1 Newton's Method

Given a general function $f(x)$, how can we determine its roots? This is a difficult problem, especially if f is intractable and analytic solutions are not feasible. Newton's method is one of the most widely known algorithms for solving this problem. It is an iterative process that requires an initial guess and the ability to evaluate, or at least approximate, $f'(x)$. The equation governing each term can be relatively simple and the process converges quadratically (the number of correct digits doubles per iteration) in many cases, making it a reasonable first effort before trying other, more complicated or specialized methods. The equation for term x_{n+1} in the Newton iteration is given by:

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

The derivation of Newton's method is as follows: Values of a function $f(x)$ in a neighborhood of a point x_0 can be approximated by the the function's tangent line at that point using the point-slope equation $g(x) = f'(x_0)(x - x_0) + f(x_0)$. Thus, given a reasonable starting guess for the root, x_0, we can obtain a better approximation for the root, x_1, by solving $0 = f'(x_0)(x_1 - x_0) + f(x_0)$ for x_1. This, of course, yields $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$.

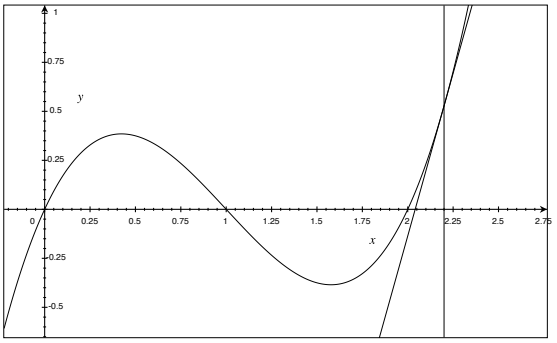

Figure 1: Visualization of Newton's Method

We visualize this in figure 1 using using Newton's method to find a root of the function $f(x) = x^3 - 3x^2 + 2x$. The intersection of the vertical line and the x axis marks our initial guess of $x_0 = 2.2$ and the intersection of the tangent line and the x axis is x_1, which is closer to the true root than our initial guess.

This raises several questions, however. How would the iteration behave if the initial guess were much farther away from the desired root? If it were closer to another root? What if the derivative were zero at one of the iteration points? When is Newton's method quick? For what starting values does Newton's method converge? One might expect that the iterations would have converged to a different root if our initial guess were much closer to that root. One might also expect slow convergence out of the process if the starting guess were extremely far way from the root. If the derivative were zero or extremely small in one of the iterations, the next approximation in the iteration would far overshoot the true value of the root and the iteration might not converge at all.

In practice, there are several cases when Newton's method can fail. We turn to Taylor's theorem to understand roughly when the method breaks down as well as its convergence rate when conditions are ideal. At this point we still consider only real valued functions that take real arguments.

Taylor's Theorem. *Let $f \in C^k$. Then, given the function's expansion about a point a, there exists $q \in (a, x)$ such that*

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + \frac{f^{(n+1)}(q)}{(n+1)!}(x-a)^{n+1}$$

for $1 \leq n \leq k$. □

Using Taylor's theorem we show that Newton's method has quadratic convergence under certain conditions.

Proposition. *Let $f \in C^2$ and z be a root of f. Let $I = [z - c, z + c]$ be an interval about the root. Let $x_0 \in I$ be an initial guess for the root. If $f'(x) \neq 0 \, \forall x \in I$ and $f''(x)$ is bounded on I, then Newton's method achieves quadratic convergence.*

Proof. Consider the expansion of the function f about x_n evaluated at z. Using Taylor's theorem to write this equation:

$$f(z)=f(x_n)+f'(x_n)(z-x_n)+\frac{f''(q)}{2}(z-x_n)^2$$

for some $q \in (x_n, z)$. Realizing that $f(z) = 0$ we can divide the equation by $f'(x_n)$ and rewrite the equality as:

$$x_n - \frac{f(x_n)}{f'(x_n)} - z = \frac{f''(q)}{2f'(x_n)}(x_n - z)^2$$

Recognizing the definition of the next term in a newton iteration, this becomes:

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

Take the absolute values of both sides. Let C $= \left|\frac{f''(q)}{2f'(x_n)}\right|$. Let $\epsilon_{n+1} = |x_{n+1} - z|$ be the absolute error in the $n+1$th term and $\epsilon_n = |x_n - z|$ be the absolute error in the nth term. We now have:

$$\epsilon_{n+1} = C\epsilon_n^2$$

If C were a constant, or approximately constant each iteration, this would signify that Newton's method achieves quadratic convergence. In other words, the number of correct digits in each approximation doubles each iteration. C is approximately constant, or at least bounded, when $f'(x)$ is nonzero for $x \in I$, $f''(x)$ is bounded for $x \in I$, and the starting guess x_0 is close enough to the root. Finally, $q \approx z$ because the interval (x_n, z) shrinks as each x_n become closer and closer to z. □

In some cases convergence can still be achieved given non-ideal condition, though the rate might be slower, possibly exponential/linear or worse, meaning that the number of correct digits increases linearly per iteration rather than doubling. For example, convergence to a root with multiplicity greater than 1 is generally slower than quadratic.

Two important but difficult elementary calculations are the square root and the reciprocal. In present day we're able to perform these calculations with ease and high accuracy due to computers and efficient algorithms. Computers have not always existed, however, and in the past, these calculations were done by hand using processes such as Newton's method. Consider the two functions $f_1(x) = x^2 - a_1$ and $f_2(x) = a_2 - \frac{1}{x}$ where a_1 and a_2 are real, nonzero constants. f_1 has roots at $\pm\sqrt{a_1}$ and f_2 has a root at $\frac{1}{a_2}$. Using $f_1'(x) = 2x$ and $f_2'(x) = -\frac{1}{x^2}$ we obtain the iterations

$$x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2} \tag{2}$$

$$x_{n+1} = x_n(2 - ax_n) \tag{3}$$

for approximating the root or reciprocal of a given number a. It's interesting to note that neither iteration converges given a starting guess of zero. For (1), it's clear from the equation that x_1 is undefined for a starting guess of $x_0 = 0$. Remembering the earlier theorem, starting with $x = 0$ guarentees that the derivative is zero-valued on I. Also, one can see from figure 2 that the root that the process converges to is dependent upon the sign of the initial guess. Intuitively, it makes sense that the process would not converge given a starting guess equidistant from each root.

