2012

Visualizing Chaos

Andrew Nicklawsky

Visualizing Chaos

Andrew Nicklawsky

Honors Thesis

Department of Numerical Computation

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

Dr. Robert Hesse, Department of Mathematics

Dr. Mike Heroux, Department of Computer Science

Dr. Thomas Sibley, Department of Mathematics

April 2012

X

Dr. Robert Hesse
Associate Professor of Mathematics

Project Advisor

X

Dr. Lynn Ziegler
Professor of Computer Science

Department Chair

X

Dr. Mike Heroux
Scientist in Residence

Department Reader

X

Dr. Thomas Sibley
Professor of Mathematics

Department Reader

X

Dr. Richard White
Professor of Chemistry

Director, Honors Thesis Program

Introduction

An important piece of information when dealing with a polynomial in the complex domain is its roots, the value or values of x for a given function f such that $f(x) = 0$. One uses iterative root finding methods, such as Newton's method, to discover an approximate value when these values cannot be explicitly solved. This iterative process can be graphically represented for complex-valued functions and has been achieved with relative ease on a 2-dimensional plane. The resulting image shows basins of attraction and the fractals that result. A basin of attraction shows the collection of points that converge to a certain root. This picture is useful but limited by the constraints of the size of the plane. However, this process could also be embodied on a sphere through the method of stereographic projection. By projecting the entire complex plane onto a sphere, one can completely visualize the dynamics, discovering the extent and location of basins of attraction. Envisaging these basin maps is an important tool in understanding the subtleties of chaotic dynamical systems on the complex plane. Therefore, I set out to write one of the first programs that would stereographically project these images onto a sphere to fully visualize the images created on the complex plane.

To further investigate the images of these basin maps I coded several programs. In this research, I utilized Newton and Halley's iterative root-finding methods on the complex plane. Upon mapping out their iterations, I wrote a program to stereographically project the resulting images upon a sphere. I also created a program for the general form as derived by Eldon Hansen and Merrell Patrick, which captures a family of iterative functions. Through my work I created a way to examine the entire picture and discover the influences of adjustment in parameters.

Background

As stated before, an important part of mathematics is discovering the roots of polynomials, where f(x) = 0. Often these polynomials can be exceedingly complicated and are difficult to solve explicitly. This problem can be overcome with the utilization of iterative root finding methods. These methods use an initial "guess" to predict what the root of a function is. Because the methods are iterative, their limits are expected to be the desired root. There are many different methods that can be utilized to achieve this. Initially, I chose to work with two of the most popular, Newton-Raphson, as well as Halley's, before moving onto a general form.

The Newton-Raphson form is one of the most commonly taught for both its simplicity and precision. It is based upon the principle of successive linearization, the technique that takes non-linear equations and replaces them with successive linear problems whose solution is that of the non-linear (**Kopecky**). First, one begins with an initial guess, $_0$, and applies it to the function:

N ($_0$) = $$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)},$$

This is sustained through the general form:

N ($_0$) = $$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Each iteration should come closer to the desired root. This is achieved because essentially with each step, one is solving for the x-intercept of each subsequent tangent line, with the idea that this is a better approximation of the root. This method is useful because of its quadratic

convergence rate, or convergence with order two, meaning it reaches its limit at a relatively quick speed, therefore fewer iterations are required. Due to the nature of the function's definition, it can only be used in the case where the function is differentiable. Newton's method is the first in the class of methods known as the Householder's methods, which are defined by their use on functions of one variable with a continuous derivative of a certain order, which is also the order of the method.

Halley's method is second in this class because it can be used on functions that possess a continuous second derivative, thus making it an improvement on Newton's Method. Halley's method is faster than Newton's for determining approximate roots because it has cubic convergence. Halley's method can be derived from Newton's method.

First consider the equation:

$$g(x) = \frac{f(x)}{\sqrt{|f'(x)|}}.$$

Taking the first derivative we get:

$$g'(x) = \frac{2[f'(x)]^2 - f(x)f''(x)}{2f'(x)\sqrt{|f'(x)|}},$$

Subsequently plugging this into Newton's results in:

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

Halley's method provides more latitude to work with when dealing with polynomials because it is a second order equation, although it is constrained by the fact that one cannot have f'(c) = 0. These two basic methods, with Newton's being a limiting case, can both be derived from the overall general Patrick-Hansen form of the Householder's methods.

The general form derived by Patrick-Hansen is as follows:

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

It is based upon the parameter α that can be adjusted to obtain the other methods. For instance, it can be shown that Newton's method can be derived when α approaches infinity and Halley's when α equals negative one. Halley's is easier to see when rewriting the general form as:

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

Which then can be seen to be Halley's when α equals negative one:

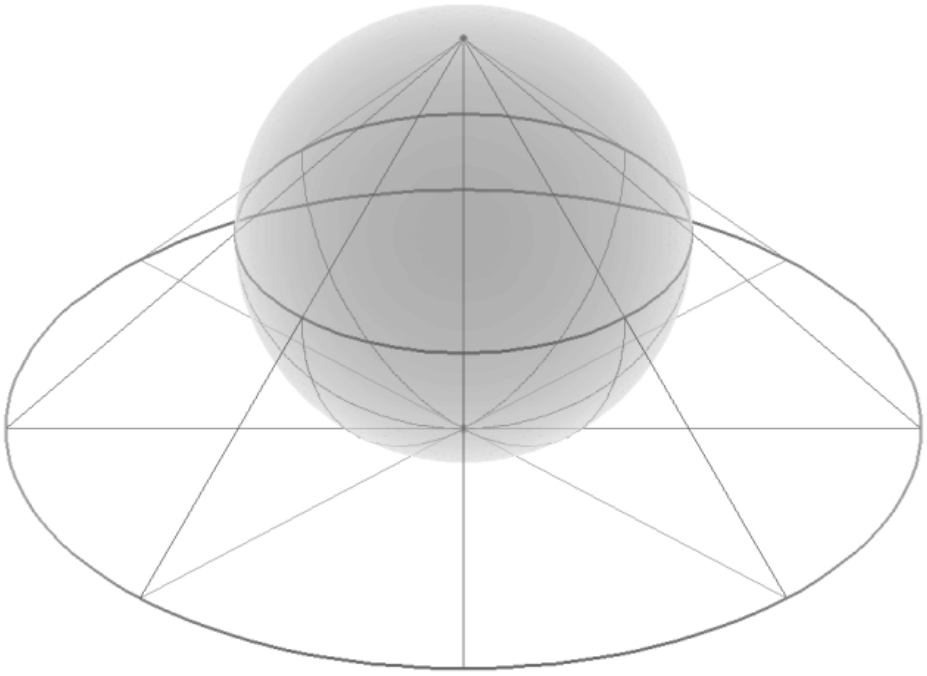
$$z' = z - \frac{f}{f' - \frac{ff''}{2f'}}$$

The Patrick-Hansen form allows for capturing both values of the square root by the use of the plus/minus in the denominator. This is a result of the derivation of the method since the result is computed from a quadratic method. Patrick-Hansen's general form captures methods that do not involve derivatives higher than the second order, thus the highest rate of convergence that can be obtained is cubic. In addition, no value of α can minimize the asymptotic error constant for the general method, so no singular form of it is asymptotically best for each function (**Hansen 259**).

This general form is useful when one wants to observe the discrepancies between models in how fast certain points converge, as well as show how they are all fundamentally related.

The basins of attraction are displayed on the complex plane through images that possess fractals. Fractals are defined as "a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole" (**Mandelbrot**). The repetitive nature of a point's convergence, in that multiple points share the same rate, leads to the display of iterative methods resulting in fractals. Because of the sensitive nature of the root finding functions, minor changes will result in drastic differences, thus classifying their resulting images as manifestations of Julia Sets. Julia Sets are the set of points on the boundary of Fatou Sets. Fatou Sets tend towards a finite value whereas Julia Sets, which are part of its complement, are either periodic or do not converge to anything, including the point at infinity (**Devaney**). These images provide a useful glimpse into the pattern of convergence, but are limited by the window size of the complex plane. Utilizing such a small frame of reference does not allow the viewer to grasp the full extent of the basins of attraction. This limited scope is problematic because it only presents a narrow view of a set of points. This setback can be ameliorated through the use of stereographic projection.

