Mid-sem Assignment

04/10/2016

# JULIA SETS

## PLOTTING JULIA SETS USING NEWTON'S METHOD

Prof. Jiten C Kalita

Dept. of Mathematics, IIT Guwahati

Aditya Parashar 140103004 --
Aman Saxena 140103007 --
Niranjan. S 140107035 --
Alok Ranjan 140121004 --

***Abstract:***

The Newton fractal is a boundary set in the complex plane which is characterised by Newton's method applied to a differentiable function.

It is a Julia set of the meromorphic function, $z \to z - \frac{f(z)}{f'(z)}$.

Julius sets are among the most widely known families of fractals. In just report, we have discussed the idea of Newton fractals, having demonstrated the method to generate Julia set using Newton-Raphson method applied to six different complex functions. The thus generated Julia set is plotted as a boundary in the complex plane.

***Introduction:***

***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. 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:

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

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

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

We visualize this in figure 1 using using Newton's method to find a root of the function $f(x) = x^3 - 3x^2 + 2x$.

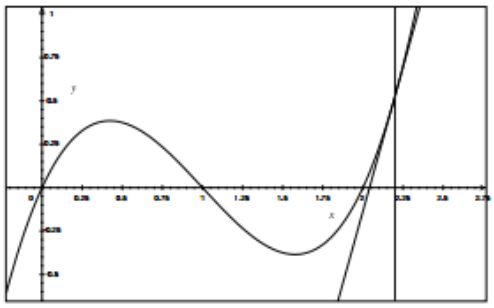


Figure 1: Visualization of Newton's Method

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 away 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 $f(x) = f(a) + f'(a)(x - a) + f''(a)(x - a)^2/2! + \cdots + f^n(a)(x - a)^n/n! + f^{n+1}(q)(x - a)^{n+1}/(n+1)!$ for $1 \leq n \leq k$. Using Taylor's theorem we show that Newton's method has quadratic convergence under certain conditions.

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 (xn, z) shrinks as each xn 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.

Figure two illustrates why the starting guess is important.

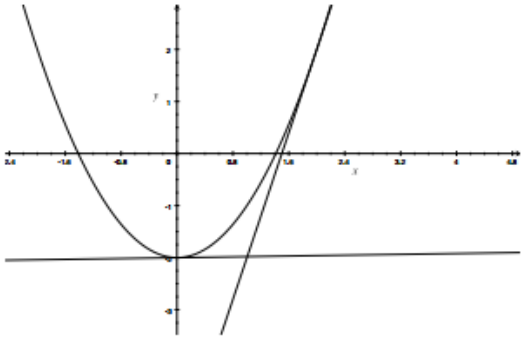


Figure 2: Bad guess vs good guess

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, 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.

***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 2iπ 3 , and teal signi- fies convergence to e −2iπ 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.

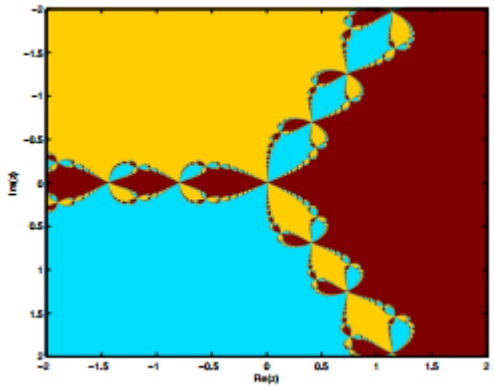


Figure 3: f(z) = z 3 − 1

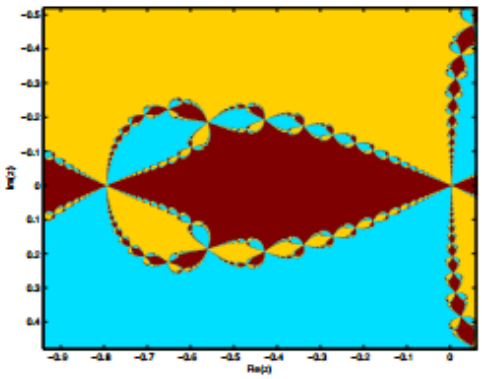


Figure 4: A closeup of a lobe with clear fractal pattern

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.

