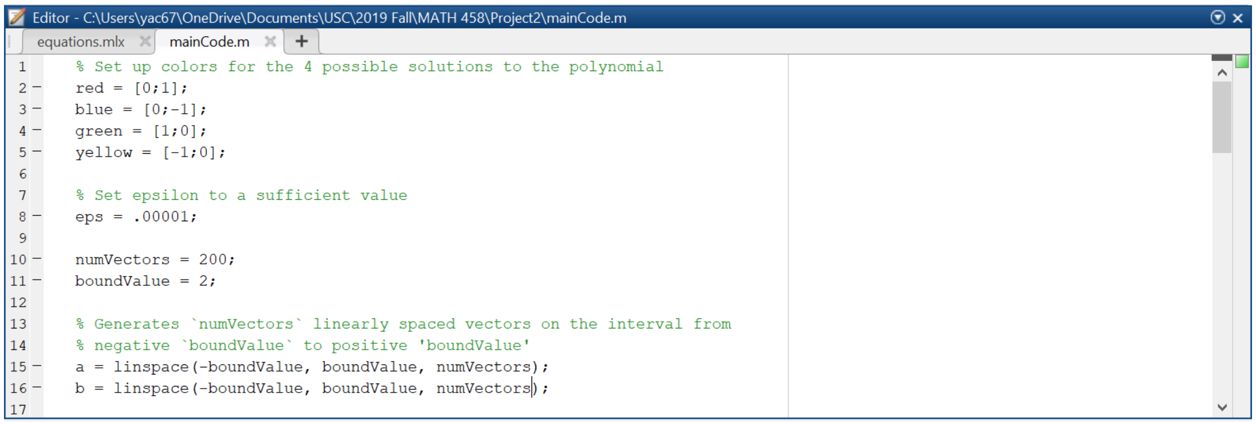
Yash Chandak

Project 2: An In-Depth Analysis of Newton's Method

Original Code

In MATLAB, I wrote the following code. The comments are particularly descriptive of each of the sections. My general approach was to generate vectors near the solutions, run Newton's method starting from that vector, and see which solution it approaches and how many iterations it takes to get within ε of a solution.

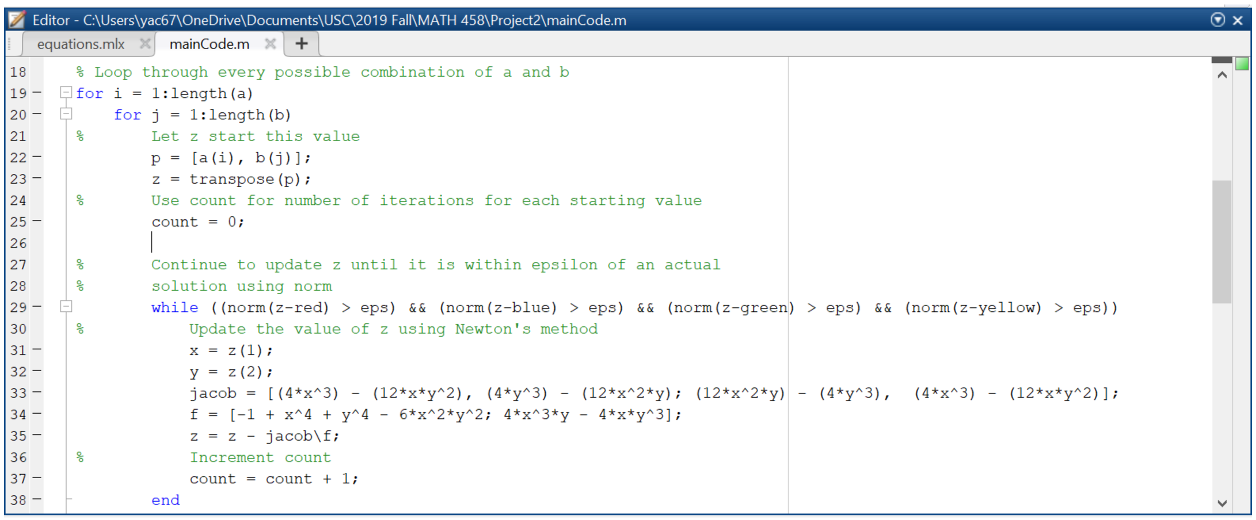

Editor - C:\Users\yac67\OneDrive\Documents\USC\2019 Fall\MATH 458\Project2\mainCode.m

