Visualizing the distributions and isosurfaces of some traditional and non-traditional quaternion fractal sets

S. Halayka[*]

July 17, 2019

Abstract

After a concise introduction, the length, displacement, and magnitude distributions and isosurfaces related to some quaternion fractal sets are visualized.

1 Introduction

As discussed in the literature (see: [1–5]), an n-dimensional field of real magnitudes $|Z|$ results from calculating (via iteration) a fractal set when using a finite n-dimensional lattice of regularly spaced points as input. Some common values for n are 1 (real), 2 (complex), and 4 (quaternion). To be considered in this paper is the iterative equation $Z = Z^2 + C$, where C is some constant. The maximum number of iterations used in this paper is 8.

Sections 2, 3, and 4 show how to generate a 1, 2, and 4-dimensional fractal, respectively. Section 5 discusses trajectories, and their properties: length, displacement, and magnitude. Section 6 discusses histograms and distributions. Section 7 discusses isosurfaces. Section 8 shows the graphical histograms for the length, displacement, and magnitude properties of all trajectories. Section 9 shows the isosurfaces for the length, displacement, and magnitude-based iteration functions. Section 10 discusses future research.

In lieu of mathematical notation, C++ code is provided in this paper. The entire code can be downloaded from the following three locations:
`https://github.com/sjhalayka/basic_fractals` (Sections 2 - 5)
`https://github.com/sjhalayka/fractals_histograms` (Section 8)
`https://github.com/sjhalayka/fractals_isosurfaces` (Section 9)

References

[1] Norton A. *Generation and display of geometric fractals in 3-D*, in: SIGGRAPH 82: proceedings of the 9th annual conference on computer graphics and interactive techniques, 1982.

[*]sjhalayka@gmail.com

[2] Hart JC, Sandin DJ, Kauffman LH. *Ray tracing deterministic 3-D fractals*, Computer Graphics Vol. 23, Issue 3, 1989.

[3] Norton A. *Julia sets in the quaternions*, Computers & Graphics, 13(2):267-278, 1989.

[4] Bourke P. *Quaternion Julia fractals*, `http://paulbourke.net/fractals/quatjulia/`, 2001.

[5] Halayka S. *Some visually interesting non-standard quaternion fractal sets*, Chaos, Solitons & Fractals Vol. 41, Issue 5, 2009.

[6] Lorensen WE, Cline HE. *Marching cubes: A high resolution 3d surface construction algorithm*, ACM Computer Graphics. 21 (4): 163-169, 1987.

[7] Bourke P. *Polygonising a scalar field*, `http://paulbourke.net/geometry/polygonise/`, 1994.

2 Introduction to 1-dimensional fractals

The following code performs the iteration process for a 1-dimensional (real) input lattice:

```
#include <iostream>
#include <cmath>
#include <vector>
using namespace std;

float iterate_1d(vector<float> &trajectory_points,
                 float Z,
                 const float C,
                 const short unsigned int max_iterations,
                 const float threshold)
{
    // Add first point to trajectory
    trajectory_points.clear();
    trajectory_points.push_back(Z);

    for (short unsigned int i = 0; i < max_iterations; i++)
    {
        // Iterative equation
        Z = Z*Z + C;

        // Add additional point(s) to trajectory
        trajectory_points.push_back(Z);

        // Abort early if magnitude tends to infinity
        if (fabsf(Z) >= threshold)
            break;
    }

    // Return magnitude
    return fabsf(Z);
}

int main(void)
{
    // Lattice parameters
    const float x_grid_max = 1.5;
    const float x_grid_min = -x_grid_max;
    const size_t x_res = 30;
    const float x_step_size = (x_grid_max - x_grid_min) / (x_res - 1);

    // Fractal parameters
    const float C = 0.2f;
    const unsigned short int max_iterations = 8;
    const float threshold = 4.0f;

    // Start at one end
    float Z = x_grid_min;

    vector<float> trajectory_points;

    for (size_t x = 0; x < x_res; x++, Z += x_step_size)
    {
        float magnitude = iterate_1d(trajectory_points,
                                     Z,
                                     C,
                                     max_iterations,
                                     threshold);

        // If point is in the set, else ...
        if(magnitude < threshold)
            cout << '*';
        else
            cout << '.';
    }

    cout << endl;

    return 0;
}
```

The output shows that some of the input points are in the set (denoted by '*'), and some are not (denoted by '.'). It's nothing to write home about:

```
........**************........
```

