College of Saint Benedict and Saint John's University
DigitalCommons@CSB/SJU

Honors Theses, 1963-2015 Honors Program

2013

Visualizing Chaos

Margaret Peterson
College of Saint Benedict/Saint John's University

Follow this and additional works at: https://digitalcommons.csbsju.edu/honors_theses

Part of the Mathematics Commons

Recommended Citation

Peterson, Margaret, "Visualizing Chaos" (2013). *Honors Theses, 1963-2015*. 20.
https://digitalcommons.csbsju.edu/honors_theses/20

This Thesis is brought to you for free and open access by DigitalCommons@CSB/SJU. It has been accepted for inclusion in Honors Theses, 1963-2015 by an authorized administrator of DigitalCommons@CSB/SJU. For more information, please contact digitalcommons@csbsju.edu.

VISUALIZING CHAOS

AN HONORS THESIS

College of St. Benedict/St. John's University

In Partial Fulfillment

of the Requirements for Distinction

In the Department of Mathematics

by

Margaret Peterson

May, 2013

Project Title: Visualizing Chaos

Approved by:
Robert Hesse

Robert Hesse
Associate Professor, Department of Mathematics
Kris A Nairn

Kris A Nairn
Associate Professor, Department of Mathematics
Bret Benesh

Bret Benesh
Assistant Professor, Department of Mathematics
Robert Hesse

Robert Hesse
Associate Professor, Department of Mathematics
Anthony Cunningham

Anthony Cunningham
Director, Honors Thesis Program

Date: April 26, 2013.

Contents

List of Figures 4

INTRODUCTION 7
Root Finding Methods 7
The Hansen-Patrick Root Finding Method 10
Varying α 11

1.**Creating the program** 13

2.**Working with the final program** 18
I. *Sphere Distortion* 18

3.**Sign choice in the denominator** 21
I. *The Plus/Minus choice* 21
II. *A sign choice analysis* 26

CONCLUSION 27

4.**Future work** 28
I. *Imaginary values for* α 28
II. *Special cases with Laguerre* α 31

REFERENCES 33

APPENDIX 1: Hansen-Patrick calculations in Java 34

APPENDIX 2: Complex numbers in Java 48

APPENDIX 3: Complex polynomials in Java 52

APPENDIX 4: Guest user interface in Java 56

APPENDIX 6: Sign choice in Java 70

APPENDIX 7: Java template 81

APPENDIX 8: Build in MATLAB 82

APPENDIX 9: Scanner in MATLAB 84

APPENDIX 10: Magnitude method plot in MATLAB 85

APPENDIX 11: Argument method plot in MATLAB 87

APPENDIX 12: Dot product method plot in MATLAB 89

List of Figures

INTRODUCTION 7
Figure 1 7
Figure 2 8
Figure 3 8
Figure 4 9

1.**Creating the program** 13
Figure 1.1 14
Figure 1.2 15
Figure 1.3 15
Figure 1.4 15
Figure 1.5 16
Figure 1.6 16
Figure 1.7 16
Figure 1.8 17

2.**Working with the final program** 18
Figure 2.1 18
Figure 2.2 18
Figure 2.3 19
Figure 2.4 19
Figure 2.5 20
Figure 2.6 20

3.**Sign choice in the denominator** 21
Figure 3.1 22
Figure 3.2 22
Figure 3.3 23
Figure 3.4 24
Figure 3.5 24
Figure 3.6 25
Figure 3.7 25

4.**Future work** 29
Figure 4.1 28
Figure 4.2 28
Figure 4.3 29
Figure 4.4 29
Figure 4.5 29
Figure 4.6 30
Figure 4.7 30
Figure 4.8 30
Figure 4.9 31
Figure 4.10 31

Acknowledgements

I would like to acknowledge my thesis advisor, Dr. Robert Hesse, for leading me in this research and lending me a generous amount of time and assistance. I would also like to acknowledge my colleague, Preston Hardy, who worked with me on the programs that were necessary for the content of this paper. I am also grateful for my co-readers, Dr. Kris Nairn and my advisor, Dr. Bret Benesh, for assisting me in my thesis writing. Finally, I would like to acknowledge the College of St. Benedict/St. John's University for their support of undergraduate research opportunities.

Abstract. Chaos is typically visualized on an infinite 2D plane. By using stereographic projection, my colleague Preston Hardy and I utilized a third dimension to plot basin maps of iterative root finding methods on a subset of the complex plane onto a sphere. These spheres are then shaded in accordance to the speed in which the particular initial point converges, creating images that can be used to visualize all basins of attraction on the complex plane on a finite 3D surface. The resulting images are used to explore efficiency of root finding methods as well as evaluating the choice of addition or subtraction in the denominator of the Hansen-Patrick root finding method. There are many theories suggesting the sign choice for positive alpha values; however, in the case of a negative alpha value, these theories do not hold. Using programs based off of those developed by Andrew Nicklawsky and Dr. Robert Hesse, we developed rules to dictate this choice between addition and subtraction in order to maximize the speed of convergence for negative and imaginary alpha values.

INTRODUCTION

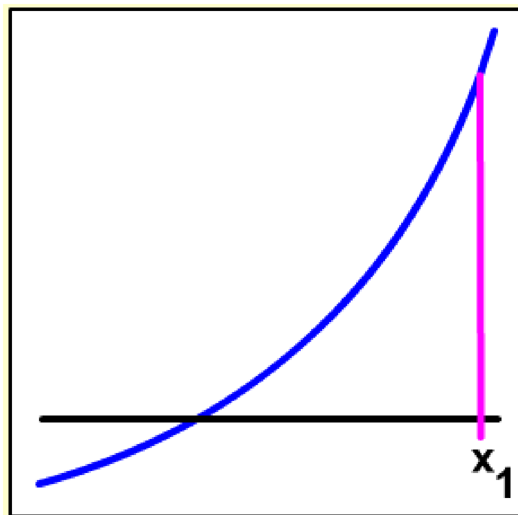
Root finding methods have applications in a wide range of fields. From iPods to weather patterns, root finding methods are crucial to technological advances and mapping natural processes. In this paper, we consider the global dynamics of the Hansen-Patrick family of root finding methods listed below (Eq.1).

$$z_{n+1} = z_n - \frac{(\alpha+1)f(z)}{\alpha f'(z) \pm \sqrt{(f'(z))^2 - (\alpha+1)f(z)f''(z)}} \quad (1)$$

Here, z_n is the initial starting point, $f(z)$ is an input polynomial, and α is an input variable that allows various pre-established root finding methods to be grouped into a single family. In 2011, Andrew Nicklawsky created a MATLAB program to visualize the basins that initial points converge to by stereographically mapping the complex plane onto a sphere. This projection allows the user to see all the basins of attraction at one time. Using Nicklawsky's program as a starting point, Preston Hardy and I created a method for choosing between plus and minus in the denominator of the Hansen-Patrick root-finding method for all α. This method resulted in converging and continuously varying images.

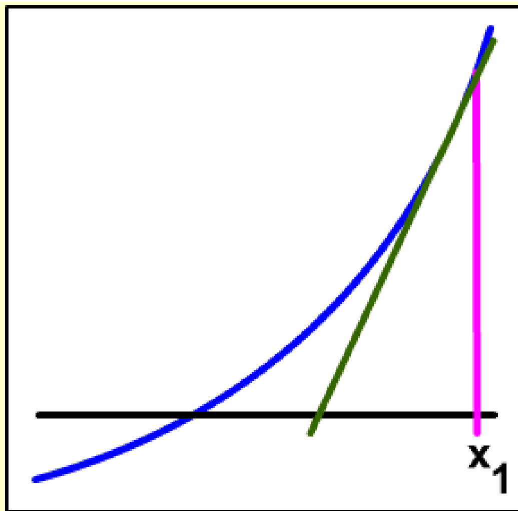
Root Finding Methods

Consider the polynomial $f(x)$ mapped on the xy-plane. The root of that function is located at the x-intercept, that is, some value x such that $f(x) = 0$. Now consider some initial guess as to what the value of that root, x, may be, and call that guess, x_1. Below is a graphical representation of this situation (Fig. 1).



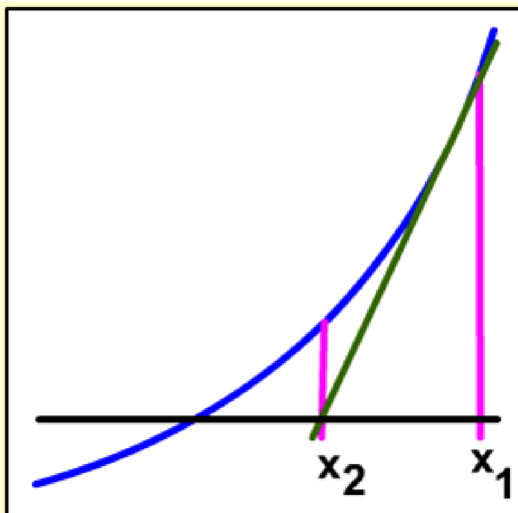
(Figure 1)

The blue line in Figure 1 is the plot of the function $f(x)$. If a line is drawn straight up from the point x_1, it intersects the plot of $f(x)$ at the point $(x_1, f(x_1))$. Consider the tangent line to $f(x)$ at this point.



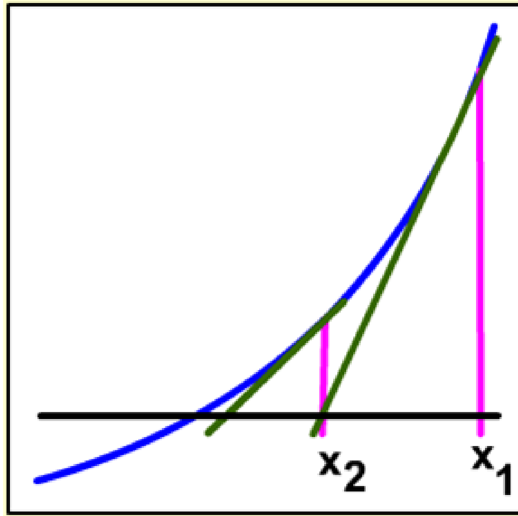
(Figure 2)

It is possible to see in Figure 2 that the tangent line also intersects the x-axis at some point, call x_2. Just as in the previous case, a line is drawn straight up from x_2, and it will intersect the function $f(x)$ at the point $(x_2, f(x_2))$.



(Figure 3)

Once again, the tangent line drawn at $(x_2, f(x_2))$ intersects the x-axis at some point x_3



(Figure 4)

Each of these new x-intercepts, x_{n+1}, is closer to the root than the previous, x_n. In order to derive a formula for this new point, x_{n+1}, it is necessary to consider the formula for the tangent line at the previous point, x_n, listed below (Eq. 2).

$$y - f(x_n) = f'(x_n)(x - x_n) \tag{2}$$

By setting $y = 0$ and solving for x, the equation returns the x-intercept, x_{n+1}, of the tangent line (Eq. 3).

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{3}$$

Equation 3 is the classic iterative root finding method known as Newton's Method. Although Newton's Method is the most common root finding method, it is not the only method. Listed below are some other standard root finding methods, which can be derived from analytic methods.

Ostrowski's Method

$$x_{n+1} = x_n - \frac{f(x_n)}{\pm\sqrt{f'(x_n)^2 - f(x_n)f''(x_n)}}$$

Halley's Method

$$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2f'(x_n)^2 - f(x_n)f''(x_n)}$$

Euler's Method

$$x_{n+1} = x_n - \frac{2f(x_n)}{f'(x_n) \pm \sqrt{f'(x_n)^2 - 2f(x_n)f''(x_n)}}$$

The Hansen-Patrick Root Finding Method

In 1976, Eldon Hansen and Merrell Patrick created a way to group these previously established root finding methods into one family by introducing a new variable, α and is written as the equation below (Eq. 1),

$$z_0 = z - \frac{(\alpha+1)f}{\alpha f' \pm \sqrt{(f')^2 - (\alpha+1)ff''}} \tag{1}$$

where $z_0 \in \mathbb{C}$ is the new point, $z \in \mathbb{C}$ is the initial point, and f is some differentiable polynomial $f(z)$ of order n. By varying the parameter α, it is possible to obtain the pre-established popular methods for finding roots. To understand the origin of the parameter α, consider a function f with a root at r written as:

$f = (z - r)g(z)$

where $g(z)$ is some function such that $g(r) \neq 0$.

Hansen-Patrick's argument is the following.

Let $s_1 = \frac{f'}{f}$

Equivalently,

$s_1 = \frac{1}{\epsilon} + \sigma_1$

where $\epsilon = (z - r)$ and $\sigma_1 = \frac{g'}{g}$

Now consider s_2 such that $s_2 = -s_1' = \frac{(f')^2 - ff''}{f^2} = \frac{1}{\epsilon^2} + (\sigma_2)$.

where $\sigma_2 = -\sigma_1 = \frac{(g')^2 - gg''}{g^2}$.

This is a root finding method, so the end result should be an approximation for r, or equivalently, ϵ. We have two relations, s_1 and s_2, to assist in doing so, if a combination of these terms occurs quadratically. Assume A, B, C to be arbitrary functions and note the exact quadratic function

$$A(s_1 - \frac{1}{\epsilon} - \sigma_1) + B(s_1 - \frac{1}{\epsilon} - \sigma_1)^2 + C(s_2 - \frac{1}{\epsilon^2} - \sigma_2) = 0$$

This can be written as an explicit quadratic.

$\frac{B-C}{\epsilon^2} - \frac{1}{\epsilon}(A + 2B(s_1 - \sigma_1)) + s_1(A - 2B\sigma_1) + Bs_1 + Cs_1 - A\sigma_1 + B\sigma_1^2 - C\sigma_2 = 0$

In an attempt to simplify this explicit quadratic, select A such that,

$A = B\sigma_1 - C\frac{\sigma_2}{\sigma_1}$

and choose B such that,

$B = -C\frac{\sigma_2}{\sigma_1}$

Plugging this into the explicit quadratic, we obtain:

$\frac{-1}{\epsilon^2}(\frac{\sigma_2}{\sigma_1^2}+1)+\frac{2s_1}{\epsilon}\frac{\sigma_2}{\sigma_1^2}-s_1^2\frac{\sigma_2}{\sigma_1^2}+s_2=0.$

In order to simplify even further, create a new variable $\sigma=\frac{\sigma_2}{\sigma_1^2}$.

When replacing σ by some fixed parameter α, there is an error δ such that $\delta=\sigma-\alpha$.

It is possible to account for this error by replacing $\epsilon=z-r$ by $z-z_0$, giving

$-\frac{\alpha+1}{z-z_0}^2+\frac{2s_1\alpha}{z-z_0}-s_1^2\alpha+s_2=0$

Solving for the root approximation, z_0, we get

$z_0=z-\frac{\alpha+1}{\alpha s_1\pm\sqrt{(\alpha+1)s_2-\alpha s_1^2}}$

Plugging in the equations for s_1 and s_2, we obtain

$$z_0=z-\frac{(\alpha+1)f}{\alpha f'\pm\sqrt{(f')^2-(\alpha+1)ff''}}$$

which is the the Hansen-Patrick root finding method.

Varying α

NEWTON'S METHOD

Consider the limit as α goes to infinity of the Hansen-Patrick root finding method:

$\lim_{\alpha\to\infty} z-\frac{(\alpha+1)f}{\alpha f'\pm\sqrt{(f')^2-(\alpha+1)ff''}}$

or equivalently,

$z-f\lim_{\alpha\to\infty}\frac{(\alpha+1)}{\alpha f'\pm\sqrt{(f')^2-(\alpha+1)ff''}}=z-f\frac{1}{f'}.$

This gives us Newton's Method for finding roots:

$z_0=z-\frac{f}{f'}.$

HALLEY'S METHOD

Consider the case where $\alpha=-1$. This appears to be undefined, as it results in a division of zero. It is possible to remedy this, however, by manipulating the Hansen-Patrick root finding method. Multiplying by both the numerator and denominator by the complex conjugate of the denominator gives the equivalent formula:

$z_0=z-\frac{\alpha f'\pm\sqrt{(f')^2-(\alpha+1)ff''}}{(\alpha-1)(f')^2+ff''}$

Plugging in $\alpha=-1$ gives the equation:

$z_0=z-\frac{f}{f'-\frac{ff''}{2f'}}$

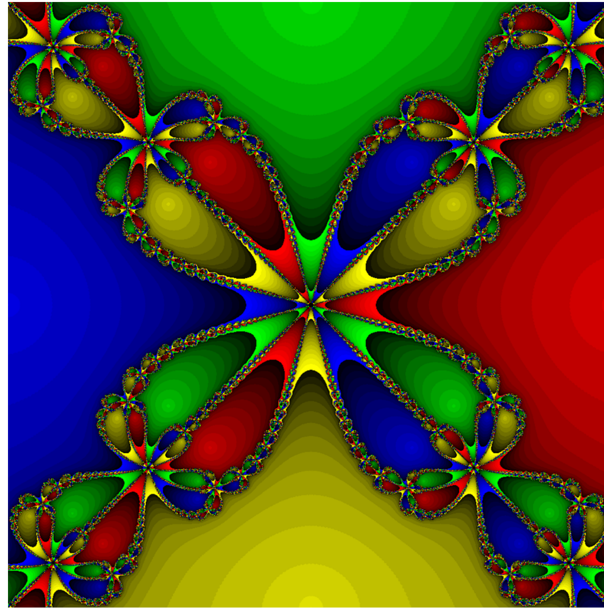
This equation is Halley's method, and it is beneficial because it contains no square root. Of all the pre-determined methods for finding roots, Halley's method is the only method that requires a negative value for α. Some other pre-determined methods and their respective alpha values include, $\alpha=0$, which gives Ostrowski's Method, $\alpha=\frac{1}{n-1}$ (recall n is the degree of the polynomial), which gives Laguerre's Method, and $\alpha=1$, which gives Euler's Method.

Given the different root-finding methods, we want to be able to visualize when an initial point $z \in \mathbb{C} \cup \infty$ will converge to a root for a given polynomial and fixed value of α. This can be accomplished by creating a program to choose the correct option of plus/minus in the denominator of the Hansen-Patrick root finding method. If the program chooses the correct option of plus/minus, we expect that while a polynomial f is held fixed as α varies continuously through positive and negative values, the boundaries of the basins of attraction will change continuously on the spheres returned by the program.

We modified Nicklawsky's project, adjusting for conclusions that were made while working with his original work. This paper explains how we improved on Nicklawsky's original program and the general rules that we developed in creating our new program used to dictate the option of plus and minus. This choice depends on the sign of the value of α input by the user. It also includes a brief analysis of the specific polynomial $x^2 - 1$ that helps to bring clarity to these rules.

1. Creating the program

Nicklawsky's program was a great base for exploring the Hansen-Patrick root finding method. The input parameters included a polynomial $f(x)$ and a value for α. It then returned a sphere colored accordingly to the number of roots found by the program. These areas of colors converging to a specific root are called **basins of attraction**. These colors were then shaded corresponding to the number of iterations run by the program until the initial point converged, meaning it came within a distance of a root of $f(x)$ specified in in the program. If a specific initial starting point did not converge to a root before the maximum number of iterations specified in the program, it was considered to be divergent, and was colored black. Consider the example of a basin map plotted on the complex plane below. Each area of color is a basin of attraction. All initial points in the green basin of attraction are converging to the same root r_1, all the initial points in the yellow basin of attraction are converging to the the same root, r_2, etc. Consider the large green area at the top of the image. As one moves from the top of the image towards the center, the color green gets darker until it reaches black. This indicates that it takes more iterations to converge to a root as z moves towards the center of the image. The initial points colored black are considered by the program to be diverging.



Recall that the spheres returned by the program are created using stereographic projection. Because stereographic projection maps the unit circle to the equator of the sphere, polynomials with small roots create basins of attraction that are easy to visualize, as such, polynomials with small roots will be used as examples in this paper. By examining the shading of the sphere, it is possible to determine the efficiency of the Hansen-Patrick root finding method corresponding to the value of α.

The images produced by Nicklawsky's program were different from the images expected, and initial starting points that converge when calculated by hand were displayed as diverging in the resulting spheres. Also, as α varied continuously, the boundaries of the basins of attraction on the sphere did not vary continuously. This indicated an issue with the choice between plus and minus in the denominator of the Hansen-Patrick root finding method.

The well-known Hansen-Patrick method for finding roots as a function of the initial point z and polynomial f is listed below (Eq. 1.3).

$$z_0 = z - \frac{(\alpha+1)f}{\alpha f' \pm \sqrt{(f')^2 - (\alpha+1)ff''}} \tag{1.1}$$

It can be written alternatively as: (Eq. 1.2)

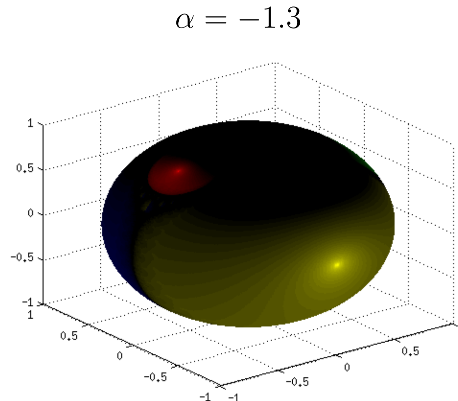
$$z - z_0 = \frac{(\alpha+1)f}{\alpha f' \pm \sqrt{(f')^2 - (\alpha+1)ff''}} \tag{1.2}$$

We are working with algorithmic methods for determining convergence to a root and we want to minimize the difference between z and z_0. Upon analyzing Equation 1.2, it is possible to see that by maximizing the denominator, the difference between our original point z and our new point z_0 is minimized. This method of selecting plus/minus is successful and leads to convergence and continuously varying images for $\alpha > 0$; however, upon testing a variety of polynomials using this method of sign choice with $\alpha = 0$ and $\alpha < 0$, the surface of the spheres are still mostly colored black, meaning that the initial points are not converging to a root.