equations.mlx | mainCode.m

```
1      % Set up colors for the 4 possible solutions to the polynomial
2 -    red = [0;1];
3 -    blue = [0;-1];
4 -    green = [1;0];
5 -    yellow = [-1;0];
6
7      % Set epsilon to a sufficient value
8 -    eps = .00001;
9
10 -   numVectors = 200;
11 -   boundValue = 2;
12
13     % Generates `numVectors` linearly spaced vectors on the interval from
14     % negative `boundValue` to positive 'boundValue'
15 -   a = linspace(-boundValue, boundValue, numVectors);
16 -   b = linspace(-boundValue, boundValue, numVectors);
17
```

To start, we have some set-up for Newton's method. We model the real and imaginary components of the solution as a two-dimensional vector. The four solutions to the given equation of $z^4 - 1 = 0$ are $z = \pm 1, \pm i$, hence the solutions are modeled as seen in lines 2-5. Additionally, we choose ε to be 10^{-5} as sufficient for us, seen in line 8.

When looking at all starting values for z, we can generate starting vectors with a MATLAB function called `linspace` seen in lines 15-16. This generates a set number of vectors, `numVectors`, that ranges from a positive to negative bound value, `boundValue`. In the code snippet above, we have set those values to 200 and 2, respectively, on lines 10-11.

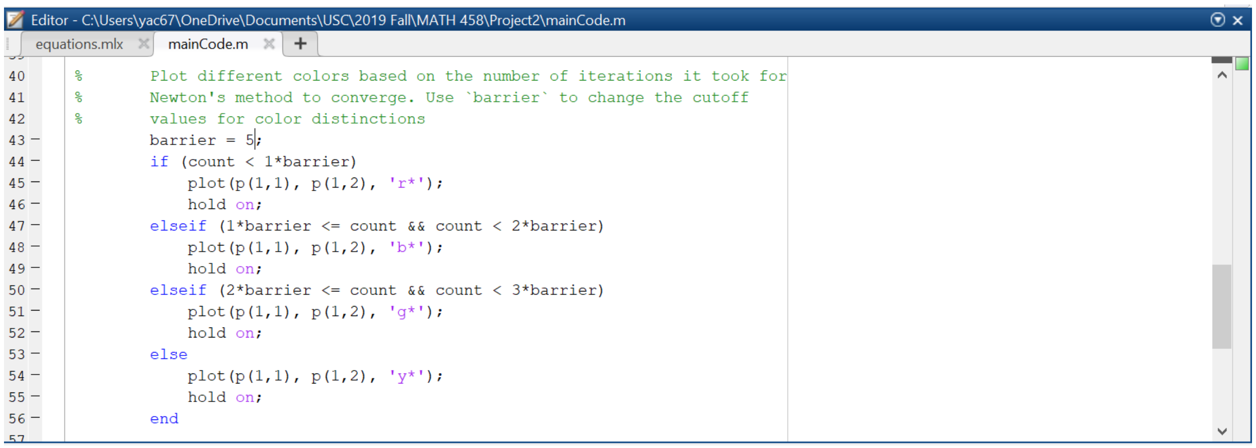

Editor - C:\Users\yac67\OneDrive\Documents\USC\2019 Fall\MATH 458\Project2\mainCode.m

equations.mlx | mainCode.m

```
18     % Loop through every possible combination of a and b
19 -  for i = 1:length(a)
20 -      for j = 1:length(b)
21     %       Let z start this value
22 -          p = [a(i), b(j)];
23 -          z = transpose(p);
24     %       Use count for number of iterations for each starting value
25 -          count = 0;
26
27     %       Continue to update z until it is within epsilon of an actual
28     %       solution using norm
29 -          while ((norm(z-red) > eps) && (norm(z-blue) > eps) && (norm(z-green) > eps) && (norm(z-yellow) > eps))
30     %           Update the value of z using Newton's method
31 -              x = z(1);
32 -              y = z(2);
33 -              jacob = [(4*x^3) - (12*x*y^2), (4*y^3) - (12*x^2*y); (12*x^2*y) - (4*y^3),  (4*x^3) - (12*x*y^2)];
34 -              f = [-1 + x^4 + y^4 - 6*x^2*y^2; 4*x^3*y - 4*x*y^3];
35 -              z = z - jacob\f;
36     %           Increment count
37 -              count = count + 1;
38 -          end
```

