5-11-2014

Generalized Mandelbrot Sets

Aaron Schlenker
Butler University

Follow this and additional works at: https://digitalcommons.butler.edu/ugtheses
Part of the Mathematics Commons, and the Theory and Algorithms Commons

Recommended Citation
https://digitalcommons.butler.edu/ugtheses/229

This Thesis is brought to you for free and open access by the Undergraduate Scholarship at Digital Commons @ Butler University. It has been accepted for inclusion in Undergraduate Honors Thesis Collection by an authorized administrator of Digital Commons @ Butler University. For more information, please contact omacisaa@butler.edu.
BUTLER UNIVERSITY HONORS PROGRAM

Honors Thesis Certification

Please type all information in this section:

Applicant: Aaron Robert Schlenker  
(Name as it is to appear on diploma)

Thesis title: Generalized Mandelbrot Sets

Intended date of commencement: 5/11/2014

Read, approved, and signed by:

Thesis adviser(s): William Johnston  
Date: 4/28/14

Reader(s): Christopher Wilson  
Date: 5/13/13

Certified by: Judith Harper Marvel  
Director, Honors Program  
Date: 5/13/14

For Honors Program use:

Level of Honors conferred: University  
Cum Laude  
Highest Honors in Mathematics

Departmental  
Highest Honors in Computer Science
Generalized Mandelbrot Sets

Aaron Schlenker *

I Introduction

A complex point \( z_0 \) is defined to be a member of the famous Mandelbrot set fractal when the iterative process using the function \( z^2 \) stays bounded when applied to \( z_0 \).

We investigate what happens if we change the iterative process so that \( z^2 \) is now composed with, for example, a Mobius transformation, indexed on a parameter \( a \). The Mandelbrot set corresponds to \( a = 0 \). What happens when we change \( a = 0 \) to other values, repeating the iterative process and then drawing the sets? Do these Generalized Mandelbrot sets have similar properties to the original Mandelbrot set?

This thesis describes some surprising results for these new sets, and it also uses transcendental functions to produce similar generalized sets. Further, it describes the algorithms that were developed and used during the study of each of these sets.

The Mandelbrot set is a famous mathematical set with many fascinating properties to both trained mathematicians and the general public. It was studied and popularized by Benoit Mandelbrot, a French and American mathematician after whom the set is named, while he worked at IBM. [6] He had been interested in processes (for example, suddenly changing physical processes, or wildly fluctuating economic pricing, or difficult-to-predict human behavior) that appeared naturally in the world.

*Mathematics and Computer Science Major, Butler University, Indianapolis, IN 46208. aschlenk@butler.edu

1We give a formal mathematical description of the iterative process in Section II.
around us but that behaved in rather unpredictable ways. The computer machinery at IBM provided him with a new investigative tool: mathematical formulas often didn’t exist for such processes, but they could be modeled using a computer with huge processing power to predict how they might behave. Mandelbrot realized some sets, just as for these processes, had this same complex structure: they were easy to define and study on a computer when they resulted from an algorithmic process, but difficult to model after, say, standard boundary lines such as arcs or sawtooth edges. One such set is the Mandelbrot set, which he first examined in 1979. He coined the descriptive term “fractal” for it and other such sets, describing their edges as “rough,” no matter how closely examined. He realized they looked jaggedly rough from a distance, but just as rough no matter how much they were magnified. This roughness is now described mathematically as a “fractal dimension”; it differs in a surprising way from the dimension of the boundary for a standard set. For example, a boundary of a standard set in the two-dimensional plane is one-dimensional. A simple case is the boundary of a circular disk is a circle, which is one-dimensional. But the Mandelbrot set’s boundary is not! It’s dimension, which mathematicians call a “Hausdorff dimension,” calculated with a complicated formula that will be explained later, turns out to be 2. This result is surprising to mathematicians—how can the boundary of a two-dimensional set be two-dimensional? Simply put, the Mandelbrot set’s boundary is so complicated, it preserves the dimensionality of the set it contains! Since Mandelbrot’s initial studies, there have been many more amazing discoveries about certain features and details of this famous set. [6]

The Mandelbrot set became of interest to mathematicians after its initial popularization by Benoit Mandelbrot. In fact, two other mathematicians are known to have studied the set in 1978 before Mandelbrot. Robert Brooks and J. Peter Matelski are the mathematicians who published a paper in 1978 discussing a set analogous to the Mandelbrot set. This paper contains the famous formula: $z^2 + c$, which is
the formula used when producing the Mandelbrot set.[5] They also produced a very crude but unmistakable picture that shows the very familiar Mandelbrot set boundary. Matelski and Brook’s paper was not published until 1981, whereas Benoit Mandelbrot was able to publish his paper, a more robust examination of the set, late in 1980. Although they may have studied the set before Mandelbrot, Brooks and Matelski both eventually came to the conclusion that the set deserved to be named after Mandelbrot because he was able to truly popularize the set. They recognized that their name for this famous set, as they lovingly called it ‘the thing with the big cardioid,’ comes up a little short. [3] Still, even though they did not truly realize the amazingly complex object they had in front of them, they still were partly responsible in pioneering its investigation. Since the initial studies into the Mandelbrot set, it has captured the attention and fascination of a mathematical community for decades, and even become immensely popular to the general public. The extra attention has resulted in fantastic discoveries of the properties of the Mandelbrot set.

John H. Hubbard and Adrien Douady made the first major mathematically theoretical contributions to the study of the Mandelbrot set. They were the first to prove many of the fundamental principles associated with the Mandelbrot set, including the local connectivity of the set. They are actually the ones responsible for the naming of this famous set, calling it the Mandelbrot set in honor of Mandelbrot. Douady wrote in 1986, “Mandelbrot was the first one to produce pictures of it, using a computer, and to start giving a description of it.”[3] The full statement acknowledges the prior work done by Brooks and Matelski, while also concluding the most significant contribution was made by Mandelbrot. It also describes why Hubbard and Douady named the set
We now describe how to calculate the Mandelbrot set. An understanding of the complex plane as a space of points is a prerequisite. The best way to introduce the complex plane is in terms of the Cartesian plane, also called the $x$-$y$ plane. A point in the Cartesian plane is represented by an $x$ value and a $y$ value. These two values are used to represent a point with the notation $(x, y)$. For example, if we wanted to represent the point located at the values of $x = 2$ and $y = 3$, we write $(2,3)$. This notation is similar for the complex plane. In the complex plane we have a real value instead of an $x$-value and an imaginary value instead of a $y$-value. The real value is just a real number, but the imaginary value will be a real number multiplied by $i$, where $i = \sqrt{-1}$. Then, the point $(2,3)$ in the complex plane represents the complex number $z$ where $z = 2 + 3i$. More generally, a complex number is $z = x + yi$, where $x$ and $y$ are real.

We can add and multiply complex numbers. If we start with complex points $z_1 = x_1 + iy_1 = (x_1, y_1)$ and $z_2 = x_2 + iy_2 = (x_2, y_2)$, then we add them by adding the real parts together and then the imaginary parts. This means $z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2) = (x_1 + x_2, y_1 + y_2)$. A quick example is to take $z_1 = 1 + i = (1,1)$ and $z_2 = 2 + 3i = (2,3)$, then $z_1 + z_2 = (1 + 2) + i(1 + 3) = 3 + 4i = (3,4)$.

Multiplication of complex numbers is a little more complicated than addition. The general formula is calculated by distributing; we get: $z_1 \cdot z_2 = (x_1 + iy_1)(x_2 + iy_2) = (x_1 \cdot x_2 - y_1 \cdot y_2) + i(x_1 \cdot y_2 + x_2 \cdot y_1) = (x_1 \cdot x_2 - y_1 \cdot y_2, x_1 \cdot y_2 + x_2 \cdot y_1)$. One detail to provide here is that $i^2 = \sqrt{-1}^2 = -1$. This is what makes $iy_1 \cdot iy_2 = -y_1 \cdot y_2$ in our formula. Now this formula is a little convoluted, but it can be understood in a better sense by looking at an example. Take $z_1 = 1 + i = (1,1)$ and $z_2 = 2 + 3i = (2,3)$, then $z_1 \cdot z_2 = (1 + i)(2 + 3i) = (1 \cdot 2 - 1 \cdot 3) + i(1 \cdot 3 + 2 \cdot 1) = -1 + 5i = (-1,5)$. Multiplying complex numbers in effect allows us to “multiply points”!

The process to calculate the Mandelbrot set is not difficult to understand once
the reader has a grasp of how to deal with complex numbers and feels comfortable iterating a value through a function. The Mandelbrot set is constructed using a simple complex function $z \mapsto z^2 + c$. The set is calculated by iterating each point in the complex plane through this function, checking at each iteration to see if the sequence is diverging off to infinity. The point is defined to be in the Mandelbrot set if the iterative process stays bounded. More formally, define the $n^{th}$ term in the sequence as $z_n = z_{n-1}^2 + z_0$. Here $z_{n-1}$ is the complex number obtained from the previous iteration, and $z_0$ is the complex number originally chosen from the complex plane. A quick example of this iteration is to look at the complex point 1, which starts with $z_0$ equal to 1. This point produces the sequence $1, 2 = 1^2 + 1, 5 = 2^2 + 1, 26 = 5^2 + 1, ...$, which diverges to infinity. However, if we start with $z_0 = -1$, then the sequence is $-1, 0 = (-1)^2 - 1, -1 = 0^2 - 1, 0 = (-1)^2 - 1, ...$, which stays bounded (more specifically, it bounces between 0 and -1). The first sequence, with $z_0 = 1$, shows 1 is not in the Mandelbrot set because the sequence diverges to infinity. The second sequence, with $z_0 = -1$, shows -1 is in the Mandelbrot set because the corresponding sequence does not diverge to infinity—it stays bounded throughout the iteration as it continues for an infinite number of steps. All other points in the complex plane are tested in the same manner. In this way, the famous Mandelbrot set is computed.

Originally, it was conjectured that the Mandelbrot set's boundary had a Hausdorff dimension greater than 1. To put this in perspective, a line has a Hausdorff dimension of 1. This conjecture about the Mandelbrot set's Hausdorff dimension was proven somewhat recently by Mitsuhiro Shishikura, with his paper published in 1998.[7] The Mandelbrot set is also a connected set which means that every point in the set is connected. Though it may not seem extraordinary, this feature is surprising. This feature is proved by showing there is a conformal isomorphism between the complement of the Mandelbrot set and the complement of the closed unit disk.[7] Another surprising fact is that you can actually use the Mandelbrot set as a way
to calculate \( \pi \), although it is extremely inefficient. Finally, the Mandelbrot set has self-similarity – zooming in on the boundary reveals further Mandelbrot sets into an infinite depth.[6] Even now, and in spite of the fact that the Mandelbrot set has been studied for more than 30 years, the area of the region the Mandelbrot set covers in the complex plane is still unknown, with only approximate answers known.[1]

These facts, along with many others, make the Mandelbrot set extremely interesting for mathematicians to study. However, it is important to note that it is not only these facts about the Mandelbrot set that make it so fascinating, but also the stunning visual beauty of the Mandelbrot set. Artists can control the coloring algorithms in the presentation of the Mandelbrot set to produce fantastically beautiful pieces of art. Among all these reasons, I hope it is clear why we were interested in looking at this set: we wanted to generalize the Mandelbrot set in ways that retained its visual beauty in our own new sets. The transference of these properties were important; without them, these sets would not be as interesting to the general public to study and visualize.
II The Generalized Mandelbrot Set

The function used to define the Mandelbrot set is simple: \( f(z) = z^2 \). We change this function to generate related sets. The functions we introduce in this section use the so-called “Blaschke factor”: \( \frac{z-a}{1-\bar{a}z} \). Motivation for the choice is based on a Blaschke factor’s properties. Products of Blaschke factors are the “atomic building blocks” of complex analytic functions on the unit disk. This atomic structure is based in the unit disk in the complex plane, which is analogous to the interior of the unit circle. For any bounded analytic function \( f \) that has zeros \( a_1, a_2, \ldots a_n \) in the unit disk, the Blaschke product \( \prod_{n=1}^{\infty} \frac{z-a_n}{1-\bar{a}_nz} \) is a multiplicative factor of \( f \). Understanding Blaschke factors is central to understanding fractals produced using Blaschke factors. We will define these “Generalized Mandelbrot sets” using a single Blaschke factor with parameter \( a \), where \( a \in \mathbb{C} \). We start by defining a set \( M_a \).^2

Definition The set \( M_a \) is the set of points \( z_0 \) producing values \( |z_n|, n = 1, 2, 3, \ldots \) that all stay bounded (i.e., no subsequence of \( \{|z_n|\}_{n=1}^{\infty} \) has infinite limit), where \( z_n \) is defined by the iterative process

\[
z_n = \left[ \frac{z_{n-1} - a}{1 - \bar{a}z_{n-1}} \right]^2 + z_0, \quad \text{where } n = 1, 2, 3, \ldots.
\]

Note \( M_0 \) is the so-called Mandelbrot set, since \( a = 0 \) produces \( z_n = z_{n-1}^2 + z_0 \).