The issue with $\alpha = 0$ is clear, as the program is written to choose the choice in the denominator that results in the largest magnitude; however, when $\alpha = 0$, the magnitude is the same for both the plus and the minus option. This can be remedied by writing the program to always choose the option of plus when the magnitudes of both options are equal. Using this method, the spheres produced indicate convergence of initial points to a root, as they are brightly colored, and continually varying basin map boundaries for $\alpha = 0$.

Unfortunately, the solution for obtaining images that vary continuously with negative values of α is not so obvious. After an extensive paper hunt, it is apparent that there is no previous work done with negative values for α, outside of the special case of Halley's method.

The images obtained by negative α using this method of choice between plus/minus do converge to a root, but it takes them a large amount of iterations to do so, resulting in dark images that appear as though a light is shining out from the inside of the sphere at the location of the roots, as shown in Figure 1.1 below. As α tends towards negative infinity, the images begin to lighten up, indicating a faster rate of initial points converging to a root, until at very large values of negative α, the images mirror those of Newton's Method.



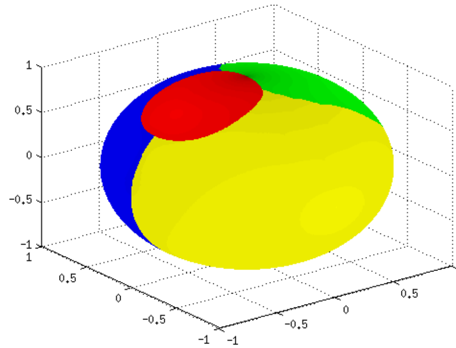
(Figure 1.1)

Because the program runs consistently and continuously for positive values of α, sign choice for negative α is considered. For positive values of α, choosing the denominator with the largest magnitude leads to convergence and continuity. So for the opposite case, or negative α, the program chooses the opposite choice in the denominator, or that which minimized the denominator's magnitude. Below are the images produced using this method and a polynomial of order four, listed below (Eq. 1.3).

$$f(x) = x^4 + x^3 - 5x^2 + x - 6 \tag{1.3}$$

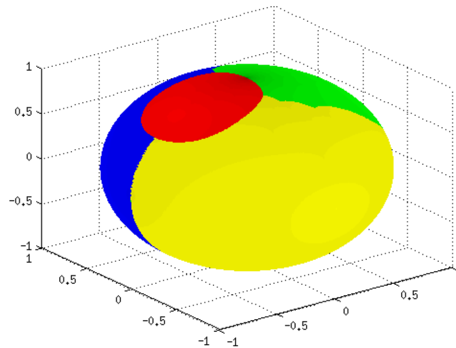
The value of α starts at $\alpha = 1.5$ and decreases by steps of 0.25 until it reaches a final value of $\alpha = -3$.

α=1.5



(Figure 1.2)

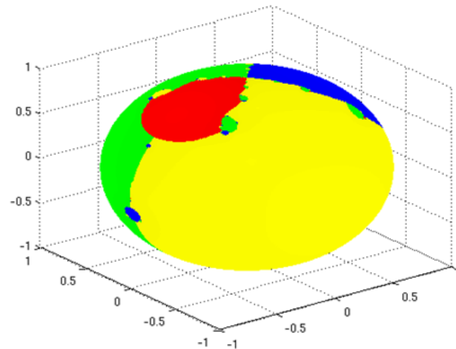
α=0.75



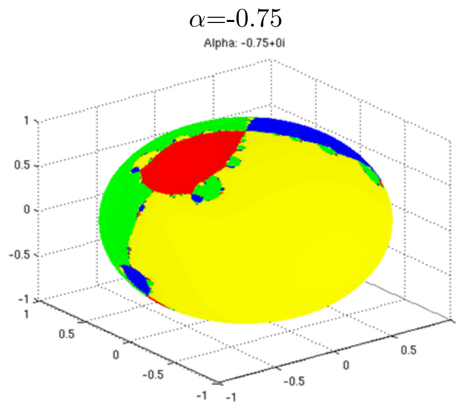
(Figure 1.3)

α=0

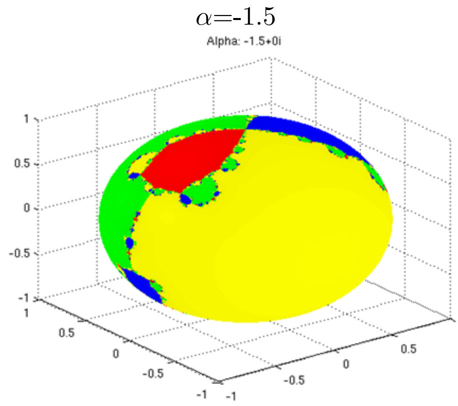
Alpha: 0+0i



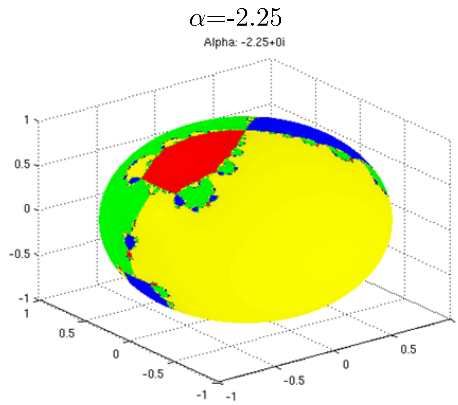
(Figure 1.4)



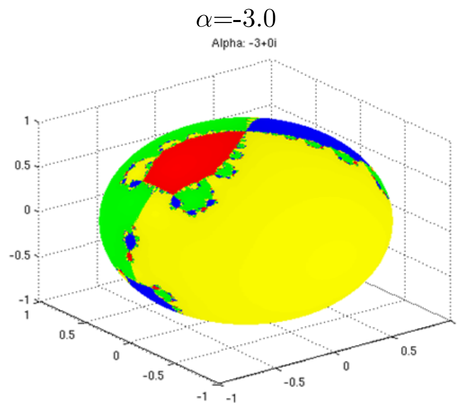
(Figure 1.5)



(Figure 1.6)



(Figure 1.7)



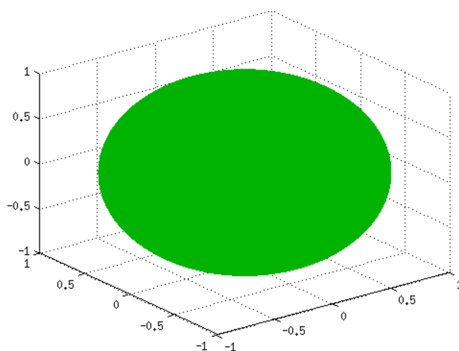
(Figure 1.8)

This method of choosing plus/minus for negative α is unexpected; however, the spheres indicates that the initial points are converging and the boundaries of the basin maps vary continuously with all real values of α.

2. Working with the final program

I. *Sphere Distortion*

On thing that is important to note when working with this program is the distortion that arises from mapping a plane onto a sphere. Because it maps an unbounded surface to a bounded surface, plots can be misleading. In many cases, an infinite area of convergence to a root may appear very small, or in some cases, may disappear entirely. Consider the example below.

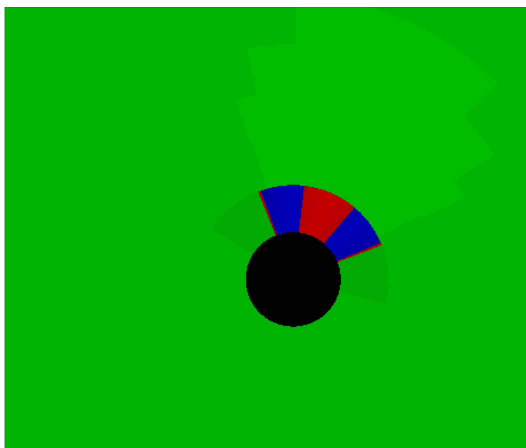


(Figure 2.1)

This sphere is showing us a map of the polynomial listed below (Eq. 2.1)

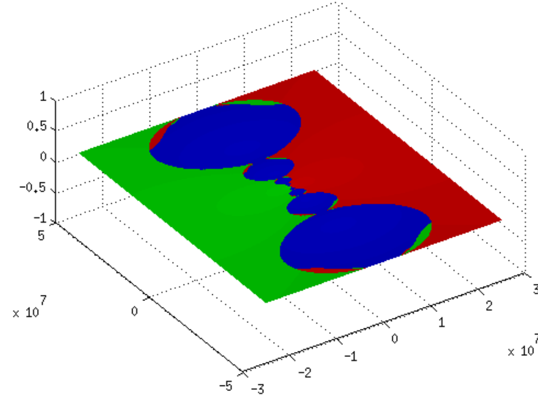
$$f(x) = x^3 - 3003x^2 + 3006002x - 1003002000 \tag{2.1}$$

Equation 2.1 has roots at $1000, 1001, 1002$. Because these roots are so large, they are almost impossible to see unless incredibly close to the north pole (Fig. 2.2). Although the area of convergence is somewhat similar for all of the images, mapping it to a sphere makes it appear as though almost all initial points are converging to the root 1000 as opposed to the other two roots.



(Figure 2.2)

Looking at it from a planar view makes it possible to get a more accurate representation of the behavior of the root convergence (See Fig 2.3).

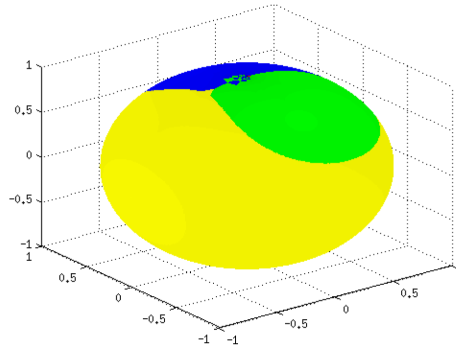


(Figure 2.3)

This is also evident when looking at polynomials with an isolated root. Take, for example, the polynomial listed below (Eq 2.2).

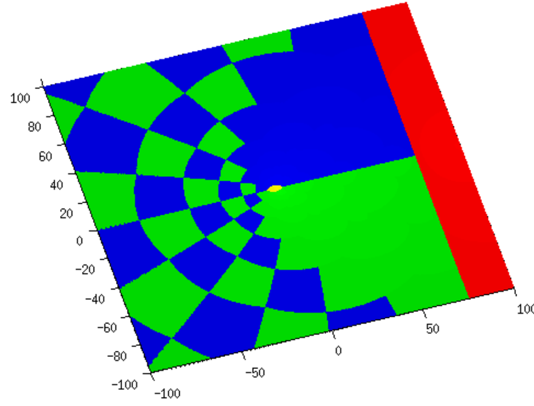
$$f(x) = x^4 - 99x^3 - 91x^2 - 891x - 900 \tag{2.2}$$

Equation 2.2 has roots at $-1, -3i, 3i, 100$. Below (Fig.2.4) is the image of Equation 2.2 with $\alpha = 0.5$.



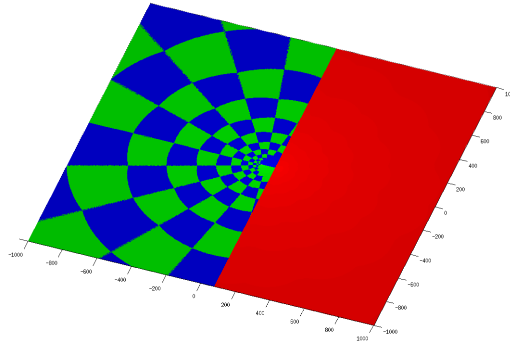
(Figure 2.4)

Like with polynomials with large roots, the images produced by polynomials with isolated roots can be misleading. Below (Fig. 2.5) is the plane that is mapped onto the sphere in Figure 2.4.



(Figure 2.5)

On this plane, with x and y ranging from -100 to 100, the area converging to -1, or the yellow area, appears small in comparison to the other colors. On the sphere however, the color yellow dominates a majority of the surface area, directly contrasting the image on the plane. When zoomed out further, the yellow area does not appear at all (Fig. 2.6).



(Figure 2.6)

We altered the code to incorporate the option for a more appropriate radius in order to reduce some of the natural distortion.

3. Sign choice in the denominator

I. *The Plus/Minus Choice*

The program runs successfully for all choices of real α; however, to do so, it must run two separate cases: one for positive α, and one for negative α. Because the images varied continuously with all real α, a single method for choosing plus/minus the denominator for all α seems plausible. Running two cases sacrifices both efficiency of program speed and convergence time.

In order to get a better idea of how the sign of α affects the sign choice in the denominator of the Hansen-Patrick method, consider a new program in MATLAB, called *Magnitude*. This program runs the Hansen-Patrick root finding method for the convenient polynomial $x^3 - 1$, and it uses the same method for selecting plus/minus as our initial program in Java. A pseudo-code for this method is stated below.

If α is positive

Choose denominator with the largest magnitude.

Else if α is negative

Choose denominator with the smallest magnitude.

The program plots each new point generated by the Hansen-Patrick method and it connects it to the previous point with a line indicting sign choice: blue indicates plus and red indicates minus.

There are two widely used methods for determining the sign in the denominator for positive values of α. The first of these methods is that which is described above. The second method aims to minimize difference between the arguments of the complex numbers z_0 and z. To compare these two methods, a second program is required that mirrors the first program, but uses this argument method, called *Argument*. A pseudo-code for this method is stated below.

If α is positive

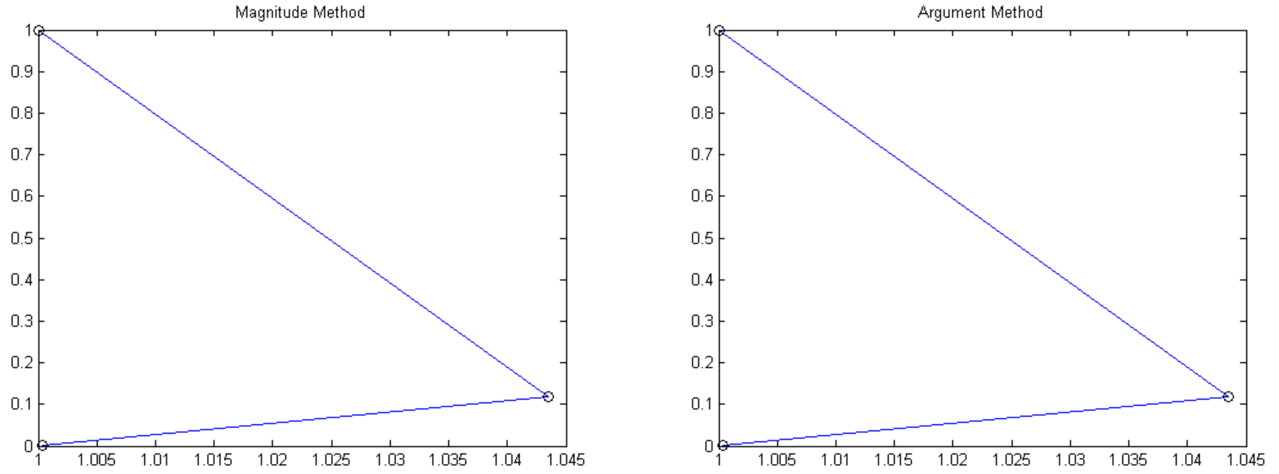
Choose sign that minimizes the difference between the arguments of z and z_0.

Else if α is negative

Choose sign that maximizes the difference between the arguments of z and z_0.

For some values of α, the images for both methods are the same (Fig. 3.1).

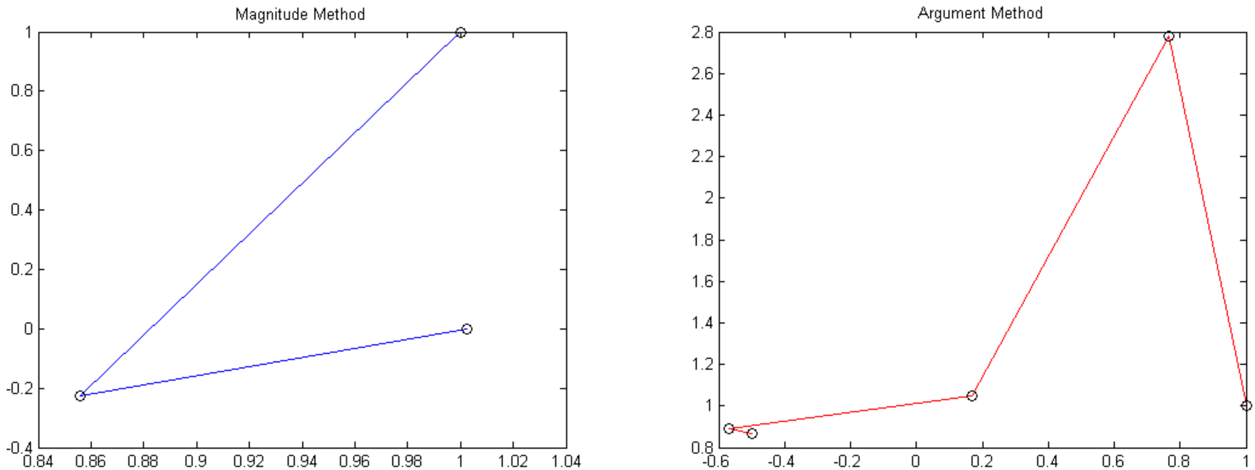
For the case $z = 1 + i$ and $\alpha = 0.75$



(Figure 3.1)

The blue lines in the plot above indicate that both sign choice methods are choosing plus in the denominator, and they connect the previous point to the next iteration. Once the program reaches a point a distance 0.01 away from a root, it stops iterating. Although the images for all values of $\alpha > 0$ converge, Figure 3.2 demonstrates how magnitude method is more efficient.

For the case $z = 1 + i$ and $\alpha = 0.1$



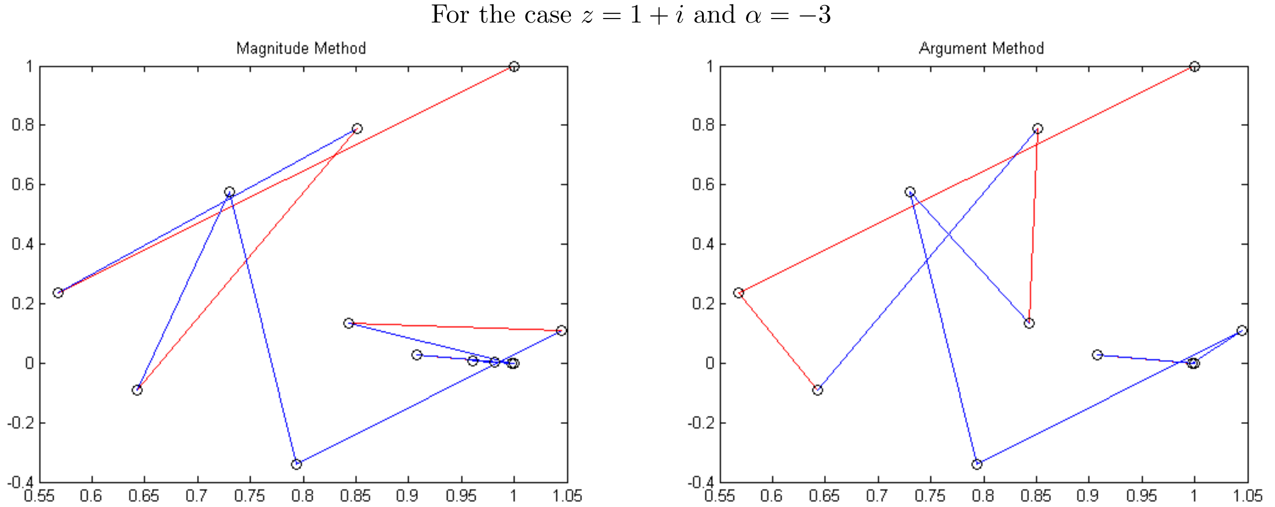
(Figure 3.2)

In this case, the two methods choose different sign choices in the denominator, as well as converge to different roots.

Unfortunately, using the argument method required two cases as well: one for positive α and one for negative α. For positive α, it minimized the difference of arguments between z_0 and z, and for negative α, it maximized the difference of arguments between z_0 and z. Unlike the case when $\alpha > 0$, when $\alpha < 0$, the method of maximizing the difference in argument was slightly more efficient than that of minimizing the

magnitude of the denominator. The figure below illustrates the difference in the paths of convergence (Fig. 3.3).