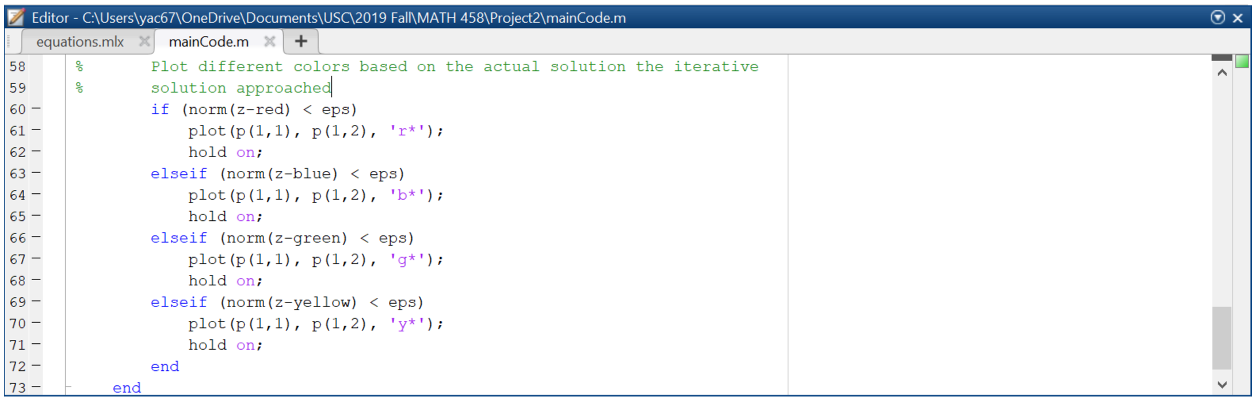
Next is our actual implementation of Newton's method in MATLAB. The `for` loops in lines 19-20 are used to test every possible starting point for z, which is set in lines 22-23. Then, the `while` loop in lines

29-38 is Newton's method. In pseudocode, it reads as follows: while z is not within epsilon of an actual solution, update the value of z using a Jacobian matrix and a matrix to simulate the function, where $z_{new} = z_{old} - inv(Jacobian(z_{old})) * function(z_{old})$ (seen in line 35). The notation uses x and y to represent the real and imaginary part of z, which is crucial to the calculations. The variable `count` will be explained later.


Editor - C:\Users\yac67\OneDrive\Documents\USC\2019 Fall\MATH 458\Project2\mainCode.m
equations.mlx
mainCode.m

```
40  %       Plot different colors based on the number of iterations it took for
41  %       Newton's method to converge. Use `barrier` to change the cutoff
42  %       values for color distinctions
43          barrier = 5;
44          if (count < 1*barrier)
45              plot(p(1,1), p(1,2), 'r*');
46              hold on;
47          elseif (1*barrier <= count && count < 2*barrier)
48              plot(p(1,1), p(1,2), 'b*');
49              hold on;
50          elseif (2*barrier <= count && count < 3*barrier)
51              plot(p(1,1), p(1,2), 'g*');
52              hold on;
53          else
54              plot(p(1,1), p(1,2), 'y*');
55              hold on;
56          end
```

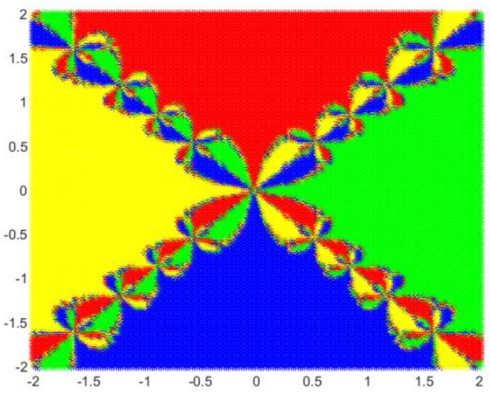
The code above is used to plot the proper color for the initial value depending on what we are plotting based on. In this plotting scheme, we are plotting a color based on the number of iterations it took for Newton's method to converge within epsilon of a solution given that starting vector. The number of iterations of Newton's method is tracked by `count`, which increments by one for each time the `while` loop is entered. As seen in line 43, we can set different barriers to change cutoff values for changing colors. The color order is red < blue < green < yellow. This code can lead to some interesting properties, which we will see later.