We visualize these sets using a computer program. The program runs a point \( z_0 \in \mathbb{C} \) through the iterative process and determines which ones have iterative values \( |z_n| \) that are bounded. Of course, the upper bound can be different for each point. The set of all such upper bounds might not be bounded in its own right. But when we use a

^2M_a actually turns out to not always possess Mandelbrot set like properties; our discussion of \( M_a \) helps motivate a more general definition of the Generalized Mandelbrot sets.
computer program, we need to fix a single number—an upper bound “test value”—to which we compare the iterative value $|z_n|$. The computer program’s chosen test value is arbitrary. For the Mandelbrot set, it turns out a chosen test value 4 completely determines the set. In other words, if $|z_n| > 4$ then the iterative value goes to infinity.[10] This allows the computer program to make an easy comparison—“Is $|z_n| > 4$?” Unfortunately, when $a \neq 0$, no single test value is decisive—the choice for the test value will affect the pictorial representation the computer produces. It turns out (as we will prove in this paper) that this condition on $|z_n| < \infty$ is fairly weak when $a \neq 0$, leading to the corresponding sets $M_a$ all being unbounded.

For example, the following computer-generated image (Figure 2), where a black point belongs to our set and white does not, is a representation of the set $M_{0.05+0i}$ with the test value set at 1,000,000 and the number of iterations at most 1,000. The number of iterations is important to determine points that will not be in the set. When the number of iterations is high, the picture has better detail, but its slow. Since the added detail is insignificant at a macro level, generally the number of iterations for our illustrations in this paper are capped at 1,000. The computer-generated sets change significantly as the test value moves towards infinity (as seen in the differences between Figures 2, 3, and 4), but it is important to note that a change in the maximum number of iterations does not appear to

Figure 2: The $M_{0.05+0i}$ set with test value 1,000,000.

Figure 3: The $M_{0.05+0i}$ set with test value 100,000.
have any effect. The reason for this property seems to be that the iterative process of the complex numbers using \( z_n = \left( \frac{z_{n-1} - a}{1 - \bar{a}z_{n-1}} \right)^2 + z_0 \) will not diverge to infinity for any point \( z_0 \) unless \( a = 0 \). This is what causes the \( M_a \) sets to be unbounded (and thus uninteresting if they are represented truly) for extremely large test values in the program.

To remedy this problem, we define for each \( a \) a related set \( S_a \), which we call the Generalized Mandelbrot set. The \( S_a \) sets have defined test values set to ensure that the pictures have interesting aspects and are often fractal in nature. These new sets allow us to study the \( M_a \) sets in a desired way—one in which their representations are all bounded like the Mandelbrot set and share qualities with the Mandelbrot that has made it so popular. Part of the Mandelbrot set’s intrinsic value comes from it’s artistic value and features. We wanted to transfer this appeal to the Generalized Mandelbrot sets to create beautiful, unique pictures known as much for the beauty as for their own mathematical qualities that we describe later. The sets \( S_a \) have these qualities desired.

**Definition** For any \( a \in \mathbb{C} \) with \( a \neq 0 \), the **Generalized Mandelbrot set** \( S_a \) is the set of points \( z_0 \) producing values \( |z_n|, n = 1, 2, 3, \ldots \), that all have \( |z_n| \leq 2/|a|^2 \) when \( |a| \leq 1 \) and \( |z_n| \leq 2|a|^2 \) when \( |a| > 1 \), where \( z_n \) is defined by the iteration process

\[
z_n = \left( \frac{z_{n-1} - a}{1 - \bar{a}z_{n-1}} \right)^2 + z_0, \text{ when } n = 1, 2, 3, \ldots
\]

We also define \( S_0 = M_0 \).

The definition of \( S_a \) is actually a specific case of a more general definition of interesting sets, all arising from a similar iteration process, and which we label
$S(f(z), K(f), N)$. To explain the notation, $f(z)$ is the function involved in the iterative process, $K(f)$ is the test value used to determine when an element is in the set, and $N$ is the number of iterations examined against that test value. Each of these parameters determine the picture produced by the computer program. Specifically, each different $f(z)$ produces different sets, the parameters $K(f)$, being the test bound, and $N$, being the number of iterations, changes the visual representation of the sets produced using a function $f(z)$. We define these sets here.

**Definition** For any given complex-valued function $f(z)$, we form the following corresponding set of points in the complex plane:

\[
S(f(z), K(f), N) = \mathbb{C} \setminus \{z_0: z_n = f(z_{n-1}) + z_0 \text{ has } |z_n| > K(f) \text{ for some } z_n \text{ where } n = 1, 2, 3, \ldots, N\}.
\]

The bound $K(f)$ is independent of $z$ or $z_0$ but will often depend on parameters involved in the definition of $f$. For example, when $K(f) = \begin{cases} 2/|a|^2 & \text{if } |a| \leq 1 \\ 2|a|^2 & \text{if } |a| > 1 \end{cases}$, $N = \infty$, and $f(z) = [\frac{z - a}{1 - az}]^2$, we get $S(f(z), K(f), N) = S_\alpha$. 
III Results

Our first visual representations of the $S_a$ and $M_a$ sets allowed us to formulate many different conjectures about their special properties. Theorem 1 was proven during the beginning of the research. This theorem corresponds to a symmetry across the real axis that some of these sets have. The result becomes easily apparent when viewing one of these sets, while intuitively it is rather easy to see why we would expect these certain sets all to have a symmetry about the real axis in the complex plane. The reason that this is intuitive relates to what a conjugate number is in complex analysis. When $z$ is a real number, then $z = \bar{z}$ where $\bar{z}$ is the conjugate $x - iy$ of $z = x + iy$, and this forms both the intuition behind this property and suggests a correct strategy for proving it.

**Theorem 1.** When $a \in \mathbb{R}$, then both $S_a$ and $M_a$ are symmetric about the real axis.

Figure 5 displays the property stated in the Theorem 1. As you can see, any point in $S_{22+0i}$ above the real axis (where the imaginary value is positive) will reflect to a point below the real axis that is also in $S_{22+0i}$. This neat result allows us technically to produce these sets knowing only half of the points that are contained in each. The proof follows from the fact that the modulus of the complex number $z = x + yi$ is equal to the modulus of $\bar{z} = x - yi$, also known as the conjugate of $z$. If $a \notin \mathbb{R}$, then we will not have this symmetry. The proof becomes apparent when you notice the Blaschke factor we use in our iterations are the same for $z_0$'s reflection across the real axis. These two numbers are only equal when $a \in \mathbb{R}$ because of the way that conjugates work. The conjugate switches the sign on the imaginary part of a complex number, and, without an imaginary part, these two complex number's moduli will be
Proof. Take \( z_0 = x_0 + iy_0 \), so \( \bar{z}_0 = x_0 - iy_0 \). Then we compare \( z_{n+1} \) for \( z_0 \) with \( z_{n+1} \) for \( \bar{z}_0 \). The iteration for \( z_0 \) is \( z_{n+1} = \left[ \frac{z_n - a}{1 - \bar{a}z_n} \right]^2 + z_0 \). When we multiply out the product with \( a = x_0 \), with \( y_0 = 0 \), we get:

\[
\begin{align*}
z_{n+1} &= \left[ \frac{(1 + x_0^2)x_n - (1 + x_n^2 + y_n^2)x_a}{1 - 2x_ax_n + (x_a^2)(x_n^2 + y_n^2)} \right]^2 - \left[ \frac{(1 - x_0^2)y_n}{1 - 2x_ax_n + (x_a^2)(x_n^2 + y_n^2)} \right]^2 + x_0 + \\
i\left[ \frac{2[(1 + x_0^2)x_n - (1 + x_n^2 + y_n^2)x_a][(1 - x_0^2)y_n]}{1 - 2x_ax_n + (x_a^2)(x_n^2 + y_n^2)} \right] + y_0.
\end{align*}
\]

Now let \( c = (1 + x_a^2)x_n - (1 + x_n^2 + y_n^2)x_a \), \( d = (1 - x_a^2)y_n \), and \( D = 1 - 2x_ax_n + (x_a^2)(x_n^2 + y_n^2) \). We look at the distance formula to show that modulus of (1) \( z_{n+1} = \left( \frac{z_n - a}{1 - \bar{a}z_n} \right)^2 + z_0 \) is the same as the modulus of (2) \( z_{n+1} = \left( \frac{z_n - a}{1 - \bar{a}z_n} \right)^2 + \bar{z}_0 \).

\[
(1) \ |z_{n+1}| = \left| \left( \frac{z_n - a}{1 - \bar{a}z_n} \right)^2 + z_0 \right| = \sqrt{\left[ \frac{c^2 - d^2}{D^2} + x_0 \right]^2 + \left[ \frac{2cd}{D^2} + y_0 \right]^2}
\]

\[
(2) \ |z_{n+1}| = \left| \left( \frac{z_n - a}{1 - \bar{a}z_n} \right)^2 + \bar{z}_0 \right| = \sqrt{\left[ \frac{c^2 - d^2}{D^2} + x_0 \right]^2 + \left[ \frac{-2cd}{D^2} - y_0 \right]^2} = \sqrt{\left[ \frac{c^2 - d^2}{D^2} + x_0 \right]^2 + \left[ \frac{2cd}{D^2} \right]^2}
\]

We see that the two moduli are equal. Hence, \( z_0 \) is in the set \( M_a \) if and only if \( \bar{z}_0 \) is also in \( M_a \). This gives symmetry about the real axis because the points above and below behave exactly similar to each other.

The next conjecture relates the set \( S_a \) with the set \( S_a \). Figures 6 and 7 illustrate an example with \( S_{1+i} \) and \( S_{1-i} \). We call these conjugate sets. They turn out to be reflections of each other over the real axis in the complex plane. We write in this case \( \overline{S_{1+i}} = S_{1-i} \).
The computer representations of similar examples depict such sets as reflected images of each other across the real axis, whenever the first set is produced with the parameter $a$ and the other is produced with $\bar{a}$. In other words, we noticed that if we have the sets $S_a$ and $S_{\bar{a}}$, then $S_a$ is the conjugate image of the set $S_{\bar{a}}$. We write $\overline{S_a} = S_{\bar{a}}$.