For the case $z = 1 + i$ and $\alpha = -3$



(Figure 3.3)

Consider a third MATLAB program that uses Nicklawsky's original method for the choice between plus/minus, called *Alpha Dot*. This method was originally interpreted as:

For positive α

If $\alpha f' \cdot \sqrt{(f')^2 - (\alpha+1)ff''} > 0$

choose plus.

Else if $\alpha f' \cdot \sqrt{(f')^2 - (\alpha+1)ff''} < 0$

choose minus.

For negative α

If $\alpha f' \cdot \sqrt{(f')^2 - (\alpha+1)ff''} > 0$

choose minus.

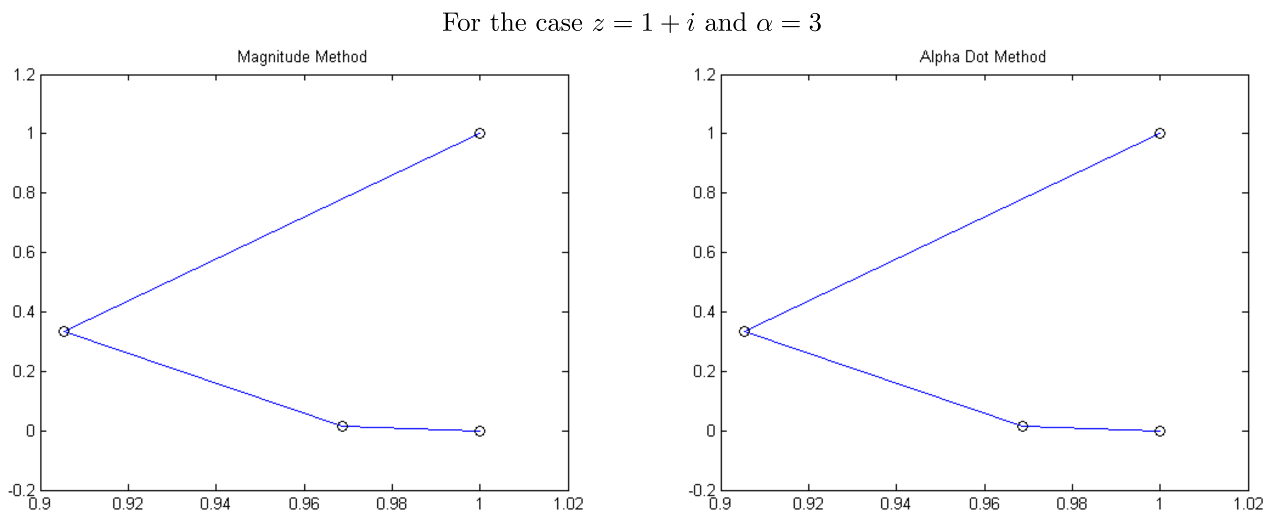
Else if $\alpha f' \cdot \sqrt{(f')^2 - (\alpha+1)ff''} < 0$

choose plus.

That is, if the sign of $\alpha f'$ and the sign of $\sqrt{(f')^2 - (\alpha+1)ff''}$ are the same, choose plus, and if they are different, choose minus. It quickly becomes apparent that this method is the same as the method used in

Magnitude. This is illustrated in the identical images below (Fig. 3.4).

For the case $z = 1 + i$ and $\alpha = 3$



(Figure 3.4)

By removing α from the first term in the *Magnitude Dot* equation, the sign of α takes care of the two separate cases. Consider a final program; we call this program *Dot*. The pseudo-code for the choice between plus/minus in the denominator is listed below.

If $f' \cdot \sqrt{(f')^2 - (\alpha + 1)ff''} > 0$

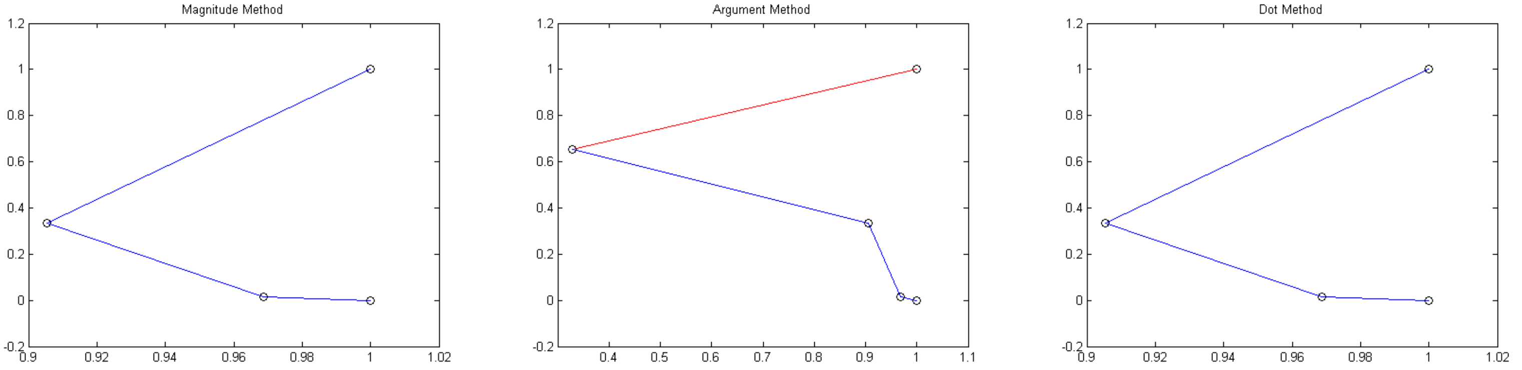
choose plus.

Else if $f' \cdot \sqrt{(f')^2 - (\alpha + 1)ff''} < 0$

choose minus.

The resulting images not only indicate convergence for all α, but the efficiency of convergence of *Dot* is superior to the other methods. For positive α, all three methods take a similar path to the root, 1 (Fig. 3.5).

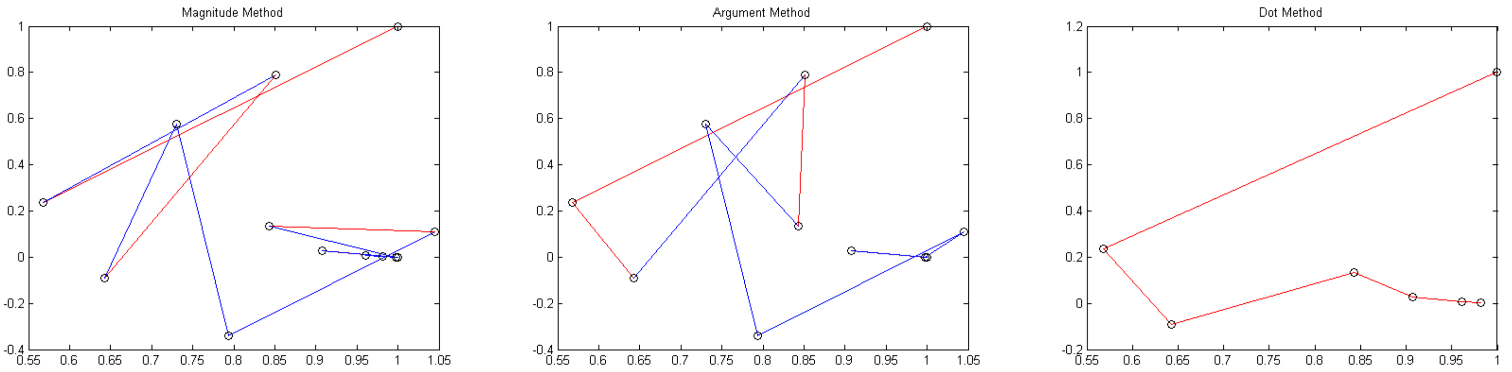
For the case $z = 1 + i$ and $\alpha = 3$



(Figure 3.5)

So far, we have only used positive values for α in our analysis for convergence. Consider now negative values of α, as in Figure 3.6.

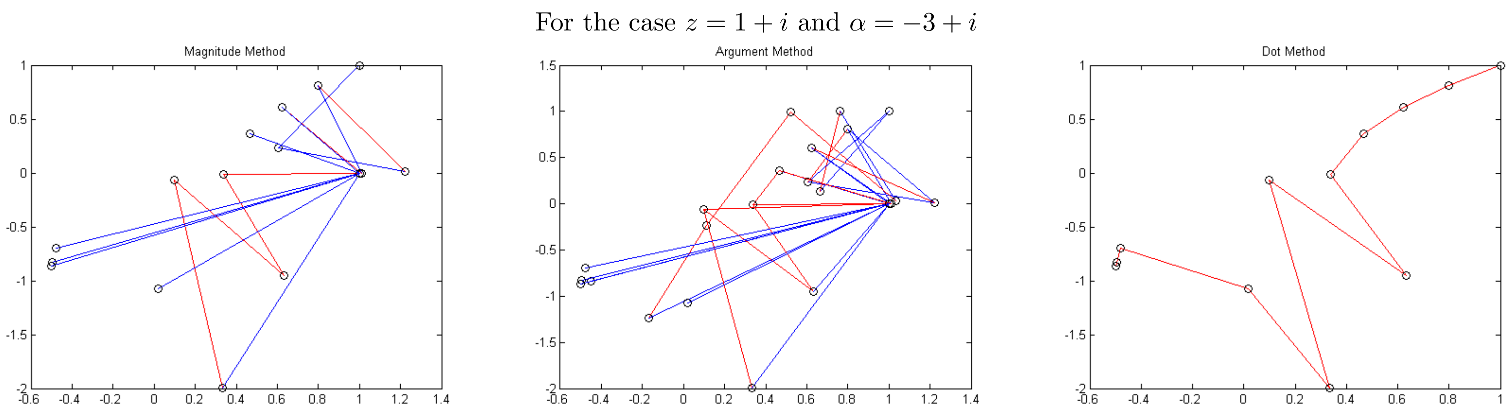
For the case $z = 1 + i$ and $\alpha = -3$



(Figure 3.6)

The *Dot* has a much more direct path to convergence. An even more drastic comparison can be seen when comparing imaginary values for α, illustrated below (Fig 3.7).

For the case $z = 1 + i$ and $\alpha = -3 + i$



(Figure 3.7)

It is interesting to note that although the programs *Magnitude* and *Dot* use the same logic, their paths to convergence are very different. We conclude that although all three methods demonstrate convergence of z and z_0 the *Dot* method is superior.

II. *A Sign Choice Analysis*

The surprising method of sign choice for negative α inspired an analysis of the polynomial $f(z) = z^2 - 1$ with roots at 1 and -1. To begin, choose the initial starting point, z, to be equal to $1+\epsilon$ where $0 < \epsilon << 1$. The images vary the most with α in the range $-1 < \alpha < 0$, so this region of variance is considered.

The necessary variables for the Hansen-Patrick Root Finding Method are as follows:

$z = 1 + \epsilon$
$f(z) = z^2 - 1 = \epsilon^2 + 2\epsilon$
$f'(z) = 2z = 2 + 2\epsilon$
$f''(z) = 2$

Proposition 3.1. Consider the polynomial $z^2 - 1$ and the Hansen-Patrick Root Finding Method. If α is negative, by choosing the sign that results in denominator with the smallest magnitude, the initial point z moves closer to the root z_0.

Proof. Using the variables listed above and the Hansen-Patrick formula, we obtain:

$$z - z_0 = \frac{(1+\alpha)(-1+(1+\epsilon)^2)}{2\alpha(1+\epsilon)\pm\sqrt{4(1+\epsilon)^2-2\alpha(-1+(1+\epsilon)^2)}}$$

We will start this analysis by focusing on the denominator, as this is where the choice between plus and minus occurs:

$$2\alpha(1+\epsilon) \pm \sqrt{4(1+\epsilon)^2 - 2\alpha(-1+(1+\epsilon)^2)}$$

Note that we have opposing signs in the denominator: $2\alpha(1+\epsilon)$ is negative, and $\sqrt{4(1+\epsilon)^2 - 2\alpha(-1+(1+\epsilon)^2)}$ is positive. We are looking to choose the sign that will minimize the magnitude of the denominator so we will choose plus. This gives us a denominator of:

$$2\alpha(1+\epsilon) + \sqrt{4(1+\epsilon)^2 - 2\alpha(-1+(1+\epsilon)^2)}$$

Let us focus on the value under the square root. We have:

$$\sqrt{4(1+\epsilon)^2 - 2\alpha(-1+(1+\epsilon)^2)} = \sqrt{4(1+\epsilon)^2 - 2\alpha(1+\epsilon)^2 + 2\alpha} = \sqrt{4(1+\epsilon)^2 - 2\alpha\epsilon^2 - 4\alpha\epsilon - 2\alpha + 2\alpha} = \sqrt{4(1+\epsilon)^2 - 2\alpha(2\epsilon+\epsilon^2)}$$

Because $\alpha < 0$, the term $-2\alpha(2\epsilon+\epsilon^2)$ is positive. So we have,

$$\sqrt{4(1+\epsilon)^2 - 2\alpha(2\epsilon+\epsilon^2)} > \sqrt{4(1+\epsilon)^2} = 2(1+\epsilon)$$

Putting this back into our original function, we have:

$$z - z_0 = \frac{(1+\alpha)(-1+(1+\epsilon)^2)}{2\alpha(1+\epsilon)+\sqrt{4(1+\epsilon)^2-2\alpha(-1+(1+\epsilon)^2)}} < \frac{(1+\alpha)(-1+(1+\epsilon)^2)}{2\alpha(1+\epsilon)+2(1+\epsilon)}$$

Simplifying, we obtain:

$$\frac{(1+\alpha)(-1+(1+\epsilon)^2)}{2\alpha(1+\epsilon)+2(1+\epsilon)} = \frac{(1+\alpha)(-1+1+2\epsilon+\epsilon)^2)}{2(1+\epsilon)(1+\alpha)} = \frac{(2\epsilon+\epsilon^2)}{2(1+\epsilon)} = \epsilon\frac{(2+\epsilon)}{(2+2\epsilon)}$$

The term $\frac{(2+\epsilon)}{(2+2\epsilon)}$ is less than one; therefore, we know that the distance between the new point z_0 and z is less than ϵ and we are moving closer to the root of 1.

CONCLUSIONS

The 3-D visualizations of the Hansen-Patrick root-finding method program are unique in the sense that they allow the user to visualize all of the basins of attraction at once by stereographically projecting an unbounded basin map onto a bounded 3D sphere. The program creates images such that the boundaries of the basins of attraction vary continuously for all values of alpha and convergence is obtained.

While creating this program, a substantial amount of time was spent considering the choice between plus and minus in the denominator of the Hansen-Patrick method. In the process, two major conclusions were made: positive and negative values for α require different cases for choosing between plus and minus in the denominator of the Hansen-Patrick root-finding method; however, it is possible to write this as a single case using the derivative of the polynomial and the value under the square root. The same logic follows for all methods of choosing between plus and minus: for positive values of α, choose the denominator with the largest magnitude, and for negative values of α, to choose the option of plus/minus that minimizes the denominator.

We have barely scratched the surface of the potential of this program. Although this paper to focuses on the choice between plus and minus in the denominator, the program has the potential to lead to many other areas of interest associated with efficiency of alpha values and potential for basins of attraction.

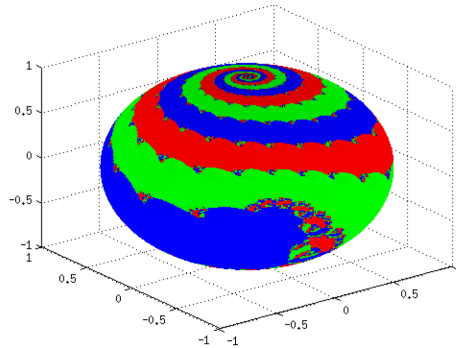
4. Future work

I. *Imaginary values for* α

The images obtained using imaginary values for α are unexpected and suggest that when using imaginary α values, a different method dictating the choice of sign in the denominator must be used. A polynomial that has small roots and clear basins of attracting is listed below (Eq. 4.1).

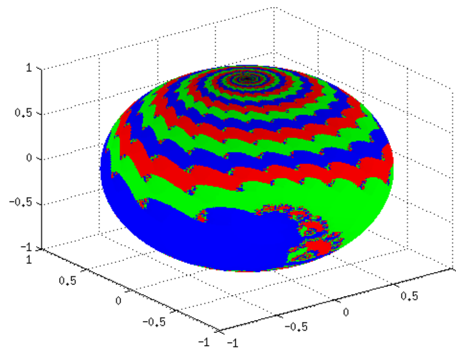
$$f(x) = x^3 - 1 \tag{4.1}$$

As stated earlier, these properties are advantageous for visualizing basins of attraction on a sphere. Convergence is obtained for most complex values of α. For example, consider the sphere obtained with $\alpha = i$ (Fig. 4.1).



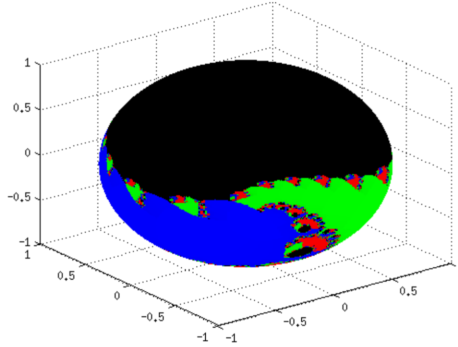
(Figure 4.1)

Following the natural progression to zero, below is the image produced by $\alpha = .9i$ (Fig. 4.2).



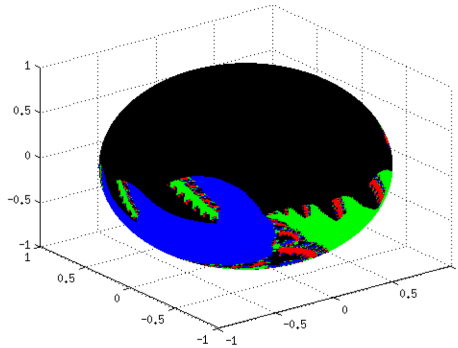
(Figure 4.2)

next step, $\alpha = 0.8i$ gives unexpected results pictured in Figure 4.3.



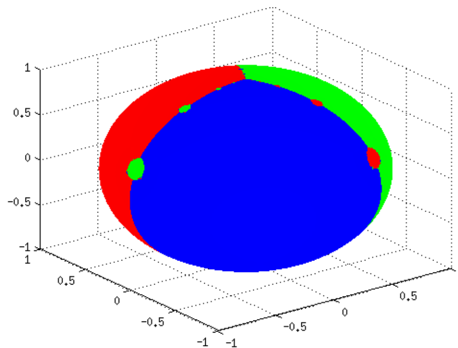
(Figure 4.3)

The entire northern hemisphere does not converge. This remained constant for $\alpha = -0.8i$ to $\alpha = 0.8i$. Pictured in Figure 4.4 is $\alpha = 0.3$.



(Figure 4.4)

This is clearly not consistent with the image for $\alpha = 0$ (Fig 4.5).



(Figure 4.5)

An interesting characteristic of imaginary α values is that complex conjugates α and α^* create images that are rotations of one another. Below is $\alpha = -0.3i$ (Fig 4.6).

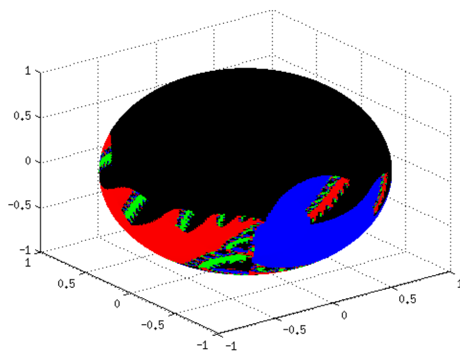
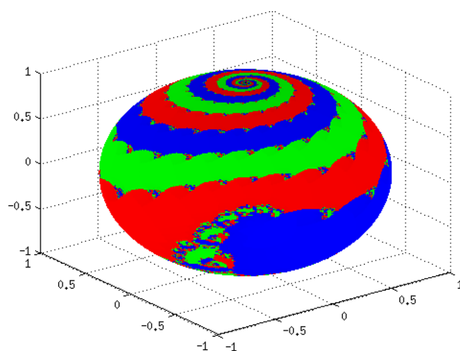


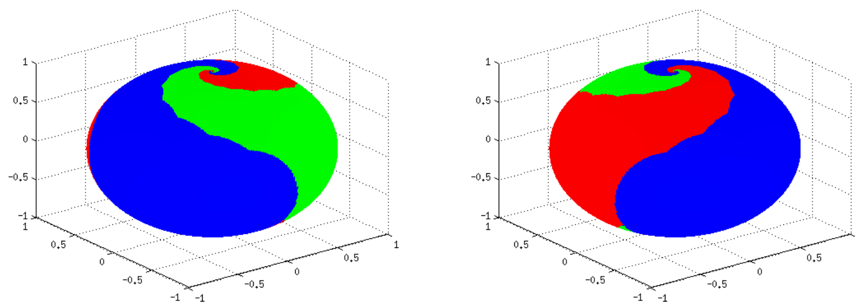
Figure 4.6

Likewise, $\alpha = -i$ (Fig 4.7) is a reversed image of $\alpha = i$



(Figure 4.7)

This is true for α values with both a real and imaginary component, as seen below with $\alpha = 1 + i$ and $\alpha = 1 - i$ (Fig. 4.8).