Editor - C:\Users\yac67\OneDrive\Documents\USC\2019 Fall\MATH 458\Project2\mainCode.m
equations.mlx
mainCode.m

```
58  %       Plot different colors based on the actual solution the iterative
59  %       solution approached
60          if (norm(z-red) < eps)
61              plot(p(1,1), p(1,2), 'r*');
62              hold on;
63          elseif (norm(z-blue) < eps)
64              plot(p(1,1), p(1,2), 'b*');
65              hold on;
66          elseif (norm(z-green) < eps)
67              plot(p(1,1), p(1,2), 'g*');
68              hold on;
69          elseif (norm(z-yellow) < eps)
70              plot(p(1,1), p(1,2), 'y*');
71              hold on;
72          end
73      end
```

The code above is used to plot the proper color for the initial value depending on what we are plotting based on. In this plotting scheme, we are plotting a color based on the actual solution the initial value of z converged to. The color order is red < blue < green < yellow. This code can lead to some interesting properties, which we will see later.

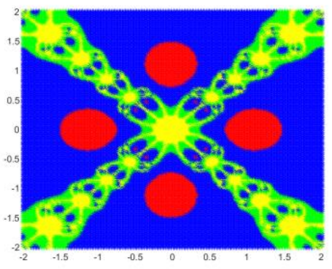
For the second plotting method, we get the following fractal:

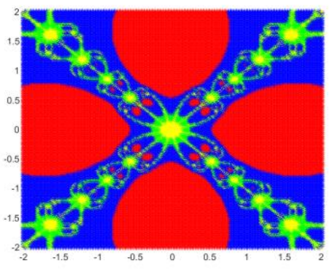


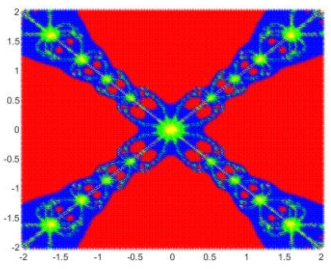
With both axes having interval $[-2,2]$, we can see how each actual solution has a quadrant specific to it (initial values closest to the actual solution seem to end up converging to that solution). Additionally, the four quadrants are separated by boundary lines where the solution of convergence doesn't follow the established rule of converging to the closest actual solution.

Around each solution, there should be a region with the property that once an approximate solution is inside this region, the convergence rate is becoming quadratic. How should you characterize quadratic convergence? How should you determine the size of the quadratic convergence region?

To determine the region of quadratic convergence, we can use the `barriers` code from above and carefully select the value of `barrier` to see some interesting properties. If we look at `barrier = 5`, `barrier = 7`, and `barrier = 9`, we get the following images, respectively (with bounds set to 2 and with 200 vectors):







These images are quite interesting. On the left, we start to see a small circle form that is centered roughly around the actual solution. Since red is the lowest iteration value, it makes sense that the graph looks as is. Additionally, a circle around the origin has been drawn in yellow, indicating the most iterations per our bounds. This also makes sense as when z starts close to 0, Newton's method needs more iterations to converge as it tends to overshoot (step-size is magnitudes greater than z).

A similar analysis can be done for when `barrier = 7`, with the additional mention that the circle for lowest iteration value is now more elliptical and significantly larger.

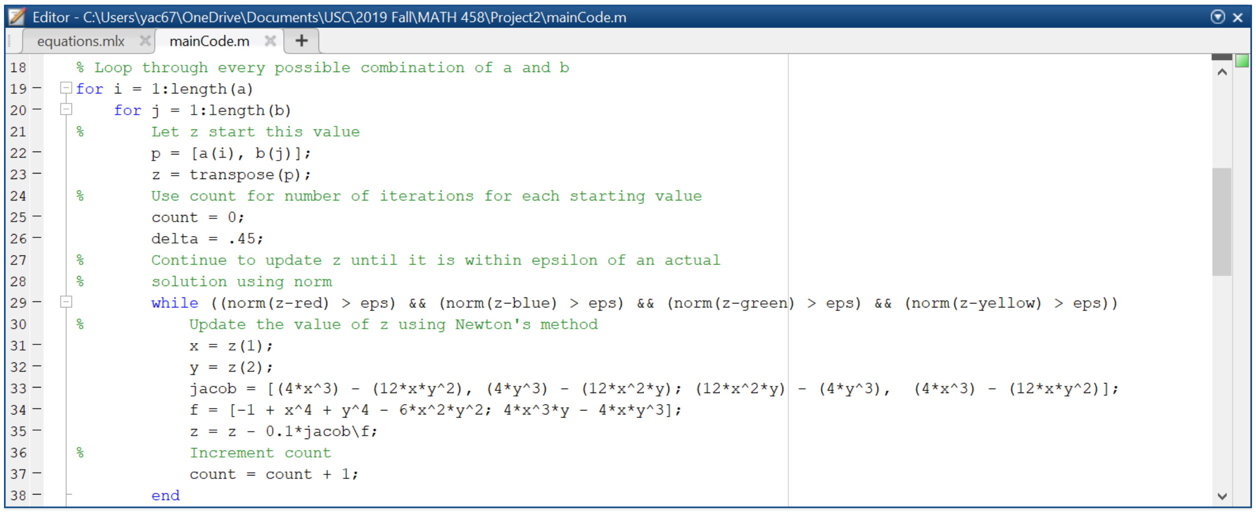
However, when `barrier = 9`, the region that is red is no longer elliptical but rather has a wavy edge. When compared to the original fractal, the red region encompasses much of the area that converged to the solution it was supposed to. Hence, I'd conclude that the region of quadratic convergence is closely related to this region.