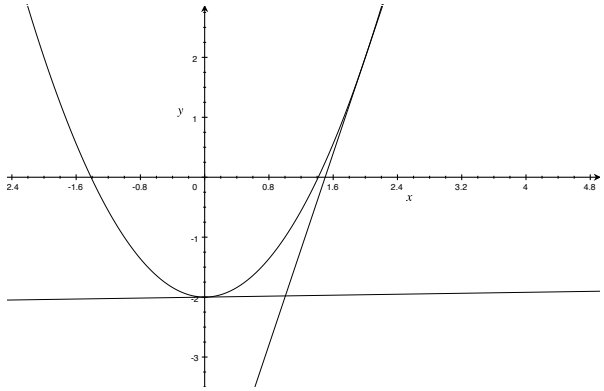

Figure 2: Bad guess vs good guess

Figure two illustrates why the starting guess is important. A starting guess of 2 yields a much more accurate approximation than one of .01, which overshoots the root by a wide margin. The intersection of the x axis and the near-horizontal line in figure 2 would be the next iteration in the series. The iteration for the reciprocal is also an example of why the starting guess is important. If the starting guess is not in the open interval $(0, \frac{2}{a})$, Newton's method will not converge at all. Generally, Newton's method does not converge if the derivative is zero for one of the iteration terms, if there is no root to be found in the first place, or if the iterations enter a cycle and alternates back and forth between different values.

2 Fractals and Newton Iterations in the Complex Plane

An interesting result is that Newton's method works for complex valued functions. Having seen that Newton's method behaves differently for different starting guesses, converging to different roots or possibly not converging at all, one might wonder what happens at problem areas in the complex plane. For example, starting points that are equidistant to multiple different roots. Attempting to visualize the convergence of each possible starting complex number results in a fractal pattern. Figure 3 is a colormap for the function $f(z) = z^3 - 1$ depicting which root Newton's method converged to given a starting complex number. Complex numbers colored red eventually converged to the root at $z = 1$. Yellow means that that starting complex number converged to $e^{\frac{2i\pi}{3}}$, and teal signifies convergence to $e^{\frac{-2i\pi}{3}}$. The set of starting points which never converge forms a fractal. This set serves as a boundary for the different basins of attraction, or the set of points which converge to a particular root.

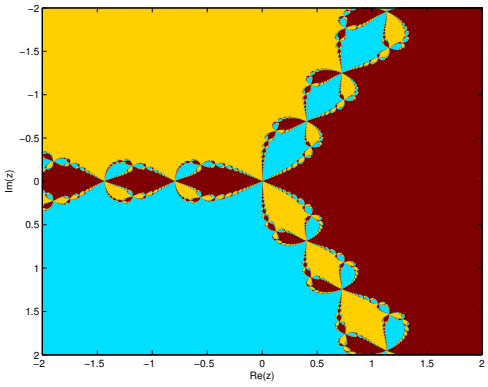

Figure 3: $f(z) = z^3 - 1$

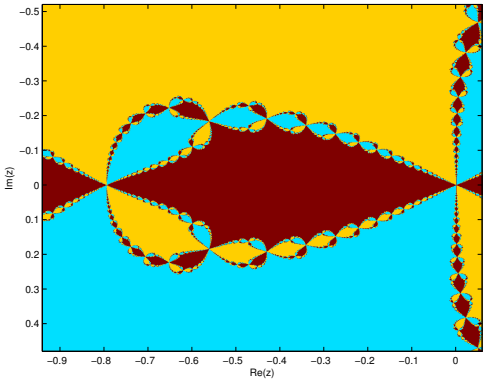

Figure 4: A closeup of a lobe with clear fractal pattern

Looking back to figure 2, visualizing the movement of the tangent line back and forth across the minimum of the parabola reveals that tiny variations in choice of starting point near a problem area can change which root the iterations converge to and also that the small area near the minimum gets sent to the majority of the real line. This behavior of a small area being mapped to a large one is characteristic of fractals and gives us an intuition as to why this phenomenon occurs: small movements from a point in the complex plane analogous to the minima from figure 2 result in the next term in the iteration being sent chaotically to some other point in the complex plane, which could then do anything.

It will now be useful to define what a fractal is and give a brief summary of different fractal properties. It turns out that there's no universally accepted definition of a fractal, but typically they're described as a complex geometric figure that does not become simpler upon magnification and exhibit at least one of the following properties: self-similarity at different scales, complicated structures at different scales, nowhere differentiable (they are quite jagged and rough), and a "fractal dimension" that's not an integer. For example, figures 3 and 4 show the self-similarity property of fractals. Zooming in to one of the lobes reveals even more lobes.