**Theorem 2.** For any chosen zero $a \in \mathbb{C}$, the conjugate images of $S_a$ and $M_a$ across the real axis are the image of $S_a$ and $M_a$, respectively. In other words, $\overline{S_a} = S_a$ and $\overline{M_a} = M_a$.

The proof is straightforward in the sense that we take the conjugate of the iterative process that determines the points in our sets.

**Proof.** $S_a$ and $M_a$ are determined according to the iteration $w = \left( \frac{z - a}{1 - \bar{a}z} \right)^2 + z_0$. Note $\bar{w} = \left( \frac{\bar{z} - \bar{a}}{1 - a\bar{z}} \right)^2 + \bar{z}_0$. Hence the iteration (at the point $z_0$) that determines $S_a$ and $M_a$ is the conjugate of the iteration that determines $S_a$ and $M_a$. Since $|w| = |\bar{w}|$ for any $w \in \mathbb{C}$, this relationship between iterations implies $\bar{z}_0$ will be an element of $M_a$ exactly when $z_0$ is an element of $M_a$. The result follows.

As we continued the research into the properties of these sets, we realized that most $S_a$ sets were bounded. This property means $S_a$ is inside a closed disk centered at $0$ with some radius $R$. A question that arises is how to find the radius $R$ of the circle given an $S_a$ set. This radius is found depending on the test value used to produce $S_a$.
We show (below) that outside of the circle, with radius R, no complex point \(z_0\) belongs to \(S_a\). This means the set is bounded.

**Theorem 3.** \(S_a\) is bounded for any \(a \in \mathbb{C}\) such that \(|a| > 1\).

The proof of Theorem 3 uses elementary principles about moduli of complex functions. The first line of the proof uses an identity for complex numbers when they are squared. We manipulate the formula to show how these points behave in their iterative values. This manipulation shows we can always choose a test value for any of these sets so that they will be bounded, or, equivalently, there will be points in the complex plane not inside the set. This bound on the set is artificial because of the way we set our test value, whereas the Mandelbrot set’s bound is absolute—the iterations for each complex point when constructing \(M_0\) either goes to infinity or produces numbers less than 4.

**Proof.** Let \(a \in \mathbb{C}\) with \(|a| > 1\). For \(z \in \mathbb{C}\),

\[
|W(z)|^2 = \left| \frac{z - a}{1 - \bar{a}z} \right|^2 = \frac{(z - a)(\bar{z} - \bar{a})}{(1 - \bar{a}z)(1 - a\bar{z})} = \frac{|z|^2 + |a|^2 - 2\text{Re}(a\bar{z})}{1 + |a|^2|z|^2 - 2\text{Re}(a\bar{z})} \quad \text{(where } W(z) = \frac{z - a}{1 - \bar{a}z})
\]

Hence \(|W(z)|^2 < 1\) if and only if

\[
|W(z)|^2 < 1 \iff |z|^2 + |a|^2 - 2\text{Re}(a\bar{z}) < 1 + |a|^2|z|^2 - 2\text{Re}(a\bar{z})
\]

\[
\iff |a|^2 - 1 < |z|^2(|a|^2 - 1)
\]

\[
\iff 1 < |z|
\]

Alternatively, \(|W(z)|^2 > 1\) if and only if \(|z| < 1\).
Therefore, for any $z_0$ with $|z_0| > 2|a|^2 + 1 > 2 > 1$,

$$|z_0 + W(z_0)^2| \geq |z_0| - |W(z_0)|^2 > |z_0| - 1.$$ 

Since $|z_0| > 2|a|^2 + 1 > 1$, $|W(z)|^2$ is a value that is in the disk centered at $z_0$ with radius 1, and so $|z_1| > 2|a|^2$. (Similarly, each $z_n$ with $n \in \mathbb{N}$ will then iteratively have absolute value larger than our bound $2|a|^2$.) In short, any point $z$ with $|z_n| > 2|a|^2 + 1$ will not be in $S_a$. Hence, $S_a$ is a bounded set when $|a| > 1$.

Theorem 3 refers strictly to sets $S_a$ with an $a$ value that has modulus greater than 1. Now we turn our attention to the other part of the plane—the points $a \in \mathbb{D}$ (where $\mathbb{D}$ is the unit disk in the complex plane). We examine $S_a$ sets with $|a| < 1$. Theorem 3 shows that all $S_a$ sets with $|a| > 1$ were bounded. Now, we prove the same for all $S_a$ sets with $|a| < 1$.

**Theorem 4.** $S_a$ is bounded for any $a \in \mathbb{C}$ such that $|a| < 1$ and $|a| \neq 0$.

In the beginning of this proof we take the points in the complex plane that are all at least a certain radius $R$ from $-a$. Namely, we choose $R > \frac{2}{|a|^2}$, the bound set for the $S_a$ sets, and go about showing that this initial condition proves the iterations’ values for each point $z_0$, with $|z_0|$ bigger than $\frac{2}{|a|^2}$, leads to $z_0$ not being in the $S_a$ sets. In the initial part of the proof we manipulate $|W(z_0)|^2$ to develop an upper bound on the modulus’s value. This allows a constant to be chosen in such a way that the point $z_0$ we begin with will not belong to $S_a$. Next, we show that none these points are in $S_a$ because $\frac{1}{|a| - \frac{1 + |a|^2}{R}} < \frac{2}{|a| - (1 + |a|^2)|a|^2}$. Finally, using the facts from earlier in the proof we show the modulus of $|z_1|$ is larger than $\frac{2}{|a|^2}$, which is the bound we set for our sets, and further that each $|z_n|$ is larger than $\frac{2}{|a|^2}$, meaning none of these $z_0$ points belong to $S_a$.
Proof. For such \( a \), and for any \( z_0 = -a + Re^{it} \), where \( R > \frac{2}{|a|^2} \) and \( |z_0| \geq \frac{2}{|a|^2} + K \), where \( K = \left[ \frac{2|a|}{|a|-(1+|a|^2)} + \frac{2|a|}{2|a|-(1+|a|^2)|a|^2} \right]^2 \), we compute:

\[
|W(z_0)|^2 = \left[ \frac{|-a + Re^{it} - a|}{|1 + \bar{a}(a - Re^{it})|} \right]^2 = \frac{|2a - Re^{it}|^2}{|(Rae^{it} - (1 + |a|^2))|}^2
\]

\[
\leq \left[ \frac{2|a| + R}{R|a| - (1 + |a|^2)} \right]^2 \quad \text{(using } |A - B| \geq |A| - |B| \text{ in the denominator)}
\]

\[
\leq \left[ \frac{2|a|}{R|a| - (1 + |a|^2)} + \frac{1}{|a| - (1+|a|^2)|a|^2} \right]^2 = K.
\]

The last inequality holds because

\[
R > \frac{2}{|a|^2} \iff \frac{1 + |a|^2}{R} < \frac{(1 + |a|^2)|a|^2}{2} \iff |a| - \frac{1 + |a|^2}{R} > \frac{2|a| - (1 + |a|^2)|a|^2}{2} \iff \frac{1}{|a| - \frac{1+|a|^2}{R}} < \frac{2}{2|a| - (1 + |a|^2)|a|^2}.
\]

Then,

\[
|z_1| = |W(z_0)|^2 + z_0 \geq |z_0| - |W(z_0)|^2 > |z_0| - K > \frac{2}{|a|^2} + K - K = \frac{2}{|a|^2},
\]

so \( z_0 \notin S_a \). In short, no point \( z \) with \( |z| > \frac{2}{|a|^2} + K \) can be in \( S_a \).

\[\square\]

Theorems 2 and 3 relied on the fact that we have a set test value that these points' moduli always exceed in the iteration. If you remember the reason we had to redefine our definition for these sets, then you might realize that if we start increasing the bound towards infinity the set will include the entire complex plane. In fact, we are
able to use the previous theorems statements to show that this will in fact be true for many of the points in the complex plane. This formally proves that each $M_a$ set with $|a| > 1$ is unbounded.

**Lemma 5.** If $a \in \mathbb{C}$ with $|a| > 1$, then $\{ z : |z| \geq 2 \} \subseteq M_a$. 

Using parts of the previous proofs help prove this lemma. Using these facts shows that each point $z_0$ has an iterative value of $|z_0| + 1$. This means that $|z_n| \leq 1 + |z_0|$. This is a very nice result because this is obvious when viewing the images of the sets, but having the idea formalized allows it to be more rigorous in the statement as a property of the $M_a$ sets.

**Proof.** Let $a \in \mathbb{C}$ with $|a| > 1$.

As before, for $z \in \mathbb{C}$,

$$|W(z)|^2 = \left| \frac{z - a}{1 - \bar{a} z} \right|^2 = \frac{(z - a)(\bar{z} - \bar{a})}{(1 - \bar{a}z)(1 - a\bar{z})} = \frac{|z|^2 + |a|^2 - 2Re(a\bar{z})}{1 + |a|^2|z|^2 - 2Re(a\bar{z})}.$$

Hence,

$$|W(z)|^2 < 1 \iff |z|^2 + |a|^2 - 2Re(a\bar{z}) < 1 + |a|^2|z|^2 - 2Re(a\bar{z})$$

$$\iff |a|^2 - 1 < |z|^2(|a|^2 - 1)$$

$$\iff 1 < |z|$$

Alternatively, $|W(z)|^2 > 1$ iff $|z| < 1$.

Therefore, for any $z_0$ with $|z_0| > 2 > 1$,

$$|W(z_0)^2 + z_0| \leq |W(z_0)|^2 + |z_0| < 1 + |z_0|.$$
Also, if $|z_0| > 2$, $z_1 = W(z_0)^2 + z_0$ is a value that is in the disk centered at $z_0$ with radius 1, and so $|z_1| > 1$.

Similarly, $z_2 = W(z_1)^2 + z_0$ has $|z_2| > 1$ (and $|z_2| < 1 + |z_0|$), which happens for each $n$:

$$z_n = W(z_n - 1)^2 + z_0$$

has $|z_n| > 1$, and $|z_n| < 1 + |z_0|$. So $z_0 \in M_a$.

The previous lemma was able to prove only that a certain unbounded portion of the complex plane belonged to $M_a$. This set represents all points in the complex plane outside of a circle centered at 0 with a radius of 2. Again, this is a nice result, but we see from the visual representations of these $M_a$ sets that the complex plane as a whole is part of the $M_a$ sets except possibly $a$ itself. This idea seems intuitive enough, but trying to prove it has been difficult. This next theorem extends the lemma previously stated. We are able also to include all of the $a$ values with $|a| = 1$, the important unit circle—the boundary of the unit disk in the complex plane.

**Theorem 6.** $M_a$ is unbounded when $|a| \geq 1$.

We have already proved this result when $|a| > 1$. We therefore need to prove $M_a$ sets are unbounded when $|a| = 1$.

**Proof.** When $|a| > 1$, the result follows from Lemma 5. If $|a| = 1$, then $a = e^{it}$ for some $t \in \mathbb{R}$, and,

$$W(z) \equiv \frac{z - a}{1 - \bar{a}z} = \frac{z - e^{it}}{1 - e^{-it}z} = e^{it} \frac{z - e^{it}}{e^{it} - z}, \text{ when } z \neq e^{it}$$

Hence $W(z)^2 = e^{2it}$ whenever $z \neq a$. This fact means the iterative process $z_n \equiv W(z_{n-1})^2 + z_0$ has $z_n$ of the form $e^{2it} + z_0$ for any $z_0 \in \mathbb{C}\{a\}$, and $|z_n|$ is the constant
\[ |e^{2it} + z_0| \text{.} \] Hence \( z_0 \in M_a \) for any \( z_0 \in \mathbb{C}\{a\}. \) \[ \square \]

In the previous theorem we were able to show that all \( M_a \) sets with \( |a| \geq 1 \) are unbounded sets. The proof of the theorem essentially shows that when \( |a| = 1 \), the set will contain the entire complex plane except possibly \( a \), rather than just a subset of it such as all points \( z \) such that \( |z_0| > 2 \). We now state this formally.