If starting from two nearby points, the Newton's method leads to two very different solutions, it may mean that the first several steps of the Newton's method are too large. If you limit the step-size of the Newton's method, would the fractal image we created with the Newton's method change?

We can analyze this question both theoretically and experimentally. Firstly, let's approach with theory.

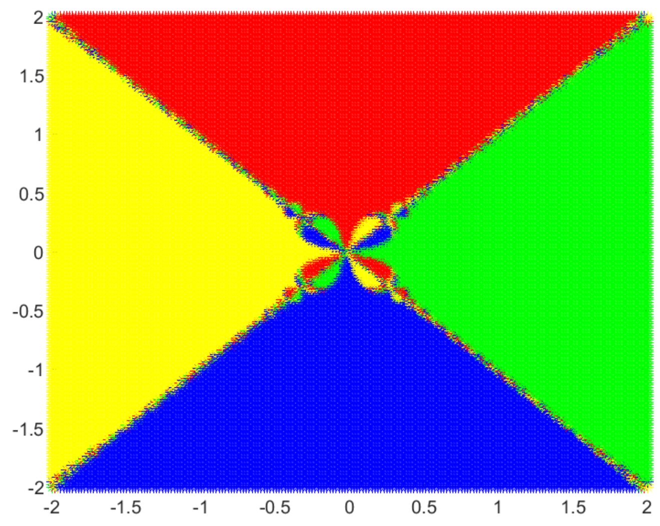
If we were to limit the step-size of Newton's method based on the norm of $z_{new} - z$, then Newton's method would converge more slowly. Equivalently, Newton's method would take more steps for the same initial value of z. Since, before, the initial values that were equally near two solutions resulted in the initial value not necessarily being classified as the solution closest to it (resulting in the fractal image we see). Additionally, we see that the number of iterations for Newton's method to converge for z starting in these areas is significantly higher than if it were to start elsewhere (outside of the region for quadratic convergence). We know that these initial value points end up converging to different solutions after many iterations because Newton's method overshoots in its initial steps. Hence, a limit on the step-size will prevent a portion of the overshoot, depending on the strictness of the limit, causing for less fractal-like boundaries between regions and more linear boundaries, as would be expected.

We can also do this experimentally. We can first try a naïve attempt at a limit on the step-size by simply scaling down the step (I chose the value of $10\% = 0.1$).


Editor - C:\Users\yac67\OneDrive\Documents\USC\2019 Fall\MATH 458\Project2\mainCode.m
equations.mlx
mainCode.m