The following code prints the 1-dimensional trajectories:

```
// ... duplicate code omitted for brevity

int main(void)
{
    // ... duplicate code omitted for brevity

    vector<float> trajectory_points;

    for (size_t x = 0; x < x_res; x++, Z += x_step_size)
    {
        float magnitude = iterate_1d(trajectory_points,
                                Z,
                                C,
                                max_iterations,
                                threshold);

        for(size_t i = 0; i < trajectory_points.size(); i++)
        {
            cout << trajectory_points[i] << endl;
        }

        cout << endl;
    }

    cout << endl;
    return 0;
}
```

Some example trajectories are:

```
-0.672414
0.65214
0.625287
0.590984
0.549262
0.501689
0.451692
0.404025
0.363236
```

and:

```
-1.5
2.45
6.2025
```

Note that the creation of this last collection of trajectory points was aborted early because the magnitude (6.2025) was greater than or equal to the threshold value (4.0). This last collection has less than ‘maximum iteration count’+1 = 9 points.

3 Introduction to 2-dimensional fractals

The following code performs the iteration process for a 2-dimensional (complex) input lattice:

```
#include <iostream>
#include <cmath>
#include <vector>
#include <complex>
using namespace std;

float iterate_2d(vector< complex<float> > &trajectory_points,
                 complex<float> Z,
                 const complex<float> C,
                 const short unsigned int max_iterations,
                 const float threshold)
{
    trajectory_points.clear();
    trajectory_points.push_back(Z);

    for (short unsigned int i = 0; i < max_iterations; i++)
    {
        Z = Z*Z + C;

        trajectory_points.push_back(Z);

        if (abs(Z) >= threshold)
            break;
    }

    return abs(Z);
}

int main(void)
{
    const float x_grid_max = 1.5;
    const float y_grid_max = 1.5;
    const float x_grid_min = -x_grid_max;
    const float y_grid_min = -y_grid_max;
    const size_t x_res = 30;
    const size_t y_res = 30;
    const float x_step_size = (x_grid_max - x_grid_min) / (x_res - 1);
    const float y_step_size = (y_grid_max - y_grid_min) / (y_res - 1);

    const complex<float> C(0.2f, 0.5f);
    const unsigned short int max_iterations = 8;
    const float threshold = 4.0f;

    complex<float> Z(x_grid_min, y_grid_min);

    vector< complex<float> > trajectory_points;

    for (size_t x = 0; x < x_res; x++)
    {
        Z = complex<float>(Z.real(), y_grid_min);

        for (size_t y = 0; y < y_res; y++)
        {
            float magnitude = iterate_2d(trajectory_points,
                                         Z,
                                         C,
                                         max_iterations,
                                         threshold);

            if(magnitude < threshold)
                cout << '*';
            else
                cout << '.';

            Z = complex<float>(Z.real(), Z.imag() + y_step_size);
        }

        Z = complex<float>(Z.real() + x_step_size, Z.imag());

        cout << endl;
    }

    cout << endl;

    return 0;
}
```

The output shows a more interesting, fractal shape:

```
..............................
..............................
..............................
..............................
..............................
....................**........
....................***.......
...................******.....
..................******......
.................********.....
.........**..***********......
.........***************......
......******************......
.....*******************......
....********************......
......********************....
......*******************.....
```

```
......******************......
......***************.........
......***********..**.........
.....********.................
......******..................
.....******...................
.......***....................
........**....................
..............................
..............................
..............................
..............................
..............................
```