Stereographic projection is utilized to project interchangeably between a sphere and a plane. Every point can be projected, barring the actual projection point itself. Due to this factor, the north pole is often chosen as the projection point.



(**Howison**)

A stereographic projection is bijective, meaning that there are exact pairings for the two sets of points, and preserves all angles between points. Unfortunately, it does not reflect all of the area of a plane unless certain measures are taken, which will be described later. The domain for the plane is (X,Y, 0) meaning z = 0, and the codomain is The transformations defining the projection of the XY-plane onto the sphere are as follows:

For Cartesian coordinates:

$$(x, y, z) = \left(\frac{2X}{1 + X^2 + Y^2}, \frac{2Y}{1 + X^2 + Y^2}, \frac{-1 + X^2 + Y^2}{1 + X^2 + Y^2} \right).$$

For polar (cylindrical):

$$(r, \theta, z) = \left(\frac{2R}{1 + R^2}, \Theta, \frac{R^2 - 1}{R^2 + 1} \right).$$

These two projections cause the origin, (0,0), to become the point on the south pole, (0,0, -1), the interior of the unit circle to map to the Southern Hemisphere, and the unit circle to fall upon the equator. The north pole, (0,0,1), is undefined but can be considered the manifestation of infinity as the points growing closer to it come from points in the plane increasingly distant from the origin.

Preliminary Work

The initial work for this project was done by exploration with Mathematica. Using the built in commands to create fractals, I began to conceptualize what I believed to be a relatively straightforward model in Java. The creation of 2-D pictures seemed attainable.

So, the first direction this project took was to implement Newton's method utilizing Java. Coming from a Computer Science background, this appeared to be the ideal setup. Newton's method could be easily executed through a *for* loop and multiple classes could be implemented to create an overall structure that would be very conducive to generating fractals. Since considerable work had already been done with fractals on the complex plane, I began searching online for resources that would have code that I could use for a backbone. I experimented with code, such as the following:

```
fdashxold = appro_derivative(xold, select_delta);
xnew = xold - (fxold/fdashxold);
absfxnew = func(xnew);
```

```
if(absfxnew < 0)
{
absfxnew = -absfxnew;
}
xold = xnew;
```

(http://www.java-forums.org/advanced-java/10989-solution-newton-raphson-method.html)

This code was simple in that it executed the exact workings of Newton's Method. Working with Java, I was able to create code that calculated the number of iterations it took for individual guesses to converge. I also discovered more complex code that visualized basins of attraction, but it was very difficult to read and had limited capability. Thus I decided to explore other options.

This preliminary experimentation taught me several lessons; most importantly, it was easy to keep track of how many iterations a single point would take to converge. However, this would prove to be exponentially more difficult to calculate and store for a larger data set. Another lesson I learned was how difficult it would be to distinguish the different roots when displaying them graphically. Because of the immense data sets, I explored several options such as parallel computing and restricting my data set. Due to its limitations, I ultimately decided Java was not the best option. Fortunately, in my online research, I found a Matlab code that I could utilize.

Matlab immediately became an attractive option for two reasons: The code I discovered created a Newtonian fractal image in the complex plane based upon the input parameters of different functions, and Matlab had built-in commands to realize my project in the third dimension.

The first step was experimentation with the base code itself to explore the parameters. As it stood, the basic code would take a predefined function and map out its basins of attraction with confining parameters such as the maximum number of iterations and resolution as seen in Figure 1. Each basin of attraction was assigned a different color, with different shades of it signifying the number of iterations a point took to reach a root.

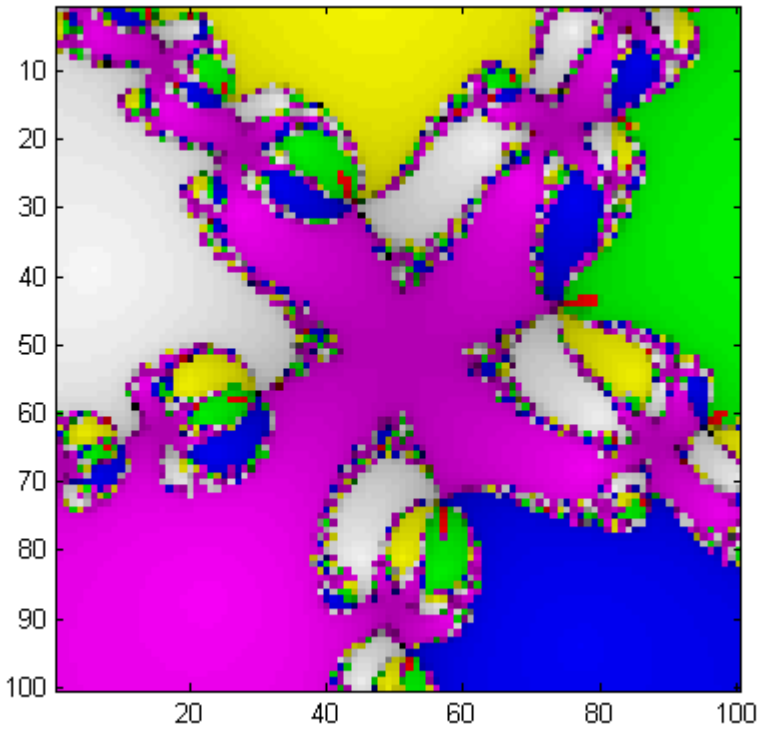


Figure 1

The following code demonstrates the initial Newton's method. Note: *polyval* is a Matlab function that returns the first polynomial argument, c, evaluated at the following argument.

```
z = polyval(c, xf);
zp = polyval(c_der, xf);
xs = xf - z/zp;
```

The calculations were broken up to expedite processing. A cap was placed upon the maximum number of iterations and resolution to make sure the program would not overwhelm a typical computer's processor. Some initial values may result in infinite iterations, which would overload a computer's memory. I began to manipulate the code to implement Halley's Method, as well as make the code itself more intuitive. This involved the implementation of calculations for the second derivative as well as an expanded method within the primary *for* loop:

```
z = polyval(c, xf);
zp = polyval(c_der, xf);
zpp = polyval(c_der2, xf);
top = 2*conv(z,zp);
bottom = 2*conv(zp,zp)-conv(z,zpp);
if (bottom == 0)
   xs = xf;
else
   xs = xf - deconv(top,bottom);
end
```

(*conv* is the Matlab command for polynomial multiplication)

The basic structure of Newton's method remained, but an *if* statement was created to catch the case where the calculation of the denominator of the equation resulted in zero. I also created shortcuts, in the form of additional input parameters, to allow the program to be executed from the command window with the previous choices.

Matlab once again proved invaluable as an ideal medium by providing commands that significantly reduced computation time. For instance, utilizing the *polyval* command created a shortcut in the evaluation of polynomials while *deconv* solved the division of the subsequent products. One of the first issues to arise was the matter of direction. Matlab was orienting the planar picture in the reverse direction. Scripting some code that transformed the matrix that

contained the individual color-coded points solved this problem. The following code shows this change in the matrix B, which held the data points for the final image, rotating each section ninety degrees.

```
B(:,:,1)=rot90(B(:,:,1));
B(:,:,2)=rot90(B(:,:,2));
B(:,:,3)=rot90(B(:,:,3));
```

I then coded Patrick-Hansen's general form. This involved further modification of the interior methods to accommodate for differing parameters, the main one being α, as well as whether to utilize a positive or negative root.

```
z = polyval(c, xf);
zp = polyval(c_der, xf);
zpp = polyval(c_der2, xf);
top = (a+1)*z;
sq_root = sqrt(zp^2-top*zpp);
bottom_dot = dot(zp,sq_root);
bottom1 = a*z+sq_root;
bottom2 = a*z-sq_root;
if (bottom_dot == 0)
   xs = xf;
elseif real(dot(zp,bottom1))>0
   xs = xf - deconv(top,bottom1);
else
   xs = xf - deconv(top,bottom2);
```

I broke up all calculations to allow for faster processing, utilizing methods that Matlab already has coded. Using the modified code as a framework allowed me to focus on the 3-D representation of the results.