Similar to how the area of a circle changes as per the square of its radius and the volume of a sphere changes as per the cube of its radius, the space-filling capacity of whatever dimension the fractal exists in does not necessarily change per an integer when its "radius analogue" changes. One way to measure the fractal dimension that can be approximated numerically is called the box-counting dimension. Picture a fractal, such as the one in Figure 5, lying on an evenly spaced grid. Now imagine a box of a certain size, and how many of those boxes it would take to completely cover the fractal. The box-counting dimension is determined by calculating how the total number of boxes required changes as the size of each box becomes smaller and smaller. Formally: Let ϵ be the size of a box and let $N(\epsilon)$ be the number of boxes required to completely cover the given fractal F given side length ϵ. Then the box-counting dimension is defined as:

$$\dim_{box}(F) = \lim_{\epsilon \to 0} \frac{\log(N(\epsilon))}{\log(N(\frac{1}{\epsilon}))} \qquad (4)$$

Let's confirm that the pattern in figure 5 generated from the Newton iterations over the complex grid for the function $f(z) = z^3 - 1$ is, in fact, a fractal using numerical approximation to determine the box-counting dimension. The number of boxes N of size R needed to cover a fractal entirely follows the equation $N = N_0 * R^{D_F}$ where N_0 is some constant and D_F is the fractal dimension, which is less than the dimension of space the fractal exists in.

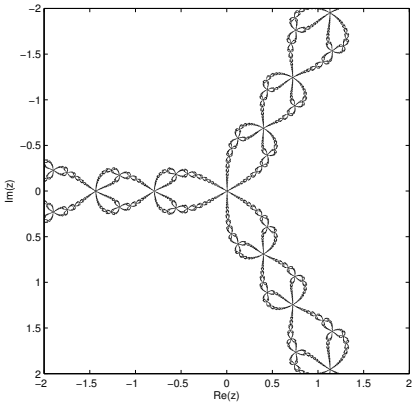

Figure 5: Fractal part of figure 3

We can see a discrepency between the number of boxes N required to cover the fractal given the size R (blue line) and the typical number required for a non-fractal two-dimensional figure (red-line) in figure 6. This is a good indication that Figure 5 is, in fact, a fractal.

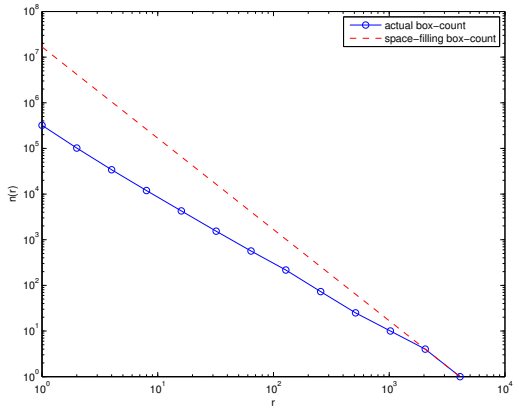

Figure 6: Number of boxes N vs size R

Figure 7 is a graph of the exponent D_F in the earlier relation vs the size of the box, R. We observe that the Newton fractal generated from $f(z) = z^3 - 1$ has a fractal dimension of appoximately $D_F = 1.5$ for R less than 100.

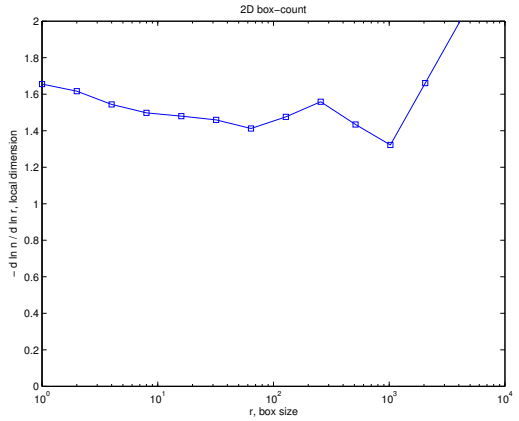

Figure 7: D_F vs size R

3 In terms of Complex Dynamics

We now turn our attention to the more sophistocated and rigorous language used to describe this phenomena developed in complex dynamics. Newton iterations are actually a discrete dynamical system. A dynamical system is a geometrical description of how the state of a set of points evolve over time based on a fixed rule. In this case, the the points are the entirety of the complex plane and the fixed rule is the newton iteration $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ for a given f, which we will hence refer to as $N_f(x_n) = x_{n+1}$. It is discrete because the system changes in discrete jumps per iteration rather than continuously.

The set of all points that a particular starting point x_0 evolves into under repeated applications of N_f is known as the trajectory or orbit of x_0. Let N_f^a represent a repeated applications of N_f i.e. $N_f^3(x) = N_f(N_f(N_f(x)))$. Then the orbit of a starting complex point x_0 is:

$$\mathcal{O}(x_0) = \{x_0, N_f(x_0), N_f^2(x_0), ...\} \quad (5)$$

If $N_f(x) = x$ for some point x, then x is a fixed point in the dynamical system. Clearly all roots of f are fixed points. For our purposes, these are the only fixed points as well. We can define the earlier mentioned term basin of attraction in terms of the notions of orbits and fixed points. Indeed, the basin of attraction for a fixed point is the set of all points whose orbit eventually reaches that fixed point and remains there. That is, for a fixed point r, its basin of attraction is:

$$\mathcal{B}_f(r) = \{z | \lim_{a\to\infty} N_f^a(z) = r\} \quad (6)$$

We point to figure 3 to illustrate this concept. Each of the different colors represents the points in one of the three different basins of attractions. Note that the borders are not in these sets. It's also possible for a starting point to enter in a cycle such that the iterations alternate back and forth, or the orbit is a finite set that does not contain a fixed point, as seen in figure 20.

The union of all the basins of attraction and attractive cycles is known as the Fatou set. Points in the Fatou set behave regularly in that their orbits behave similarly to their neighbors' orbits, eventually converging to the same fixed point or attractive cycle.

The complement of the Fatou set is known as the Julia set. This is the set of points whose orbits are complicated, meaning that they do not eventually rest upon a fixed point and generally behave chaotically. Orbit of a point in the Julia set is a subset of the Julia set, and small perturbations in points of the Julia set result in a variety of different orbits. The Julia set occurs at the boundaries of the different basins of attraction. In figure 8, the lighter points are a visualization of the Julia set and the darker the Fatou set.