**Corollary 7.** When \( |a| = 1 \), \( \mathbb{C}\{a\} \subseteq M_a \).

**Proof.** The result is a restatement of the fact that \( z_0 \in M_a \) for any \( z_0 \in \mathbb{C}\{a\} \).

\[ \square \]

This section discussed the current results that we have obtained from our study of the Generalized Mandelbrot sets. There are other principles that we have conjectured. For example, we believe \( M_a \) is unbounded when \( |a| < 1 \). In fact, we conjecture that \( \mathbb{C} \setminus \{a\} \subseteq M_a \) for all \( a \in \mathbb{C} \). Because of time limitations, however, the results were not completed. This research only represents a beginning look into the properties of these sets and their properties. The vast number of possible fractals represents further avenues for research. One of the most exciting aspects of the research into these fractals is the stunning pictures that are created. These visual representations provide a strong basis to conjecture properties. The pictures also give tangible results that can create excitement in others when viewing the research.

---

\[ ^3 \text{Sometimes } a \text{ turns out to be in } M_a \text{ (such as for } a = -1) \text{ and sometimes it does not (such as for } a = 1). \]
IV Further Sets

The $S_d$ sets introduce examples of a vast number of sets using complex functions, including ones other than the Blaschke factor. We used the general definition: $S(f(z), N, K(f))$ given in Section II. We examined many different functions into the process and produced several fantastic pictures. The first type of functions we present here are transcendental functions. These functions have interesting qualities and properties which are a nice lead into the reason to investigate the sets they produce. The first transcendental function used was $\cos z$. The iterative process for this set reads as: $z_{n+1} = \cos z_n + z_0$. This process produces wonderful pictures in terms of complexity and visual beauty.

As you might notice in the Figure 9, there are many interesting aspects to the picture. This represents an avenue for research to investigate the specific properties that occur inside this set. This set was produced using a test value 10,000 and 1,000 iterations. Since the sets generated are able to be done with multiple test values and iterations, one question that might come to mind is “What will happen to this set as we change the test value?” The study of the Blaschke factor showed that if their test value is increased to a large enough value, the whole complex plane will be inside of the set. When producing $S(\cos z, K(f), N)$, we find the opposite occurs. One will notice this effect in Figure 10.

The interior of the set, the black part, stays essentially the same in each of these reproductions. The difference lies in the coloring outside of the main set. This leads to some interesting questions, such as: What is the test value needed to properly represent this set? Is it similar to the Mandelbrot set? For example, if $|z_n| > 1000$ for some $n$, is it always the case that $z_0$’s iteration produces moduli that go to infinity? These questions are still being explored to try to understand these sets in a deeper sense. The extent of the research into these sets has amounted to some conjectures being formed. However, the conjectures derived from the study of the pictures have
not led to new theorems or proven properties as of yet.

The set $S(\cos z, K(f), N)$ has many different interesting aspects and qualities. One such quality that is very noticeable in the set is something we have labeled the ‘red-veiling effect’. This ‘red-veiling’ leads to the boundaries of the set being in a sense chaotic. The effect appears in the pictures as a kind of cloud over the set, which is colored black in our pictures. The previous pictures, Figures 9 and 10, allows a look into this effect first-hand and may be one of the first things noticed by many who view this set. The reason for this effect is not really understood at this moment, but many preliminary ideas to as why it has shown up have been explored.
Many of these original ideas were based on the 'limited' ability of the computer. In a sense, the computer limits the ability to produce pictures that have an infinite number of iterations that may give a more true representation of the set. An infinite number may make a clear difference in the set, but remember that the Mandelbrot set is pretty easily generated using a small number of iterations for each complex point, even as low as 50 in some cases. It therefore seems reasonable to assume that having 1,000 iterations for each point gives a relatively true representation of the set.

The next idea centered around the precision of the calculations used in the program. Java in particular has built-in functions to calculate $\sin z, \cos z, \cosh z, \sinh z,$ and $e^z$. This leads to a question of the accuracy of these functions. In order to alleviate this concern, specific functions that are numerically stable were written in order to calculate each of these individual values. This was tested and shown not to make any significant difference between the sets, except in how the screen was colored outside of the set, showing that there was no discernible difference in the depiction of the set itself. Figure 11 displays the differences between the depictions, calculating them using the exact same parameters but different ways of calculating the functions in question.

Again, as noted before there isn't a difference between the two depictions in terms...
of points in the set. This alleviated the concern over the precision problem. This issue was the last thought tested as to why the red-veiling effect happens. This veiling effect needs to be studied more to understand why it happens. It is one of the more interesting aspects of the set. Another feature of this set is a periodicity across the complex plane. Because $\cos z$ is a periodic function, so is the set; the iterative process preserves it. Figure 12 displays what happens when you produce a picture of the set from $-5\pi$ to $5\pi$.

The pictures that were produced using $\cos z$ had fantastic results. Although there wasn’t enough time to study many of the properties of this set, it was nonetheless very satisfying to work on because of the visual beauty of the produced pictures. The next avenue was to study other generalized sets using different functions. These sets produced using $\sin z$ also created some fascinating pictures. Figure 13 is a standard depiction of the set.

These $S(\sin z, K(f), N)$ sets have a similar flavor to the $S(\cos z, K(f), N)$ sets, but they have very distinct features. One aspect to note are the leaves that come off of the main ‘bulbs’ of the sets; this effect also happens in the $\cos z$ sets that were produced. Even the red-veiling effect shows up in the $\sin z$ sets. The fact that the $\sin z$ sets have this effect is a quite pleasant result. This leads to the conclusion that some property of trigonometric functions are the cause for why this effect shows up.
The $\sin z$ sets also have a $2\pi$ periodicity along the real axis as can be seen in Figure 14.

The fact that the $\sin z$ sets has the periodicity is a good sign because the $\cos z$ sets have the same property. As noted before, it is very reasonable to expect the sets produced by $\sin z$ and $\cos z$ to have similar flavors in terms of the visual representations. Since the pictures have many similar properties, it gives a sense that the representations are verified, or at least substantiated in a strong way. These two functions represent the extent of the research into trigonometric functions to produce the generalized sets. The next transcendental functions that we studied stemmed from the exponential function. Again, the process $z_{n+1} = e^{z_n} + z_0$ produces a fascinating
set. Figure 15 is a picture of the set.

Figure 15 also has a figure in the set that repeats in the complex plane, but here the figure repeats about the imaginary axis, rather than the real axis as in \( \sin z \) and \( \cos z \). The set also has a boundary on the real line where every point to the left will be in the set. This is an initial conjecture that is pretty obvious from a picture of this set. The interesting parts of this set lie in a small interval on the real line. This interval contains the figure that is of the most interest. The figure that repeats has a lot of complexity (in terms of the points that are not in the set), more so than just on the boundary of the actual set colored in black.

We then constructed \( S(e^{-z^2}, 1000, 1000) \). The function \( e^{-z^2} \) is of particular interest to mathematicians. It is, for example, the bell curve in statistics when \( z \) is real. The set produced using this process has some interesting aspects like the other sets, but it is still unique. Figure 16 is a picture of the set.

The most interesting parts of these sets come from the chaos present in the middle of the picture. This chaotic behavior in the middle of this depiction is possibly the best avenue for continued research. Time constraints have limited the study of the sets produced by transcendental functions. The eventual goal is to have a continuation of this research that looks fully into these properties, and produces more fractals that
are visually appealing.

The next generalized sets discussed in this section are an extension of the original definition used for the $S(f(z), K(f), N)$ sets. This extension allows the complex constant that is being added in the process to be put through the main function. In more explicit terms, instead of having just $z_{n+1} = f(z_n) + z_0$, we can have a process $z_{n+1} = f(z_n) + f(z_0)$. We label these sets formally as $T(f(z), K(f), N)$. This produces still more interesting pictures that can be studied and enjoyed for their beauty. The pictures below show many of the sets produced using this extended definition. The captions display the functions used to produce each individual set.

The last set studied in the scope of this research was the one produced with
Figure 18: $T(\sin z + \sin z_0, 1000, 1000)$

Figure 19: $T((z-a)^2 + (z_0-a)^2 \text{ at } .1 + .1i, 1000, 1000)$
the singular inner function. Here the iterative function is \( z_{n+1} = e^{(z_{n-1})^2} + z_0 \). The singular inner function is of interest because of the special properties that it has. This includes mapping the entire unit disk \( \mathbb{D} \) to itself, along with being analytic on the unit disk. We have used a color scheme for the representation of this set to highlight this fact. The points that lie outside the unit disk are colored white, while the other points are colored according to how they act in the iterative process. This set was produced at the final stages of the research and the depth of the study of the set was strictly constrained to the production of the set. However, the picture of the set is still interesting and has fascinating qualities. The pole at \( z = 1 \) of the singular inner function is clearly influencing the behavior of the set. There is a ray emanating from this pole at a 45° angle to the bottom left where much of the chaos in the set occurs.

The next section of this thesis discusses the algorithms used for the production of these sets. The computer program used when creating these pictures is given in Appendix A. The discussion of the algorithms simply present pseudo-code when describing the program. Any user can use their preferred language when programming. It is important to remember that much of the interest from mathematicians in fractals comes from the special mathematical properties, but the public’s interest generally stems from the visual beauty and complexity that these sets have intrinsically. This
fact makes the computer production of the sets extremely important; the need to create excitement for the study of these sets. For example, we want someone to be able to use the program to visualize whatever portion of the set they want to see. The next section describes how such algorithms of the computer program work.
V Generalized Mandelbrot Set Algorithms

V.I Introduction

This section discusses the algorithms used to display the Generalized Mandelbrot sets. Some of these algorithms are well known, while others were developed during the scope of the programming in order to study our sets. The first main program produced visualizations of the Blaschke factor generalized sets. Many of the algorithms that are discussed were developed during this initial phase. This program provided a framework for the specific programs used to produce the generalized sets with other functions. We discuss and analyze the general algorithms used, then the specific algorithm used to calculate the fractals associated with the Blaschke factor. Finally, the algorithms for the other functions that used in the production of sets are discussed and analyzed, including \( \cos{z}, \sin{z}, e^z \).

V.II Mandelbrot Algorithm

The general algorithm used to produce the Mandelbrot set is well known and easy to follow. It was first published in 1985 in an article in *Scientific American.*\[9\] We used this same algorithm with some modifications.

We begin by computing a correspondence between the pixels in the window to points in the complex plane as follows: 1. The window size is set by the user. The range the user enters for the real and imaginary axes are used to find two scales between pixels in our window. Two scales are calculated: one for the vertical direction and one for the horizontal direction. The formula for these are: 

\[
x_{-\text{gap}} = \frac{x_2 - x_1}{\text{width}}, \quad y_{-\text{gap}} = \frac{y_2 - y_1}{\text{height}}.
\]

2. Each pixel in the window, represented as a complex number \( z_0 \), is iterated through \( z_{n+1} = z_n^2 + z_0 \) where the pixel's complex value is \( z_0 \). Since \( z_n \)'s are complex values, they have an imaginary and real part, which are calculated as described in the introduction. Next, \( z_{n+1} \) is compared to a test value (this test value
is 4 for the Mandelbrot set). This tells us if the pixel is in the set. 3. The coloring of pixels is then determined by two cases. First, if the pixel is in the set, the pixel is colored black. Second, the pixel is colored by the coloring algorithm according to the number of iterations it takes for \( z_0 \) to become larger than the test value.