To achieve a 3-D representation of the results, I would make use of the matrix output of the current version code. The code created a matrix that stored the resulting data point's color

based upon the number of iterations needed to reach a root. Thus, it appeared to be a simple matter of translating that matrix into 3-D coordinates. The *surf* command became the primary instrument through which this was realized where: "*surf* (X,Y,Z,C) creates a shaded surface, with color defined by {the matrix} C. MATLAB performs a linear transformation on this data to obtain colors from the current colormap" (Matlab). Some of the first issues I ran into were the use of the expressions within the commands themselves. It took several days to figure out what combinations of commands within *surf* would result in a clear picture. Many different coding options existed that would alter the viewing of the axis, the orientation of the sphere, and how the points were to be projected upon the sphere. Without the correct sequence, the subsequent image would become distorted into wave-like patterns and shapeless forms, or would not be displayed at all. With that figured out, the next step was to execute the stereographic transformation.

Stereographic projection involved coding several lines that took the values in my resultant matrix, which stored the colors, and transformed them into the correct 3-dimensional points. Initially, there was some confusion based on whether the program was consolidating the data points into vectors or matrices. This was an issue because the algebra involved needed the data to be strictly one or the other. Eventually I solved this issue when I deduced that matrices were being used. The next major issue was the question of how to create both hemispheres.

Adjusting the positive or negative orientation of the Z coordinates individually could create each hemisphere. When placing these two hemispheres together, though, large gaps appeared on the equator because of the nature of stereographic projection.

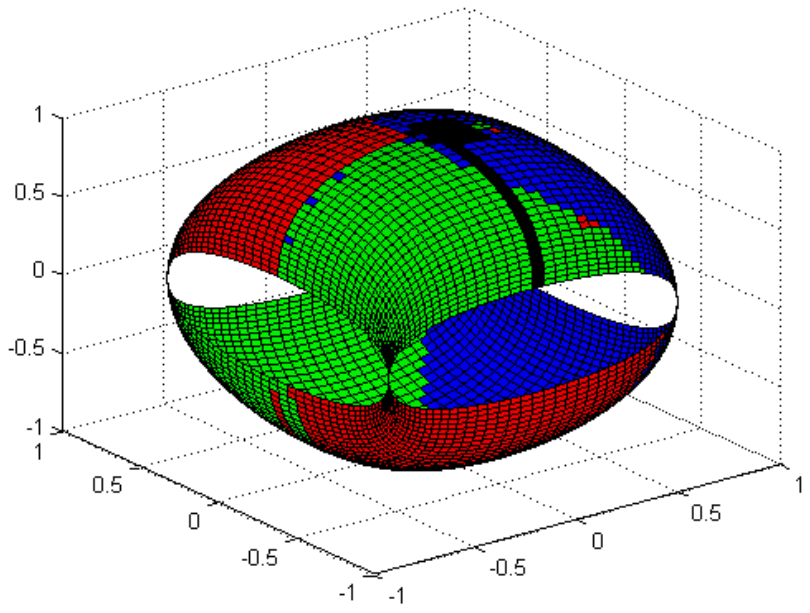


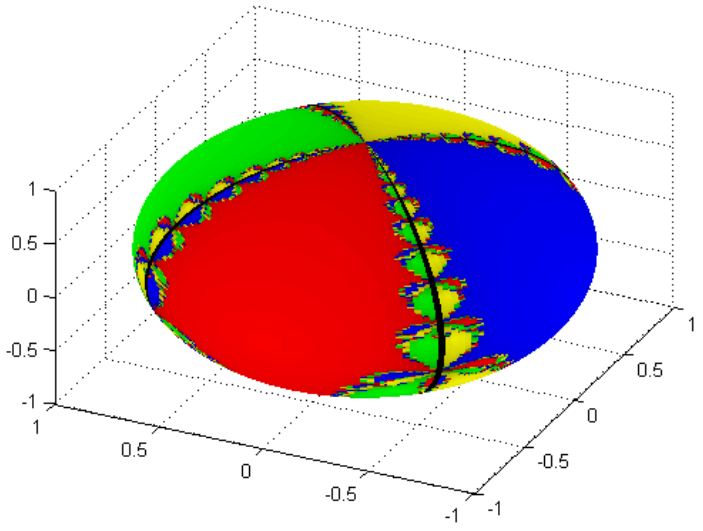
Figure 2

The complex images I was attempting to project were square areas. Translating them onto a sphere mapped them correctly, but neither hemisphere captured the appropriate points along the equator. The corners of the squares met, but their sides were curved, creating the open spaces (see Figure 2). It was immediately apparent that this would have to be done through the use of polar coordinates. It became difficult to determine just exactly where the conversion from Cartesian to polar should occur, and which state the points should be in upon mapping.

The main issue was that the base code worked beautifully within its Cartesian premises and to change that would result in a massive process. Based on the way calculations were already executed, changing the code to completely polar would cause more problems than it solved. Modifying the resultant matrix at the end would do nothing since it would contain the same

problematic coordinates. It was decided that the best route would be to begin with a polar circle and translate that into Cartesian coordinates.

This solution worked perfectly, creating a continuous sphere that graphically represented all points (see Figure 3).



(Figure 3. Execution of Newton's Method Program on the equation z^3-1 with resolution at 200)

The Result

My final code, as seen starting on page 34, is a combination of the program created by Cahit Güngör, a graduate student, with my methods sutured in.

The first line of the code is intuitive. It allows a user to call the program within Matlab given the parameters of an *a* (alpha), the *dim* (resolution), and *c* (the function). The alpha determines which form of the Householder methods will result, thus c needs to be a one variable equation due to the nature of Householder methods. Line 3 establishes a matrix, colorArr, which will be used to define the different colors. It is a six by three matrix, allowing for seven different

colors as determined by values of one's and zero's in the three columns. As it stands, a function can only have seven basins of attraction, as there are only seven colors. More can be added, though, by creating a bigger colorArr matrix. The following lines up until 17 define each individual color. The command roots, in line 21, returns a vector of all the roots for the given function c, stored as rootArr. The next lines of code, 24-38, are utilized to calculate the first two derivatives of the given function. First a variable is established that will hold the vector for the derivative. A *for* loop is then utilized to establish the size of the vector of the resulting derivative in both cases. After exiting that loop the actual derivative is computed. The use of the *deconv* function speeds up the process of calculating the derivative by deconvolving the first parameter from the second through long division. The command in line 39 ensures that the given dimensions are integers. Lines 42-46 establish the range of data points. This is where I restricted the planar image to that of a circle, through the use of polar coordinates, in order to avoid creating gaps in the stereographic projection. This is accomplished by creating a *rho* that tracks through the points 0 to 1 by step size of the inverse of the dimension and a *theta* that does the same over 360 degrees. *Rho* is first established and squared to increase the increments in step size between 0 and 1. *Theta* is transformed into radians as well during the parameter's creation. The command *meshgrid* transforms these vectors into arrays, which make them compatible with the execution of the rest of the program. A warning is also established to tell Matlab to ignore if division by zero occurs. Line 49 contains the set value for the maximum number of iterations the program allows for before "painting" that value black. This constraint, although allowing the results of some data points to be missed, keeps this program from crashing through too many calculations. The matrix *A*, created in line 51, is utilized later in line 53 to generate what will be the final array of data points. *A* is necessary because it sets the size through the resolution, as