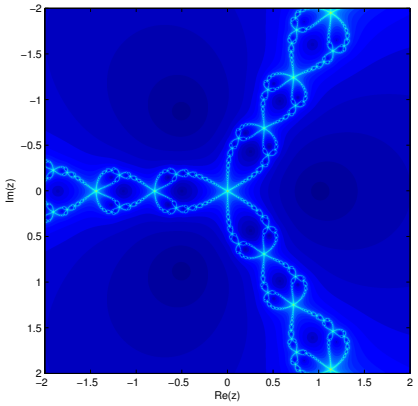

Figure 8: Colormap of iterations until convergence. Darker means faster convergence.

4 Variations of Newton's Method

The most desireable feature of Newton's method is its quadratic convergence, which it often loses if conditions are not ideal. For example, attempting to use Newton's method to find a root of multiplicity greater than one results in linear convergence. Methods have been developed to account for such cases in order to recover the quadratic convergence property. For example, given a function $f(x)$ with a root of multiplicity m, we can instead use Newton's method on the function $g(x) = f(x)^{\frac{1}{m}}$ which has the same root but with multiplicity one. Using $g'(x) = \frac{1}{n}f(x)^{\frac{1}{m}-1}f'(x)$ the iteration becomes

$$x_{n+1} = x_n - m\frac{f(x_n)}{f'(x_n)} \tag{7}$$

which is called the relaxed Newton's method.

Let's observe the difference in the convergence rates for the function $f_1(z) = (z-1)(z-2)^2(z-3)$ which has a root at two of multiplicity two using the regular and the relaxed Newton's method.

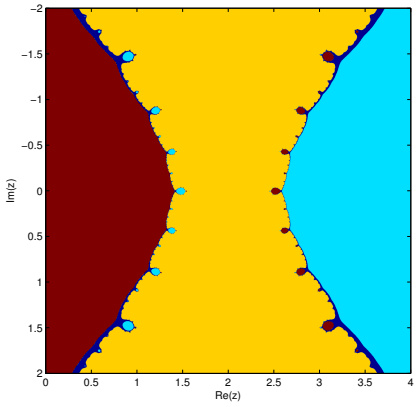

Figure 9: 10 iterations using regular Newton, all simple roots

In figure 9 we exhibit a colormap of the convergence in 10 iterations for $f_2(x) = (z-1)(z-2)(z-3)$, which has a simple root where f_1 has a double root. Red, yellow, and teal signify convergence to the roots 1, 2, and 3 respectively. Dark blue means that number did not converge, and is seen only near the borders for the different basins of attraction.

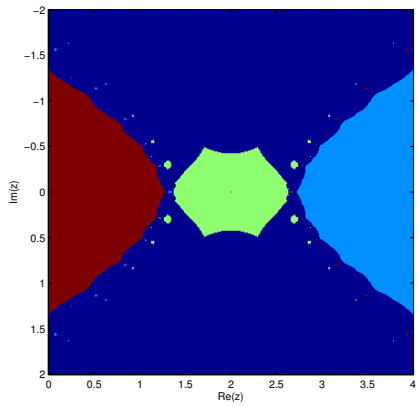

Figure 10: 10 iterations using regular Newton

Figure 10 depicts the reults of 10 iterations of newton's method on f_1 for each starting complex number on a small grid about the roots. Red, lime, and light blue indicate convergence to the roots 1, 2, and 3 respectively. Dark blue means that the iterations did not converge to the root for that given value.

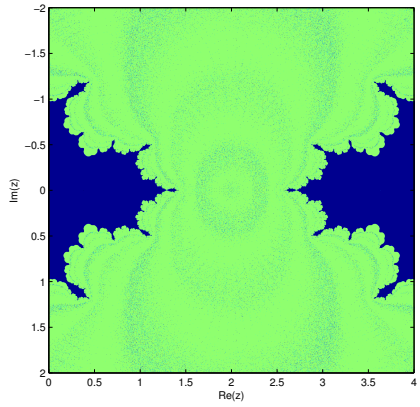

Figure 11: 10 iterations using relaxed Newton

Contrasting figures 9 and 10, we observe that the presence of a double root indeed results in slower convergence to that root. Nearly all the values in figure 9 converged, while most of the values in figure 10 did not converge.

The relaxed Newton's method fixes this, however, as seen in figure 11. The colors in figure 11 have a different interpretation. Lime here means convergence to 2 and blue means no convergence at all. We observe that convergence to the double root is achieved much more quickly for a larger amount of starting values using the relaxed newton's method. Iterations using the relaxed Newton's method also results in a different fractal pattern.

In general, the relaxed Newton's method can be used on any function and the choice of m, the root, affects the convergence and the sharpness of the fractal pattern, Choosing $0 < m < 1$ softens the fractal pattern as seen in figure 13. This is because exponentiating the original polynomial with something greater than 1 introduces a multiple root resulting in slower convergence but less instances of overshooting near a problem point. On the other hand, choices of $m > 1$ can result in faster convergence, but the iterations are more prone to chaotic behavior near boundary points resulting in a sharper fractal pattern, as seen in figure 12.