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 xn+1 = xn− f(xn) f 0(xn) for a given f, which we will hence refer to as Nf (xn) = xn+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 x0 evolves into under repeated applications of Nf is known as the trajectory or orbit of x0. Let Na f represent a repeated applications of Nf i.e. N3 f (x) = Nf (Nf (Nf (x))). Then the orbit of a starting complex point x0 is: O(x0) = {x0, Nf (x0), N2 f (x0), ...} (5) If Nf (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: Bf (r) = {z| lima→∞ N a f (z) = r} (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.

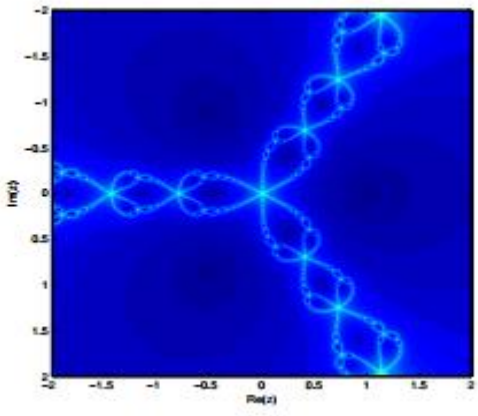



Figure 8: Colormap of iterations until convergence. Darker means faster convergence.

***Numerical Procedure:***

We consider a grid of size 1024*1024 pixels, on the complex plane.

For each pixel, we iteratively calculate the convergence in each pixel by using the newton's method. This is done by recording the nth number of iteration, where the convergence is reached, with respect to a pre-set tolerance of $10^{-6}$.

However, we limit the maximum number of iterations to 360. Hence for every such convergent point, there exists its iteration point. Now, we normalise this number on a scale of 0 – 359, by using an appropriate normalisation formula to set in the hue values (in HSV colour code).

For all those pixels, where the convergence is not achieved even after 360, the iteration is stopped with the default value given to be 360.

If i represents the convergent iteration, then

hue = (i % MAX_ITER)/(MAX_ITER – 1) ;

After this statement the hue value are between 0 to 1, and now we extend it to the range of 0 – 360 using the below formula,

hue = 240 * sqrt(hue) + 120;

Any such normalisation formula depending on the need can be used. Now, we convert the HSV colour format values to corresponding RGB colour code values.

We save the RGB colour format value for each pixel in a 1-D array of size n*n*3.

Then the 1-D array is converted as a PNG image using OpenCV®

***HSV:***

Given a colour with hue H ϵ [0°, 360°), saturation SHSV ϵ [0, 1], and value V ϵ [0, 1], we first find chroma:

$$C = V \times S_{HSV}$$

Then we can find a point (R1, G1, B1) along the bottom three faces of the RGB cube, with the same hue and chroma as our colour (using the intermediate value X for the second largest component of this colour):

$$H' = \frac{H}{60^\circ}$$
$$X = C(1 - |H' \mod 2 - 1|)$$
$$(R_1, G_1, B_1) = \begin{cases} (0,0,0) & \text{if } H \text{ is undefined} \\ (C,X,0) & \text{if } 0 \le H' < 1 \\ (X,C,0) & \text{if } 1 \le H' < 2 \\ (0,C,X) & \text{if } 2 \le H' < 3 \\ (0,X,C) & \text{if } 3 \le H' < 4 \\ (X,0,C) & \text{if } 4 \le H' < 5 \\ (C,0,X) & \text{if } 5 \le H' < 6 \end{cases}$$