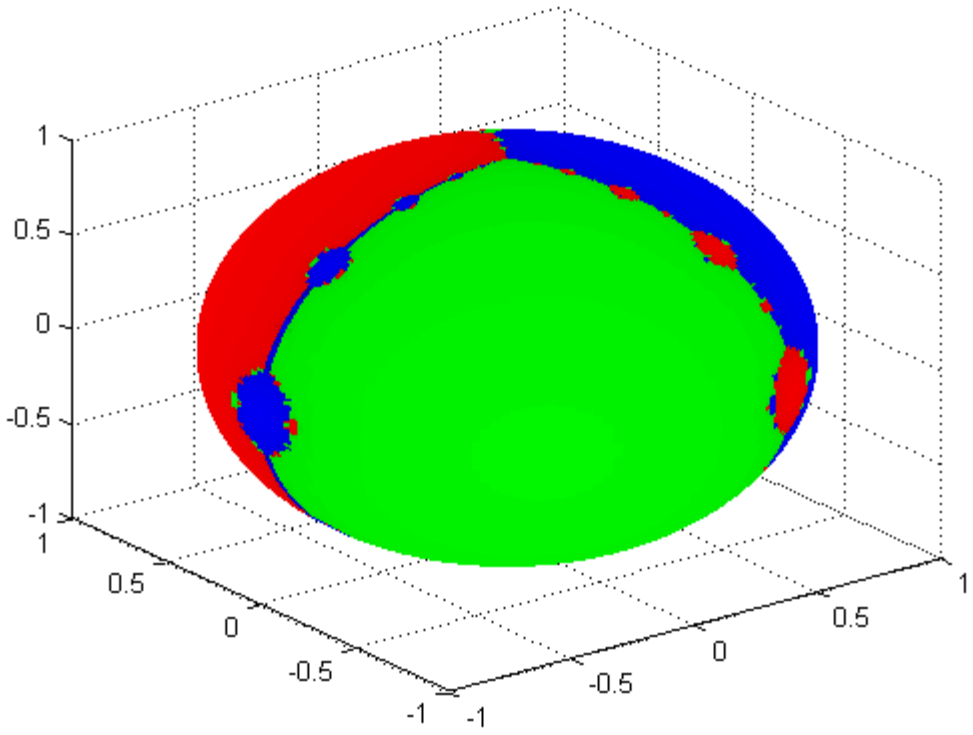
desired by the user with *dim*. It is first established as being all zeros, as data will be transferred to it later. The command *uint8* creates an array composed of 8 bit integers, which will be utilized for graphics. The *for* loop on line 54 is what creates the final sphere. The two halves are created separately, thus the *for* loop acts as a placeholder to unite them. The following two *for* loops contain the main inner workings of the program. For the size of *dim*, which is the size of the array *B*, each data point will be executed. Line 58 forms a variable, *xf*, that is a complex number, as constructed from values obtained from the arrays *X* and *Y*, as determined by the *for* loops. Lines 60-65 allow the calculation of the second half of the sphere. The *if* statement determines that the second hemisphere is being constructed and then *xf* becomes the complex conjugate. The check for the case of *xf* being 0 is also included. Patrick-Hansen's general loop is executed in lines 69-84. Using placeholders, the individual calculations are broken down so as to speed up the processing of the entire function. After the completion of the loop, the variable *tmp* is created. This variable helps to determine which root the point will tend towards. The variable *tmp* is the absolute value of a tiled array the size of the number of allotted roots for the function. In line 87 *rootIndex* is set as all the values in *tmp* that are less than the minimum difference, as established by the *find* command, which locates all values in the array that satisfy a condition. The color for the point is set accordingly, with the value of the *rootIndex* determining the color, which is saved to the matrix B. After all points have been calculated, an array is created for "z" values, while lines 103-5 perform the stereographic projection on the Cartesian coordinates saved in the arrays. The command *surf,* which creates a shaded surface as determined by the matrix *B*, is then used to create the sphere. The *for* loop determining the hemispheres is utilized for the two *surf* commands, one of them containing inverted *z* values. And thus melding the two halves of the sphere creates the sphere.

Exploration

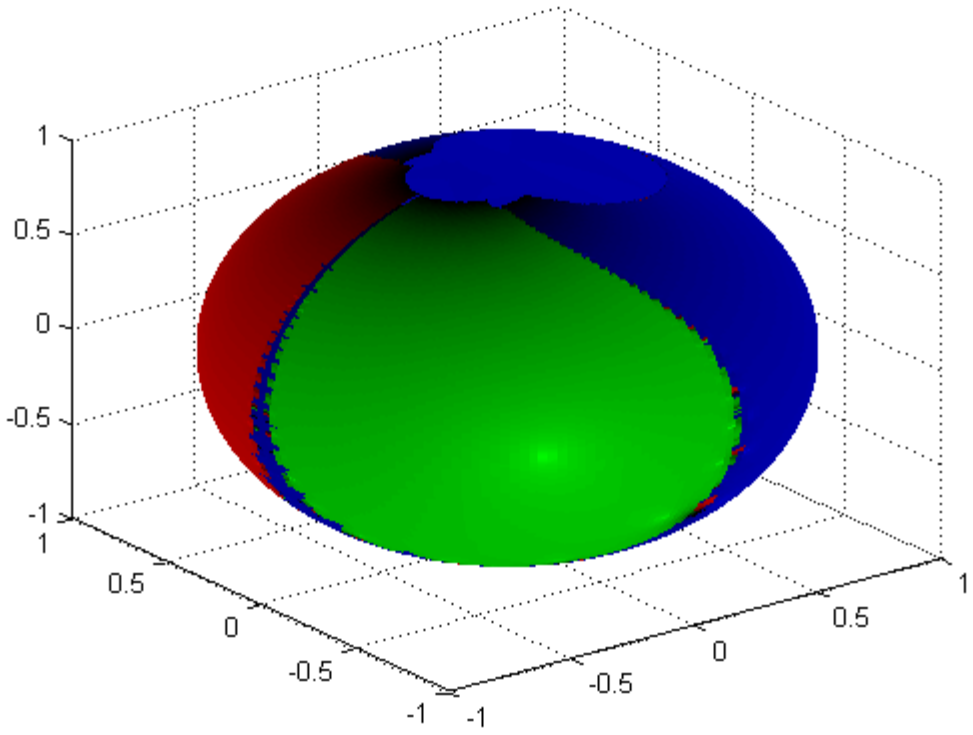
With an executable program, the next step was to explore the different effects made by adjusting parameters. Although my program had three input parameters to adjust, the primary focus was the influence of differing values of *alpha* on set exponential equations. I was interested in how a slight change in this parameter altered the stereographic images of a selected equation. The results of this investigation would show the differences within the Patrick-Hansen's family of functions in the rate in which they calculated roots, as well as if points would still tend towards the same basin. As noted in *A Family of Root Finding Methods*, certain values of *alpha* work better with different polynomials and values of *z*. Thus, I produced some images to verify these results. The outcomes of some of this exploration are as follows:

The following figures use values of *alpha* starting at -1 (Halley's Method- Figure 4) and ending at 1 by increments of 0.2 on the equation z^3-1 with resolution set at 200. Two well-known special cases, Laguerre's (Figure 15) and Newton's (Figure 16) method are also included. Laguerre's method is derived from *alpha* equaling the inverse of the degree of the polynomial minus 1. So in this case, since I used a 3rd degree equation, $\alpha = \frac{1}{3-1} = \frac{1}{2}$. Newton's method is *alpha* equaling infinity. Through this sequence one can examine the effect different values of *alpha* have on rates of convergence. The first noticeable difference is the step between Halley's method and -0.8 (Figure 5). The brightness in color, especially around the roots, indicates that Halley's is superior in calculating a point's convergence. The sizes of the basins of attraction are also changed between the two. In Halley's they are all approximately equal whereas when *alpha* equals -0.8, one grows very large, engulfing the other two. The size of the basins becomes more equalized and the brightness increases as alpha approaches 0 (Figure 9), which is another