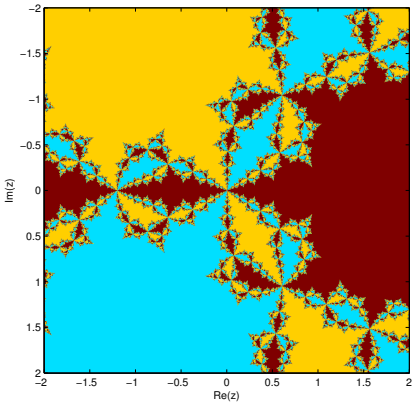

Figure 12: 100 iterations of relaxed Newton on $f(z) = z^3 - 1$ with m = 1.9

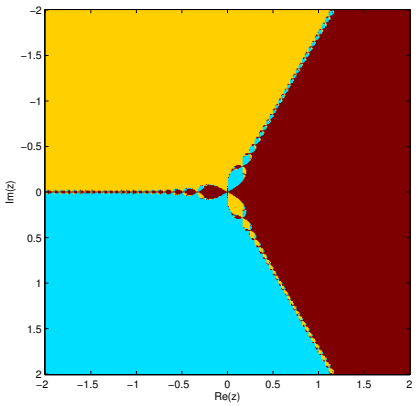

Figure 13: 500 iterations of relaxed Newton on $f(z) = z^3 - 1$ with m = .1

Another variation of Newton's method is called Newton's method for a multiple root. This approach trades simplicity of iteration terms for more robustness in achieving quadratic convergence. If a polynomial $f(x)$ has a root of multiplicity m at a point x_0, its derivative f' will have a root of multiplicity $m-1$ at that point. Thus we can obtain a function $g(x) = \frac{f(x)}{f'(x)}$ that has all simple roots at all of f's roots. The iteration then becomes:

$$x_{n+1} = x_n - \frac{f(x_n)f'(x_n)}{f'(x_n)^2 - f(x_n)f''(x_n)} \qquad (8)$$

The presence of a derivative in the original Newton iteration was troubling enough. Here we now must be able to calculate the second derivative as well. The benefit of Newton's method for multiple roots is that it converges quadratically at every root. We observe this through visualising the convergence of $f(z) = (z-1)^2(z-2)^2(z-3)^2$.

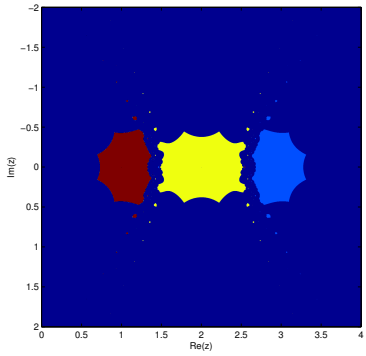

Figure 14: 10 iterations using regular Newton

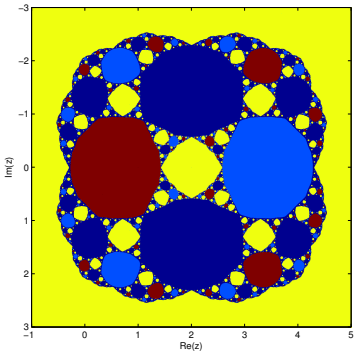

Figure 15: 10 iterations using Newton's method for multiple roots

We see in figure 14 that not many starting values achieve convergence within the desired tolerance when all three roots have multiplicity two using the standard Newton iteration. Compare this to using Newton's method for multiple roots in figure 15 and 16. Dark blue signifies convergence to something other than one of the three desired roots or failure to converge at all. Even though values that would have converged to one of the roots in regular Newton's now converge to something else, largely due to the presence of the derivative in the denominator, the values that converge to the desired roots do so much more quickly. In figure 15 we only use 10 iterations. There is little difference between figure 15 and 16 when 100 iterations are used.

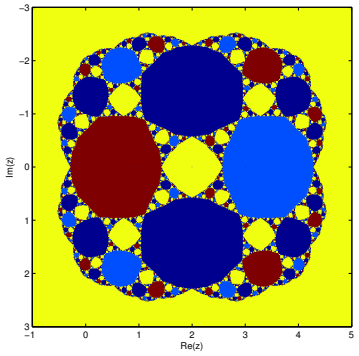

Figure 16: 100 iterations using Newton's method for multiple roots

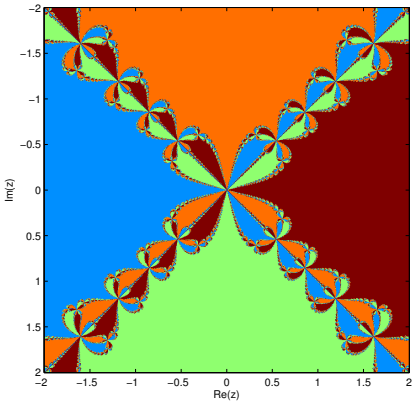

Figure 17: $f(z) = z^4 - 1$

Other, more complicated and specialized, variations of Newton's method exist.

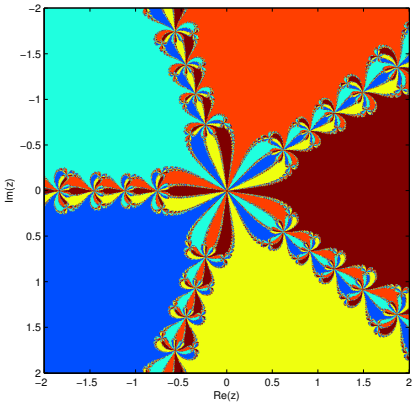

Figure 18: $f(z) = z^5 - 1$

5 A Survey of Various Newton Fractals

In this section we visualize different Newton fractals, their properties, and their convergence. We saw the newton fractal several times earlier for the polynomial $f(z) = z^3 - 1$ with roots at the 3rd roots of unity. Visually it has three monochrome areas with the fractal pattern on the borders of these areas, on the lines with points equidistant from two different roots. All the other polynomials with roots at the nth roots of unity follow the similar pattern of having large monochrone areas and then chaotic fractal behavior on the lines that separate these areas.