4 Introduction to 4-dimensional fractals

The following code performs the iteration process for a 4-dimensional (quaternion) input lattice:

```cpp
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;


class quaternion
{
public:
    inline quaternion(void) :
    x(0.0f), y(0.0f), z(0.0f), w(0.0f)
    { /* default constructor */ }

    inline quaternion(const float src_x,
        const float src_y,
        const float src_z,
        const float src_w) :
        x(src_x), y(src_y), z(src_z), w(src_w)
        { /* custom constructor */ }

    inline float self_dot(void) const
    {
        return x*x + y*y + z*z + w*w;
    }

    inline float magnitude(void) const
    {
        return sqrtf(self_dot());
    }

    quaternion operator*(const quaternion &right) const
    {
        quaternion ret;

        ret.x = x*right.x - y*right.y - z*right.z - w*right.w;
        ret.y = x*right.y + y*right.x + z*right.w - w*right.z;
        ret.z = x*right.z - y*right.w + z*right.x + w*right.y;
        ret.w = x*right.w + y*right.z - z*right.y + w*right.x;

        return ret;
    }
    quaternion operator+(const quaternion &right) const
    {
        quaternion ret;

        ret.x = x + right.x;
        ret.y = y + right.y;
        ret.z = z + right.z;
        ret.w = w + right.w;

        return ret;
    }

    quaternion operator-(const quaternion &right) const
    {
        quaternion ret;

        ret.x = x - right.x;
        ret.y = y - right.y;
        ret.z = z - right.z;
        ret.w = w - right.w;

        return ret;
    }

    float x, y, z, w;
};


float iterate_4d(vector< quaternion > &trajectory_points,
                 quaternion Z,
                 const quaternion C,
                 const short unsigned int max_iterations,
                 const float threshold)
{
    trajectory_points.clear();
    trajectory_points.push_back(Z);

    for (short unsigned int i = 0; i < max_iterations; i++)
    {
        Z = Z*Z + C;

        trajectory_points.push_back(Z);

        if (Z.magnitude() >= threshold)
            break;
    }

    return Z.magnitude();
}


int main(void)
{
    float x_grid_max = 1.5;
    float y_grid_max = 1.5;
    float z_grid_max = 1.5;
    float x_grid_min = -x_grid_max;
    float y_grid_min = -y_grid_max;
    float z_grid_min = -z_grid_max;
    size_t x_res = 30;
    size_t y_res = 30;
    size_t z_res = 30;

    const float x_step_size = (x_grid_max - x_grid_min) / (x_res - 1);
    const float y_step_size = (y_grid_max - y_grid_min) / (y_res - 1);
    const float z_step_size = (z_grid_max - z_grid_min) / (z_res - 1);

    // 4th dimension is constant
    const float z_w = 0.0f;

    quaternion C;
    C.x = 0.3f;
    C.y = 0.5f;
    C.z = 0.4f;
    C.w = 0.2f;
    unsigned short int max_iterations = 8;
    float threshold = 4.0f;

    quaternion Z(x_grid_min, y_grid_min, z_grid_min, z_w);

    for (size_t z = 0; z < z_res; z++, Z.z += z_step_size)
    {
        cout << "Z␣slice␣" << z + 1 << "␣of␣" << z_res << endl;

        Z.x = x_grid_min;

        for (size_t x = 0; x < x_res; x++, Z.x += x_step_size)
        {
            Z.y = y_grid_min;

            for (size_t y = 0; y < y_res; y++, Z.y += y_step_size)
            {
                vector<quaternion> trajectory_points;

                float magnitude = iterate_4d(trajectory_points,
                                        Z,
                                        C,
                                        max_iterations,
                                        threshold);

                if (magnitude < threshold)
                    cout << '*';
                else
                    cout << '.';
            }

            cout << endl;
        }

        cout << endl << endl;
    }

    return 0;
}
```

The output shown below is one of the thirty 2-dimensional slices that make up a 3-dimensional fractal shape. This fractal shape is 3-dimensional, and not 4-dimensional, since the 4th dimension input value is held constant (0.0); this fractal shape is a 3-dimensional slice of a 4-dimensional fractal shape.

```
...

Z slice 15 of 30
..............................
..............................
..............................
..............................
..............................
```

```
..............................
.....................**.......
....................*****.....
....................*****.....
...................******.....
..................*******.....
.........*.......********.....
.........*.....**********.....
.....*.****.....*******.......
....********....*********.....
.....*********....********....
.......********....****.*.....
......**********....*.........
......*******.......*.........
.....*****....................
.....****.....................
.....****.....................
.....****.....................
.......**.....................
..............................
..............................
..............................
..............................
..............................
..............................

...
```

5 Introduction to trajectory properties

See Figure 1 for an example 2-dimensional (complex) trajectory. We visualize the complex trajectory because both the paper and the trajectory are 2-dimensional.