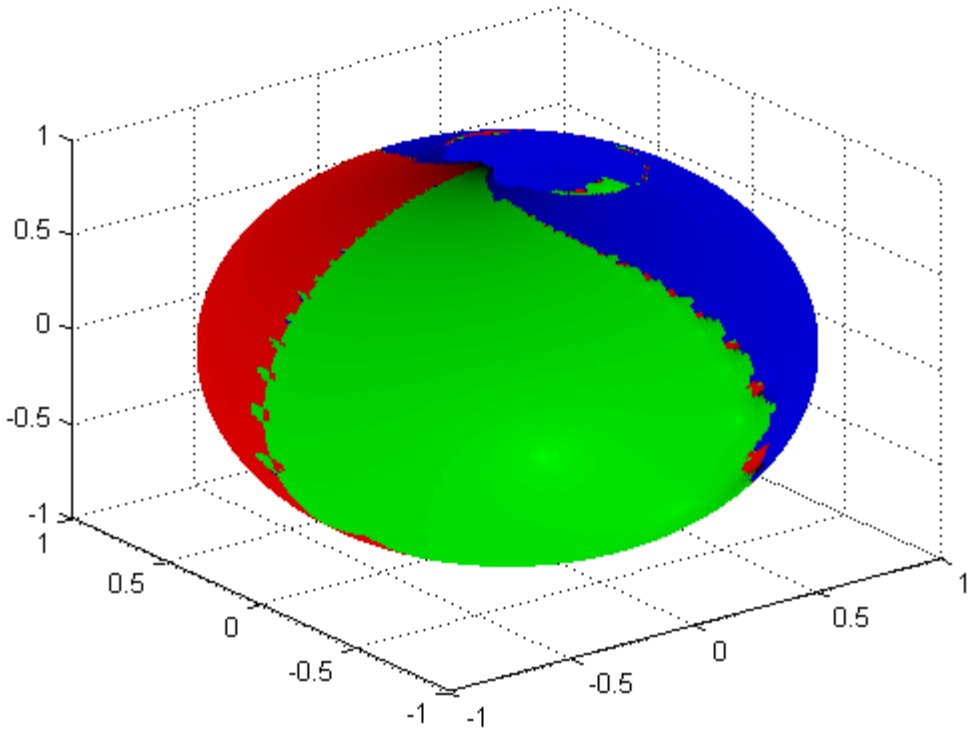
Householder method known as Ostrowski's Square Root Iteration. By 0, the rates of convergence appear roughly equivalent to those seen in Halley's, although there are fewer fractals along the borders of the basins. As alpha continues to increase, more fractals appear and the colors dim. This is indicative that once again rates of convergence are severely slowing. This is quite evident when by 0.8 (Figure 13) large masses of black have appeared, indicating that these points have not even converged within the limit of 60 iterations exercised in the program. With the next step, *alpha* equaling 1 (Figure 14), which is Euler's Method, the entire sphere basically becomes black. At this point, subsequent increases in *alpha* fail to cause points to converge within 60 iterations. However, taking the special case of Newton's method, one can see that acceptable rates of convergence occur and at almost as fast of rate as Halley's. This would appear to run contrary to pattern, as increased values of *alpha* seemed to result in worse rates of convergence and Newton's method is the limit method as *alpha* goes to infinity. This reversal could be due to the simplification of the actual equation itself manifested in Newton's method. Laguerre's Method appears similar to 0.4 and 0.6, as expected.