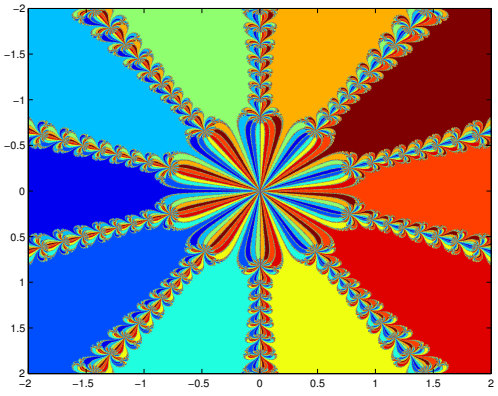

Figure 19: $f(z) = z^{10} - 1$

A given starting point either converges to a root, enters into a cycle, or behaves chaotically and wanders about the complex plane. Figure 20 shows all three of these possibilities. Light blue, orange, and lime are the basins of attraction while dark blue and maroon are the attractive cycles. The fractal pattern is the set of points that wander chaotically about the complex plane.

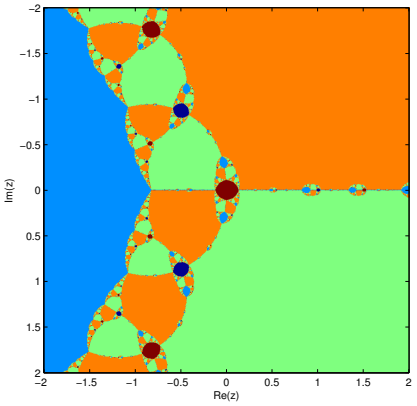

Figure 20: The maroon and dark blue points enter cycles

Let's observe a typical Newton fractal with all real roots. The color below signifies how long that starting point took to converge. Darker means faster convergence, lighter means slower. Each number converged reasonably quickly, with the slowest taking 31 iterations.

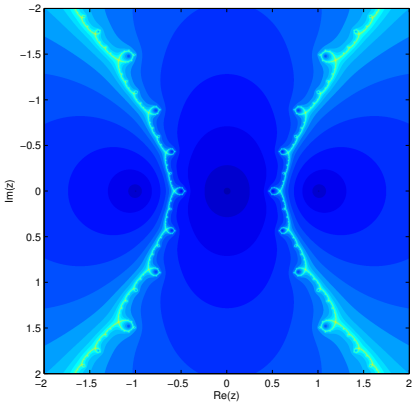

Figure 21: $f(z) = z(z-1)(z+1)$

Now let's look at a function with all imaginary roots. The roots of the function in figure 22 are at $-2i, -i, i, 2i$.

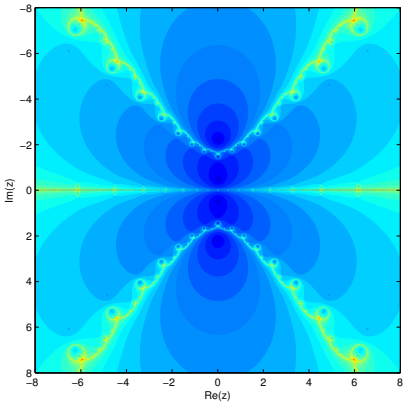

Figure 22: $f(z) = x^4 + 5x^2 + 4$

This fractal behaved pretty regularly, nothing too exotic. How about one with real and imaginary roots? Figure 23 shows the Newton fractal with roots at $0, 1, -1, -i$ and i.

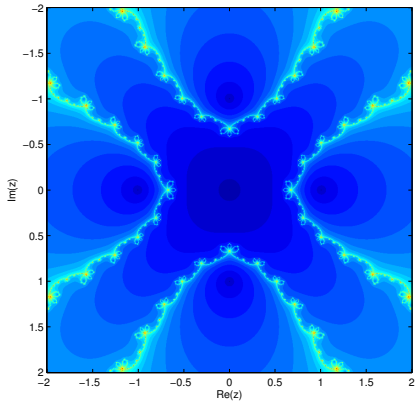

Figure 23: $f(z) = z^5 - z$

Figure 24 revisits the polynomial with all real roots from figure 21, except here relaxed Newton's method is performed with $m = 1.9$. The pattern looks like two insects facing each other. I tried different values of m and found that complex values of m apply a twisting effect to the fractal. Given a real part of 1, an increasing imaginary part starting at 0 and going to 1 results in a clockwise twist while a decreasing imaginary part results in a counterclockwise twist.

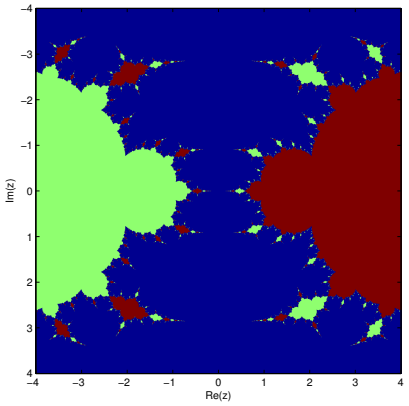

Figure 24: $f(z) = z(z-1)(z+1)$, $m = 1.9$

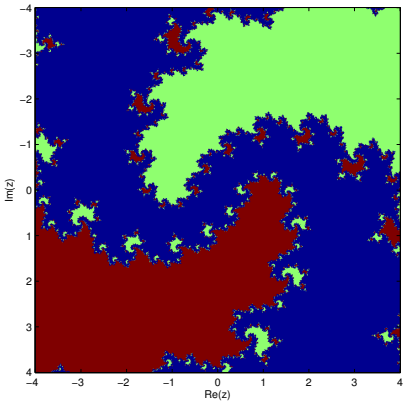

Figure 25: $f(z) = z(z-1)(z+1)$, $m = .9i+1$

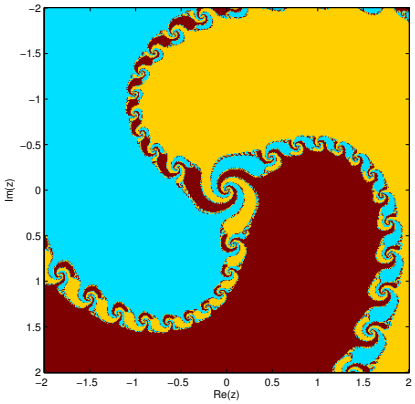

Figure 26: $f(z) = z^3 - 1$, $m = .5i + 1$

6 Code

Original code for this project was written in Matlab. I used Frederic Moisy's box-counting code as well. Code for regular/relaxed Newton iteration performing calculations on every value:

```
function r = oldnewtp(p, x, N, m, dotime)
    dp = polyder(p);
    for i = 1:N
        if dotime
            tic;
        end
        x = x - m*polyval(p,x)./polyval(dp,x);
        if dotime
            toc
        end
    end
    r = x;
end
```