(Figure 4.8)

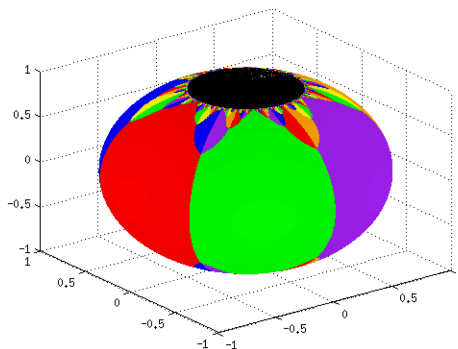
There is much work to be done with imaginary values for α. The divergence indicated in the northern hemisphere of the figures above suggest that the method of choosing between plus and minus in the denominator of the Hansen-Patrick root finding method must be adjusted for α values with an imaginary component.

II. *Special cases with Laguerre*

In our work with complex alpha values, we came across some polynomials that did not agree with what we had been previously been obtaining for Laguerre's method.
Take, for example, the following polynomial (Eq. 4.2).

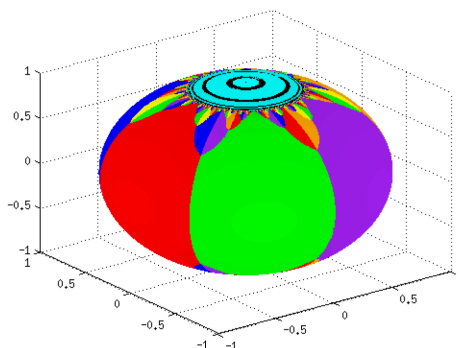
$$f(x) = x^6 - 1 \tag{4.2}$$

Plugging it into the program using Laguerre's method, produces the image below (Fig. 4.9).



(Figure 4.9)

We tested this points individually to see if we were getting the correct results and we found that while they are not converging to a root, they are in fact converging to zero. By creating a color indicating convergence to zero, it is possible to see that this is the case (Fig 4.10).



(Figure 4.10)

We found that this occurs with cases outside of the cubic function.

In the process of exploring these unique functions, we came across some important things. First of all, if a polynomial has a root at zero, Laguerre's method will never be an issue. In fact, the only time a polynomial has an issue with Laguerre's method is when the derivative has no constant. Also, multiplying by a constant (including i) does not affect the convergence of a root, and the images will be identical. This can be shown by a simple proof.

Proposition 4.1. Multiplying a polynomial $f(z)$ by some constant $C \in \mathbb{C}$ does not affect the behavior of

convergence of the Hansen-Patrick Method.

Proof. We want to show that z_0 for some polynomial $f(z)$ is the same when that polynomial is multiplied by a constant $C \in \mathbb{C}$.

We know the Hansen-Patrick root-finding method to be the following:

$z_0 = z - \frac{(\alpha+1)f}{\alpha f' \pm \sqrt{(f')^2 - (\alpha+1)ff''}}$

Let C be a constant such that $C \in \mathbb{C}$ and define $g(z)$ such that

$g(z) = C * f(z)$

Then by the chain rule we have

$g'(z) = C * f'(z)$

$g''(z) = C * f''(z)$

Plugging $g(z)$ back into the Hansen-Patrick method, we have:

$z_0 = z - \frac{(\alpha+1)g(z)}{\alpha g'(z) \pm \sqrt{(g'(z)^2 - (\alpha+1)g(z)g''(z)}}$

Which is equivalent to

$z_0 = z - \frac{(\alpha+1)Cf(z)}{\alpha Cf'(z) \pm \sqrt{(Cf'(z))^2 - (\alpha+1)Cf(z)Cf''(z)}}$

Simplifying we get $z_0 = z - \frac{(\alpha+1)Cf(z)}{\alpha Cf'(z) \pm \sqrt{C^2 f'(z)^2 - C^2(\alpha+1)f(z)f''(z)}} = z - \frac{(\alpha+1)f(z)}{\alpha Cf'(z) \pm C\sqrt{f'(z)^2 - (\alpha+1)f(z)f''(z)}}$

We get that C cancels and

$z_0 = z - \frac{(\alpha+1)f(z)}{\alpha f'(z) \pm \sqrt{(f')^2 - (\alpha+1)f(z)f''(z)}}$

which is the same value for z_0 from $f(z)$.

REFERENCES

Alligood, K. T., Sauer, T., & Yorke, J. A. (1997). *Chaos: an introduction to dynamical systems*. New York: Springer.

Birkhoff, G. D. (1927). *Dynamical systems,*. New York: American Mathematical Society.

Curry, J. H., & Fielder, S. L. (1988). On the dynamics of Laguerre's iterations: Zn-1. *Physica D. Nonlinear phenomena, 30*(1-2), 124-134.

Ford, L. R. (1955). *Differential equations* (2d ed.). New York: McGraw-Hill.

Hansen, E., & Patrick, M. L. (1976). *A family of root finding methods*. Durham, N.C.: Dept. of Computer Science, Duke University.

Kneisl, K. (2001). Julia sets for the super-Newton method, Cauchy's method, and Halley's method. *Chaos, 11*(2), 359-370.

MATLAB. (1998). Natick, Mass.: MathWorks Inc..

Osada, N. (2005). Asymptotic error constants of cubically convergent zero finding methods. *Elsevier science, 1*, 1-16.

Petkovic, M. S., Petkovic, L. D., & Rancic, L. (2003). Higher-order simultaneous methods for the determination of polynomial multiple zeros. *Intern J. Computer Math, 00*, 1-21.

APPENDIX 1: Hansen-Patrick calculations in Java

```
package summer2012;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.PrintWriter;
/**
 *
 * @author pghardy
 */
public class MathResearch{
    //The Hansen-Patrick variable alpha used throught the program set by the constructor
    private Complex alpha;

    //The dimension of the sperical viewing. A picture of dim=250 is decent for overall viewing
        //with 1000 being a good upper limit
    private int dim;

    //This is the main function used throught the program and initialized by the constructor
    private RationalFunction c;

    //The array that holds the roots (or other fixed points you would like to add) of c
    private Complex rootArray[];

    //How close the points must be to a root before we safely say that it converged
    private double MIN_DIFFER = 0.01;

    //Number of iterations we try before shading the point black and consider it non-convergent
    private double ITER = 60;

    //The dimension of the zoom-in feature. 1000 would break up the the x and y pieces into 1000 slice
    private int ZOOM_DIM = 1000;

    //Only used for the Newton feature to distinguish the upper limit for shading
    private double THRESHOLD = 1;

    //Used to zoom-in on a particular area
    private double X1, X2, Y1, Y2;

    //For spherical viewing, this can be changed so that any real number sits around the equator.
    private double equator = 1;

    //This is the path that all the output files will be sent to to be accessed by MATLAB
    private static final String path = "/home/s12/mmpeterson/Desktop/MATLAB/MATLAB_CONVERT/";

    //Simply an easier way to use 0+0i throughout the program
    private static final Complex ZERO = new Complex();  //The number 0 + 0i.

    //A static variable that is used to name the files. For example, when we loop over
        //many alpha values, we name the files ending in _w. Thus file 1 will end in _0
        //and file 2 will end in _1.
    private static int w = 0;
```

```
//the array to hold the colors of each particular root. Colors set in method runColors()
private static double colorArr[][] = new double[10][3];

//The first derivative of the main function c
private static RationalFunction c_der;

//The second deriviative of the main function c
private static RationalFunction c_der2;

//For spherical viewing:
    //B1_1 holds the red portion of the lower hemisphere points
    //B2_1 holds the green portion of the lower hemisphere points
    //B3_1 holds the blue portion of the lower hemisphere points
    //B1_2 holds the red portion of the upper hemisphere points
    //B2_2 holds the green portion of the upper hemisphere points
    //B3_2 holds the blue portion of the upper hemisphere points
    //x holds the x-values of all points on the sphere
    //y holds the y-values of all points on the sphere
    //z holds the z-values of all points on the sphere
private int[][] B1_1, B2_1, B3_1, B1_2, B2_2, B3_2;
private double[][]x,y,z;


/*
 * The constructor is used to set all of the needed information. This is called in
 * GUI which passes all of the information here.
 *
 * @param alpha The Hansen-Patrick value to be used
 * @param newton True if you would like to use the Newton feature
 * @param zoom True if you would like to use the zoom feature
 * @param loop True if you would like to use the looping feature
 * @param real_part True if you would like to look over the real part of alpha
 * @param start When looping, the inital alpha value
 * @param end When looping, the ending alpha value
 * @param loopIter When looping the number of sections to break up (end-start).
 *          Ex. To loop over -10 to 10, use loopIter = 21 to include both endpoints properly
 * @param dim The dimension used in the sperical viewing feature
 * @param c The main function to be viewed
 * @param UPPER_LEFT_X The upper left x-value of the point to zoom in on
 * @param UPPER_LEFT_Y The upper left y-value of the point to zoom in on
 * @param UPPER_LEFT_Z The upper left z-value of the point to zoom in on (only for spherical zoomi
 * @param LOWER_RIGHT_X The upper left x-value of the point to zoom in on
 * @param LOWER_RIGHT_Y The upper left y-value of the point to zoom in on
 * @param LOWER_RIGHT_Z The upper left z-value of the point to zoom in on (only for spherical zoom
 * @param rootArray The array of roots of the function as Complex numbers.
 * @param MIN_DIFFER How close a point must be to be considered converged
 * @param maxIter Top number of iterations to try per point
 * @param Zoom_dim Dimension of the zoomed-in area
 * @param threshold The maximum distance the point can be away from actual Newtons method before p
 * @param equator For spherical viewing, this is the real values that is displayed about the equat
 */
public MathResearch(Complex alpha, boolean newton, boolean zoom, boolean loop, boolean real_part,
```

```
                Complex start, Complex end, int loopIter, int dim, RationalFunction c,
                double UPPER_LEFT_X, double UPPER_LEFT_Y, double UPPER_LEFT_Z,
                double LOWER_RIGHT_X, double LOWER_RIGHT_Y, double LOWER_RIGHT_Z,
                Complex[] rootArray, double MIN_DIFFER, int maxIter, int Zoom_dim,
                double threshold, double equator){
super();
this.alpha = alpha;
this.dim = dim;
this.c = c;
this.rootArray = rootArray;
this.MIN_DIFFER = MIN_DIFFER;
this.ITER = maxIter;
this.ZOOM_DIM = Zoom_dim;
this.THRESHOLD = threshold;
this.X1 = UPPER_LEFT_X / (1 - UPPER_LEFT_Z);
this.Y1 = UPPER_LEFT_Y / (1 - UPPER_LEFT_Z);
this.X2 = LOWER_RIGHT_X / (1 - LOWER_RIGHT_Z);
this.Y2 = LOWER_RIGHT_Y / (1 - LOWER_RIGHT_Z);
this.equator = equator;
c_der = c.differentiate();
c_der2 = c_der.differentiate();

/*
 * This is a big if/else block which simply navigates to the desired feature to be used
 * The choices are: Newton, Loop, Real-part and Zoom. Therefore the 12 possible features in or
 *  1. Zooming in while looping over the real part of Newton
 *  2. Sphere viewing while looping over the real part of Newton (not implemented)
 *  3. Zooming in while looping over the imaginary part of Newton
 *  4. Sphere viewing while looping over the imaginary part of Newton (not implemented)
 *  5. Zooming in without looping using with Newton
 *  6. Spherical viewing without looping using with Newton (not implemented)
 *  7. Zooming in while looping over the real part
 *  8. spherical viewing while looping over the real part
 *  9. Zooming in while looping over the imaginary part
 *  10. spherical viewing while looping over the imaginary part
 *  11. Original Zooming without looping
 *  12. Original Sphere without looping
 */
if(newton){
    if(loop){
        if(real_part){
            if(zoom){
                double step = (end.getRe() - start.getRe()) / (loopIter - 1);
                for (w = 0; w < loopIter; w++) {
                    this.alpha = start.add(new Complex(w * step, 0));
                    newtonEx();
                }
            }
            else{//do not zoom
                System.out.println("Looping over Newton without Zooming is not yet implemented
            }
        }
        else{//imaginary part
```

```
                if(zoom){
                    for (w = 0; w < loopIter; w++) {
                        double step = (end.getIm() - start.getIm()) / (loopIter - 1);
                        this.alpha = start.add(new Complex(0, w * step));
                        newtonEx();
                    }
                }
                else{//do not zoom
                    System.out.println("Looping over Newton without Zooming is not yet implemented
                }
            }
        }
        else{//do not Loop
            if (zoom) {
                newtonEx();
            }
            else {//do not zoom
                System.out.println("Spherical viewing of Newton is not yet implemented");
            }
        }
    }
    else{//not Newton
        if(loop){
            if(real_part){
               if(zoom){
                    double step = (end.getRe() - start.getRe()) / (loopIter - 1);
                    for (w = 0; w < loopIter; w++) {
                        this.alpha = start.add(new Complex(w * step, 0));
                        run2();
                    }
                }
                else{//do not zoom
                    double step = (end.getRe() - start.getRe()) / (loopIter - 1);
                    for (w = 0; w < loopIter; w++) {
                        this.alpha = start.add(new Complex(w * step, 0));
                        run();
                    }
                }
            }
            else{//imaginary part
                if(zoom){
                    for (w = 0; w < loopIter; w++) {
                        double step = (end.getIm() - start.getIm()) / (loopIter - 1);
                        this.alpha = start.add(new Complex(0, w * step));
                        run2();
                    }
                }
                else{//do not zoom
                    double step = (end.getIm() - start.getIm()) / (loopIter - 1);
                    for (w = 0; w < loopIter; w++) {
                        this.alpha = start.add(new Complex(0, w * step));
                        run();
                    }
```

```
                }
            }
        }
        else{//do not Loop
            if(zoom){
                   run2();
                }
                else{//do not zoom
                   run();
                }
        }
    }
}
/*
 * A method to run when you want to view the spherical version
 */
private void run(){
    //Initialize the colors for rootArray
    runColors();

    //Create 9 PrintWriters for outputting the 9 matrices
    PrintWriter[] myOuts = new PrintWriter[9];

    //Initialize the 9 PrintWriters
    //For example, the very first files would have the path:
    //    home/f10/pghardy/Desktop/MATLAB/MATLAB_CONVERT/0_0
    for(int mat = 0; mat < 9; mat++){
        try{
            myOuts[mat] = new PrintWriter(new BufferedWriter(new FileWriter(path+mat+"_"+w)));
        }catch(Exception e){System.out.println(e);}
    }
    B1_1 = new int[dim+1][dim+1]; //REDNESS //B = uint8(round(A * 255)); //Lower Hemisphere
    B2_1 = new int[dim+1][dim+1]; //GREENESS                             //Lower Hemisphere
    B3_1 = new int[dim+1][dim+1]; //BLUENESS                             //Lower Hemisphere
    B1_2 = new int[dim+1][dim+1]; //REDNESS //B = uint8(round(A * 255)); //Upper Hemisphere
    B2_2 = new int[dim+1][dim+1]; //GREENESS                             //Upper Hemisphere
    B3_2 = new int[dim+1][dim+1]; //BLUENESS                             //Upper Hemisphere
    x = new double[dim+1][dim+1];
    y = new double[dim+1][dim+1];
    z = new double[dim+1][dim+1];

    //An array for temporarily holding the radius of the poitns
    double[] temp = new double[dim+1];

    //An array for holding the angle of the point (both are polar coordinates)
    double[] tempAngles = new double[dim+1];

    //This loop initializes the points to be tested so they will be evenly spaced once on the sphe
    for(int t = 0; t <= dim; t++){
        temp[t] = (Math.cos((-(Math.PI)*t)/(2.0*dim))/(1-Math.sin((-(Math.PI)*t)/(2.0*dim)))*equat
        tempAngles[t] = (t*2*Math.PI)/dim;
    }
```

```
//A 'matrix' to hold the x-values to be tested
double[][] outputX = new double[tempAngles.length][temp.length];
//A 'matrix' to hold the y-values to be tested
double[][] outputY = new double[tempAngles.length][temp.length];

//This loop converts the points to cartesian coordinates
for(int m = 0; m < tempAngles.length; m++){
    for(int n = 0; n < temp.length; n++){
        outputX[m][n] = temp[m]*Math.cos(tempAngles[n]);
        outputY[m][n] = temp[m]*Math.sin(tempAngles[n]);
    }
}

//This is where the colors are made. When o=1, we are doing the southern hemishpere
for(int o = 1; o <= 2; o++){ //for o=1:2

    //v and w are simply used to traverse outputX and outputY
    //v corresponds to the row and w is the column
    for(int v = 0; v <= dim; v++){ //for v=1:dim+1
        for(int w = 0; w <= dim; w++){ //for w = 1:dim+1

            //xf is created using the coordinates we already created
            Complex xf = new Complex(outputX[v][w],outputY[v][w]); //xf = complex(X(v,w),Y(v,w

            //For the northern hemishpere, we basically take the reciprocal of the points that
                //evenly distributed between zero and the equator
            if(o == 2){ //if (o == 2)
                if(xf.equals(ZERO)){ //if (xf == 0)
                    break;
                }
                xf = new Complex(equator,0).mul((new Complex(equator,0).div(xf)).getConjugate(
            }

            //General Loop, k goes up to the maximum number of specified iterations
            for(int k = 1; k <= ITER; k++) { //for k=1:iter

               //xs becomes the new point that xf went to under the Hansen-Patrick method
               Complex xs = newPoint(xf);

               //Now we check if the point is within MIN_DIFFER of any of the roots
               Integer rootIndex = findRootIndex(xs);

               //If it is NOT close enough, then rootIndex==null and we try again
               //If it is close enough we color that point according to the root that was
                    //found and the number of iterations (k) it took to converge
               if(rootIndex != null){ //if ~isempty(rootIndex)
                   if(o==1){
                        B1_1[v][w]=(int)Math.round(colorArr[rootIndex][0]*(1-(k/ITER))); //Rec
                        B2_1[v][w]=(int)Math.round(colorArr[rootIndex][1]*(1-(k/ITER))); //Gre
                        B3_1[v][w]=(int)Math.round(colorArr[rootIndex][2]*(1-(k/ITER))); //Blu
                        break;
                   }
                   else{
```

```
                        B1_2[v][w]=(int)Math.round(colorArr[rootIndex][0]*(1-(k/ITER))); //Red
                        B2_2[v][w]=(int)Math.round(colorArr[rootIndex][1]*(1-(k/ITER))); //Gre
                        B3_2[v][w]=(int)Math.round(colorArr[rootIndex][2]*(1-(k/ITER))); //Blu
                        break;
                    }
                }

                //Then the next guess is set to the old point
                xf=xs; //xf = xs;
            }

        }
    }
}

//These loops convert the points into the sperical coordinates
for (int sigma = 0; sigma <= dim; sigma++) {
    for (int zeta = 0; zeta <= dim; zeta++) {
        double tempX = outputX[sigma][zeta]/equator;
        double tempY = outputY[sigma][zeta]/equator;
        x[sigma][zeta] = (2 * tempX) / (1 + tempX * tempX + tempY * tempY); //x=(2*X)./(1+X.^2
        y[sigma][zeta] = (2 * tempY) / (1 + tempX * tempX + tempY * tempY); //y=(2*Y)./(1+X.^2
        z[sigma][zeta] = (tempX * tempX + tempY * tempY - 1) / (tempX * tempX + tempY * tempY
    }
}

//These 9 lines output the 9 matrices we need to construct the MATLAB surf function
outputInt(B1_1, myOuts[0]); myOuts[0].close();
outputInt(B2_1, myOuts[1]); myOuts[1].close();
outputInt(B3_1, myOuts[2]); myOuts[2].close();
outputInt(B1_2, myOuts[3]); myOuts[3].close();
outputInt(B2_2, myOuts[4]); myOuts[4].close();
outputInt(B3_2, myOuts[5]); myOuts[5].close();
outputDouble(x, myOuts[6]); myOuts[6].close();
outputDouble(y, myOuts[7]); myOuts[7].close();
outputDoubleVer(z, myOuts[8]); myOuts[8].close();
}

/*
 * A method that is used only when wanting to zoom in on a specified area
 */
private void run2() {
    //Initialize the colors for rootArray
    runColors();

    //Create 5 PrintWriters for outputting the 5 matrices
    PrintWriter[] myOuts = new PrintWriter[5];

    //Initialize the 5 PrintWriters
    //For example, the very first files would have the path:
    //    home/f10/pghardy/Desktop/MATLAB/MATLAB_CONVERT/ZOOM_0_0
    for (int mat = 0; mat < 5; mat++) {
        try {
```

```
        myOuts[mat] = new PrintWriter(new FileWriter(path + "ZOOM_" + mat+"_"+w));
    } catch (Exception e) {System.out.println(e);}

}
B1_1 = new int[ZOOM_DIM+1][ZOOM_DIM+1]; //REDNESS //B = uint8(round(A * 255));
B2_1 = new int[ZOOM_DIM+1][ZOOM_DIM+1]; //GREENESS
B3_1 = new int[ZOOM_DIM+1][ZOOM_DIM+1]; //BLUENESS

//A temporary array to hold the x-values to be tested
double[] Xtemp = new double[ZOOM_DIM + 1];
//A temporary array to hold the y-values to be tested
double[] Ytemp = new double[ZOOM_DIM + 1];

//A 'matrix' of all the x-values of points to be tested
x = new double[ZOOM_DIM + 1][ZOOM_DIM + 1];
//A 'matrix' of all the y-values of points to be tested
y = new double[ZOOM_DIM + 1][ZOOM_DIM + 1];

//This breaks up the change in x and the change in y into ZOOM_DIM pieces
for (int i = 0; i <= ZOOM_DIM; i++) {
    Xtemp[i] = X1 + ((X2 - X1) * i) / (ZOOM_DIM); //real parts of the points to test
    Ytemp[i] = Y1 + ((Y2 - Y1) * i) / (ZOOM_DIM); //imaginary parts of the points to test
}

//This loop places all the combinations together. If Xtemp = [1 2] and Ytemp=[3 4],
    //then x = [1 2] and y = [3 3] and now all four points are matched up.
    //        [1 2]         [4 4]
for (int m = 0; m <= ZOOM_DIM; m++) {
    for (int n = 0; n <= ZOOM_DIM; n++) {
        x[m][n] = Xtemp[n];
        y[m][n] = Ytemp[m];
    }
}

//v and w are simply used to traverse outputX and outputY
//v corresponds to the row and w is the column
for (int v = 0; v <= ZOOM_DIM; v++) {
    for (int w = 0; w <= ZOOM_DIM; w++) {

        //xf is created using the coordinates we already created
        Complex xf = new Complex(x[v][w], y[v][w]);

        //General Loop, k goes up to the maximum number of specified iterations
        for (int k = 0; k < ITER; k++) {

            //xs becomes the new point that xf goes to under the Hansen-Patrick method
            Complex xs = newPoint(xf);

            //Now we check if the point is within MIN_DIFFER of any of the roots
            Integer rootIndex = findRootIndex(xs);

            //If it is NOT close enough, then rootIndex==null and we try again
            //If it is close enough we color that point according to the root that was
```

```
                    //found and the number of iterations (k) it took to converge
                if (rootIndex != null) { //if ~isempty(rootIndex)
                    B1_1[v][w] = (int) Math.round(colorArr[rootIndex][0] * (1 - (k / ITER))); //Re
                    B2_1[v][w] = (int) Math.round(colorArr[rootIndex][1] * (1 - (k / ITER))); //Gr
                    B3_1[v][w] = (int) Math.round(colorArr[rootIndex][2] * (1 - (k / ITER))); //Bl
                    break;
                }

                //Then the next guess is set to the old point
                xf = xs; //point = xs;
            }
        }
    }
    //These 5 lines output the 5 matrices we need to construct the MATLAB surf function
    outputInt(B1_1, myOuts[0]); myOuts[0].close();
    outputInt(B2_1, myOuts[1]); myOuts[1].close();
    outputInt(B3_1, myOuts[2]); myOuts[2].close();
    outputDouble(x, myOuts[3]); myOuts[3].close();
    outputDouble(y, myOuts[4]); myOuts[4].close();
}

/*
 * The bulk of the program which computes the new point of the iteration
 */
private Complex newPoint(Complex oldPoint){

    //A Complex holder for the next guess
    Complex newPoint = oldPoint;

    //The value of the function evaluated at the old point
    Complex z = c.evaluate(oldPoint); //z = polyval(c, oldPoint);

    //The value of the derivative evaluated at the old point
    Complex zp = c_der.evaluate(oldPoint); //zp = polyval(c_der, oldPoint);

    //The value of the second derivative evaluated at the old point
    Complex zpp = c_der2.evaluate(oldPoint); //zpp = polyval(c_der2, oldPoint);

    //The top of the fractional part of the Hansen-Patrick method
    Complex top = (alpha.add(new Complex(1, 0))).mul(z); //top = (a+1)*z;

    //The square root portion in the denominator
    Complex sq_root = (zp.mul(zp).sub(top.mul(zpp))).sq_root(); //sq_root = sqrt(zp^2-top*zpp);

    //The value of the bottom if the plus sign is used
    Complex bottom1 = (alpha.mul(zp)).add(sq_root); //bottom1 = a*z+sq_root;

    //The value of the bottom if the minus sign is used
    Complex bottom2 = (alpha.mul(zp)).sub(sq_root); //bottom2 = a*z-sq_root;

    //The absolute value of the plus version of the bottom
    double bottom1Abs = bottom1.modulus(); //bottom1Abs = abs(bottom1);
```

```
    //The absolute value of the minus version of the bottom
    double bottom2Abs = bottom2.modulus(); //bottom2Abs = abs(bottom2);

    //for determining sign choice
           if(alpha.getRe()>=0){
        if (bottom1Abs.equals(ZERO) && bottom2Abs.equals(ZERO)) {
            newPoint = oldPoint;
        } else if (bottom1Abs.getRe() == bottom2Abs.getRe()) {
            if ((zp.getConjugate().mul(bottom1)).getRe() > 0) {
                newPoint = oldPoint.sub(top.div(bottom1)); //newPoint = oldPoint - top/bottom1;
            } else {
                newPoint = oldPoint.sub(top.div(bottom2)); //newPoint = oldPoint - top/bottom2;
            }
        } else if (bottom1Abs.getRe() > bottom2Abs.getRe()) { //elseif(bottom1Abs > bottom2Abs) /
            newPoint = oldPoint.sub(top.div(bottom1)); //newPoint = oldPoint - top/bottom1;
        } else {
            newPoint = oldPoint.sub(top.div(bottom2)); //newPoint = oldPoint - top/bottom2;
        }
    } else if (alpha.getRe() != -1) {
        if (bottom1Abs.equals(ZERO) && bottom2Abs.equals(ZERO)) {
            newPoint = oldPoint;
        } else if (bottom1Abs.getRe() >= bottom2Abs.getRe()) { //elseif(bottom1Abs > bottom2Abs) /
            newPoint = oldPoint.sub(top.div(bottom2)); //newPoint = oldPoint - top/bottom1;
        } else {
            newPoint = oldPoint.sub(top.div(bottom1)); //newPoint = oldPoint - top/bottom2;
        }

    } else {//alpha.getRe()==-1 //Halley's Method
        top = new Complex(2, 0).mul(z).mul(zp);
        bottom1 = (new Complex(2, 0).mul(zp).mul(zp)).sub(z.mul(zpp));
        if (bottom1.equals(ZERO)) {
            newPoint = oldPoint;
        }
        else{
            newPoint = oldPoint.sub(top.div(bottom1));
        }
    }
    return newPoint;
}



/*
 * This method is used for finding if a point is within MIN_DIFFER of a root
 * Note: If more than one root is found, it will use the 'last' one
 */
public Integer findRootIndex(Complex newPoint){
    Integer rootIndex = null;
    Complex tempA[] = new Complex[rootArray.length];
    for (int rep = 0; rep < rootArray.length; rep++) {
        tempA[rep] = new Complex((newPoint.sub(rootArray[rep])).modulus(),0);//tmp = abs(repmat(po
        if (tempA[rep].getRe() < MIN_DIFFER) { //rootIndex = find(tmp<min_differ);
            rootIndex = rep;
```

```
        }
    }
    return rootIndex;
}

/*
 * This method computes the pieces common to run() and run2()
 */
public void runColors(){
    colorArr[0][0] = 255;colorArr[0][1] = 0;  colorArr[0][2] = 0;   //RED
    colorArr[1][0] = 0;  colorArr[1][1] = 0;  colorArr[1][2] = 255; //BLUE
    colorArr[2][0] = 0;  colorArr[2][1] = 255;colorArr[2][2] = 0;   //GREEN
    colorArr[3][0] = 255;colorArr[3][1] = 255;colorArr[3][2] = 0;   //YELLOW
    colorArr[4][0] = 160;colorArr[4][1] = 32; colorArr[4][2] = 240; //PURPLE
    colorArr[5][0] = 255;colorArr[5][1] = 165;colorArr[5][2] = 0;   //ORANGE
    colorArr[6][0] = 0;  colorArr[6][1] = 255;colorArr[6][2] = 255; //CYAN
    colorArr[7][0] = 140;colorArr[7][1] = 70; colorArr[7][2] = 20;  //BROWN
    colorArr[8][0] = 255;colorArr[8][1] = 105;colorArr[8][2] = 180; //PINK
    colorArr[9][0] = 255;colorArr[9][1] = 255;colorArr[9][2] = 255; //GRAY

}

private void newtonEx(){
    runColors();
    FileWriter[] outFiles = new FileWriter[3];
    PrintWriter[] myOuts = new PrintWriter[3];
    for (int mat = 0; mat < 3; mat++) {
        try {
            outFiles[mat] = new FileWriter(path + "NEWT_" + mat+"_"+w);
            myOuts[mat] = new PrintWriter(outFiles[mat]);
        } catch (Exception e) {System.out.println(e);}

    }
    B1_1 = new int[ZOOM_DIM+1][ZOOM_DIM+1]; //GRAYNESS //B = uint8(round(A * 255));

    double[] Xtemp = new double[ZOOM_DIM + 1];
    double[] Ytemp = new double[ZOOM_DIM + 1];
    x = new double[ZOOM_DIM + 1][ZOOM_DIM + 1];
    y = new double[ZOOM_DIM + 1][ZOOM_DIM + 1];
    for (int i = 0; i <= ZOOM_DIM; i++) {
        Xtemp[i] = X1 + ((X2 - X1) * i) / (ZOOM_DIM); //real parts of the points to test
        Ytemp[i] = Y1 + ((Y2 - Y1) * i) / (ZOOM_DIM); //imaginary parts of the points to test
    }
    for (int m = 0; m <= ZOOM_DIM; m++) {
        for (int n = 0; n <= ZOOM_DIM; n++) {
            x[m][n] = Xtemp[n];
            y[m][n] = Ytemp[m];
        }
    }
    for (int v = 0; v <= ZOOM_DIM; v++) {
        for (int w = 0; w <= ZOOM_DIM; w++) {
            Complex oldNormal = new Complex(x[v][w], y[v][w]);
            Complex oldNewton = new Complex(x[v][w], y[v][w]);
```

```
                Complex temp1 = oldNormal;
                Complex temp2 = oldNewton;
                Complex newNormal = null;
                Complex newNewton = null;
                for (int k = 0; k < ITER; k++) {
                   newNormal = newPoint(oldNormal);
                   newNewton = newPointNewton(oldNewton);
                   oldNormal = newNormal;
                   oldNewton = newNewton;
//                   if(k == 24){
//                       System.out.println(newNormal.sub(newNewton).modulus().getRe());
//                   }
                }

                double modDif  = (newNormal.sub(newNewton)).modulus();

                if(modDif >THRESHOLD){
                    System.out.println("old: " + temp1);
                    System.out.println("old: " + temp2);
                    System.out.println("norm: " + newNormal);
                    System.out.println("newt: " + newNewton);
                    System.out.println(modDif);
                    System.out.println();
                    modDif = THRESHOLD;
                }
                B1_1[v][w] = (int) Math.round(255 *(1-(THRESHOLD-modDif)/THRESHOLD));


            }
        }
        outputInt(B1_1, myOuts[0]); myOuts[0].close();
        outputDouble(x, myOuts[1]); myOuts[1].close();
        outputDouble(y, myOuts[2]); myOuts[2].close();
    }


    public Complex newPointNewton(Complex oldPoint){
        Complex z = c.evaluate(oldPoint);
        Complex zp = c_der.evaluate(oldPoint);

        return oldPoint.sub(z.div(zp));
    }
    /*
     * A method for outputting files of the form int[][]
     */
    public void outputInt(int[][] ob, PrintWriter pw){
        for(int i = 0; i < ob.length; i++){
            for(int j = 0; j < ob[i].length; j++){
                pw.print(ob[i][j] + " ");
            } pw.println();
        } pw.close();
    }
```

```
    /*
     * A method for outputting files of the form double[][]
     */
    public void outputDouble(double[][] ob, PrintWriter pw){
        for(int i = 0; i < ob.length; i++){
            for(int j = 0; j < ob[i].length; j++){
                pw.print(ob[i][j] + " ");
            } pw.println();
        } pw.close();
    }

    public void outputDoubleVer(double[][] ob, PrintWriter pw){
        for(int i = 0; i < ob.length; i++){
            pw.println(ob[i][0] + " ");
        } pw.close();
    }

    /*
     * A method for printing out and array of the form int[][]
     */
    public void printArrayInt(int[][] ob){
        for(int i = 0; i < ob.length; i++){
            for(int j = 0; j < ob[i].length; j++){
                System.out.print(ob[i][j] + " ");
            }
            System.out.println();
        }
        System.out.println();
        System.out.println();
    }

    /*
     * A method for printing out and array of the form double[][]
     */
    public void printArrayDouble(double[][] ob){
        for(int i = 0; i < ob.length; i++){
            for(int j = 0; j < ob[i].length; j++){
                System.out.print(ob[i][j] + " ");
            }
            System.out.println();
        }
        System.out.println();
        System.out.println();
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

    }

}
```