This algorithm produces a visual representation of the Mandelbrot set, and it can be easily generalized and used for the Generalized Sets. This algorithm's runtime can be calculated by looking at three values used in the program. These values are the dimensions of the window and then the largest number of iterations a point can run through. Theoretically, the number of iterations is unbounded for any given complex point. So, in practice we limit \( n \) to 1,000. The time complexity for the worst case for this algorithm, or where every pixel in the window would be in one of our sets, is \( \text{width} \cdot \text{height} \cdot \text{iterations} \). Typically, we set the window around \( 1200 \cdot 800 \approx 1 \) million pixels. This leads to the number of operations for the worst case in the program taking around a billion operations. One possible concern that arises from limiting the number the number of iteration centers around the integrity of the visual representation of the set—is the picture compromised in any way? This concern, however, becomes unwarranted because it turns out the detail of the set doesn't change in any significant way when the number of iterations is 1,000 or 10,000 or even 100,000. It turns out from a quick test case that the number of points inside of the Mandelbrot set using our algorithm goes from 130,795 pixels at 1,000 iterations to 130,522 at 10,000 and then to 130,505 at 100,000. The increased computing time then represents a diminishing return. In this case, 1,000 iterations serves the production of these sets fine, and the tests reveal similar results for the Generalized Sets. We see from the above calculations that only .2 percent of the pixels in the set at 1,000 iterations are lost once when the number of iterations is increased to 100,000. It's safe to assume this is practically insignificant and we conclude 1,000 iterations serves as a nice cutoff point between precision and performance. The tests also show that 100
iterations generally is much less accurate compared to 1000 iterations. The number of pixels in the Mandelbrot using our program at 100 is 133,936. This is an almost 2.5 percent increase from 1000 iterations. This could possibly change the boundary of the set and is another reason we use 1000 iterations for our depictions of the Generalized Sets.

V.III Blaschke Algorithm

The Mandelbrot algorithm above was the framework for producing Generalized Sets. The general iteration for the Mandelbrot set is \( z_{n+1} = z_n^2 + z_0 \), using the function \( z^2 \) in the iterative process and checking to see if each point diverges to infinity or stays bounded, as discussed in the introduction. We generalized this formula to \( z_{n+1} = f(z_n) + z_0 \) or \( z_{n+1} = f(z_n) + f(z_0) \). We also introduce a “test bound” for each iteration to be compared against rather than just checking for divergence. Here \( f \) is any complex function. The challenge then lies in calculating the iterative value for each complex point in our window. The solution to this challenge lies in rewriting each in terms of the real and imaginary parts, not dissimilar to the process for the Mandelbrot set. The first function used in creating Generalized Sets was \( f(z_n) = \left[ \frac{z_n - a}{1 - az_n} \right]^2 \), using the iterative process \( z_{n+1} = \left[ \frac{z_n - a}{1 - az_n} \right]^2 + z_0 \).

First, we calculate \( z_{n+1} = \left[ \frac{z_n - a}{1 - az_n} \right]^2 + z_0 \). We then write \( z_{n+1} \) as \( x + iy \). This formula then becomes:

\[
z_{n+1} = \frac{[(1 + x_n^2 - y_n^2)x_0 - (1 + x_0^2 + y_0^2)x_n + 2x_n y_n y_0 + i((1 - x_n^2 + y_n^2)y_0 - (1 + x_0^2 + y_0^2)y_n + 2x_n y_n x_0)]^2 + (x_0 + iy_0)}{[1 - 2x_0x_n - 2y_0y_n + (x_n^2 + y_n^2)(x_0^2 + y_0^2)]^2}
\]

Now, to simplify let \( c = [(1 + x_n^2 - y_n^2)x_0 - (1 + x_0^2 + y_0^2)x_n + 2x_n y_n y_0] \), \( d = [(1 - x_n^2 + y_n^2)y_0 - (1 + x_0^2 + y_0^2)y_n + 2x_n y_n x_0] \), and \( e = [1 - 2x_0 x_n - 2y_0 y_n + (x_n^2 + y_n^2)(x_0^2 + y_0^2)] \). Then, the equation for \( z_{n+1} \) becomes: \( \left( \frac{c^2 - d^2}{c^2} + x_0 \right) + i \left( \frac{2cd}{c^2} + y_0 \right) \). This equation is used in the algorithm to produce the Blaschke generalized sets. A loop iterates each pixel to determine which pixels lie inside, and outside, of the set. The pseudo-code
for this part of the program is as follows, given we start with a complex number $a$
for our Blaschke factor (remember $a = 0$ for the Mandelbrot set) and $z_0$ which is the
complex number tested:
Pseudo-code:

\textit{Iteration}()

\begin{align*}
    x_a &= \text{real}(a) \\
    y_a &= \text{imag}(a) \\
    x_0 &= \text{real}(z_0) \\
    y_0 &= \text{imag}(z_0)
\end{align*}

\text{for}(n < \text{iterations})

\begin{align*}
    c &= ((1 + (x_a^2) - (y_a^2)) \cdot x_n) - ((1 + (x_n^2) + (y_n^2)) \cdot x_a) + (2 \cdot x_a \cdot y_a \cdot y_n) \\
    d &= ((1 - (x_a^2) + (y_a^2)) \cdot y_n) - ((1 + (x_n^2) + (y_n^2)) \cdot y_a) + (2 \cdot x_a \cdot y_a \cdot x_n) \\
    e &= (1 - (2 \cdot (x_n \cdot x_a)) - (2 \cdot (y_n \cdot y_a)) + (((x_a^2) + (y_a^2)) \cdot ((x_n^2) + (y_n^2))))
\end{align*}

\begin{align*}
    x &= (((c^2) - (d^2))/((e^2))) + x_0 \\
    y &= ((2 \cdot c \cdot d)/((e^2))) + y_0 \\
    \text{if}((x \cdot x) + (y \cdot y) > \text{bound})
\end{align*}

Notice this formula also produces the Mandelbrot set, so two tasks are accom-
plished at the same time. The runtime for this algorithm is essentially identical to
the runtime for the Mandelbrot set algorithm.

\section*{V.IV Coloring Algorithm}

The Mandelbrot set's most stunning pictures rely on the coloring near the bound-
dary. Coloring of the Generalized Sets then becomes important. Developing a strong
coloring algorithm helps display the beauty of these sets. The coloring algorithm
determines the color of each point in the window of the Generalized Sets. Many coloring algorithms use the “escape times”—the number of iterations a point takes to exceed a certain test value—of each point (the same as we do for our program). The true visual beauty of these sets lie in the chaos of their boundaries. Providing the stunning visual aspect adds to the excitement of studying these sets.

The coloring algorithm’s main input, as noted in the previous paragraph, is a variable that is determined according to the escape times of each point. Specifically, the escape times are parameterized in such a way to ensure that they fall between 0 and 1. In other words, if the maximum number of iterations is 1,000 and a point escapes after \( n \) iterations, then escape—the variable—equals \( \frac{n}{1000} \). Escape is saved for each individual point in the window. Then the variable is put into a function to vary the color. The functions used in this algorithm relate to the Bernstein polynomials. A Bernstein polynomial is of the form: \( b_{v,n}(x) = \binom{n}{v} x^v (1-x)^{n-v} \). The specific functions used are shown in the pseudo-code below. Colors are made on a computer by taking three or four values that are then placed on an RGB scale. This scale stands for red, green, and blue, each being a component of a color. Another variable that can be added to these three is an alpha value. This value determines the transparency of a color, with a high value being non-transparent and a low value being completely transparent. In Java specifically, the color is constructed by taking these four values as integers, and a color is represented as \((r, g, b, \alpha)\). It is preferable to have “continuous”—a gradient coloring pattern—coloring of these sets rather than having “bands”—single colors that form a band around the set—of colors around the sets. To have a continuous coloring algorithm, a function is needed that produces a specific value for each “escape time”. This function allows the computer to create very specific colors for each individual point. As discussed above, the functions created used an adaptation of the Bernstein polynomials. This scheme is perfect for this algorithm because of the Bernstein polynomial behavior in the interval \([0, 1]\). These
functions have an output ranging from 0 to 255, with 255 being the highest integer that can be used for the red, green, blue, or alpha value in creating a color, as we move across the interval $[0, 1]$. Specifically, we are able to make them vary from 0 to 255 by multiplying the base functions by a certain constant. Further, these functions are constructed so that each peaks in a specific area in this interval. This creates better color complexity for the points that are escaping towards infinity.

Pseudo-code:

```plaintext
//array containing escape points for pixels in window
mb[]][loop(i < height)
    loop(j < width)
        if(1 < mb[j][i] < 2) // meaning the iteration escaped in the first or second step
            color = white
            fillpixel(j, i, 1, 1) // fill the pixel at position j, i with white
        if(mb[j][i] = 0)
            color = black
            fillpixel(j, i, 1, 1) // fill pixel at position j, i with black
        if(2 < mb[j][i] < MaxIterations) // meaning the iteration escaped late
            cont = mb[j][i]/MaxIterations // variable we create that is between 0 and 1
            color = Color(14 * (1 - cont)^5 * cont * 255),
                    (29 * (1 - cont)^3 * cont^2 * 255),
                    (9.4 * (1 - cont) * cont^3 * 255), 255); // red color to white
            fillpixel(j, i, 1, 1); // fill pixel at position j, i with color
```

The runtime of the coloring algorithm is essentially equal to the time to save the
escape time of each point and to produce a color for each pixel. This means that the time complexity is equal to the number of pixels that are in the windows of the sets. More specifically, if a window has dimensions 1200 pixels by 800 pixels, then the runtime is about a million operations. This saving operation is done inside of the loops that iterate each point. Then the parameterized variable is calculated for each individual point. Next, the variable is plugged into the functions to create a new color, which is then used to color the pixel appropriately. The time complexity for these steps essentially boil down to a small constant times the total number of pixels. In other words, this algorithm takes $n \times 960,000$ operations, where $n < 50$.

**V.V Movie Algorithm-Line**

During the study of these sets, it was important to showcase how these sets changed as the parameter $a$ moved about the complex plane. In order to show how these sets changed as $a$ moved about the complex plane, a “movie”—a slide show consisting of many pictures—needed to be created. The frames in the movies are each a picture of a set produced by a specific $a$ value, then the pictures are used to produce the movie in a movie-making program.

The user inputs the values for the parameter $a$ to determine what movement in the complex plane the movie shows. For instance, one could have the movie show the change in the sets as $a$ moves from $-2$ to $2$ on the real line. Once the inputs are obtained, a parameterization is calculated for $a$. The program then runs the general algorithm for each single $a$ value, producing a picture for each. Specifically, let’s say the user wants to see how the sets change from $-1$ to $1$ on the real axis. The program creates pictures of each Generalized Set from $-1$ to $1$ by adding $\frac{1-(-1)}{\text{numofpics}} = \frac{2}{\text{numofpics}}$ to $a = -1$ until we reach $1$. The user can also input values for $a$ to vary in the vertical direction.

Pseudo-code:
\[
x_{\text{gap}} = ((x_2 - x_1)/200)
\]
\[
y_{\text{gap}} = ((y_2 - y_1)/200)
\]

// start for loop to make images for generalized sets
for (yloop = y_1; yloop < y_2; yloop = yloop + y_{\text{gap}})
    for (xloop = x_1; xloop < x_2; xloop = xloop + x_{\text{gap}})
        Iteration()

Note, \(x_1, x_2, y_1, y_2\) are all inputs from the user where \(x_1\) is the start for the parameterization of the real values and \(x_2\) the stopping point. The same is true for \(y_1\) and \(y_2\).