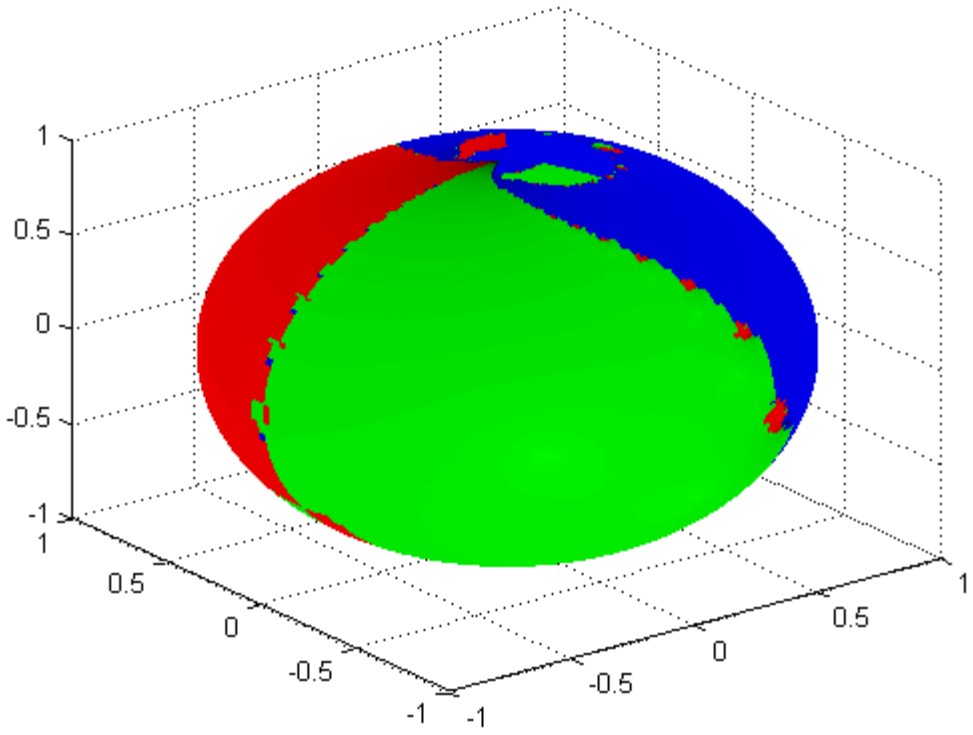
Halley's Method – Figure 4



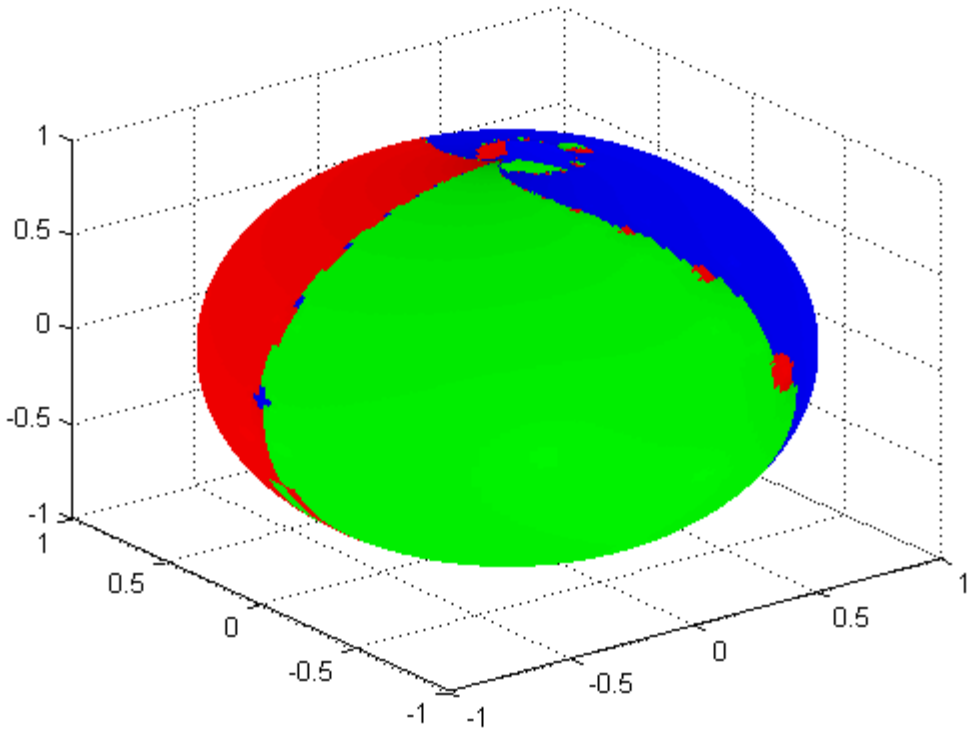
-0.8 – Figure 5



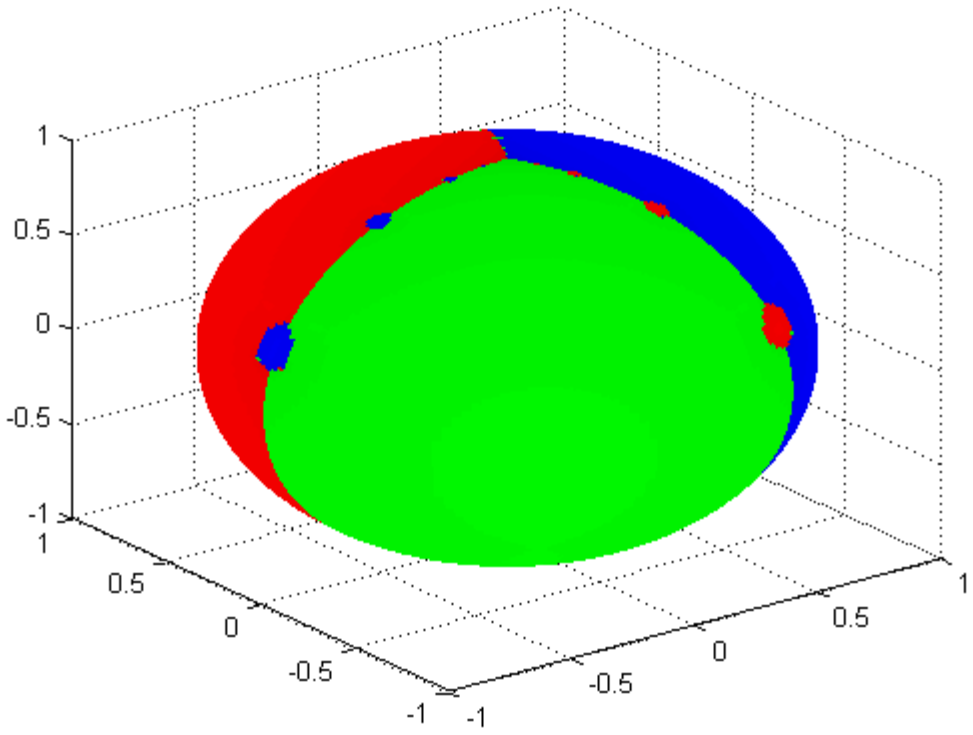
-0.6 – Figure 6



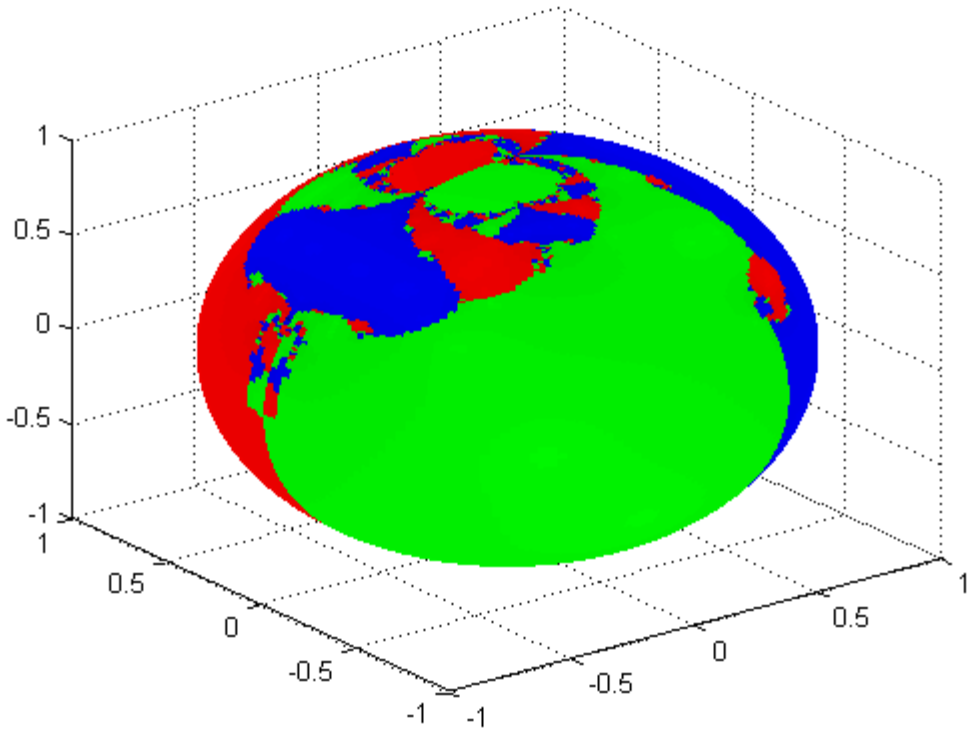
-0.4 – Figure 7



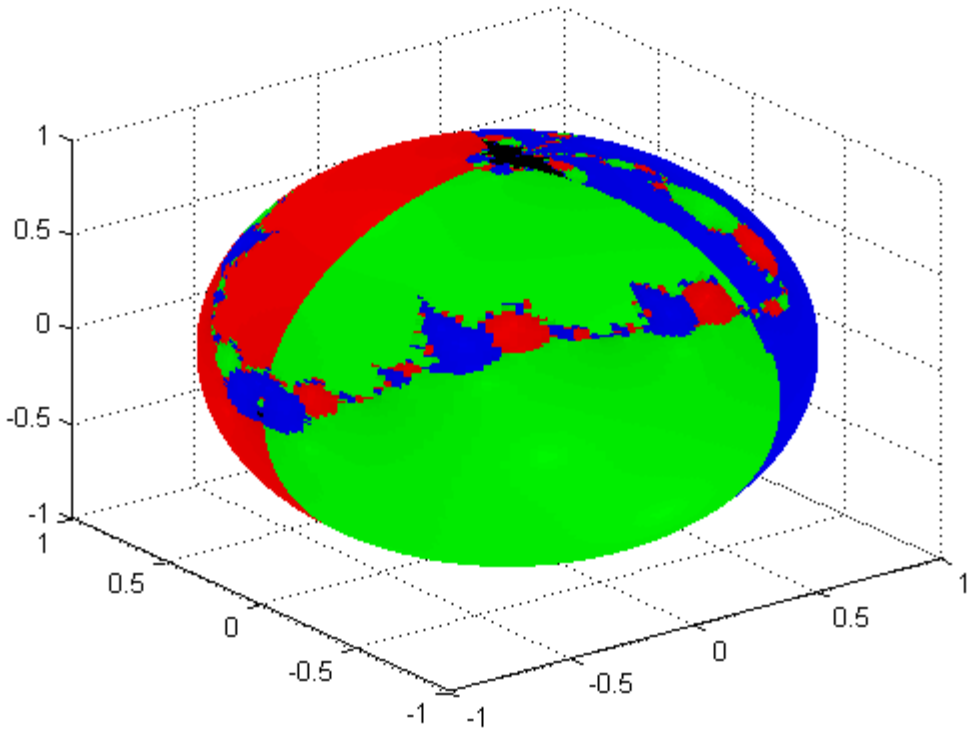
-0.2 – Figure 8



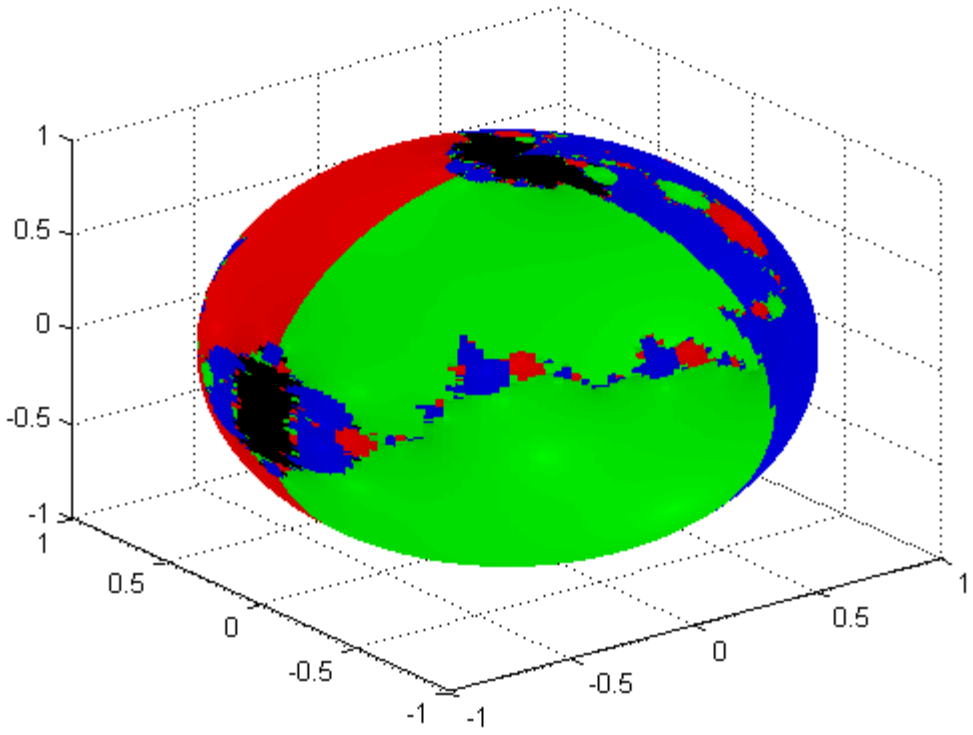
Ostrowski's Square Root Iteration (0.0) – Figure 9