APPENDIX 2: Complex numbers in Java

```
package summer2012;

/* Copyright (c) 2012 the authors listed at the following URL, and/or
the authors of referenced articles or incorporated external code:
http://en.literateprograms.org/Complex_numbers_(Java)?action=history&offset=20090126143509

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Retrieved from: http://en.literateprograms.org/Complex_numbers_(Java)?oldid=16044
*/

public class Complex {

    private double re;
    private double im;

    public Complex () {
      this.re = 0;
      this.im = 0;
    }

    public Complex(double re, double im) {
      this.re = re;
      this.im = im;
    }
    public Complex(double re, double im, String print) {
      System.out.println(print + "|" + re + "," + im);
      this.re = re;
      this.im = im;
    }

    public Complex(Complex input) {
      this.re = input.getRe();
      this.im = input.getIm();
```

```
}
public double getRe() {
  return this.re;
}

public double getIm() {
  return this.im;
}
public void setRe(double re) {
  this.re = re;
}

public void setIm(double im) {
  this.im = im;
}
public Complex add(Complex op) {
  Complex result = new Complex();
  result.setRe(this.re + op.getRe());
  result.setIm(this.im + op.getIm());
  return result;
}

public Complex sub(Complex op) {
  Complex result = new Complex();
  result.setRe(this.re - op.getRe());
  result.setIm(this.im - op.getIm());
  return result;
}

public Complex mul(Complex op) {
  Complex result = new Complex();
  result.setRe(this.re * op.getRe() - this.im * op.getIm());
  result.setIm(this.re * op.getIm() + this.im * op.getRe());
  return result;
}

public Complex mul(double op) {
  Complex result = new Complex();
  result.setRe(this.re * op);
  result.setIm(this.im * op);
  return result;
}

public Complex div(Complex op) {
  Complex result = new Complex(this);
  result = result.mul(op.getConjugate());
 double opNormSq = op.getRe()*op.getRe()+op.getIm()*op.getIm();
  result.setRe(result.getRe() / opNormSq);
  result.setIm(result.getIm() / opNormSq);
  return result;
}

public double modulus(){
```

```
        return Math.sqrt(this.re*this.re + this.im*this.im);
    }
//    public Complex fromPolar(double magnitude, double angle) {
//      Complex result = new Complex();
//      result.setRe(magnitude * Math.cos(angle));
//      result.setIm(magnitude * Math.sin(angle));
//      return result;
//    }
//
//    public double getNorm() {
//      return Math.sqrt(this.re * this.re + this.im * this.im);
//    }

    public double getAngle() {
      return Math.atan2(this.im, this.re);
    }
    public Complex getConjugate() {
      return new Complex(this.re, this.im * (-1));
    }

    public boolean equals(Complex op){
        if(this.re == op.getRe() && this.im == op.getIm()){
            return true;
        }
        return false;
    }

    public Complex sq_root(){
        Complex sq = this;
        double a = this.getRe();
        double b = this.getIm();
        if (b==0){
            if(a>=0){
                return new Complex(Math.sqrt(a),0);
            }//a<0
            return new Complex(0,Math.sqrt(-a));
        }
        if(b>0){
            return new Complex(Math.sqrt(((Math.sqrt(a*a+b*b))+a)/(2)),Math.sqrt(((Math.sqrt(a*a+b*b))
        }
        //b<0
        return new Complex(Math.sqrt(((Math.sqrt(a*a+b*b))+a)/(2)),-Math.sqrt(((Math.sqrt(a*a+b*b))-a)
    }

    @Override
    public String toString() {
      if (this.re == 0) {
        if (this.im == 0) {
          return "0";
        } else {
          return (this.im + "i");
        }
      } else {
```

```
      if (this.im == 0) {
        return String.valueOf(this.re);
      } else if (this.im < 0) {
        return("[" + this.re + " " + this.im + "i]");
      } else {
        return("[" + this.re + " +" + this.im + "i]");
      }
    }
  }

  public static void main(String argv[]) {
    Complex a = new Complex(3, 4);
    Complex b = new Complex(1, -100);
    //System.out.println(a.getNorm());
    System.out.println(b.getAngle());
    System.out.println(a.mul(b));
    System.out.println(a.div(b));
    System.out.println(a.div(b).mul(b));
    Complex c = new Complex(5, -2);
    Complex d = new Complex(5, -2);
    System.out.println(c.equals(d));
    System.out.println(c == d);

    Complex e = new Complex(5, 0);
    Complex f = new Complex(-5, 0);
    Complex g = new Complex(4, 1);
    Complex h = new Complex(4, -1);
    System.out.println(e.sq_root());
    System.out.println(f.sq_root());
    System.out.println(g.sq_root());
    System.out.println(h.sq_root());

    Complex j = new Complex(8, -6);
    System.out.println(j.modulus());
  }
}
```