Code for performing the different newton iterations on only nonconverged values:

```
% implements newton's algorithm for root finding.

% p - a row vector for the coefficients of a polynomial. i.e. [1 2 3 4] is
% x^3 + 2*x^2 + 3*x^1 + 4
% x - complex number,  is a starting guess which should be close to the final
    root
% N - integer, is the number of times to run the iteration
% m - constant multiple in relaxed newton iterations
% tol - stop iterating when the difference in iteration becomes this
% critical - iteration at which calculations switch over to only performing
    newton on nonconverged values
% r - matrix that is the result of the N iterations
% s - paired matrix with each # of iterations until under desired tolerance
function [r,s] = newtp(p, x, N, m, tol, mr, critical,dotime)
    % get the derivative and second derivative if necessary
    dp = polyder(p);
    if mr dp2 = polyder(dp); end

    % set up a logical matrix to keep track of which numbers have converged
    % and keep track of previous iterations
    notconverged = true(size(x,1),size(x,2));
    prev = x;

    % set up matrix to keep track of when each entry converges
    conv = zeros(size(x,1),size(x,2));

    for j = 1:N
        if dotime
            tic;
        end
        % perform the iterations on only the values that have not converged
```

```
        % to within the desired tolerance

        if j <= critical
            if mr
                x = x - (polyval(p,x).*polyval(dp,x))./(polyval(dp,x).^2 -
    polyval(p,x).*polyval(dp2,x));
            else
                x = x - m*polyval(p,x)./polyval(dp,x);
            end
        else
            if mr
                x(notconverged) = x(notconverged) - (polyval(p,x(notconverged)
    ).*polyval(dp,x(notconverged)))./(polyval(dp,x(notconverged)).^2 - polyval
    (p,x(notconverged)).*polyval(dp2,x(notconverged)));
            else
                x(notconverged) = x(notconverged) - m*polyval(p,x(notconverged
    ))./polyval(dp,x(notconverged));
            end
        end

        % logical matrix with 1 if the entry has not converged and 0 if it
        % has
        t = abs(prev - x) > tol;

        % if an entry converges this iteration record how many iterations
        % it took
        conv(xor(notconverged, t)) = j;

        % update the logical matrix
        notconverged = t;

        % break out of the loop if every entry has converged
        if ~notconverged
            fprintf('converged after %g iterations \n', j)
            break
        end


         if dotime
            toc
         end

        prev = x;
    end

    s = conv;
    r = x;
end
```

Function used to convert a fractal colormap to a binary image:

```
% takes a colormap of a fractal and converts it into a binary image.
function r = tobinary(img)
```

```
    nrows = size(img,1);
    ncols = size(img,2);

    result = ones(nrows,ncols);

    for row = 1:nrows
        for col = 1:ncols
            left = col - 1;
            right = col + 1;
            top = row - 1;
            bot = row + 1;

            if (left >= 1) && (img(row,left) ~= img(row,col))
                result(row,col) = 0;
            end
            if (right <= ncols) && (img(row,right) ~= img(row,col))
                result(row,col) = 0;
            end

            if (top >= 1) && (img(top,col) ~= img(row,col))
                result(row,col) = 0;
            end

            if (bot <= nrows) && (img(bot,col) ~= img(row,col))
                result(row,col) = 0;
            end

            if (left >= 1) && (top >= 1) && (img(top,left) ~= img(row,col))
                result(row,col) = 0;
            end

            if (right <= ncols) && (top >= 1) && (img(top,right) ~= img(row,
    col))
                result(row,col) = 0;
            end

            if (left >= 1) && (bot <= nrows) && (img(bot,left) ~= img(row,col)
    )
                result(row,col) = 0;
            end

            if (right <= ncols) && (bot <= nrows) && (img(bot,right) ~= img(
    row,col))
                result(row,col) = 0;
            end
        end
    end
    r = result;
end
```