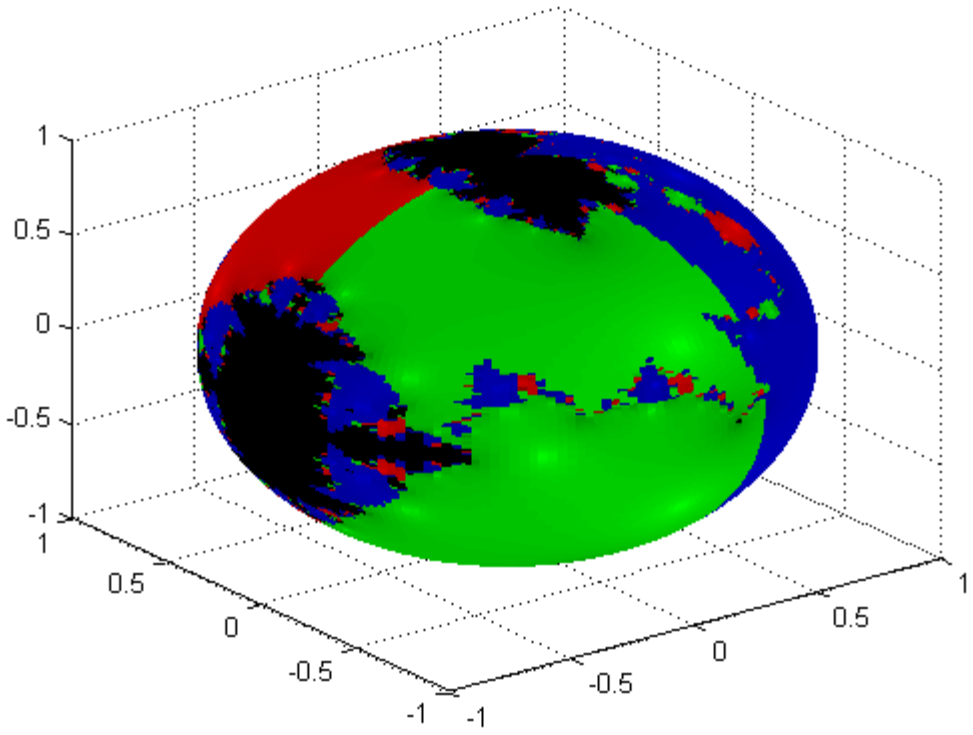
0.2 – Figure 10



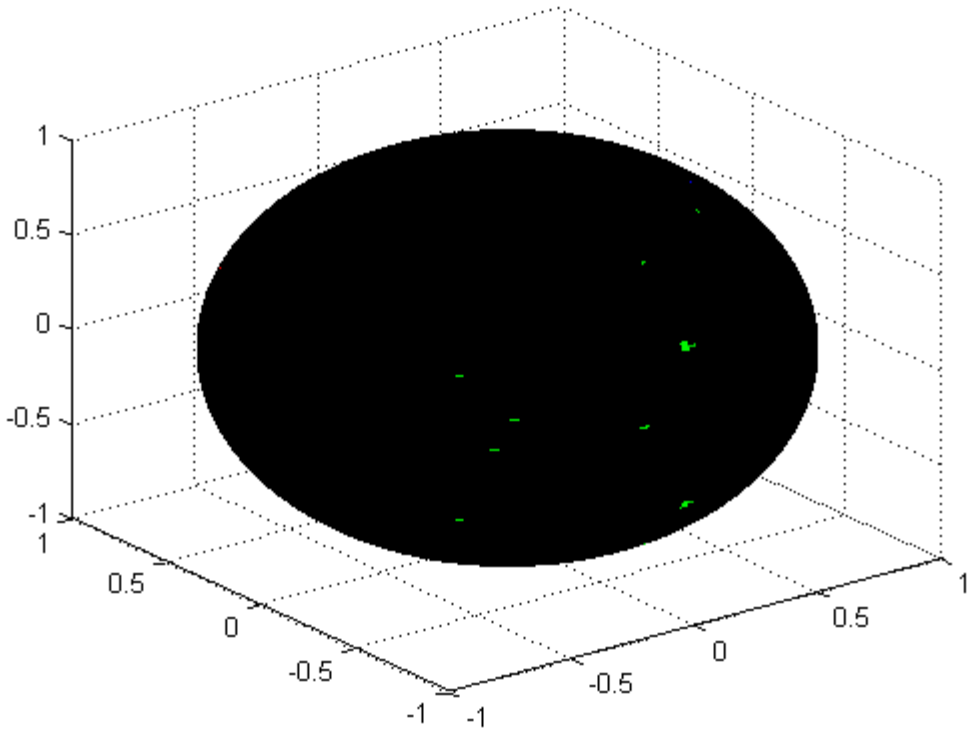
0.4 – Figure 11



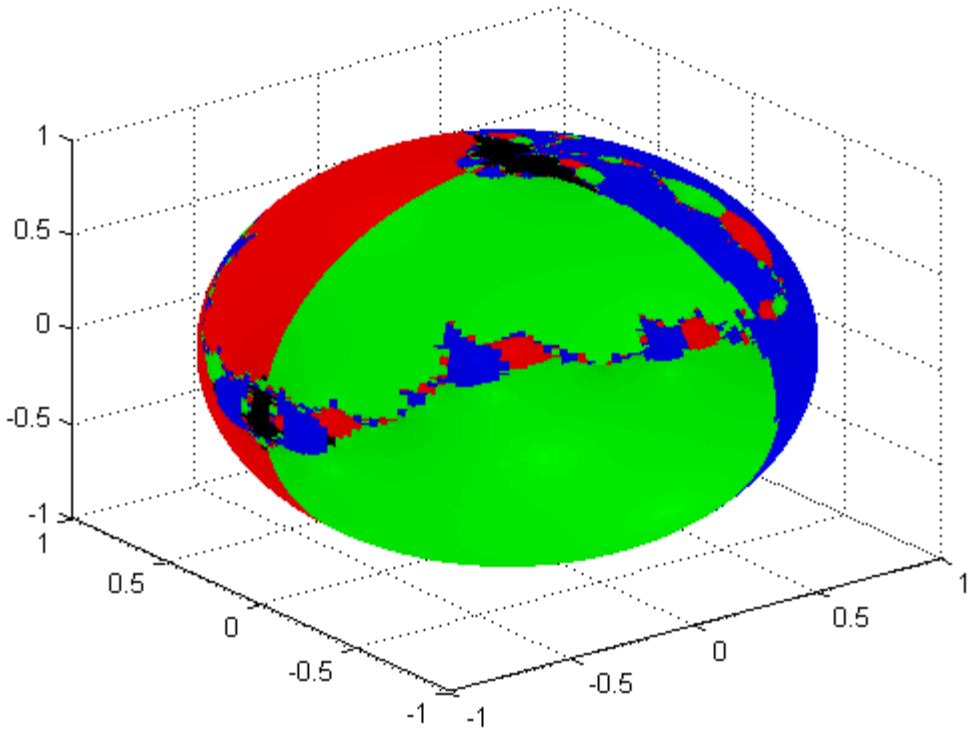
0.6 – Figure 12



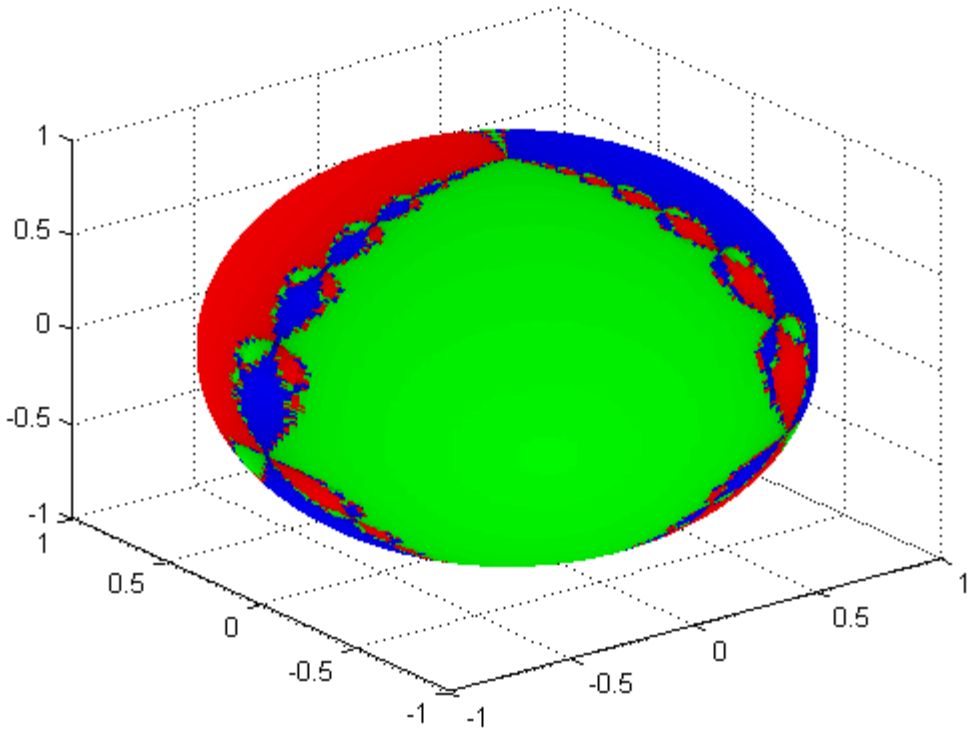
0.8 – Figure 13



Euler's Method – Figure 14



Laguerre's Method – Figure 15



Newton's Method – Figure 16

Conclusion

It is not often in math that we get to visualize the entire picture of a problem. The ability to represent these iterative methods in the third dimension is a very powerful tool. I did not know if implementing stereographic projection would give a good overall picture or if I would be able to get enough points with large enough |z| so that there would not be a gap at infinity. However, with these spherical representations I have been able to graphically show what occurs at all points on the complex plane. This information is quite valuable as it allows for a visualization of which method is best for certain equations.

These images can be used to help determine which method is best utilized for differing values of z, as well as complete pictures of the basins of attraction dependent on method. One can visualize Patrick-Hansen's theory that Laguerre's method is the best for large values of both positive and negative z. Further investigation can be applied to the series of fractals created by different values of alpha and the effect they have on the size of the basins of attraction.

Although the pictures themselves are limited by processing power and specific data from each point, I believe this could be obtained in a reduced nature with more formidable computers. With greater processing power, further exploration of these fascinating images is possible.

```
function tp2(a,dim,c)
%clear all;
colorArr = [6,3];
%RED
colorArr(1,1) = 1;colorArr(1,2) = 0;colorArr(1,3) = 0;
%GREEN
colorArr(2,1) = 0;colorArr(2,2) = 1;colorArr(2,3) = 0;
%BLUE
colorArr(3,1) = 0;colorArr(3,2) = 0;colorArr(3,3) = 1;
%YELLOW
colorArr(4,1) = 1;colorArr(4,2) = 1;colorArr(4,3) = 0;
%MAGENTA
colorArr(5,1) = 0;colorArr(5,2) = 1;colorArr(5,3) = 1;
%PURPLE
colorArr(5,1) = 1;colorArr(5,2) = 0;colorArr(5,3) = 1;
%GRAY
colorArr(6,1) = 1;colorArr(6,2) = 1;colorArr(6,3) = 1;

% matrix representation of this function
% c = [1 0 0 -1];
rootArr = roots(c);

%derivative of the function
c_der = [];
for i = 1:length(c)-1
c_der(i)=(length(c)-i)*c(i);
end
[fdivfp, rem] = deconv(c, c_der);
while rem(1) == 0
rem = rem(2:end);
end

%second derivative of the function
c_der2 = [];
for i = 1:length(c_der)-1
c_der2(i)=(length(c_der)-i)*c_der(i);
end
[fdivfp, rem] = deconv(c_der, c_der2);
 dim = round(dim);

%variable settings for the complex guesses
rt=(0:1/dim:1).^2;
[th,r] = meshgrid((0:360/dim:360)*pi/180,rt);
[X,Y]=pol2cart(th,r);
warning off MATLAB:divideByZero
min_differ = 0.01;

```

```
%maximum iteration
iter = 60;