APPENDIX 3: Complex polynomials in Java

```
package summer2012;

public class ComplexPolynomial {
    private Complex[] coef;  // coefficients
    private int deg;     // degree of ComplexPolynomial (0 for the zero ComplexPolynomial)
    private Complex ZERO = new Complex();

    // a * x^b
    public ComplexPolynomial(Complex a, int b) {
        coef = new Complex[b+1];
        for(int l = 0; l < b; l++){
            coef[l] = ZERO;
        }
        coef[b] = a;
        deg = degree();
    }
    //non-complex coefficient, a * x^b
//    public ComplexPolynomial(double a, int b) {
//        coef = new Complex[b+1];
//        for(int l = 0; l < b; l++){
//            coef[l] = ZERO;
//        }
//        coef[b] = new Complex(a,0);
//        deg = degree();
//    }

//    public Complex getCoef(int i){
//        return coef[i];
//    }

    public void setCoef(Complex a, int b){
        coef[b] = a;
    }


    // return the degree of this ComplexPolynomial (0 for the zero ComplexPolynomial)
    private int degree() {
        int d = 0;
        for (int i = 0; i < coef.length; i++)
            if (!coef[i].equals(ZERO)) d = i;
        return d;
    }

    // return c = a + b
    public ComplexPolynomial plus(ComplexPolynomial b) {
        ComplexPolynomial a = this;
        ComplexPolynomial c = new ComplexPolynomial(ZERO, Math.max(a.deg, b.deg));
        for (int i = 0; i <= a.deg; i++) c.coef[i] = c.coef[i].add(a.coef[i]);
        for (int i = 0; i <= b.deg; i++) c.coef[i] = c.coef[i].add(b.coef[i]);
        c.deg = c.degree();
        return c;
```

```
    }

    // return (a - b)
    public ComplexPolynomial minus(ComplexPolynomial b) {
        ComplexPolynomial a = this;
        ComplexPolynomial c = new ComplexPolynomial(ZERO, Math.max(a.deg, b.deg));
        for (int i = 0; i <= a.deg; i++) c.coef[i] = c.coef[i].add(a.coef[i]);
        for (int i = 0; i <= b.deg; i++) c.coef[i] = c.coef[i].sub(b.coef[i]);
        c.deg = c.degree();
        return c;
    }

    // return (a * b)
    public ComplexPolynomial times(ComplexPolynomial b) {
        ComplexPolynomial a = this;
        ComplexPolynomial c = new ComplexPolynomial(ZERO, a.deg + b.deg);
        for (int i = 0; i <= a.deg; i++)
            for (int j = 0; j <= b.deg; j++)
                c.coef[i+j] = c.coef[i+j].add((a.coef[i].mul(b.coef[j])));
        c.deg = c.degree();
        return c;
    }

    public RationalFunction divide(ComplexPolynomial b){
        return new RationalFunction(this, b);
    }

    // return a(b(x))  - compute using Horner's method
//    public ComplexPolynomial compose(ComplexPolynomial b) {
//        ComplexPolynomial a = this;
//        ComplexPolynomial c = new ComplexPolynomial(ZERO, 0);
//        for (int i = a.deg; i >= 0; i--) {
//            ComplexPolynomial term = new ComplexPolynomial(a.coef[i], 0);
//            c = term.plus(b.times(c));
//        }
//        return c;
//    }


    // do a and b represent the same ComplexPolynomial?
    public boolean eq(ComplexPolynomial b) {
        ComplexPolynomial a = this;
        if (a.deg != b.deg) return false;
        for (int i = a.deg; i >= 0; i--)
            if (!a.coef[i].equals(b.coef[i])) return false;
        return true;
    }


    // use Horner's method to compute and return the ComplexPolynomial evaluated at x
    public Complex evaluate(Complex x) {
        Complex p = ZERO;
        p = p.add(coef[0]);
```

```
    for(int i = 1; i <= this.deg; i++){
        Complex temp = x;
        for(int e = 1; e < i; e++){
            temp = temp.mul(x);
        }
        p = p.add(coef[i].mul(temp));
    }

    return p;
}

// differentiate this ComplexPolynomial and return it
public ComplexPolynomial differentiate() {
    if (deg == 0) return new ComplexPolynomial(ZERO, 0);
    ComplexPolynomial deriv = new ComplexPolynomial(ZERO, deg - 1);
    deriv.deg = deg - 1;
    for (int i = 0; i < deg; i++)
        deriv.coef[i] = new Complex(i + 1,0).mul(coef[i + 1]);
    return deriv;
}

// convert to string representation
@Override
public String toString() {
    if (deg ==  0) return "" + coef[0];
    if (deg ==  1) return coef[1] + "x + " + coef[0];
    String s = coef[deg] + "x^" + deg;
    for (int i = deg-1; i >= 0; i--) {
        if      (coef[i].equals(new Complex(0,0))) continue;
        else s = s + " + " + ( coef[i]);
        if      (i == 1) s = s + "x";
        else if (i >  1) s = s + "x^" + i;
    }
    return s;
}

// test client
public static void main(String[] args) {
    ComplexPolynomial zero = new ComplexPolynomial(new Complex(), 0);

    ComplexPolynomial p1   = new ComplexPolynomial(new Complex(4,1), 3);
    ComplexPolynomial p2   = new ComplexPolynomial(new Complex(3,0), 2);
    ComplexPolynomial p3   = new ComplexPolynomial(new Complex(1,0), 0);
    ComplexPolynomial p4   = new ComplexPolynomial(new Complex(2,0), 1);
    ComplexPolynomial p    = p1.plus(p2).plus(p3).plus(p4);   // (4+i)x^3 + 3x^2 + 2x + 1

    ComplexPolynomial q1   = new ComplexPolynomial(new Complex(3,0), 2);
    ComplexPolynomial q2   = new ComplexPolynomial(new Complex(5,0), 0);
    ComplexPolynomial q    = q1.plus(q2);                     // 3x^2 + 5


    ComplexPolynomial r    = p.plus(q);
    ComplexPolynomial s    = p.times(q);
```

```
        //ComplexPolynomial t    = p.compose(q);

        System.out.println("zero(x) =     " + zero);
        System.out.println("p(x) =        " + p);
        System.out.println("q(x) =        " + q);
        System.out.println("p(x) + q(x) = " + r);
        System.out.println("p(x) * q(x) = " + s);
        //System.out.println("p(q(x))     = " + t);
        System.out.println("0 - p(x)    = " + zero.minus(p));
        System.out.println("p(3+0i)     = " + p.evaluate(new Complex(3,0)));
        System.out.println("p(3-2i)     = " + p.evaluate(new Complex(3,-2)));
        System.out.println("p'(x)       = " + p.differentiate());
        System.out.println("p''(x)      = " + p.differentiate().differentiate());

        ComplexPolynomial one = new ComplexPolynomial(new Complex(1,0), 0);
        ComplexPolynomial four = new ComplexPolynomial(new Complex(4,0), 0);
        ComplexPolynomial nine = new ComplexPolynomial(new Complex(9,0), 0);
        ComplexPolynomial sixteen = new ComplexPolynomial(new Complex(16,0), 0);
        ComplexPolynomial twentyFive = new ComplexPolynomial(new Complex(25,0), 0);

        ComplexPolynomial squared = new ComplexPolynomial(new Complex(1,0), 2);

        ComplexPolynomial m1 = squared.plus(one);
        ComplexPolynomial m2 = squared.plus(four);
        ComplexPolynomial m3 = squared.plus(nine);
        ComplexPolynomial m4 = squared.plus(sixteen);
        ComplexPolynomial m5 = squared.plus(twentyFive);
        ComplexPolynomial M = m1.times(m2).times(m3).times(m4).times(m5);
        System.out.println(M);

        System.out.println("*******************");

        ComplexPolynomial test1 = new ComplexPolynomial(new Complex(1,0), 0);
        ComplexPolynomial test2 = new ComplexPolynomial(new Complex(4,0), 2);
        ComplexPolynomial test = test1.plus(test2);
        System.out.println(test);
        test.setCoef(new Complex(8,0),2);
        System.out.println(test);
    }
}
```

APPENDIX 4: Guest user interface in Java

```
package summer2012;

import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.PrintWriter;
import javax.swing.*;

/**
 *
 * @author pghardy
 */
public class GUI extends JFrame implements ActionListener {

    PrintWriter myOut;
    BufferedReader myIn;
    Container thisFrame;
    JPanel function, roots;
    JLabel[] powersNum = new JLabel[11];
    JLabel[] powersDen = new JLabel[11];
    JLabel[] plusNum = new JLabel[11];
    JLabel[] plusDen = new JLabel[11];
    JLabel[] imNum = new JLabel[11];
    JLabel[] imDen = new JLabel[11];
    JLabel NUMERATOR = new JLabel("NUMERATOR");
    JButton DENOMINATOR;
    JTextField[] numeratorLeft = new JTextField[11];
    JTextField[] numeratorRight = new JTextField[11];
    JTextField[] denominatorLeft = new JTextField[11];
    JTextField[] denominatorRight = new JTextField[11];

    JLabel[] blanks = new JLabel[30];

    JLabel ROOTS = new JLabel("FUNCTION ROOTS");
    JTextField[] rootsLeft = new JTextField[10];
    JTextField[] rootsRight = new JTextField[10];
    JLabel[] plusRoots = new JLabel[10];
    JLabel[] imRoots = new JLabel[10];

    JButton /*normal,*/ newton, zoom, loop;
    //JTextField normalBox = new JTextField("true");
    JTextField newtonBox = new JTextField("false");
    JTextField zoomBox = new JTextField("false");
    JTextField loopBox = new JTextField("false");

    JLabel dim = new JLabel("DIM");
```

```
JLabel loopIts = new JLabel("loopIts");
JLabel alpha = new JLabel("alpha");
JLabel end = new JLabel("END");

JTextField dimBox = new JTextField("500");
JTextField loopItsBox = new JTextField("51");

JTextField alphaLeft = new JTextField("0");
JLabel alphaPlus = new JLabel("+");
JTextField alphaRight = new JTextField("0");
JTextField endLeft = new JTextField("0");
JLabel endPlus = new JLabel("+");
JTextField endRight = new JTextField("0");

JLabel ulX = new JLabel("Upper-Left-X");
JLabel ulY = new JLabel("Upper-Left-Y");
JLabel ulZ = new JLabel("Upper-Left-Z");
JLabel lrX = new JLabel("Lower-Right-X");
JLabel lrY = new JLabel("Lower-Right-Y");
JLabel lrZ = new JLabel("Lower-Right-Z");

JTextField ulXbox = new JTextField("-100.0");
JTextField ulYbox = new JTextField("100.0");
JTextField ulZbox = new JTextField("0.0");
JTextField lrXbox = new JTextField("100.0");
JTextField lrYbox = new JTextField("-100.0");
JTextField lrZbox = new JTextField("0.0");

JLabel min_differ = new JLabel("MIN_DIF");
JLabel zoom_dim = new JLabel("ZOOM_DIM");
JLabel maxIter = new JLabel("MaxIterations");
JLabel threshold = new JLabel("Threshold");

JTextField min_differBox = new JTextField("0.01");
JTextField zoom_dimBox = new JTextField("1000");
JTextField maxIterBox = new JTextField("25");
JTextField thresholdBox = new JTextField("1");

JLabel equator = new JLabel("Equator");
JTextField equatorBox = new JTextField("1");

JButton completed;

public GUI() {

    //normal = new JButton("NORMAL");
    newton = new JButton("NEWTON");
    zoom = new JButton("ZOOM");
    loop = new JButton("LOOP");
    DENOMINATOR = new JButton("DENOMINATOR");
    completed = new JButton("RUN");
    //normal.addActionListener(this);
    newton.addActionListener(this);
```

```
zoom.addActionListener(this);
loop.addActionListener(this);
DENOMINATOR.addActionListener(this);
completed.addActionListener(this);

//normalBox.setEditable(false);
newtonBox.setEditable(false);
zoomBox.setEditable(false);
loopBox.setEditable(false);

ulXbox.setEditable(false);
ulYbox.setEditable(false);
ulZbox.setEditable(false);
lrXbox.setEditable(false);
lrYbox.setEditable(false);
lrZbox.setEditable(false);
zoom_dimBox.setEditable(false);

loopItsBox.setEditable(false);
endRight.setEditable(false);
endLeft.setEditable(false);

thresholdBox.setEditable(false);

function = new JPanel(new GridLayout(24,5));
roots = new JPanel(new GridLayout(22,4));

for(int i = 0; i < 30; i++) blanks[i] = new JLabel("");


for(int i = 0; i < 11; i++){
    numeratorLeft[i] = new JTextField("0");
    numeratorRight[i] = new JTextField("0");
    denominatorLeft[i] = new JTextField("0");
    denominatorRight[i] = new JTextField("0");
    powersNum[i] = new JLabel("x^"+i);
    powersDen[i] = new JLabel("x^"+i);
    plusNum[i] = new JLabel("+");
    plusDen[i] = new JLabel("+");
    imNum[i] = new JLabel("i");
    imDen[i] = new JLabel("i");
}
denominatorLeft[0].setText("1");
function.add(NUMERATOR);function.add(blanks[0]);function.add(blanks[1]);function.add(blanks[2]
for(int i = 0; i < 11; i++){
    function.add(numeratorLeft[i]);
    function.add(plusNum[i]);
    function.add(numeratorRight[i]);
    function.add(imNum[i]);
    function.add(powersNum[i]);
}
function.add(DENOMINATOR);function.add(blanks[4]);function.add(blanks[5]);function.add(blanks[
for(int i = 0; i < 11; i++){
```

```
            function.add(denominatorLeft[i]);
            function.add(plusDen[i]);
            function.add(denominatorRight[i]);
            function.add(imDen[i]);
            function.add(powersDen[i]);
        }

        roots.add(ROOTS);roots.add(blanks[8]);roots.add(blanks[9]);roots.add(blanks[10]);
        for(int i = 0; i < 10; i++){
            rootsLeft[i] = new JTextField("");
            rootsRight[i] = new JTextField("");
            plusRoots[i] = new JLabel("+");
            imRoots[i] = new JLabel("i");
            roots.add(rootsLeft[i]);
            roots.add(plusRoots[i]);
            roots.add(rootsRight[i]);
            roots.add(imRoots[i]);
        }
        roots.add(blanks[11]);roots.add(newton);roots.add(zoom);roots.add(loop);
        roots.add(blanks[12]);roots.add(newtonBox);roots.add(zoomBox);roots.add(loopBox);
        roots.add(dim);roots.add(dimBox);roots.add(loopIts);roots.add(loopItsBox);
        roots.add(alpha);roots.add(alphaLeft);roots.add(alphaPlus);roots.add(alphaRight);
        roots.add(end);roots.add(endLeft);roots.add(endPlus);roots.add(endRight);
        roots.add(ulX);roots.add(ulXbox);roots.add(lrX);roots.add(lrXbox);
        roots.add(ulY);roots.add(ulYbox);roots.add(lrY);roots.add(lrYbox);
        roots.add(ulZ);roots.add(ulZbox);roots.add(lrZ);roots.add(lrZbox);
        roots.add(min_differ);roots.add(min_differBox);roots.add(zoom_dim);roots.add(zoom_dimBox);
        roots.add(maxIter);roots.add(maxIterBox);roots.add(threshold);roots.add(thresholdBox);
        roots.add(completed);roots.add(blanks[13]);roots.add(equator);roots.add(equatorBox);

//        JTextField min_differBox = new JTextField("0.01");
//        JTextField zoom_dimBox = new JTextField("1000");
//        JTextField NewtonIterBox = new JTextField("10");
//        JTextField thresholdBox = new JTextField("1");
        for(int i = 0; i < 11; i++){
            denominatorLeft[i].setEditable(false);
            denominatorRight[i].setEditable(false);
        }

        thisFrame = getContentPane();

        thisFrame.add(function, BorderLayout.WEST);
        thisFrame.add(roots, BorderLayout.EAST);

        thisFrame.validate();
        loadValues();

        setVisible(true);
        setSize(1300, 750);
    }

    private void loadValues(){
        try{
```

```
    myIn = new BufferedReader(new FileReader("values"));
}
catch(Exception ex){
    System.out.println(ex);
}
try{
    for(int i = 0; i < 11; i++){
        numeratorLeft[i].setText(myIn.readLine());
        numeratorRight[i].setText(myIn.readLine());
        denominatorLeft[i].setText(myIn.readLine());
        denominatorRight[i].setText(myIn.readLine());
    }
    for(int i = 0; i < 10; i++){
        rootsLeft[i].setText(myIn.readLine());
        rootsRight[i].setText(myIn.readLine());
    }
    newtonBox.setText(myIn.readLine());
    zoomBox.setText(myIn.readLine());
    loopBox.setText(myIn.readLine());
    dimBox.setText(myIn.readLine());
    loopItsBox.setText(myIn.readLine());
    alphaLeft.setText(myIn.readLine());
    alphaRight.setText(myIn.readLine());
    endLeft.setText(myIn.readLine());
    endRight.setText(myIn.readLine());
    ulXbox.setText(myIn.readLine());
    ulYbox.setText(myIn.readLine());
    ulZbox.setText(myIn.readLine());
    lrXbox.setText(myIn.readLine());
    lrYbox.setText(myIn.readLine());
    lrZbox.setText(myIn.readLine());
    min_differBox.setText(myIn.readLine());
    zoom_dimBox.setText(myIn.readLine());
    maxIterBox.setText(myIn.readLine());
    thresholdBox.setText(myIn.readLine());
    equatorBox.setText(myIn.readLine());

    myIn.close();

}
catch(Exception ex2){
    System.out.println(ex2);
}
if (newtonBox.getText().equals("true")) {
    thresholdBox.setEditable(true);
    for(int i = 0; i < 10; i++){
        rootsLeft[i].setEditable(false);
        rootsRight[i].setEditable(false);
    }
    maxIter.setText("NewtonIter");
    equatorBox.setEditable(false);
} else {
    thresholdBox.setEditable(false);
```

```
            maxIter.setText("MaxIterations");
            equatorBox.setEditable(true);
        }
        if(zoomBox.getText().equals("true")) {
            ulXbox.setEditable(true);
            ulYbox.setEditable(true);
            ulZbox.setEditable(true);
            lrXbox.setEditable(true);
            lrYbox.setEditable(true);
            lrZbox.setEditable(true);
            zoom_dimBox.setEditable(true);
            dimBox.setEditable(false);
            equatorBox.setEditable(false);
        } else {
            ulXbox.setEditable(false);
            ulYbox.setEditable(false);
            ulZbox.setEditable(false);
            lrXbox.setEditable(false);
            lrYbox.setEditable(false);
            lrZbox.setEditable(false);
            zoom_dimBox.setEditable(false);
            dimBox.setEditable(true);
            equatorBox.setEditable(true);
        }
        if (loopBox.getText().equals("false")) {
            loopItsBox.setEditable(false);
            endRight.setEditable(false);
            endLeft.setEditable(false);
            alpha.setText("alpha");
        } else {
            loopItsBox.setEditable(true);
            endRight.setEditable(true);
            endLeft.setEditable(true);
            alpha.setText("START");
        }
    }

     private void storeValues(){
        try{
            myOut = new PrintWriter(new FileWriter("values"));
        }
        catch(Exception ex){
           System.out.println(ex);
        }
        for(int i = 0; i < 11; i++){
            myOut.println(numeratorLeft[i].getText());
            myOut.println(numeratorRight[i].getText());
            myOut.println(denominatorLeft[i].getText());
            myOut.println(denominatorRight[i].getText());

        }
        for(int i = 0; i < 10; i++){
            myOut.println(rootsLeft[i].getText());
```

```
            myOut.println(rootsRight[i].getText());
        }
        myOut.println(newtonBox.getText());
        myOut.println(zoomBox.getText());
        myOut.println(loopBox.getText());
        myOut.println(dimBox.getText());
        myOut.println(loopItsBox.getText());
        myOut.println(alphaLeft.getText());
        myOut.println(alphaRight.getText());
        myOut.println(endLeft.getText());
        myOut.println(endRight.getText());
        myOut.println(ulXbox.getText());
        myOut.println(ulYbox.getText());
        myOut.println(ulZbox.getText());
        myOut.println(lrXbox.getText());
        myOut.println(lrYbox.getText());
        myOut.println(lrZbox.getText());
        myOut.println(min_differBox.getText());
        myOut.println(zoom_dimBox.getText());
        myOut.println(maxIterBox.getText());
        myOut.println(thresholdBox.getText());
        myOut.println(equatorBox.getText());

        myOut.close();

    }

    @Override
    public void actionPerformed(ActionEvent evt) {
        Object actionObject = evt.getSource();
        /*if(actionObject==normal){
            if(normalBox.getText().equals("true")){
                normalBox.setText("false");
            }
            else{
                normalBox.setText("true");
            }
        }
        else */if(actionObject==newton){
            if(newtonBox.getText().equals("true")){
                newtonBox.setText("false");
                for(int i = 0; i < 10; i++){
                    rootsLeft[i].setEditable(true);
                    rootsRight[i].setEditable(true);
                }
                thresholdBox.setEditable(false);
                maxIter.setText("MaxIterations");
                equatorBox.setEditable(true);
            }
            else{
                newtonBox.setText("true");
                for(int i = 0; i < 10; i++){
                    rootsLeft[i].setEditable(false);
```

```
                rootsRight[i].setEditable(false);
            }
            thresholdBox.setEditable(true);
            maxIter.setText("NewtonIter");
            equatorBox.setEditable(false);
        }
    }
    else if(actionObject==zoom){
        if(zoomBox.getText().equals("true")) {
            zoomBox.setText("false");
            ulXbox.setEditable(false);
            ulYbox.setEditable(false);
            ulZbox.setEditable(false);
            lrXbox.setEditable(false);
            lrYbox.setEditable(false);
            lrZbox.setEditable(false);
            dimBox.setEditable(true);
            zoom_dimBox.setEditable(false);
            equatorBox.setEditable(true);
        }
        else{
            zoomBox.setText("true");
            ulXbox.setEditable(true);
            ulYbox.setEditable(true);
            ulZbox.setEditable(true);
            lrXbox.setEditable(true);
            lrYbox.setEditable(true);
            lrZbox.setEditable(true);
            zoom_dimBox.setEditable(true);
            dimBox.setEditable(false);
            equatorBox.setEditable(false);
        }
    }
    else if(actionObject==loop){
        if(loopBox.getText().equals("false")){
            loopBox.setText("true-Real");
            loopItsBox.setEditable(true);
            endRight.setEditable(true);
            endLeft.setEditable(true);
            alpha.setText("START");
        }
        else if(loopBox.getText().equals("true-Real")){
            loopBox.setText("true-Imaginary");
        }
        else{
            loopBox.setText("false");
            loopItsBox.setEditable(false);
            endRight.setEditable(false);
            endLeft.setEditable(false);
            alpha.setText("alpha");
        }
    }
    else if(actionObject==DENOMINATOR){
```

```
            if(denominatorLeft[0].isEditable()){
                for(int i = 0; i < 11; i++){
                    denominatorLeft[i].setEditable(false);
                    denominatorRight[i].setEditable(false);
                }
            }
            else{
                for(int i = 0; i < 11; i++){
                    denominatorLeft[i].setEditable(true);
                    denominatorRight[i].setEditable(true);
                }
            }

        }
        else if(actionObject == completed) {
            Complex alpha = new Complex(Double.parseDouble(alphaLeft.getText()),Double.parseDouble(alp

//              boolean normal;
//              if(normalBox.getText().equals("true")){normal = true;}
//              else{normal = false;}

            boolean newton;
            if(newtonBox.getText().equals("true")){newton = true;}
            else{newton = false;}

            boolean zoom;
            if(zoomBox.getText().equals("true")){zoom = true;}
            else{zoom = false;}

            boolean loop;
            boolean realPart;
            if(loopBox.getText().equals("false")){loop = false; realPart = false;}
            else if(loopBox.getText().equals("true-Real")){loop = true; realPart = true;}
            else{loop = true; realPart = false;}

            Complex start = alpha;
            Complex end = new Complex(Double.parseDouble(endLeft.getText()),Double.parseDouble(endRigh

            int loopIter = Integer.parseInt(loopItsBox.getText());

            int dim = Integer.parseInt(dimBox.getText());

            Complex[] numeratorCoef = new Complex[11];
            Complex[] denominatorCoef = new Complex[11];
            ComplexPolynomial[] numeratorPieces = new ComplexPolynomial[11];
            ComplexPolynomial[] denominatorPieces = new ComplexPolynomial[11];
            ComplexPolynomial c1 = new ComplexPolynomial(new Complex(),0);
            ComplexPolynomial c2 = new ComplexPolynomial(new Complex(),0);

            for(int i = 0; i < 11; i++){
                numeratorCoef[i]= new Complex(Double.parseDouble(numeratorLeft[i].getText()),Double.pa
                denominatorCoef[i]= new Complex(Double.parseDouble(denominatorLeft[i].getText()),Doubl
                numeratorPieces[i] = new ComplexPolynomial(numeratorCoef[i],i);
```

```
        denominatorPieces[i] = new ComplexPolynomial(denominatorCoef[i],i);
        c1 = c1.plus(numeratorPieces[i]);
        c2 = c2.plus(denominatorPieces[i]);
    }
    RationalFunction c = new RationalFunction(c1,c2);

    double UPPER_LEFT_X = Double.parseDouble(ulXbox.getText());
    double UPPER_LEFT_Y = Double.parseDouble(ulYbox.getText());
    double UPPER_LEFT_Z = Double.parseDouble(ulZbox.getText());
    double LOWER_RIGHT_X = Double.parseDouble(lrXbox.getText());
    double LOWER_RIGHT_Y = Double.parseDouble(lrYbox.getText());
    double LOWER_RIGHT_Z = Double.parseDouble(lrZbox.getText());

    Complex[] rootArray = null;
    for(int i = 0; i <= 10; i++){
        if(i==10){
            rootArray = new Complex[10];
            break;
        }
        if(rootsLeft[i].getText().equals("") && rootsRight[i].getText().equals("")){
            rootArray = new Complex[i];
            break;
        }
    }
    for(int i = 0; i < rootArray.length; i++){
        rootArray[i] = new Complex(Double.parseDouble(rootsLeft[i].getText()),Double.parseDoub
    }

    double minDiffer = Double.parseDouble(min_differBox.getText());

    int maxIter = Integer.parseInt(maxIterBox.getText());

    int zoom_dim = Integer.parseInt(zoom_dimBox.getText());

    double threshold  = Double.parseDouble(thresholdBox.getText());

    double equator = Double.parseDouble(equatorBox.getText());

    storeValues();
    if (maxIter == -1) {
        MathResearchSign mr = new MathResearchSign(alpha, newton, zoom, loop, realPart,
                start, end, loopIter, dim, c,
                UPPER_LEFT_X, UPPER_LEFT_Y, UPPER_LEFT_Z,
                LOWER_RIGHT_X, LOWER_RIGHT_Y, LOWER_RIGHT_Z,
                rootArray, minDiffer, maxIter, zoom_dim,
                threshold, equator);
    } else {
        MathResearch mr = new MathResearch(alpha, newton, zoom, loop, realPart,
                start, end, loopIter, dim, c,
                UPPER_LEFT_X, UPPER_LEFT_Y, UPPER_LEFT_Z,
                LOWER_RIGHT_X, LOWER_RIGHT_Y, LOWER_RIGHT_Z,
                rootArray, minDiffer, maxIter, zoom_dim,
                threshold, equator);
```

```
            }

            System.exit(0);

        }
    }

    public static void main(String[] args) {
        GUI g = new GUI();
    }
}
```

