width / 2) * 4.0 / width;
				double im = (i - height / 2) * 4.0 / width;
				double x = 0, y = 0;
				int mand = chk(x,y,re,im);
				if (mand < max) img.setRGB(j, i, colors[mand]);
	   else img.setRGB(j, i, black);
			}
			System.out.println(Integer.toString(i));
		}
		ImageIO.write(img, "png", new File("Out.png"));
	}
	public static int chk(double x, double y, double re, double im) {
		for (int i = 0; i<max; i++) {
			if(x*x+y*y < 4) {
				double x_new = x*x-y*y+re;
	   y = 2*x*y+im;
	   x = x_new;
			}else {
				return i;
			}
		}
		return max;
} }
```