Finally, we can find R, G, and B by adding the same amount to each component, to match value:

$$m = V - C$$
$$(R,G,B) = (R_1 + m, G_1 + m, B_1 + m)$$

***PROGRAM CODE:***

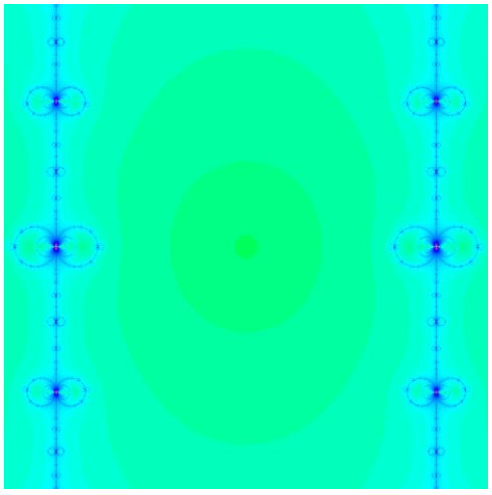
```
#include <stdio.h>
#include <stdlib.h>
#include <complex>
#include <string.h>
#include <opencv2/opencv.hpp>
#include <opencv2/highgui.hpp>
#include <vector>

using namespace cv;
using namespace std;

#define N 1024
#define SQRT_2 1.4142
#define MAX_ITER 120

void HSVtoRGB( float *r, float *g, float *b, float h, float s, float v
);
void saveImage(int width, int height, unsigned char * bitmap,
complex<float> seed);
void compute_julia(complex<float> c, unsigned char * image);


int main()
{
    complex<float> c(-1.00f, 0.000f);
    unsigned char *image = new unsigned char[N*N*3]; //RGB image
    compute_julia(c, image);
```

```
    saveImage(N, N, image, c);
    delete[] image;
}

void compute_julia(complex<float> c, unsigned char * image)
{
    complex<float> z_old(0.0f, 0.0f);
    complex<float> z_new(0.0f, 0.0f);
    complex<float> c1(0.667f,0.0f);
    complex<float> c2(0.333f,0.0f);
    float w;
    //loop for iterating over all x,y co-ordinates
    for(int y=0; y<N; y++)
        for(int x=0; x<N; x++)
        {
            //initial seed
            z_new.real(12.0f * x / (N) - 6.0f);//x-range=-2.0 to 2.0
            z_new.imag(4.0f * y / (N) - 2.0f);//y-range=-2.0 to 2.0
            int i;
            for(i=0; i<MAX_ITER; i++)
            {
                z_old.real(z_new.real());
                z_old.imag(z_new.imag());
                //z_new = c1*z_old+c2*pow(z_old,-2);//f(z)=z^n -1;
                z_new=z_old-tan(z_old);
                //z_new = z_old + (1.0f/2.0f*z_old)*(1.0f-
exp(z_old*z_old));//f(z)=exp(-z^2);
                if(norm(z_new-z_old) < 10e-8f) break;
            }
            //color control parameters
            float brightness = (i<MAX_ITER) ? 1.0f : 0.0f;
            float hue = (i % MAX_ITER)/float(MAX_ITER - 1);
            hue = ((240)*pow(hue,0.5)+120);
            //change power to control smoothness of contours
            float r, g, b;
            HSVtoRGB(&r, &g, &b, hue, 1.0f, brightness);
            image[(x + y*N)*3 + 0] = (unsigned char)(b*255);
            image[(x + y*N)*3 + 1] = (unsigned char)(g*255);
            image[(x + y*N)*3 + 2] = (unsigned char)(r*255);
        }
}

void saveImage(int width, int height, unsigned char * bitmap,
complex<float> seed)
{
    //saaving image buffer unsigned char array of length N*N*3
    Mat rawData  =  Mat( 1024,1024,CV_8UC3,bitmap);


    /*display window
    namedWindow("winname",WINDOW_AUTOSIZE );
    imshow("winname", rawData);
    waitKey(0);*/
```