A = zeros(dim+1, dim+1);
%RGB representation
B = uint8(round(A * 255));
for o=1:2
for v=1:dim+1
for w = 1:dim+1

xf = complex(X(v,w),Y(v,w));

if (o == 2)
    if (xf == 0)
    break;
    end
 xf =conj(1/xf);
end

xs = xf + eps + 1;
% General loop
for k=1:iter
z = polyval(c, xf);
zp = polyval(c_der, xf);
zpp = polyval(c_der2, xf);
top = (a+1)*z;
sq_root = sqrt(zp^2-top*zpp);
bottom_dot = dot(zp,sq_root);
bottom1 = a*z+sq_root;
bottom2 = a*z-sq_root;
if (bottom_dot == 0)
    xs = xf;
elseif real(dot(zp,bottom1))>0
    xs = xf - deconv(top,bottom1);
else
    xs = xf - deconv(top,bottom2);
end
tmp = abs(repmat(xf, size(rootArr))-rootArr);
%find the root associated with the one found in Newton Raphson
rootIndex = find(tmp<min_differ);
if ~isempty(rootIndex)
%color associated with roots and rate of convergence
B(v,w,1)=colorArr(rootIndex,1) * (1-(k/iter)) * 255;
B(v,w,2)=colorArr(rootIndex,2) * (1-(k/iter)) * 255;
B(v,w,3)=colorArr(rootIndex,3) * (1-(k/iter)) * 255;
break;
end
```

```
 95   xf = xs;
 96   end
 97   end
 98   end
 99
100   Z=zeros(size(X));
101
102
103   x=(2*X)./(1+X.^2+Y.^2);
104   y=(2*Y)./(1+X.^2+Y.^2);
105   z=(-1+X.^2+Y.^2)./(1+X.^2+Y.^2);
106
107   if (o == 1)
108    surf(x,y,z,B,'FaceColor','texturemap'), shading flat, hold on
109   else
110    surf(x,y,-z,B,'FaceColor','texturemap'), shading flat, hold off
111   end
112   end
```

Bibliography

Devaney, Robert L., and Bodil Branner. *Complex Dynamical Systems: The Mathematics behind the Mandelbrot and Julia Sets*. Providence, RI: American Mathematical Society, 1994. Print.

Gaston, Julia. "Mémoire sur l'iteration des fonctions rationnelles," *Journal de Mathématiques Pures et Appliquées* vol. 8. (1918): 47–245.

Gleick, James. *Chaos: Making a New Science*. 1. New York, NY:Penguin Books. 1987. Print.

Güngör, Cahit. *Fractals with Newton Raphson Method*. Thesis. Middle East Technical University, 2009. Print.

Hansen, Patrick and Eldon, Merrell. "A Family of Root Finding Methods." *Numerische Mathematik* 27. (1977): 257-269. Print.

Hurley, James. *Multivariable Calculus*. 1. Philadelphia, PA:Saunders College Publishing. 1981. Print.

Howison, Mark. This image illustrates in 3D a stereographic projection from the north pole onto a plane below the sphere. Digital image. *Wikipedia*. 13 Jan. 2008. Web. 27 Dec. 2011.

Kopecky, Karen. "Root Finding Methods." . N.p., 2007. Web. 27 Dec 2011. <http://www.karenkopecky.net/Teaching/eco613614/Notes_RootFindingMethods.pdf>.

Mandelbrot, Beloit B. *The Fractal Geometry Of Nature*. W. H. Freeman, 1983.

McGoodwin, Michael. "Julia Jewels: An Exploration of Julia Sets." *McGoodwin Family Website Home Page*. Mar. 2000. Web. 01 Jan. 2012. <http://www.mcgoodwin.net/julia/juliajewels.html>.

Van Loan, Charles. *Introduction to Scientific Computing*. 1. Upper Saddle River, NJ:Prentice-Hall, Inc. 1997. Print.