As you can see from the above code, we create the pictures desired for the movie as the program moves the parameter \(a\) about the complex plane. Notice this specific movement algorithm only moves in a horizontal or vertical line about the plane. This construction is a natural place to start when developing algorithms that create movies to display how these sets change when moving about the complex plane. The question then is—What other movements should we consider? Would moving about a linear function in the complex plane show us interesting aspects of these sets? Would moving in a circle about the complex plane give us different insight into these sets? These types of questions led us to develop different algorithms to produce slide shows of the sets utilizing other movement patterns.

The time complexity of this movement algorithm is the time required to create a single picture of one of the Generalized Sets multiplied by the number of pictures that are created for the movie. In this way, the runtime is \(p \cdot \text{iterations} \cdot 960,000\), where \(p = \text{number of pictures}\). For the case where both the imaginary and real values are parameterized, the runtime is: \(p_1 \cdot p_2 \cdot \text{iterations} \cdot 960,000\), where \(p_1 = \text{number of pictures for real values}\) and \(p_2 = \text{number of pictures for imaginary values}\). When both directions vary, the process of producing the pictures generally takes a substantially
longer period of time.

V.VI Movie Algorithm-Circle

The next algorithm produced to illustrate a movement of the parameter $a$ about the complex plane is one that moves in a circle. The study of the change in the sets by this movement could give insight into their properties. The unit disk is one of the most studied areas in the complex plane and this movement can demonstrate how these sets change as we move about it. Developing this algorithm allows one to see these sets in a unique way, while also providing something interesting for the general public.

The algorithm begins by taking a single input. This input is the radius that is desired for the circle that moves about the complex plane. Then, from this, we move in a circle by calculating what the next imaginary number should be for $a$ as we add $\frac{2\pi}{\text{numofpics}}$ to the angle. The angle addition uses a similar idea to the first movement algorithm discussed, where we have a number of pictures to create for the slide show, and depending upon the number of pictures desired, we calculate how much we move from one frame to the next. This is done by recalculating $a$ using the new angle, while keeping the radius constant.

Pseudo-code:

$$\text{anglegap} = \frac{2\pi}{\text{numofpics}}$$

for(double angle = 0; angle <= (2 * Math.PI); angle+= anglegap)

xloop = radius * cos(angle)
yloop = radius * sin(angle)

Iteration()

This circular movement algorithm runs in similar time to the previous movement algorithm. In this sense, the runtime is $n \cdot \text{iterations} \cdot 960,000$. Here, $n$ is the number
of pictures desired for the movie.

V.VII Zoom Algorithm

One of the most amazing features of the Mandelbrot set lies in the complexity of the chaotic boundary. The boundary shows the self-similarity of the Mandelbrot set. Since this is one of the most fascinating properties of the Mandelbrot set there is a need to view this aspect for our research. The Generalized Sets also tend to have a similar property. Many do not have self-similarity, but they tend to have chaotic boundaries that resemble Julia sets. [6] This is an exciting aspect and one that requires further study. Zooming in on the boundary of these sets, or other “interesting” parts of the sets, creates visually appealing pictures.

The zooming algorithm works by creating the depiction of the set in an active window. The user can then click the pointer in the location they want to zoom. The program then saves the picture that is currently being viewed and zooms into the new location, resetting the dimensions of where the set is being viewed. The user can continue to zoom until a desired level is reached. At each level of the zoom, once the dimensions of the set are recalculated, each pixel in the window is recalculated and colored accordingly, in a sense “repainting” the window. This repainting works in the same way as the general algorithm for producing the Generalized Sets. The pictures in Figure 21 show the ability of the zoom function in our program and some of the beautiful pictures that it can create.

The runtime for this algorithm is essentially the same as the algorithm used to produce a single Generalized Set. Hence, the runtime is \( \text{iterations} \cdot 960,000 \). There are instances where this zoom algorithm tends toward the worst case in time complexity, but also instances where the runtime is faster. This occurs when one zooms into an area where almost every pixel in the window is in the set, or when the opposite is true. Then each point will either go through the maximum number of iterations
or close to the minimum number of iterations (namely under 10 iterations). Still, the technical worst case runtime of this zoom algorithm at each step of a zoom is \( \text{iterations} \cdot 960,000 \) and generally is faster in terms of actual performance.

**V.VIII \( \cos z, \sin z, e^z, e^{-z^2}, \text{and } e^{z+1} \text{ Algorithms} \)**

The definition we provide in the first chapter allows the production of many other sets where \( f(z) \) is chosen. One such complex iteration we used in the further study of the generalized sets was \( z_{n+1} = \cos z + z_0 \). The motivation for the further study was brought in part by the fact that the \( M_a \) sets using \( f(z) = \left( \frac{z-a}{1-az} \right)^2 \) are all unbounded. This means that every single point in the complex plane belongs to these sets as long as a large enough test value is used. Pictorially, this can be understood as every point in the picture of an \( M_a \) set being colored black. This does not create an interesting picture. So we extended the definition to allow for any \( f(z) \), which brings a multitude of new sets that have interesting qualities. For instance, when using \( f(z) = \cos z \) we produce stunning pictures. Figure 22 is one of the visual representations we produced of this set using our coloring algorithm.
The challenge in programming these new functions lie in altering the formulas into something that can be interpreted by a computer. The rest of the computer program is the same for these extended sets. The process for calculating this iteration is similar to what was done for the Blaschke factor discussed earlier in this chapter. Now we present the pseudo-code for \( Z_{n+1} = \cos Z_n + Z_0 \), with \( Z_n = x_n + iy_n \) and \( Z_0 = x_0 + iy_0 \). It needs to be noted that \( \cos z = \frac{e^{-iz} + e^{iz}}{2} \). Then, the iteration \( Z_{n+1} = \cos Z_n + Z_0 = \frac{e^{iz_n} + e^{-iz_n}}{2} + Z_0 \). Through some manipulation this can be rewritten in such a way that it can be easily formulated into a computer. In this case \( Z_{n+1} = \left[ \frac{1}{2}(e^{-y_n} + e^{y_n}) \cos x_n + x_0 \right] + i \left[ \frac{1}{2}(e^{y_n} - e^{-y_n}) \sin x_n + y_0 \right] \).

The next function we studied was \( \sin z \). The function \( \sin z \) has the complex representation \( \frac{e^{iz} - e^{-iz}}{2i} \). Using this representation we can rewrite the formula for the iteration \( Z_{n+1} = \sin Z_n + Z_0 \) as \( Z_{n+1} = \left[ \frac{1}{2}(e^{-y_n} + e^{y_n}) \sin x_n + x_0 \right] + i \left[ \frac{1}{2}(e^{y_n} - e^{-y_n}) \cos x_n + y_0 \right] \).

The rest of the functions that we studied all start with the \( e \) as a base raised to some power. We start with \( e^z \) used as the function in our extended definition for the Generalized Mandelbrot sets. The iteration is then \( Z_{n+1} = e^z_n + Z_0 \). As before we need to write this formula in terms of the real and imaginary parts so that we can program it into a computer. The iteration then becomes \( Z_{n+1} = [e^{x_n} \cos y_n + x_0] + i[e^{x_n} \sin y_n + y_0] \).
The next function we used in the production of the Generalized Mandelbrot sets was $e^{-z^2}$. The iteration is then $z_{n+1} = e^{-z_n^2} + z_0 = [e^{(y_n^2-x_n^2)} \cdot \cos(2x_ny_n) + x_0] + i[-e^{(y_n^2-x_n^2)} \cdot \sin(2x_ny_n) + y_0]$. Again, the pseudo-code is not presented as it is almost similar to the previous codes for $\cos z$ and $\sin z$ and requires a small change in the way that the real and imaginary parts are calculated for the $z_n$ term.

The last function used in the production of Generalized Sets was $e^{\frac{z+1}{z-1}}$. The iteration is then $z_{n+1} = e^{\frac{z_{n+1}}{z_n-1}} + z_0 = [e^{\frac{2x_n^2+y_n^2-2}{x_n^2+y_n^2-2x_n+1}} \cdot \cos(\frac{4y_n}{x_n^2+y_n^2-2x_n+1}) + x_0] + i[e^{\frac{2x_n^2+y_n^2-2}{x_n^2+y_n^2-2x_n+1}} \cdot \sin(\frac{4y_n}{x_n^2+y_n^2-2x_n+1}) + y_0]$. Using these formulas we then change the calculation method in the above pseudo-codes for $\cos z$ or $\sin z$ to reflect the new formula when calculating $z_n$.

The runtimes for all of these algorithms run very similar to the algorithm for the general Mandelbrot set. These algorithms generally take somewhat longer because of function calls to calculate the sine and cosine values, along with the exponential function $e^z$. Still the overall time complexity falls into the range of the general algorithm for the Mandelbrot set with a small increase in the overall running time.
VI Conclusion

Nonlinear complex functions provide Generalized Sets with chaotic features. This paper used, in addition to Mandelbrot's $z^2$, the functions $f(z) = \left(\frac{z-a}{1-\bar{a}z}\right)^2$, $f(z) = \cos z$, $f(z) = \sin z$, $f(z) = e^z$, $f(z) = e^{-z^2}$, and $f(z) = e^{z+1}$. Each set produced fantastic pictures. We zoomed into the boundaries of these sets to view the chaotic behavior in a microscopic sense. We also parameterized input values for the Blaschke factor, which created movies and slide shows that viewed these sets in very interesting ways. It is important to note we are not limited by just a single picture that is produced of the sets. Rather, we have a plethora of research opportunities that were followed and can continue to be researched.

The research had many different avenues at the onset. The initial development of the computer program to view the Generalized Mandelbrot sets $M_a$ was extremely successful. It provided the framework for the study and production of the other Generalized Sets produced. The development of the zooming algorithm helped view some of the most interesting aspects in some of these sets, including the appearance of the Mandelbrot set in the set produced by $\cos z$. The algorithms developed to create movies of the Generalized Sets proved to be extremely useful in displaying how these sets change to an audience. These movies are interesting by themselves, but when coupled with an understanding of the Generalized Set's behavior it allowed for a deeper understanding of the properties governing these sets. In fact, the computer program provided a basis for the deeper mathematical study of the Generalized Sets. Without the program, many of the theorems proven and conjectures made would not have occurred. Overall, this research has had numerous successes in terms of new research into an area that is somewhat unknown, producing interesting pictures and movies that would induce excitement, and sometimes awe, in people who view them.
A  The Java Program Code

import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.awt.geom.*;
import java.awt.event.*;
import java.awt.Color;

class Closer extends WindowAdapter{
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }
}

class Mandelbrot1 extends JPanel implements MouseListener, MouseWheelListener{

    //variables
    int MaxIterations = 1000;
    double x = 0;
    double y = 0;
    double xtemp = 0;
    double ytemp = 0;
    double check = 0;
    Color color;
    int rgb = 0;

    //blaschke variables
    public double x1 = 0;
    public double y1 = 0;
    public double x2 = 0;
    public double y2 = 0;
    public double realx = 0;
    public double imagx = 0;
    public double den = 0;
    public double real0 = 0;
    public double imag0 = 0;
    public double den0 = 0;
    public double cent = 0;

    //window size
    public static final int WINDOW_WIDTH = 1024;
    public static final int WINDOW_HEIGHT = 768;