APPENDIX 5: Branches for Java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package summer2012;

import java.io.FileWriter;
import java.io.PrintWriter;

/**
 *
 * @author pghardy
 */
public class Branches {
    Complex alpha;
    Complex point;
    static RationalFunction c;
    static RationalFunction c_der;
    static RationalFunction c_der2;
    private static Complex rootArray[];
    private final int maxIt = 1;
    static FileWriter outFiles;
    static PrintWriter myOuts = null;


    public Branches(Complex point, int currentIt){



        this.alpha = new Complex(0.2,0);
        this.point = point;

        for(Complex lex: rootArray){
            if((lex.sub(point)).modulus() < 0.001){
                System.out.println("*************************************");
                System.out.println("*************************************");
                System.out.println("*************************************");
                System.out.println("|");
                System.out.println("V");
                myOuts.print(point + " | LEVEL: " + currentIt);
                myOuts.println();
            }
        }

        //this.point = new Complex(-0.12813570386711456,0.09309603821535975);

        Complex z = c.evaluate(point);
        Complex zp = c_der.evaluate(point);
        Complex zpp = c_der2.evaluate(point);
```

```
        Complex alphaPlusOne = alpha.add(new Complex(1,0));
        Complex alphaTimesFp = alpha.mul(zp);
        Complex zpSquared = zp.mul(zp);



        Complex sq_root = ((zpSquared).sub(alphaPlusOne.mul(z).mul(zpp))).sq_root();
        Complex top = alphaPlusOne.mul(z);

        Complex p = point.sub(top.div(alphaTimesFp.add(sq_root)));
        System.out.println(point + " | LEVEL: " + currentIt);
        System.out.println("PLUS POINT:     " + p);

        Complex m = point.sub(top.div(alphaTimesFp.sub(sq_root)));
        System.out.println("MINUS POINT:    " + m);
        System.out.println();
        System.out.println();

        if(currentIt < maxIt){
            new Branches(p, currentIt+1);
            new Branches(m, currentIt+1);

        }
    }

    public static void main(String[] args){
        rootArray = new Complex[6];
        rootArray[0] = new Complex(-1,0); //RED
        rootArray[1] = new Complex(-0.5, 0.866025403784439); //BLUE
        rootArray[2] = new Complex(-0.5,-0.866025403784439); //GREEN
        rootArray[3] = new Complex(0.5, 0.866025403784439); //YELLOW
        rootArray[4] = new Complex(0.5,-0.866025403784439); //PURPLE
        rootArray[5] = new Complex(1,0); //ORANGE

        //Complex point = new Complex(1/(4*Math.sin(-0.4*Math.PI)-4),((Math.sin(0.8*Math.PI))*(1+Math.
        Complex point = new Complex(100,100);

        Complex a = new Complex(-1,0);        //The constant coefficient,  ex: -3      --> Complex(-3,
        Complex b = new Complex(0,0);        //The linear coefficient, ex: (4-i)x     --> Complex(4,-1
        Complex d = new Complex(0,0);        //The quadratic coefficient, ex: (2i)x^2 --> Complex(0,2)
        Complex e = new Complex(0,0);        //...
        Complex f = new Complex(0,0);
        Complex g = new Complex(0,0);
        Complex h = new Complex(1,0);
        Complex i = new Complex(0,0);
        Complex j = new Complex(0,0);
        Complex k = new Complex(0,0);
        Complex l = new Complex(0,0);
        //**********************************

        ComplexPolynomial constant = new ComplexPolynomial(a,0);
        ComplexPolynomial linear   = new ComplexPolynomial(b,1);
        ComplexPolynomial quadratic= new ComplexPolynomial(d,2);
```

```
        ComplexPolynomial cubic    = new ComplexPolynomial(e,3);
        ComplexPolynomial quartic  = new ComplexPolynomial(f,4);
        ComplexPolynomial quintic  = new ComplexPolynomial(g,5);
        ComplexPolynomial sixthDeg = new ComplexPolynomial(h,6);
        ComplexPolynomial sevDeg   = new ComplexPolynomial(i,7);
        ComplexPolynomial eightDeg = new ComplexPolynomial(j,8);
        ComplexPolynomial nineDeg  = new ComplexPolynomial(k,9);
        ComplexPolynomial tenDeg   = new ComplexPolynomial(l,10);

        c = new RationalFunction(constant.plus(linear).plus(quadratic).plus(cubic).plus(quartic).plus(
        c_der = c.differentiate();
        c_der2 = c_der.differentiate();
        try{
            outFiles = new FileWriter("/home/f10/pghardy/Desktop/MATLAB/MATLAB_CONVERT/branches");
            myOuts = new PrintWriter(outFiles);
        }
        catch(Exception ex){
            System.out.println(ex);
        }

        Branches br = new Branches(point, 0);
        myOuts.close();
    }
}
```

APPENDIX 6: Sign choice in Java

```
package summer2012;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.PrintWriter;

/**
 *
 * @author pghardy
 */
public class MathResearchSign {
    //The Hansen-Patrick variable alpha used throught the program set by the constructor

    private Complex alpha;
    //The dimension of the sperical viewing. A picture of dim=250 is decent for overall viewing
    //with 1000 being a good upper limit
    private int dim;
    //This is the main function used throught the program and initialized by the constructor
    private RationalFunction c;
    //The dimension of the zoom-in feature. 1000 would break up the the x and y pieces into 1000 slice
    private int ZOOM_DIM = 1000;
    //Used to zoom-in on a particular area
    private double X1, X2, Y1, Y2;
    //For spherical viewing, this can be changed so that any real number sits around the equator.
    private double equator = 1;
    //This is the path that all the output files will be sent to to be accessed by MATLAB
    private static final String path = "/home/s12/mmpeterson/Desktop/MATLAB/MATLAB_CONVERT/";
    //Simply an easier way to use 0+0i throughout the program
    private static final Complex ZERO = new Complex();  //The number 0 + 0i.
    //A static variable that is used to name the files. For example, when we loop over
    //many alpha values, we name the files ending in _w. Thus file 1 will end in _0
    //and file 2 will end in _1.
    private static int w = 0;
    //The first derivative of the main function c
    private static RationalFunction c_der;
    //The second deriviative of the main function c
    private static RationalFunction c_der2;
    //For spherical viewing:
    //B1_1 holds the red portion of the lower hemisphere points
    //B2_1 holds the green portion of the lower hemisphere points
    //B3_1 holds the blue portion of the lower hemisphere points
    //B1_2 holds the red portion of the upper hemisphere points
    //B2_2 holds the green portion of the upper hemisphere points
    //B3_2 holds the blue portion of the upper hemisphere points
    //x holds the x-values of all points on the sphere
    //y holds the y-values of all points on the sphere
    //z holds the z-values of all points on the sphere
    private int[][] B1_1, B2_1, B3_1, B1_2, B2_2, B3_2;
    private double[][] x, y, z;

    /*
```

```
 * The constructor is used to set all of the needed information. This is
 * called in GUI which passes all of the information here.
 *
 * @param alpha The Hansen-Patrick value to be used @param newton True if
 * you would like to use the Newton feature @param zoom True if you would
 * like to use the zoom feature @param loop True if you would like to use
 * the looping feature @param real_part True if you would like to look over
 * the real part of alpha @param start When looping, the inital alpha value
 * @param end When looping, the ending alpha value @param loopIter When
 * looping the number of sections to break up (end-start). Ex. To loop over
 * -10 to 10, use loopIter = 21 to include both endpoints properly @param
 * dim The dimension used in the sperical viewing feature @param c The main
 * function to be viewed @param UPPER_LEFT_X The upper left x-value of the
 * point to zoom in on @param UPPER_LEFT_Y The upper left y-value of the
 * point to zoom in on @param UPPER_LEFT_Z The upper left z-value of the
 * point to zoom in on (only for spherical zooming) @param LOWER_RIGHT_X The
 * upper left x-value of the point to zoom in on @param LOWER_RIGHT_Y The
 * upper left y-value of the point to zoom in on @param LOWER_RIGHT_Z The
 * upper left z-value of the point to zoom in on (only for spherical
 * zooming) @param rootArray The array of roots of the function as Complex
 * numbers. @param MIN_DIFFER How close a point must be to be considered
 * converged @param maxIter Top number of iterations to try per point @param
 * Zoom_dim Dimension of the zoomed-in area @param threshold The maximum
 * distance the point can be away from actual Newtons method before painted
 * white @param equator For spherical viewing, this is the real values that
 * is displayed about the equator.
 */
public MathResearchSign(Complex alpha, boolean newton, boolean zoom, boolean loop, boolean real_pa
        Complex start, Complex end, int loopIter, int dim, RationalFunction c,
        double UPPER_LEFT_X, double UPPER_LEFT_Y, double UPPER_LEFT_Z,
        double LOWER_RIGHT_X, double LOWER_RIGHT_Y, double LOWER_RIGHT_Z,
        Complex[] rootArray, double MIN_DIFFER, int maxIter, int Zoom_dim,
        double threshold, double equator) {
    this.alpha = alpha;
    this.dim = dim;
    this.c = c;
    this.ZOOM_DIM = Zoom_dim;
    this.X1 = UPPER_LEFT_X / (1 - UPPER_LEFT_Z);
    this.Y1 = UPPER_LEFT_Y / (1 - UPPER_LEFT_Z);
    this.X2 = LOWER_RIGHT_X / (1 - LOWER_RIGHT_Z);
    this.Y2 = LOWER_RIGHT_Y / (1 - LOWER_RIGHT_Z);
    this.equator = equator;
    c_der = c.differentiate();
    c_der2 = c_der.differentiate();

    /*
     * This is a big if/else block which simply navigates to the desired
     * feature to be used The choices are: Newton, Loop, Real-part and Zoom.
     * Therefore the 12 possible features in order are: 1. Zooming in while
     * looping over the real part of Newton 2. Sphere viewing while looping
     * over the real part of Newton (not implemented) 3. Zooming in while
     * looping over the imaginary part of Newton 4. Sphere viewing while
     * looping over the imaginary part of Newton (not implemented) 5.
```

```
 * Zooming in without looping using with Newton 6. Spherical viewing
 * without looping using with Newton (not implemented) 7. Zooming in
 * while looping over the real part 8. spherical viewing while looping
 * over the real part 9. Zooming in while looping over the imaginary
 * part 10. spherical viewing while looping over the imaginary part 11.
 * Original Zooming without looping 12. Original Sphere without looping
 */
if (newton) {
    if (loop) {
        if (real_part) {
            if (zoom) {
                double step = (end.getRe() - start.getRe()) / (loopIter - 1);
                for (w = 0; w < loopIter; w++) {
                    this.alpha = start.add(new Complex(w * step, 0));
                    newtonEx();
                }
            } else {//do not zoom
                System.out.println("Looping over Newton without Zooming is not yet implemented
            }
        } else {//imaginary part
            if (zoom) {
                for (w = 0; w < loopIter; w++) {
                    double step = (end.getIm() - start.getIm()) / (loopIter - 1);
                    this.alpha = start.add(new Complex(0, w * step));
                    newtonEx();
                }
            } else {//do not zoom
                System.out.println("Looping over Newton without Zooming is not yet implemented
            }
        }
    } else {//do not Loop
        if (zoom) {
            newtonEx();
        } else {//do not zoom
            System.out.println("Spherical viewing of Newton is not yet implemented");
        }
    }
} else {//not Newton
    if (loop) {
        if (real_part) {
            if (zoom) {
                double step = (end.getRe() - start.getRe()) / (loopIter - 1);
                for (w = 0; w < loopIter; w++) {
                    this.alpha = start.add(new Complex(w * step, 0));
                    run2();
                }
            } else {//do not zoom
                double step = (end.getRe() - start.getRe()) / (loopIter - 1);
                for (w = 0; w < loopIter; w++) {
                    this.alpha = start.add(new Complex(w * step, 0));
                    run();
                }
            }
```

```
            } else {//imaginary part
                if (zoom) {
                    for (w = 0; w < loopIter; w++) {
                        double step = (end.getIm() - start.getIm()) / (loopIter - 1);
                        this.alpha = start.add(new Complex(0, w * step));
                        run2();
                    }
                } else {//do not zoom
                    double step = (end.getIm() - start.getIm()) / (loopIter - 1);
                    for (w = 0; w < loopIter; w++) {
                        this.alpha = start.add(new Complex(0, w * step));
                        run();
                    }
                }
            }
        } else {//do not Loop
            if (zoom) {
                run2();
            } else {//do not zoom
                run();
            }
        }
    }
}
/*
 * A method to run when you want to view the spherical version
 */

private void run() {

    //Create 9 PrintWriters for outputting the 9 matrices
    PrintWriter[] myOuts = new PrintWriter[9];

    //Initialize the 9 PrintWriters
    //For example, the very first files would have the path:
    //    home/f10/pghardy/Desktop/MATLAB/MATLAB_CONVERT/0_0
    for (int mat = 0; mat < 9; mat++) {
        try {
            myOuts[mat] = new PrintWriter(new BufferedWriter(new FileWriter(path + mat + "_" + w))
        } catch (Exception e) {
            System.out.println(e);
        }
    }
    B1_1 = new int[dim + 1][dim + 1]; //REDNESS //B = uint8(round(A * 255)); //Lower Hemisphere
    B2_1 = new int[dim + 1][dim + 1]; //GREENESS                             //Lower Hemisphere
    B3_1 = new int[dim + 1][dim + 1]; //BLUENESS                             //Lower Hemisphere
    B1_2 = new int[dim + 1][dim + 1]; //REDNESS //B = uint8(round(A * 255)); //Upper Hemisphere
    B2_2 = new int[dim + 1][dim + 1]; //GREENESS                             //Upper Hemisphere
    B3_2 = new int[dim + 1][dim + 1]; //BLUENESS                             //Upper Hemisphere
    x = new double[dim + 1][dim + 1];
    y = new double[dim + 1][dim + 1];
    z = new double[dim + 1][dim + 1];
```

```
//An array for temporarily holding the radius of the poitns
double[] temp = new double[dim + 1];

//An array for holding the angle of the point (both are polar coordinates)
double[] tempAngles = new double[dim + 1];

//This loop initializes the points to be tested so they will be evenly spaced once on the sphe
for (int t = 0; t <= dim; t++) {
    temp[t] = (Math.cos((-(Math.PI) * t) / (2.0 * dim)) / (1 - Math.sin((-(Math.PI) * t) / (2.
    tempAngles[t] = (t * 2 * Math.PI) / dim;
}

//A 'matrix' to hold the x-values to be tested
double[][] outputX = new double[tempAngles.length][temp.length];
//A 'matrix' to hold the y-values to be tested
double[][] outputY = new double[tempAngles.length][temp.length];

//This loop converts the points to cartesian coordinates
for (int m = 0; m < tempAngles.length; m++) {
    for (int n = 0; n < temp.length; n++) {
        outputX[m][n] = temp[m] * Math.cos(tempAngles[n]);
        outputY[m][n] = temp[m] * Math.sin(tempAngles[n]);
    }
}

//This is where the colors are made. When o=1, we are doing the southern hemishpere
for (int o = 1; o <= 2; o++) { //for o=1:2

    //v and w are simply used to traverse outputX and outputY
    //v corresponds to the row and w is the column
    for (int v = 0; v <= dim; v++) { //for v=1:dim+1
        for (int w = 0; w <= dim; w++) { //for w = 1:dim+1

            //xf is created using the coordinates we already created
            Complex xf = new Complex(outputX[v][w], outputY[v][w]); //xf = complex(X(v,w),Y(v,

            //For the northern hemishpere, we basically take the reciprocal of the points that
            //evenly distributed between zero and the equator
            if (o == 2) { //if (o == 2)
                if (xf.equals(ZERO)) { //if (xf == 0)
                    break;
                }
                xf = new Complex(equator, 0).mul((new Complex(equator, 0).div(xf)).getConjugat
            }

            Boolean plusSign = newPoint(xf);

            if (plusSign == null) {
                if (o == 1) {
                    B1_1[v][w] = 0; //Red
                    B2_1[v][w] = 0; //Green
                    B3_1[v][w] = 0; //Blue
                } else {
```

```
                    B1_2[v][w] = 0; //Red
                    B2_2[v][w] = 0; //Green
                    B3_2[v][w] = 0; //Blue
                }
            } else if (plusSign) {
                if (o == 1) {
                    B1_1[v][w] = 255; //Red
                    B2_1[v][w] = 0; //Green
                    B3_1[v][w] = 0; //Blue
                } else {
                    B1_2[v][w] = 255; //Red
                    B2_2[v][w] = 0; //Green
                    B3_2[v][w] = 0; //Blue
                }
            } else {//minus sign chosen
                if (o == 1) {
                    B1_1[v][w] = 0; //Red
                    B2_1[v][w] = 0; //Green
                    B3_1[v][w] = 255; //Blue
                } else {
                    B1_2[v][w] = 0; //Red
                    B2_2[v][w] = 0; //Green
                    B3_2[v][w] = 255; //Blue
                }
            }
        }
    }
}

//These loops convert the points into the sperical coordinates
for (int sigma = 0; sigma <= dim; sigma++) {
    for (int zeta = 0; zeta <= dim; zeta++) {
        double tempX = outputX[sigma][zeta] / equator;
        double tempY = outputY[sigma][zeta] / equator;
        x[sigma][zeta] = (2 * tempX) / (1 + tempX * tempX + tempY * tempY); //x=(2*X)./(1+X.^2
        y[sigma][zeta] = (2 * tempY) / (1 + tempX * tempX + tempY * tempY); //y=(2*Y)./(1+X.^2
        z[sigma][zeta] = (tempX * tempX + tempY * tempY - 1) / (tempX * tempX + tempY * tempY
    }
}

//These 9 lines output the 9 matrices we need to construct the MATLAB surf function
outputInt(B1_1, myOuts[0]);
myOuts[0].close();
outputInt(B2_1, myOuts[1]);
myOuts[1].close();
outputInt(B3_1, myOuts[2]);
myOuts[2].close();
outputInt(B1_2, myOuts[3]);
myOuts[3].close();
outputInt(B2_2, myOuts[4]);
myOuts[4].close();
outputInt(B3_2, myOuts[5]);
myOuts[5].close();
```