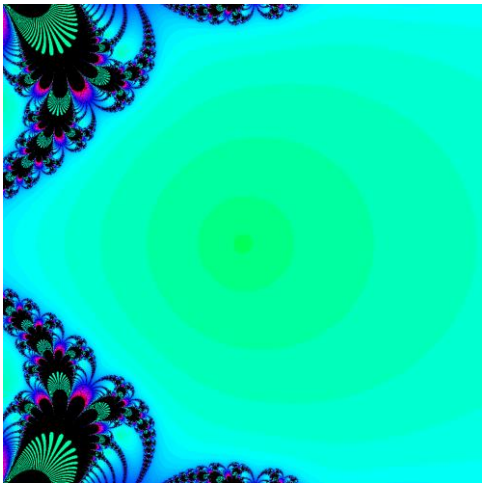
```
    vector<int> compression_params;
    compression_params.push_back(CV_IMWRITE_PNG_COMPRESSION);
    compression_params.push_back(9);
    try {
        imwrite("alpha.png", rawData, compression_params);
    }
    catch (runtime_error& ex) {
        fprintf(stderr, "Exception converting image to PNG format:
%s\n", ex.what());
    }

    fprintf(stdout, "Saved PNG file with alpha data.\n");

}

// r,g,b values are from 0 to 1
// h = [0,360], s = [0,1], v = [0,1]
//      if s == 0, then h = -1 (undefined)
void HSVtoRGB( float *r, float *g, float *b, float h, float s, float v )
{
    int i;
    float f, p, q, t;
    if( s == 0 ) {
        // achromatic (grey)
        *r = *g = *b = v;
        return;
    }
    h /= 60;            // sector 0 to 5
    i = floor( h );
    f = h - i;          // factorial part of h
    p = v * ( 1 - s );
    q = v * ( 1 - s * f );
    t = v * ( 1 - s * ( 1 - f ) );
    switch( i ) {
        case 0:
            *r = v;
            *g = t;
            *b = p;
            break;
        case 1:
            *r = q;
            *g = v;
            *b = p;
            break;
        case 2:
            *r = p;
            *g = v;
            *b = t;
            break;
        case 3:
            *r = p;
            *g = q;
            *b = v;
            break;
```

```
            case 4:
                *r = t;
                *g = p;
                *b = v;
                break;
            default:        // case 5:
                *r = v;
                *g = p;
                *b = q;
                break;
        }
}
```

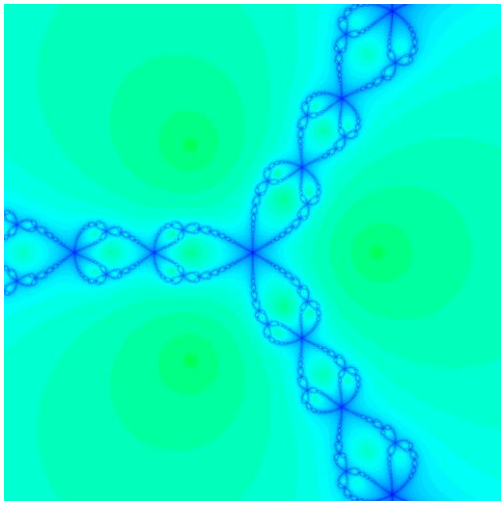
***IMAGES OBTAINED BASED ON THE CODE:***