    //gaps
    public double firstDim = 0;
public double secondXDim = 0;
public double firstYDim = 0;
public double secondYDim = 0;
public double tempfirstXDim = 0;
public double tempsecondXDim = 0;
public double tempfirstYDim = 0;
public double tempsecondYDim = 0;
double bound = 0;
int count = 0;
int [][] sb;
double [][] zdistance;

//fill Mandelbrot Array
double xgap = 0;
double ygap = 0;
double xagap = 0;
double yagap = 0;
double xatemp = 0;

constructor
public Mandelbrotf(){
setSize(1024,760);
addMouseListener(this);
addMouseWheelListener(this);
mbCalc();
}

//calculates new mb[] once values are changed for coloring
public void mbCalc(){
xgap = (secondXDim-firstXDim)/WINDOW_WIDTH;
ygap = (secondYDim-firstYDim)/WINDOW_HEIGHT;
xagap = (double)((xa2-xa1)/200);
yagap = (double)((ya2-ya1)/200);
xatemp = xa1;

//start for loop to make images for Mandelbrot sets
for(double yaloop = ya1; yaloop<ya2; yaloop+=yagap){
    if(yagap==0) yagap++;
    //xa1 = xatemp;
    for(double xaloop = xa1; xaloop<xa2; xaloop+=xagap){
        if(xagap==0) xagap++;
        //bound for calculations
        if(xaloop==0 && yaloop==0)
            bound = 4;
        else if(xaloop+xaloop+yaloop+yaloop<=1)
            bound = 2/(xaloop+xaloop+yaloop+yaloop);
        else if(xaloop+xaloop+yaloop+yaloop>1)
            bound = 2*(xaloop+xaloop+yaloop+yaloop);
        System.out.println("Bound is : "+bound);
        mb = new int[1024][743];
        zdistance = new double[1024][743];
        int xdimension=0;
        int ydimension=0;
        //go through each of the points in our designated area
for(double i=secondYDim; i>=firstYDim; i-=ygap){
    if(ygap==0) i-=10;
    /move for dimensions in big array
    xdimension+=0;
    ydimension+=0;
    for(double j=firstXDim; j<=secondXDim; j+=xgap) {
        if(xgap==0) j+=10;
        /dimensions and resetting variables at each step as we move across real line
        xdimension+=0;
        y = 0;
        realz = 0;
        imagz = 0;
        den = 0;
        x0 = j;
        y0 = 1;
        //for adding blaschke factor constant
        realz0 = ((1+(xaloop*xaloop)+(yaloop*yaloop))*x0)-(((1+(x0*x0)+(y0*y0))*xaloop)+(2*xaloop*yaloop*y0);
        imagz0 = (((1+(xaloop*xaloop)+(yaloop*yaloop))*y0)-((1+(x0*x0)+(y0*y0))*yaloop)+(2*xaloop*yaloop*x0);
        den0 = (1+(x0*x0)+(y0*y0));
        if(den==0) {
            //adding blaschke constant
            x = (((realz+realz)-(imagz+imagz))/((realz0+realz0)-(imagz0+imagz0));
            y = (((2*realz+imagz)/(den+den))+(2*realz0+imagz0)/(den0+den0));
            //adding complex constant
            x = (((realz+realz)-(imagz+imagz))/((den+den)))+x0;
            y = ((2*realz+imagz)/(den+den))+(2*realz0+imagz0)/(den0+den0));
        }
        else{
            //adding blaschke constant
            x = (((realz+realz)-(imagz+imagz))/((den+den)))+x0;
            y = ((2*realz+imagz)/(den+den))+(2*realz0+imagz0)/(den0+den0));
            //check for colors
            if(check>bound) {
                zdistance[xdimension][ydimension] = check;
                mb[xdimension][ydimension] = k;
                break;
            }
        }
    }
}
//end if for break in iterations to color/fill nb array

if (check == true) {
    mb[dimension][ydimension] = 0;
}

//end else statement for break to fill mb array

//end for loop for iterative process to check numbers
//end for loop to go through x dimensions
//end for loop to go through y dimensions

//create the buffered images
mandelbrotBufferedImage(mb, xdistance);

//end for loop to go through x values of a
//end for loop to go through y values of a

} //Zoom in function(3x zoom)
public void mouseClicked(MouseEvent e) {
    //variables
double xposition, yposition, xZoom, yZoom, xZoomGap, yZoomGap;
    double getX = (double) e.getX();
    double getY = (double) e.getY();
    //set variable values to zoom in to correct place
    xposition = (double) ((double) e.getX() / 1024); //calculates where we are on the page
    yposition = (double) ((double) e.getY() / 1024); //where we are on the page
    xposition = xposition * (secondXDim - firstXDim); //gives position according to scale
    yposition = yposition * (secondYDim - firstYDim); //gives position according to scale

    //where we clicked according to graph position
    xZoom = firstXDim + xposition;
    yZoom = secondXDim - yposition;
    xZoomGap = ((secondXDim - firstXDim) / 3); //implements 3x zoom
    yZoomGap = ((secondYDim - firstYDim) / 3);

    //new gap calculations
    secondXDim = xZoom + xZoomGap;
    firstXDim = xZoom - xZoomGap;
    secondYDim = yZoom + yZoomGap;
    firstYDim = yZoom - yZoomGap;

    //recalculate the mb[][] array and repaint it
    mbCalc();
    repaint();
}

//Zoom out function(zooms all the way out to original picture)
public void mouseWheelMoved(MouseWheelEvent e) {
    //new gap calculations
    secondXDim = tempSecondXDim;
    firstXDim = tempFirstXDim;
secondYDim = tempsecondYDim;
firstYDim = tempfirstYDim;

// recalculate the mb [] [] array and repaint it
mbCalcO;
repaint();

public void mandeLbronuffered ImageOnt (int [] [] mb, double [] [] zdf snance) {

// create buffered image with Mandelbrot set
BufferedImage MandelbrotPic = new BufferedImage (1024, 768, BufferedImage.TYPE_INT_RGB);
for (int i = 0; i < 768; i++) {
    for (int j = 0; j < 1024; j++) {
        if (i < mb[j][i] && mb[j][i] < 2) {
            color = new Color (255, 255, 255); // white
            rgb = color.getRGB();
            MandelbrotPic.setRGB(j, i, rgb);
        } else if (mb[j][i] <= 0) {
            MandelbrotPic.setRGB(j, i, 0);
        } else if (mb[j][i] <= MaxIterations)
            cont = ((double)mb[j][i] - Math.log(Math.log(1 + mb[j][i])) / Math.log(bound)) / MaxIterations;
        if ((double)mb[j][i] / (double)MaxIterations) < 0.05
            cont = ((double)mb[j][i] / (double)MaxIterations) + 0.05;
        else
        if (mb[j][i] <= Math.log(Math.log(Math.log(mb[j][i]))))
            cont = (double)mb[j][i] / (double)MaxIterations;
        else
            cont = ((double)mb[j][i] - Math.log(Math.log(Math.log(mb[j][i]))) / Math.log(bound)) / MaxIterations;
    }
}

for (int i = 0; i < 768; i++)
    try {
        // save image to file
        try {
            // retrieve image
            File outputfile = new File("a" + count + ".png");
            ImageIO.write(MandelbrotPic, "png", outputfile);
            System.out.println("The picture saved.");
        } catch (IOException e) {
            System.out.println("This didn't work. ");
        }
    }

    // paint new picture for the window
    public void paintComponent(Graphics g) {
        for (int i = 0; i < 760; i++)
            if (count++;
}
for(int j=0; j<1024; j++)
/
if((i<mb[j][1] && mb[j][1]<2))
    color = new Color(255, 255, 255);
g.setColor(color);
g.fillRect(j, i, 1, 1);
} else if(mb[j][1]==0){
g.setColor(Color.BLACK);
g.fillRect(j, i, 1, 1);
} else if((i<mb[j][1] && mb[j][1]<=MaxIterations){
    //cont = ((double)mb[j][1]-Math.log((Math.log(distance[j][1]))/(Math.log(bound))))/MaxIterations;
    if((double)mb[j][1]/(double)MaxIterations <= 0.05)
        cont = ((double)mb[j][1]/(double)MaxIterations)*0.025;
    else
        cont = (double)mb[j][1]/(double)MaxIterations;
}
else
for(int j=0; j<1024; j++)
    color = new Color((int)(14*(1-cont)*(1-cont)*(1-cont)*(1-cont)*cont*cont*cont),
    (int)(29*(1-cont)*(1-cont)*(1-cont)*cont*cont*cont+255), (int)(9.4*(1-cont)*cont*cont*cont*cont+255), 255); // red color to white
    g.setColor(color);
g.fillRect(j, i, 1, 1);

public void mouseEntered(MouseEvent e){ }
public void mouseExited(MouseEvent e){ }
public void mousePressed(MouseEvent e){ }
public void mouseReleased(MouseEvent e){ }

public class MandelbrotBlaschkeZoom extends JFrame{

    // parts of our window
    public Mandelbrot map = new Mandelbrot1();
    public MandelbrotBlaschkeZoom(){

        // Get x and y dimensions from user
        JTextField x1Field = new JTextField(10);
        JTextField x2Field = new JTextField(10);
        JTextField y1Field = new JTextField(10);
        JTextField y2Field = new JTextField(10);
        JTextField x1Field = new JTextField(10);
        JTextField x2Field = new JTextField(10);
        JTextField y1Field = new JTextField(10);
        JTextField y2Field = new JTextField(10);
        JPanel myPanelx = new JPanel();
        myPanelx.setLayout(new GridLayout(4,2));
        myPanelx.add(new JLabel("Start X Dimension: "));
        myPanelx.add(x1Field);
        myPanelx.add(Box.createHorizontalStrut(15)); // a spacer
        myPanelx.add(new JLabel("End X Dimension: "));
        myPanelx.add(x2Field);
        myPanelx.add(new JLabel("Start Y Dimension: "));
        myPanelx.add(y1Field);
        myPanelx.add(Box.createHorizontalStrut(15)); // a spacer
        myPanelx.add(new JLabel("End Y Dimension: "));
        myPanelx.add(y2Field);
}
```java
myPanelx.add(Box.createHorizontalStrut(15)); // a spacer
myPanelx.add(new JLabel("Start Y Dimension:"));
myPanelx.add(y1Field);
myPanelx.add(Box.createHorizontalStrut(15)); // a spacer
myPanelx.add(new JLabel("End Y Dimension:"));
myPanelx.add(y2Field);
myPanelx.add(Box.createHorizontalStrut(15)); // a spacer
myPanelx.add(new JLabel("Start X Dimension:"));
myPanelx.add(x1Field);
myPanelx.add(Box.createHorizontalStrut(15)); // a spacer
myPanelx.add(new JLabel("End X Dimension:"));
myPanelx.add(x2Field);
myPanelx.add(Box.createHorizontalStrut(15)); // a spacer
myPanelx.add(new JLabel("Start Y value for a:"));
myPanelx.add(y1Field);
myPanelx.add(Box.createHorizontalStrut(15)); // a spacer
myPanelx.add(new JLabel("End Y value for a:"));
myPanelx.add(y2Field);
myPanelx.add(Box.createHorizontalStrut(15)); // a spacer
myPanelx.add(new JLabel("Start X value for a:"));
myPanelx.add(x1Field);
myPanelx.add(Box.createHorizontalStrut(15)); // a spacer
myPanelx.add(new JLabel("End Y value for a:"));
myPanelx.add(y2Field);

int resultx = JOptionPane.showConfirmDialog(null, myPanelx,
   "Please Enter X-Dimensions:", JOptionPane.OK_CANCEL_OPTION);
if (resultx == JOptionPane.OK_OPTION) {
   map.firstXDim = Double.parseDouble(x1Field.getText());
   map.secondXDim = Double.parseDouble(x2Field.getText());
   map.firstYDim = Double.parseDouble(y1Field.getText());
   map.secondYDim = Double.parseDouble(y2Field.getText());
   map.tempfirstXDim = Double.parseDouble(x1Field.getText());
   map.tempsecondXDim = Double.parseDouble(x2Field.getText());
   map.tempfirstYDim = Double.parseDouble(y1Field.getText());
   map.tempsecondYDim = Double.parseDouble(y2Field.getText());
   map.x1 = Double.parseDouble(x1Field.getText());
   map.x2 = Double.parseDouble(x2Field.getText());
   map.y1 = Double.parseDouble(y1Field.getText());
   map.y2 = Double.parseDouble(y2Field.getText());
}

//create stuff
addWindowListener( new Closer());
setTitle("Mandelbrot Set");
setSize(1024, 740);
//content pane
Container set = getContentPane();
set.setLayout( new BorderLayout());
map = new MandelbrotBlaschkoZoom();
map.repaint();
set.add(map, "Center");

//show everything
show();
}

public static void main( String args[])
{
   new MandelbrotBlaschkoZoom();
}
```
Program snippet for Inner Function