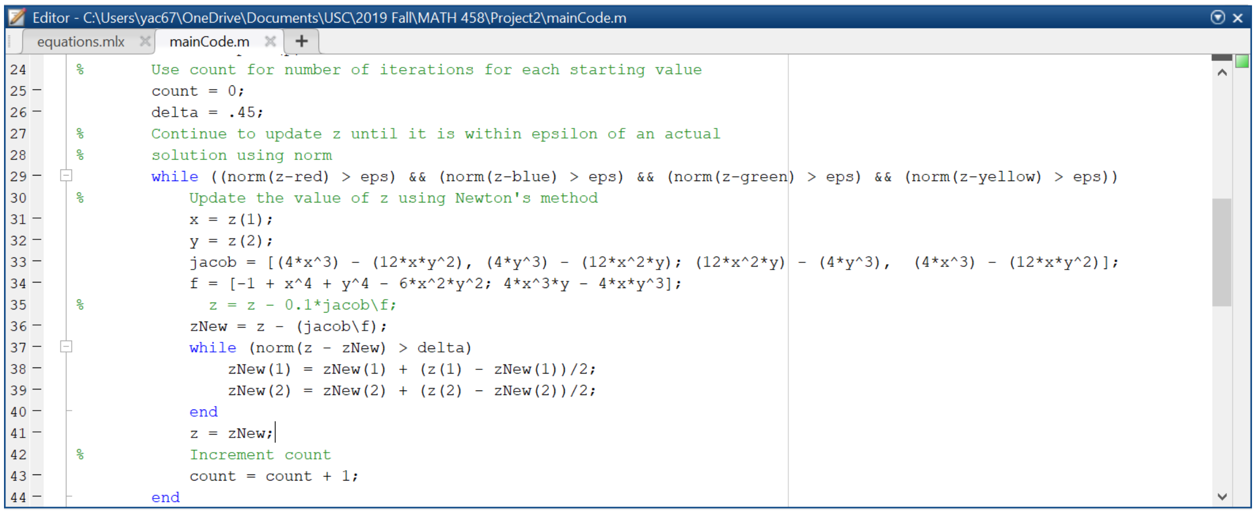
```
18     % Loop through every possible combination of a and b
19 -  for i = 1:length(a)
20 -      for j = 1:length(b)
21    %        Let z start this value
22 -          p = [a(i), b(j)];
23 -          z = transpose(p);
24    %        Use count for number of iterations for each starting value
25 -          count = 0;
26 -          delta = .45;
27    %        Continue to update z until it is within epsilon of an actual
28    %        solution using norm
29 -          while ((norm(z-red) > eps) && (norm(z-blue) > eps) && (norm(z-green) > eps) && (norm(z-yellow) > eps))
30    %            Update the value of z using Newton's method
31 -              x = z(1);
32 -              y = z(2);
33 -              jacob = [(4*x^3) - (12*x*y^2), (4*y^3) - (12*x^2*y); (12*x^2*y) - (4*y^3),  (4*x^3) - (12*x*y^2)];
34 -              f = [-1 + x^4 + y^4 - 6*x^2*y^2; 4*x^3*y - 4*x*y^3];
35 -              z = z - 0.1*jacob\f;
36    %            Increment count
37 -              count = count + 1;
38 -          end
```

In the code above, we see that it is the same as the original code with one change. In line 35, I've multiplied the step-size of z by 0.1. This yields the following fractal when using bounds of $[-2,2]$, 200 vectors, and plotting based on actual solution it converges to:



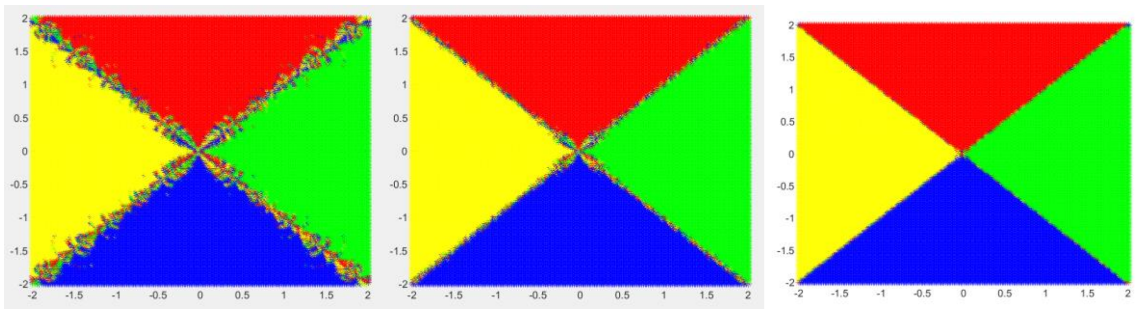
This seems to uphold, at least partially, our hypothesis that the boundaries between regions would become more linear. However, the linear bounds near the origin seem to be unaffected by the scalar multiple. This is because the multiple does not take into account the size of the step relative to the value of z, but rather scales down by the same portion each time. So, if the step-size was initially very large (orders of magnitude larger than z), then a scalar multiple will do little to deter overshooting. As this is the case for initial values near the origin (even 'small' steps can be orders of magnitude larger than an initial z that starts very close to the origin), we must find a more robust method of scaling down the step-size.