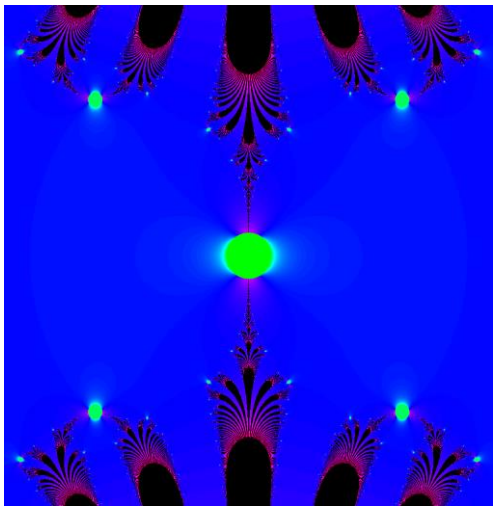
***Img (1)*** : sin(z) =0



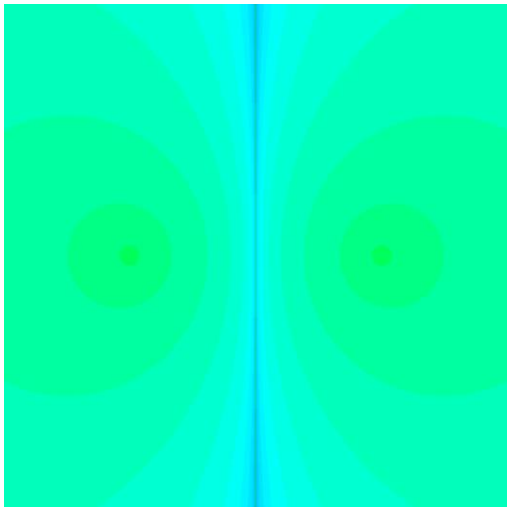
***Img (3)*** : $e^z - 1 = 0$



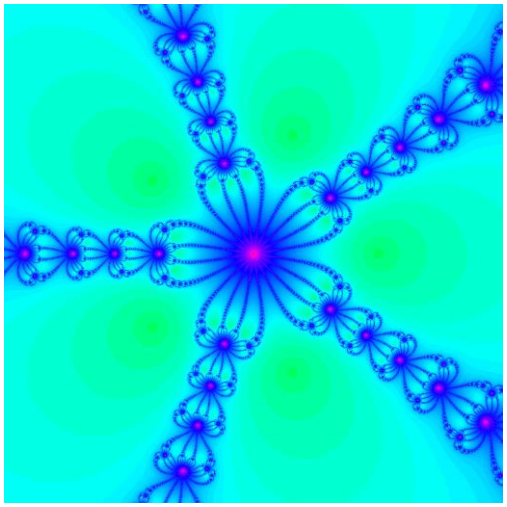
***Img (2)*** : $z^3 - 1 = 0$



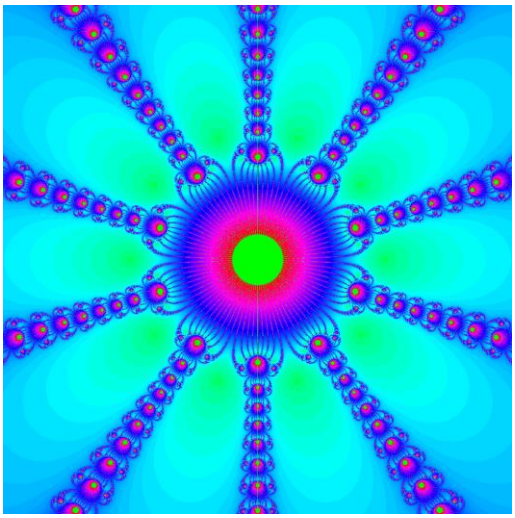
***Img (4)*** : $e^{z^2} - 1 = 0$



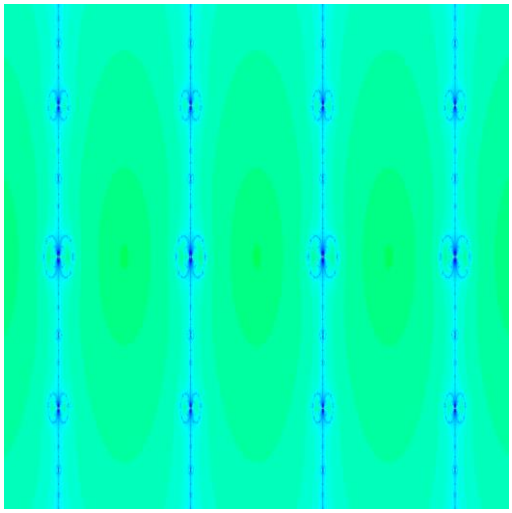
***Img (5)*** : $z^2 - 1 = 0$



***Img (7)*** : $z^5 - 1 = 0$



***Img (6)*** : $z^{10} - 1 = 0$



***Img (8)*** : sin(z) = 0, On a broader scale of view

***CONCLUSION:***

Newton's method is a process for approximating roots if a function. Usually, the method produces a sequence that convergence rapidly to a root. However if the initial point is taken on a Julia set, the behaviour can be chaotic as seen in the report.