```
        outputDouble(x, myOuts[6]);
        myOuts[6].close();
        outputDouble(y, myOuts[7]);
        myOuts[7].close();
        outputDoubleVer(z, myOuts[8]);
        myOuts[8].close();
    }

    /*
     * A method that is used only when wanting to zoom in on a specified area
     */
    private void run2() {

        //Create 5 PrintWriters for outputting the 5 matrices
        PrintWriter[] myOuts = new PrintWriter[5];

        //Initialize the 5 PrintWriters
        //For example, the very first files would have the path:
        //    home/f10/pghardy/Desktop/MATLAB/MATLAB_CONVERT/ZOOM_0_0
        for (int mat = 0; mat < 5; mat++) {
            try {
                myOuts[mat] = new PrintWriter(new FileWriter(path + "ZOOM_" + mat + "_" + w));
            } catch (Exception e) {
                System.out.println(e);
            }

        }
        B1_1 = new int[ZOOM_DIM + 1][ZOOM_DIM + 1]; //REDNESS //B = uint8(round(A * 255));
        B2_1 = new int[ZOOM_DIM + 1][ZOOM_DIM + 1]; //GREENESS
        B3_1 = new int[ZOOM_DIM + 1][ZOOM_DIM + 1]; //BLUENESS

        //A temporary array to hold the x-values to be tested
        double[] Xtemp = new double[ZOOM_DIM + 1];
        //A temporary array to hold the y-values to be tested
        double[] Ytemp = new double[ZOOM_DIM + 1];

        //A 'matrix' of all the x-values of points to be tested
        x = new double[ZOOM_DIM + 1][ZOOM_DIM + 1];
        //A 'matrix' of all the y-values of points to be tested
        y = new double[ZOOM_DIM + 1][ZOOM_DIM + 1];

        //This breaks up the change in x and the change in y into ZOOM_DIM pieces
        for (int i = 0; i <= ZOOM_DIM; i++) {
            Xtemp[i] = X1 + ((X2 - X1) * i) / (ZOOM_DIM); //real parts of the points to test
            Ytemp[i] = Y1 + ((Y2 - Y1) * i) / (ZOOM_DIM); //imaginary parts of the points to test
        }

        //This loop places all the combinations together. If Xtemp = [1 2] and Ytemp=[3 4],
        //then x = [1 2] and y = [3 3] and now all four points are matched up.
        //         [1 2]         [4 4]
        for (int m = 0; m <= ZOOM_DIM; m++) {
            for (int n = 0; n <= ZOOM_DIM; n++) {
                x[m][n] = Xtemp[n];
```

```
            y[m][n] = Ytemp[m];
        }
    }

    //v and w are simply used to traverse outputX and outputY
    //v corresponds to the row and w is the column
    for (int v = 0; v <= ZOOM_DIM; v++) {
        for (int w = 0; w <= ZOOM_DIM; w++) {

            //xf is created using the coordinates we already created
            Complex xf = new Complex(x[v][w], y[v][w]);

            //xs becomes the new point that xf goes to under the Hansen-Patrick method
            Boolean plusSign = newPoint(xf);

            //If it is NOT close enough, then rootIndex==null and we try again
            //If it is close enough we color that point according to the root that was
            //found and the number of iterations (k) it took to converge
            if (plusSign == null) {
                B1_1[v][w] = 0; //Red
                B2_1[v][w] = 0; //Green
                B3_1[v][w] = 0; //Blue
                //break;
            } else if (plusSign) {
                B1_1[v][w] = 255; //Red
                B2_1[v][w] = 0; //Green
                B3_1[v][w] = 0; //Blue
            } else {//minus sign chosen
                B1_1[v][w] = 0; //Red
                B2_1[v][w] = 0; //Green
                B3_1[v][w] = 255; //Blue
            }
        }
    }
    //These 5 lines output the 5 matrices we need to construct the MATLAB surf function
    outputInt(B1_1, myOuts[0]);
    myOuts[0].close();
    outputInt(B2_1, myOuts[1]);
    myOuts[1].close();
    outputInt(B3_1, myOuts[2]);
    myOuts[2].close();
    outputDouble(x, myOuts[3]);
    myOuts[3].close();
    outputDouble(y, myOuts[4]);
    myOuts[4].close();
}

/*
 * The bulk of the program which computes the new point of the iteration
 */
private Boolean newPoint(Complex oldPoint) {

    Boolean plusSign = null;
```

```
//A Complex holder for the next guess
//Complex newPoint;

//The value of the function evaluated at the old point
Complex z = c.evaluate(oldPoint); //z = polyval(c, oldPoint);

//The value of the derivative evaluated at the old point
Complex zp = c_der.evaluate(oldPoint); //zp = polyval(c_der, oldPoint);

//The value of the second derivative evaluated at the old point
Complex zpp = c_der2.evaluate(oldPoint); //zpp = polyval(c_der2, oldPoint);

//The top of the fractional part of the Hansen-Patrick method
Complex top = (alpha.add(new Complex(1, 0))).mul(z); //top = (a+1)*z;

//The square root portion in the denominator
Complex sq_root = (zp.mul(zp).sub(top.mul(zpp))).sq_root(); //sq_root = sqrt(zp^2-top*zpp);

//The value of the bottom if the plus sign is used
Complex bottom1 = (alpha.mul(zp)).add(sq_root); //bottom1 = a*z+sq_root;

//The value of the bottom if the minus sign is used
Complex bottom2 = (alpha.mul(zp)).sub(sq_root); //bottom2 = a*z-sq_root;

//The absolute value of the plus version of the bottom
double bottom1Abs = bottom1.modulus(); //bottom1Abs = abs(bottom1);

//The absolute value of the minus version of the bottom
double bottom2Abs = bottom2.modulus(); //bottom2Abs = abs(bottom2);

if (alpha.getRe() >= 0) {
    if (bottom1Abs == 0 && bottom2Abs == 0) {
        //newPoint = oldPoint;
        plusSign = null;
    } else if (bottom1Abs == bottom2Abs) {
        if ((zp.getConjugate().mul(bottom1)).getRe() > 0) {
            //newPoint = oldPoint.sub(top.div(bottom1)); //newPoint = oldPoint - top/bottom1;
            plusSign = true;
        } else {
            //newPoint = oldPoint.sub(top.div(bottom2)); //newPoint = oldPoint - top/bottom2;
            plusSign = false;
        }
    } else if (bottom1Abs > bottom2Abs) { //elseif(bottom1Abs > bottom2Abs)
        //newPoint = oldPoint.sub(top.div(bottom1)); //newPoint = oldPoint - top/bottom1;
        plusSign = true;
    } else {
        //newPoint = oldPoint.sub(top.div(bottom2)); //newPoint = oldPoint - top/bottom2;
        plusSign = false;
    }
} else if (alpha.getRe() != -1) {
    if (bottom1Abs == 0 && bottom2Abs == 0) {
        //newPoint = oldPoint;
        plusSign = null;
```

```
            } else if (bottom1Abs >= bottom2Abs) { //elseif(bottom1Abs > bottom2Abs)
                //newPoint = oldPoint.sub(top.div(bottom2)); //newPoint = oldPoint - top/bottom1;
                plusSign = false;
            } else {
                //newPoint = oldPoint.sub(top.div(bottom1)); //newPoint = oldPoint - top/bottom2;
                plusSign = true;
            }

        }
//        else {//alpha.getRe()==-1 //Halley's Method
//            top = new Complex(2, 0).mul(z).mul(zp);
//            bottom1 = (new Complex(2, 0).mul(zp).mul(zp)).sub(z.mul(zpp));
//            if (bottom1.equals(ZERO)) {
//                newPoint = oldPoint;
//            }
//            else{
//                newPoint = oldPoint.sub(top.div(bottom1));
//            }
//        }
        return plusSign;
    }

    private void newtonEx() {
    }

    /*
     * A method for outputting files of the form int[][]
     */
    public void outputInt(int[][] ob, PrintWriter pw) {
        for (int i = 0; i < ob.length; i++) {
            for (int j = 0; j < ob[i].length; j++) {
                pw.print(ob[i][j] + " ");
            }
            pw.println();
        }
        pw.close();
    }

    /*
     * A method for outputting files of the form double[][]
     */
    public void outputDouble(double[][] ob, PrintWriter pw) {
        for (int i = 0; i < ob.length; i++) {
            for (int j = 0; j < ob[i].length; j++) {
                pw.print(ob[i][j] + " ");
            }
            pw.println();
        }
        pw.close();
    }

    public void outputDoubleVer(double[][] ob, PrintWriter pw) {
        for (int i = 0; i < ob.length; i++) {
```

```
            pw.println(ob[i][0] + " ");
        }
        pw.close();
    }

    /*
     * A method for printing out and array of the form int[][]
     */
    public void printArrayInt(int[][] ob) {
        for (int i = 0; i < ob.length; i++) {
            for (int j = 0; j < ob[i].length; j++) {
                System.out.print(ob[i][j] + " ");
            }
            System.out.println();
        }
        System.out.println();
        System.out.println();
    }

    /*
     * A method for printing out and array of the form double[][]
     */
    public void printArrayDouble(double[][] ob) {
        for (int i = 0; i < ob.length; i++) {
            for (int j = 0; j < ob[i].length; j++) {
                System.out.print(ob[i][j] + " ");
            }
            System.out.println();
        }
        System.out.println();
        System.out.println();
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
    }
}
```

APPENDIX 7: Java template

```
/**
 *
 * @author pghardy
 */
public class temp {


    public static void main(String[] args){
        ComplexPolynomial top1 = new ComplexPolynomial(new Complex(1,0),3);
        ComplexPolynomial top2 = new ComplexPolynomial(new Complex(-1,0),0);
        ComplexPolynomial bottom1 = new ComplexPolynomial(new Complex(1,0),3);
        ComplexPolynomial bottom2 = new ComplexPolynomial(new Complex(1,0),0);

        RationalFunction rf = new RationalFunction(top1.plus(top2), bottom1.plus(bottom2));
        System.out.println(rf.differentiate());
        System.out.println(rf.differentiate().differentiate());
    }
}
```

APPENDIX 8: Build in MATLAB

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- You may freely edit this file. See commented blocks below for -->
<!-- some examples of how to customize the build. -->
<!-- (If you delete it and reopen the project it will be recreated.) -->
<!-- By default, only the Clean and Build commands use this build script. -->
<!-- Commands such as Run, Debug, and Test only use this build script if -->
<!-- the Compile on Save feature is turned off for the project. -->
<!-- You can turn off the Compile on Save (or Deploy on Save) setting -->
<!-- in the project's Project Properties dialog box.-->
<project name="summer2012" default="default" basedir=".">
    <description>Builds, tests, and runs the project summer2012.</description>
    <import file="nbproject/build-impl.xml"/>
    <!--

    There exist several targets which are by default empty and which can be
    used for execution of your tasks. These targets are usually executed
    before and after some main targets. They are:

      -pre-init:                 called before initialization of project properties
      -post-init:                called after initialization of project properties
      -pre-compile:              called before javac compilation
      -post-compile:             called after javac compilation
      -pre-compile-single:       called before javac compilation of single file
      -post-compile-single:      called after javac compilation of single file
      -pre-compile-test:         called before javac compilation of JUnit tests
      -post-compile-test:        called after javac compilation of JUnit tests
      -pre-compile-test-single:  called before javac compilation of single JUnit test
      -post-compile-test-single: called after javac compilation of single JUunit test
      -pre-jar:                  called before JAR building
      -post-jar:                 called after JAR building
      -post-clean:               called after cleaning build products

    (Targets beginning with '-' are not intended to be called on their own.)

    Example of inserting an obfuscator after compilation could look like this:

        <target name="-post-compile">
            <obfuscate>
                <fileset dir="${build.classes.dir}"/>
            </obfuscate>
        </target>

    For list of available properties check the imported
    nbproject/build-impl.xml file.


    Another way to customize the build is by overriding existing main targets.
    The targets of interest are:

      -init-macrodef-javac:     defines macro for javac compilation
      -init-macrodef-junit:     defines macro for junit execution
```

```
-init-macrodef-debug:     defines macro for class debugging
-init-macrodef-java:      defines macro for class execution
-do-jar-with-manifest:    JAR building (if you are using a manifest)
-do-jar-without-manifest: JAR building (if you are not using a manifest)
run:                      execution of project
-javadoc-build:           Javadoc generation
test-report:              JUnit report generation
```

An example of overriding the target for project execution could look like this:

```
<target name="run" depends="summer2012-impl.jar">
    <exec dir="bin" executable="launcher.exe">
        <arg file="${dist.jar}"/>
    </exec>
</target>
```

Notice that the overridden target depends on the jar target and not only on the compile target as the regular run target does. Again, for a list of available properties which you can use, check the target you are overriding in the nbproject/build-impl.xml file.

APPENDIX 9: Scanner in MATLAB

```
function C = SCANNER()

B1_1 = load('
F_CONVERT_0_myTest');
B2_1 = load('MATLAB_CONVERT_1_myTest');
B3_1 = load('MATLAB_CONVERT_2_myTest');
B1_2 = load('MATLAB_CONVERT_3_myTest');
B2_2 = load('MATLAB_CONVERT_4_myTest');
B3_2 = load('MATLAB_CONVERT_5_myTest');
x = load('MATLAB_CONVERT_6_myTest');
y = load('MATLAB_CONVERT_7_myTest');
z = load('MATLAB_CONVERT_8_myTest');

A = zeros(size(x));
B1 = uint8(round(A * 255));
B2 = uint8(round(A * 255));

B1(:,:,1) = B1_1;
B1(:,:,2) = B2_1;
B1(:,:,3) = B3_1;

B2(:,:,1) = B1_2;
B2(:,:,2) = B2_2;
B2(:,:,3) = B3_2;

surf(x,y,z,B1,'FaceColor','texturemap'), shading flat, hold on
surf(x,y,-z,B2,'FaceColor','texturemap'), shading flat, hold off
```

APPENDIX 10: Magnitude method plot in MATLAB

```
\newpage
\noindent{\bf \small APPENDIX 1}
\begin{verbatim}
function test(alpha,z0,tol)
%function test(alpha,z0,tol)
%
maxtol=10.0;
x=real(z0);
y=imag(z0);
plot(x,y)
hold on
pnum = 1+alpha-z0^3+2*alpha*z0^3+(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
pden = 3*alpha*z0^2+(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
pz1 = pnum/pden;
mnum = 1+alpha-z0^3+2*alpha*z0^3-(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
mden = 3*alpha*z0^2-(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
mz1 = mnum/mden;
pdis = min(abs(pz1-z0),abs(mz1-z0));
ndis = max(abs(pz1-z0),abs(mz1-z0));
if real(alpha)>=0;
    while pdis > tol
        if abs(pz1-z0)<=abs(mz1-z0)
            x=[real(z0),real(pz1)];
            y=[imag(z0),imag(pz1)];
            plot(x,y,'-b',x,y,'ok')
            title('Magnitude Method')
            z0 = pz1;
            pnum = 1+alpha-z0^3+2*alpha*z0^3+(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            pden = 3*alpha*z0^2+(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            pz1 = pnum/pden;

            display(angle(z0));
        else
            x=[real(z0),real(mz1)];
            y=[imag(z0),imag(mz1)];
            plot(x,y,'-r',x,y,'ok')
            title('Magnitude Method')
            z0 = mz1;
            mnum = 1+alpha-z0^3+2*alpha*z0^3-(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            mden = 3*alpha*z0^2-(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            mz1 = mnum/mden;

            display(angle(z0));
        end
        pdis = min(abs(pz1-z0),abs(mz1-z0));
    end
else
    while ndis > tol
        if abs(pz1-z0)>abs(mz1-z0)
            x=[real(z0),real(pz1)];
```

```
            y=[imag(z0),imag(pz1)];
            plot(x,y,'-b',x,y,'ok')
            title('Magnitude Method')
            z0 = pz1;
            pnum = 1+alpha-z0^3+2*alpha*z0^3+(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            pden = 3*alpha*z0^2+(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            pz1 = pnum/pden;

            display(angle(z0));
        else
            x=[real(z0),real(mz1)];
            y=[imag(z0),imag(mz1)];
            plot(x,y,'-r',x,y,'ok')
            title('Magnitude Method')
            z0 = mz1;
            mnum = 1+alpha-z0^3+2*alpha*z0^3-(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            mden = 3*alpha*z0^2-(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            mz1 = mnum/mden;

            display(angle(z0));
        end
        ndis = max(abs(pz1-z0),abs(mz1-z0));
        if ndis>maxtol
            break
        end

    end
end
hold off
```

APPENDIX 11: Argument method plot in MATLAB

```
function arg(alpha,z0,rad)
%function argchoose(alpha,z0,rad)
%
maxtol=10.0;
x=real(z0);
y=imag(z0);
plot(x,y)
hold on
pnum = 1+alpha-z0^3+2*alpha*z0^3+(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
pden = 3*alpha*z0^2+(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
pz1 = pnum/pden;
mnum = 1+alpha-z0^3+2*alpha*z0^3-(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
mden = 3*alpha*z0^2-(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
mz1 = mnum/mden;
parg = min(abs(angle(pz1)-angle(z0)),abs(angle(mz1)-angle(z0)));
narg = max(abs(angle(pz1)-angle(z0)),abs(angle(mz1)-angle(z0)));
if real(alpha)>=0;
    while parg > rad
        if abs(angle(pz1)-angle(z0))<=abs(angle(mz1)-angle(z0));
            x=[real(z0),real(pz1)];
            y=[imag(z0),imag(pz1)];
            plot(x,y,'-b',x,y,'ok')
            title('Argument Method')
            z0 = pz1;
            pnum = 1+alpha-z0^3+2*alpha*z0^3+(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            pden = 3*alpha*z0^2+(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            pz1 = pnum/pden;

            display(angle(z0));
        else
            x=[real(z0),real(mz1)];
            y=[imag(z0),imag(mz1)];
            plot(x,y,'-r',x,y,'ok')
            title('Argument Method')
            z0 = mz1;
            mnum = 1+alpha-z0^3+2*alpha*z0^3-(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            mden = 3*alpha*z0^2-(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            mz1 = mnum/mden;

            display(angle(z0));
        end
        parg = min(abs(angle(pz1)-angle(z0)),abs(angle(mz1)-angle(z0)));
    end
else
    while narg > rad
        if abs(angle(pz1)-angle(z0))>abs(angle(mz1)-angle(z0));
            x=[real(z0),real(pz1)];
            y=[imag(z0),imag(pz1)];
            plot(x,y,'-b',x,y,'ok')
            title('Argument Method')
            z0 = pz1;
```

```
            pnum = 1+alpha-z0^3+2*alpha*z0^3+(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            pden = 3*alpha*z0^2+(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            pz1 = pnum/pden;

            display(angle(z0));
        else
            x=[real(z0),real(mz1)];
            y=[imag(z0),imag(mz1)];
            plot(x,y,'-r',x,y,'ok')
            title('Argument Method')
            z0 = mz1;
            mnum = 1+alpha-z0^3+2*alpha*z0^3-(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            mden = 3*alpha*z0^2-(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            mz1 = mnum/mden;

            display(angle(z0));
        end
        narg = max(abs(angle(pz1)-angle(z0)),abs(angle(mz1)-angle(z0)));
        if narg>maxtol
            break
        end

    end
end
hold off
```

APPENDIX 12: Dot product method plot in MATLAB

```
function dot(alpha,z0,tol)
%function dot(alpha,z0,tol)
%
maxtol=10.0;
x=real(z0);
y=imag(z0);
plot(x,y)
hold on
pnum = 1+alpha-z0^3+2*alpha*z0^3+(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
pden = 3*alpha*z0^2+(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
pz1 = pnum/pden;
mnum = 1+alpha-z0^3+2*alpha*z0^3-(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
mden = 3*alpha*z0^2-(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
mz1 = mnum/mden;
dot = (3*z0)^2*sqrt((9*z0^4)-(alpha+1)*((6*z0^4)-6));
pdis = min(abs(pz1-z0),abs(mz1-z0));

    while pdis > tol
        if dot >= 0;
            x=[real(z0),real(pz1)];
            y=[imag(z0),imag(pz1)];
            plot(x,y,'-b',x,y,'ok')
            title('Dot Method')
            z0 = pz1
            pnum = 1+alpha-z0^3+2*alpha*z0^3+(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            pden = 3*alpha*z0^2+(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            pz1 = pnum/pden;

        else
            x=[real(z0),real(mz1)];
            y=[imag(z0),imag(mz1)];
            plot(x,y,'-r',x,y,'ok')
            title('Dot Method')
            z0 = mz1
            mnum = 1+alpha-z0^3+2*alpha*z0^3-(z0*sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            mden = 3*alpha*z0^2-(sqrt(3)*sqrt(-z0*(-2-2*alpha-z0^3+2*alpha*z0^3)));
            mz1 = mnum/mden;

        end
        pdis = min(abs(pz1-z0),abs(mz1-z0));
    end

hold off
```