To account for this and ensure that our step size does not , we could write more code that could scale down the step size so that z_{new} is within a certain range of z. This range can be mathematically expressed with the norm of $z_{new} - z$ being less than `delta`, where we can change `delta`. If $z_{new} - z$ is greater than or equal to `delta`, we can update z_{new} to come halfway to z. Thus, z_{new} logarithmically approaches z until the norm of $z_{new} - z$ being less than `delta`. This implementation can be seen in the code below:


Editor - C:\Users\yac67\OneDrive\Documents\USC\2019 Fall\MATH 458\Project2\mainCode.m
equations.mlx
mainCode.m

```
%       Use count for number of iterations for each starting value
        count = 0;
        delta = .45;
%       Continue to update z until it is within epsilon of an actual
%       solution using norm
        while ((norm(z-red) > eps) && (norm(z-blue) > eps) && (norm(z-green) > eps) && (norm(z-yellow) > eps))
%           Update the value of z using Newton's method
            x = z(1);
            y = z(2);
            jacob = [(4*x^3) - (12*x*y^2), (4*y^3) - (12*x^2*y); (12*x^2*y) - (4*y^3),  (4*x^3) - (12*x*y^2)];
            f = [-1 + x^4 + y^4 - 6*x^2*y^2; 4*x^3*y - 4*x*y^3];
%             z = z - 0.1*jacob\f;
            zNew = z - (jacob\f);
            while (norm(z - zNew) > delta)
                zNew(1) = zNew(1) + (z(1) - zNew(1))/2;
                zNew(2) = zNew(2) + (z(2) - zNew(2))/2;
            end
            z = zNew;
%           Increment count
            count = count + 1;
        end
```

In line 26, we set a value for delta. Then, in line 26 instead of re-writing the value of z with the result of Newton's method, we put it into a variable called `zNew`. This way, we can compare z_{new} to z. Then, in lines 37-40, the `while` loop reads: `while` the norm of the difference between `zNew` and `z` is greater than our assigned `delta`, update `zNew` to be halfway closer to `z`. Then, in line 41, we can update the value of `z`.

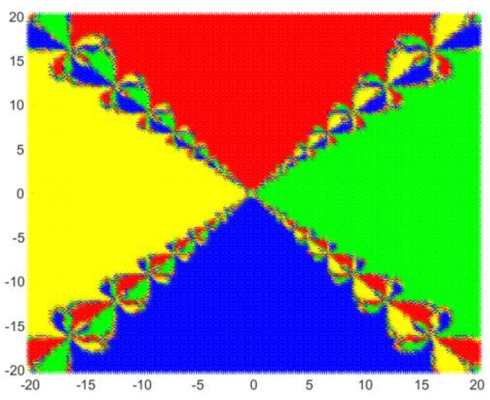


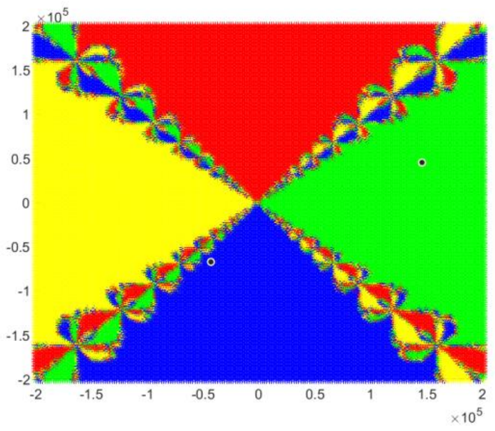
For different deltas, we get the following figures with bounds set to 2 and with 200 vectors.

The images above are for `delta = 0.7`, `delta = 0.4`, and `delta = 0.1`, respectively (note that choosing `delta = 1.0` would result in the original fractal). Once again, we see that our hypothesis was correct; as `delta` becomes stricter, it looks more and more like a linear boundary.

Does the Newton's method always converge with the exception of starting from z = 0?

To experiment with this, we can change our bounds to something much larger. The following images are for when `boundValue = 20` and `boundValue = 200,000`.





As expected, the overall structure of the image stayed the same, even as the bounds increased. This indicates that Newton's method will likely converge for all starting values of z, where $z \neq 0$.