Main fractal producing function. Usage of varargin and parameter parser taken from RazerM of stackexchange:

```
% function to produce an image of the fractal developed when using newton's
% method to find roots given starting points. takes a polynomial, performs
% N newton iterations over a grid in the complex plane and produces a
% colormap based on the results of the iterations at that point

% p - a row vector of the input polynomial's coefficients ie [1 0 0 -1] =
% z^3 -1
% xmin, xmax, ymin, ymax - screen bounds
% tol1 - float, typically of the format 1ex where x is an integer.
% digits of accuracy in the rounding. for example, an input of 1e-3
% rounds the result of the newton iterations to 3 decimal places and rounds
% the results of the call to root to 3 decimal places in order to compare
% if a function converged to a root
% tol2 - float, typically of the format 1ex where x is an integer.
%stops performing calculations a number when the different between
% iterations is under this tolerance.
% step - how fine the grid is. "resolution"
% N - integer, number of newton iterations to perform
% m - coefficient to multiplty by for the relaxed newton's method. keep
% this at 1 for the regular newton's method
% mr - boolean, perform calculations using newton's method for multiple roots
    instead
% centerize - boolean, centerize the data before plotting the image.
% usually the matrix will be a bunch of integers. this centerizes the data
% about the mean.
% iterationscale - boolean, display graphic color is based on root converged
    to AND
% number of iterations until convergence
% onlyiteration - boolean, display graphic is colored based only on number of
% iterations until convergence
% binaryimg - boolean, if true output will be a matrix of all ones and zeros.
% use to display the fractal in isolation.
% critical - integer. argument that's passed to the newtp function that
% determines at which iteration it should switch over to only doing newton
    iterations on
% converged value
% dotime - boolean, display the time in each iteration of the newton
% iteration loop
% scale - float, scales the axes by this number
% random - integer. if nonzero uses a random polynomial of degree given by
    random in order to generate the fractal
function r = fractalp(varargin)

    % this code was taken from RazerM of stack exchange. thanks RazerM
    %# define defaults at the beginning of the code so that you do not need to
    %# scroll way down in case you want to change something or if the help is
    %# incomplete
    options = struct('p', [1 0 0 -1],'xmin',-2,'xmax',2,'ymin',-2,'ymax',2,'
    step',.01,'n',100,'tol1',1e-3,'tol2',1e-4,'m',1,'mr',false,'old',false,'
    cent',true,'iterationscale',false,'onlyiteration',false,'binaryimg',false,
    'critical',10,'dotime',false,'scale',1,'random',0);

    %# read the acceptable names
```

```
    optionNames = fieldnames(options);

    %# count arguments
    nArgs = length(varargin);
    if round(nArgs/2)~=nArgs/2
        error('EXAMPLE needs propertyName/propertyValue pairs')
    end

    for pair = reshape(varargin,2,[]) %# pair is {propName;propValue}
        inpName = lower(pair{1}); %# make case insensitive

        if any(strmatch(inpName,optionNames))
            %# overwrite options. If you want you can test for the right class here
            %# Also, if you find out that there is an option you keep getting wrong,
            %# you can use "if strcmp(inpName,'problemOption'),testMore,end"-statements
            options.(inpName) = pair{2};
        else
            error('%s is not a recognized parameter name',inpName)
        end
    end

    % set up the grid:
    scale = options.('scale');
    xmin = options.('xmin')*scale;
    xmax = options.('xmax')*scale;
    ymin = options.('ymin')*scale;
    ymax = options.('ymax')*scale;
    x = xmin:options.('step'):xmax;
    y = ymin:options.('step'):ymax;
    [X,Y] = meshgrid(x, y);

    % if random was enabled create the random polynomial
    if options.('random') > 0
        options.('p') = 5*randn(1,options.('random')+1) + 5*i*randn(1,options.('random')+1);
    end

    % find out which roots each complex number converges to
    if options.('old') % constant time per iteration
        Z = oldnewtp(options.('p'), X + Y*i, options.('n'), options.('m'), options.('dotime'));
    else % quicker time per iteration
        [Z,S] = newtp(options.('p'), X + Y*i, options.('n'), options.('m'), options.('tol2'), options.('mr'),options.('critical'),options.('dotime'));
    end
    % round each numbers
    Z = round(Z * 1/options.('tol1'))/options.('tol1');

    % determine all roots of the function and store them in an array
    r = roots(options.('p'));
    n = length(r);
```

```
    % round each root to 8 decimal places
    r = round(r * 1/options.('tol1'))/1/options.('tol1');

    % add a zero in the end
    if ~ismember(0, r)
        r = [r.' 0].';
    end

    % create a matrix full of the indices to which the number converged
    [~, loc] = ismember(Z,r);

    if options.('cent') loc = (loc - n/2)/(n/2); end



    if options.('binaryimg')
        r = tobinary(loc);

        imagesc(x,y,r);
        colormap gray;
    elseif options.('onlyiteration')
        imagesc(x,y,S);
    else
        r = loc;
        if options.('iterationscale')
            S = S/max(S(:));
            loc = loc.*S;
        end
        imagesc(x,y,loc);
    end
    xlabel('Re(z)')
    ylabel('Im(z)')
    axis square
    options
end
```

My original algorithm for computing the Newton iterations ran at constant time per loop because it performed the iteration on every single entry in the input matrix. Upon suggestion from Prof. Barnett, I improved this by only performing calculations on the values that had not converged. I also broke off from the iterations if every value had converged within the desired tolerance. This improves the run time:

```
% time it takes for naive implementation to compute using .001 resolution
tic;
r = fractalp(p,-1,1,-1,1,1e-1,1e-2,.001,55,1,false,true);
toc

% time it takes for improved implementation to compute using .001
% resolution
% time it takes for naive implementation to compute using .001 resolution
tic;
```

```
r = fractalp(p,-1,1,-1,1,1e-1,1e-2,.001,55,1,false,false);
toc
```

```
>> test
Elapsed time is 9.724580 seconds.
converged after 51 iterations
Elapsed time is 6.840386 seconds.
```

This change, however, resulted in initially longer iteration times before dropping off and become faster than the old method. What I ended up doing was adding a critical value parameter at which the iteration switches over to only performing the calculation on the nonconverged values. This value defaults to 10 and has to be changed in order to achieve the fastest calculation.

7 References

https://www.math.uwaterloo.ca/~wgilbert/Research/GilbertFractals.pdf
http://www.chiark.greenend.org.uk/~sgtatham/newton/
https://www.whitman.edu/mathematics/SeniorProjectArchive/2009/burton.pdf
http://eprints.maths.ox.ac.uk/1323/1/NA-96-14.pdf
http://en.wikipedia.org/wiki/Box_counting_dimension
Alligood, Kathleen T., Tim Sauer, and James A. Yorke. "Chapter 4." Chaos: An Introduction to Dynamical Systems. New York: Springer, 1997. N. pag. Print.
http://en.wikipedia.org/wiki/Fractal
http://en.wikipedia.org/wiki/Newton%27s_method
http://en.wikipedia.org/wiki/Newton_fractal
http://www.fast.u-psud.fr/~moisy/ml/boxcount/html/demo.html
http://en.wikipedia.org/wiki/Julia_set