for (double i=secondXDim-1; i>=firstXDim; i-=xgap) {
    if (xgap==0) { i-=10; }
    //Move for dimensions in big array
    xDimension++; yDimension++;?
    for (double j=firstYDim; j<=secondYDim; j+=ygap) {
        if (ygap==0) { j+=10; }
        //Dimensions and resetting variables at each step as we move across real line
        xDimension++; yDimension++;?
        x = 0; y = 0; realz = 0; imagz = 0; denz = 0; z0 = j; y0 = 1;
        //Inner(z0) parts
        realz0 = Math.cos(x0)*Math.cosh(y0);
        imagz0 = Math.sin(x0)*Math.sinh(y0);
        for(int k=1; k<MaxIterations; k++) {
            //Calculate new z
            realz = Math.exp((2*x*x+2*y*y-2)/(x*x+y*y-2*x+1))*Math.cos((4*y)/(x*x+y*y-2*x+1));
            imagz = Math.exp((2*x*x+2*y*y-2)/(x*x+y*y-2*x+1))*Math.cos((-4)*y/(x*x+y*y-2*x+1));
            //code for not adding z0
            x = realz+x0;
            y = imagz+y0;
            //x = realz+realz0;
            y = imagz+imagz0; /*
            check = (x*x)+(y*y);
            if ((x0*x0+y0*y0)>1) {
                mDistance[xDimension][yDimension]=check;
                mb[xDimension][yDimension]=1;
                break;
            }
            //Check for colors at fractal points
            if (check>bound) {
                mDistance[xDimension][yDimension]=check;
                mb[xDimension][yDimension]=k;
                break;
            }
        //end if for break in iterations to color/fill mb array
        }
        //if in set color black
        if (check==bound) {
            mb[xDimension][yDimension]=0;
        }
        //end else statement for break to fill mb array
    }
    //end for loop for iterative process to check numbers
}
Program snippet for calculating Inner Product Squared

```java
for (double i=secondYDim; i>=firstYDim; i-=ygap) {
    if (ygap==0) { i-=10; }
    //move for dimensions in big array
    xdimension--;  
ydimension++;  
for (double j=firstXDim; j<=secondXDim; j+=xgap) {
    if (xgap==0) { j+=10; }
    //dimensions and resetting variables at each step as we move across real line
    xdimension++; 
x = 0;  
y = 0;  
realz = 0;  
image = 0;  
den = 0;  
x0 = j;  
y0 = i;
    //inner(z0) parts
    realz0 = Math.cos(x0) + Math.cos(y0);  
image0 = Math.sin(x0) + Math.sin(y0);  
for (int k=1; k<MaxIterations; k++) {
        //calculate new z
        realz = Math.exp((2*x*x+2*y*y-2)/(x*x+y*y-2*x+y)) + Math.cos((6*y)/(x*x+y*y-2*x+y)) + Math.cos((12*y)/(x*x+y*y-2*x+y));  
        image = Math.exp((2*x*x+2*y*y-2)/(x*x+y*y-2*x+y)) + Math.cos((12*y)/(x*x+y*y-2*x+y)) + Math.cos((24*y)/(x*x+y*y-2*x+y));  
        realz2 = realz + realz0;  
        image2 = 3*realz + image;  
        //code for not adding z0
        x = realz2;  
y = image2;  
    }
        //check for colors at fractal points
        if (check<bound) {  
zdistance[xdimension][ydimension] = check;  
                 mb[xdimension][ydimension] = k;  
                 break;  
        }
        //end if for break in iterations to color/fill mb array
        //if in not color black
        if (check<bound) {
                 mb[xdimension][ydimension] = 0;
        }
        //end else statement for break to fill mb array
}
```

 }//end for loop for iterative process to check numbers

52
Program snippet for calculating Cosine Sets

```java
for (double i = secondYDim; i >= firstYDim; i -= ygap) {
    if (ygap < 0) { i += ygap; }
    // move for dimensions in big array
    xDimension = 0;
    yDimension++;
    for (double j = firstXDim; j <= secondXDim; j += xgap) {
        if (xgap < 0) { j += xgap; }
        // dimensions and resetting variables at each step as we move across real line
        xDimension++;
        x = 0;
        y = 0;
        realz = 0;
        imagz = 0;
        den = 0;
        x0 = j;
        y0 = i;
        // cos(z0) parts with functions
        realz0 = calcCosSeries(x0) * calcCoshSeries(y0);
        imagz0 = calcSinSeries(x0) * calcSinhSeries(-y0);
        // cos(z0) parts
        realz0 = Math.cos(x0) * Math.cosh(y0);
        imagz0 = Math.sin(x0) * Math.sinh(y0);
        for (int k = 1; k <= MaxIterations; k++) {
            // calculate new z for iteration functions
            realz = 0.5 * ((calcExpSeries(-y0) + calcExpSeries(y0)) * calcCosSeries(x0));
            imagz = 0.5 * ((calcExpSeries(-y0) - calcExpSeries(y0)) * calcSinSeries(x0));
            // calculate new z
            realz = 0.5 * ((Math.exp(y0) + Math.exp(-y0)) * Math.cos(x0));
            imagz = 0.5 * ((Math.exp(-y0) - Math.exp(y0)) * Math.sin(x0));
            // code for not adding z0
            x = realz + x0;
            y = imagz + y0;
            // x = realz + realz0;
            y = imagz + imagz0; // check = (x*x) + (y*y);
            // check for colors at fractal points
            if (check < bound) {
                zDistance[xDimension][yDimension] = check;
                mb[xDimension][yDimension] = k;
                break;
            }
        } // end if for break in iterations to color/fill mb array
        // if in set color black
        if (check < bound) {
            mb[xDimension][yDimension] = 0;
        }
    }
}
```
Program snippet for calculating Sine Sets

```java
for (double i = secondYDim; i >= firstYDim; i -= ygap) {
    if (ygap == 0) { i -= 10; }
    // move for dimensions in big array
    xdimension = 0;
    ydimension = i;
    for (double j = firstXDim; j <= secondXDim; j += xgap) {
        if (xgap == 0) { j += 10; }
        // dimensions and resetting variables at each step as we move across real line
        xdimension++;
        x = 0;
        y = 0;
        realz = 0;
        imagz = 0;
        den = 0;
        x0 = j;
        y0 = i;
        // sin(x0) parts
        realiz0 = 0.5 * ((Math.exp(-y0) + Math.exp(y0)) * (double) Math.sin(x0));
        imagz0 = 0.5 * ((Math.exp(y0) - Math.exp(-y0)) * (double) Math.cos(x0));
        for (int k = 1; k <= MaxIterations; k++) {
            // calculate new z for iteration
            realiz = 0.5 * ((Math.exp(-y) + Math.exp(y)) * (double) Math.sin(x));
            imagz = 0.5 * ((Math.exp(y) - Math.exp(-y)) * (double) Math.cos(x));
            // x = realz + realiz;
            // y = imagz + imagz0;
            // code for not adding x0
            if (x == 0 && y == 0) {
                x = 0;
                y = 0;
            } else {
                x = realiz + x0;
                y = imagz + y0;
            }
            check = (x * x) + (y * y);
            // check for colors at fractal points
            if (check > bound) {
                mb[xdimension][ydimension] = check;
                mb[xdimension][ydimension] = k;
                break;
            }
        }
    }
}
// if in set color black
if (check <= bound) {
    mb[xdimension][ydimension] = 0;
}
```
Program snippet for calculating $e^z$ Sets

for(double i=secondYDim; i>=firstYDim; i--) {
  for(double j=firstXDim; j<=secondXDim; j++) {
    if((ygap'-O) { i--; }
    //move for dimensions in big array
    xdimension=0;
    ydimension++;  
    for(double j=firstXDim; j<=secondXDim; j++) {
      if((xgap-0) { j--; }
      //dimensions and resetting variables at each stop as we move across real line
      xdimension++;
      x = 0;
      y = 0;
      realz = 0;
      imagz = 0;
      den = 0;
      x0 = j;
      y0 = i;
      //cos(z) parts
      for(int k=1; k<=MaxIterations; k++){
        //calculate new z for iteration
        //code for $e^{-z^2 + z}$
        realz = (Math.exp(y*y-x*x)*Math.cos(-2*x*y));
        imagz = (Math.exp(y*y-x*x)*Math.sin(-2*x*y));
        //code for $e^z + z$
        realz = (Math.exp(z)) * Math.cos(y));
        imagz = (Math.exp(z)) * Math.sin(y));
        x = realz+x0;
        y = imagz+y0;
        check = (x*x)+(y*y);
        //check for colors at fractal points
        if(check<bound){
          zdistance[xdimension][ydimension]=check;
          mb[xdimension][ydimension]=k;
          break;
        } //end if for break in iterations to color/fill mb array
        //if in set color black
        if(check<bound){
          mb[xdimension][ydimension]=9;
        }
      } //end else statement for break to fill mb array
for loop for iterative process to check numbers

for loop to go through x dimensions

for loop to go through y dimensions
B Interesting Pictures

Figure 23: A zoom into “Seahorse Valley” in $S_{1+1i}$

Figure 24: A zoom into an interesting part of an $S_a$ set.
Figure 25: Part of a movie of the $S_a$ sets.
Figure 26: A zoom into an $S_\alpha$ set.
Figure 27: A zoom into parts of the Mandelbrot set.

Figure 28: A zoom into "Seahorse Valley" of the Mandelbrot set.

Figure 29: A zoom into "Elephant Valley" of the Mandelbrot set.
Figure 30: Part of the $S_{-1+1i}$ set.

Figure 31: Part of the $S_{-1+1i}$ set.
Figure 32: Part of the $S_{-1+.1i}$ set.

Figure 33: Part of the $S_{-1+.1i}$ set.
Figure 34: Part of the $S_{2-.3i}$ set.

Figure 35: Part of the $S_{2-.3i}$ set.
Figure 36: Part of the $S_{0.05-0.5i}$ set.

Figure 37: Part of the $S_{0.05-0.5i}$ set.
Figure 38: Part of the $S_{05-05i}$ set.

Figure 39: Part of the $S_{05-05i}$ set.
Figure 40: Part of the $S_{22+0i}$ set.

Figure 41: Part of the $S_{22+0i}$ set.
Figure 42: Part of the $S_{22+0i}$ set.

Figure 43: Part of the $S_{22+0i}$ set.
References