The following code shows how to measure the length, displacement, and magnitude of a quaternion trajectory:

```
void get_trajectory_properties(
    const vector<quaternion> &points,
    float &length,
    float &displacement,
    float &magnitude)
{
        if (points.size() == 0)
        {
                length = displacement = magnitude = 0.0f;
                return;
        }
        else if (points.size() == 1)
        {
                length = displacement = 0.0f;
                magnitude = points[0].magnitude();
                return;
        }

        length = 0.0f;

        for (size_t i = 0; i < points.size() - 1; i++)
                length += (points[i + 1] - points[i]).magnitude();

        displacement = (points[points.size() - 1] - points[0]).magnitude();
        magnitude = points[points.size() - 1].magnitude();
}
```

The following code shows how to obtain and use the trajectories' properties:

```
// ... duplicate code omitted for brevity

for (size_t y = 0; y < y_res; y++, Z.y += y_step_size)
{
        vector<quaternion> trajectory_points;

        iterate_4d(trajectory_points, Z, C, max_iterations, threshold);

        float length = 0.0f;
        float displacement = 0.0f;
        float magnitude = 0.0f;

        get_trajectory_properties(trajectory_points,
                length,
                displacement,
                magnitude);

        if (magnitude < threshold)
                cout << '*';
        else
                cout << '.';
}

// ... duplicate code omitted for brevity
```

6 Introduction to histograms

A histogram is a diagram that illustrates the frequency of some data. For instance, the histogram of the following collection of 1-dimensional (integer) input data:

```
0, 0, 1, 1, 1, 1, 2, 3, 3, 4, 5, 5, 5
```

is

```
4  *
3  *   *
2 ** * *
1 ******
  012345
```

Note that the value 1 is the most frequent (four times), with 5 close behind (three times). This is reflected in the histogram. This histogram's maximum value is 5, and its mode is 1. Also note that the number of '*' symbols in the histogram is equal to the number of elements in the collection.

For real (generally non-integer) data, in this paper, the input is distributed into bins of some constant width.

7 Introduction to isosurfaces

An isosurface is a 2-dimensional surface that exists in 3-dimensional space. To generate an isosurface, the (effectively 3-dimensional) field of real magnitudes $|Z|$ is used as input, along with a single real isovalue. The Marching Cubes [6, 7] algorithm converts those inputs into an isosurface. In this paper, the isovalue is equal to the threshold value (like 4.0, or 20.0). In essence, any field locations with a value less than the threshold value are inside of the isosurface, and any locations with a value greater than the threshold value are outside of the isosurface. The output generated by Marching Cubes is a collection of triangle primitives.

See Figure 2 for an example isosurface, rendered using OpenGL.

8 Graphical length, displacement, and magnitude histograms

The following code shows how to create some data collections (stored in a C++ STL vector), and ideally convert them into histograms and saved as .PNG files:

```
// ... duplicate code omitted for brevity

void generate_and_save_histogram(const vector<float> &input,
                                 const char *const file_name)
{
    // ... insert OpenCV code here to generate and save a histogram
    // ... see: https://github.com/sjhalayka/fractals_histograms
}


int main(void)
{
    // ... duplicate code omitted for brevity

    vector<float> lengths;
    vector<float> displacements;
    vector<float> magnitudes;

    for (size_t z = 0; z < z_res; z++, Z.z += z_step_size)
    {
        cout << "Z␣slice␣" << z + 1 << "␣of␣" << z_res << endl;

        Z.x = x_grid_min;

        for (size_t x = 0; x < x_res; x++, Z.x += x_step_size)
        {
            Z.y = y_grid_min;

            for (size_t y = 0; y < y_res; y++, Z.y += y_step_size)
            {
                vector<quaternion> trajectory_points;

                iterate_4d(trajectory_points,
                        Z,
                        C,
                        max_iterations,
                        threshold);

                float length = 0.0f;
                float displacement = 0.0f;
                float magnitude = 0.0f;

                get_trajectory_properties(trajectory_points,
                                          length,
                                          displacement,
                                          magnitude);

                lengths.push_back(length);
                displacements.push_back(displacement);
                magnitudes.push_back(magnitude);
            }

            cout << endl;
        }

        cout << endl << endl;
    }

    generate_and_save_histogram(lengths, "lengths.png");
    generate_and_save_histogram(displacements, "displacements.png");
    generate_and_save_histogram(magnitudes, "magnitudes.png");

    return 0;
}
```

Let's visualize the distributions of the trajectories' lengths, displacements, and magnitudes for all points (both those points in the set as well as those points not in the set), as generated in the above code.

The histograms in Figures 3 - 5, where threshold equals 4.0, show how the maximum length is generally greater than the maximum displacement. This is also generally the case for the length and displacement per individual trajectory, which is generally indicative of curved trajectories – the trajectories generally meander because there are bends.

In a lot of the cases (but not all cases) a curved trajectory forms a loop, which gives rise to the commonly-used name 'orbit'. However, most of the time the loop is not quite exact, and so all 'maximum iteration count'+1 = 9 points per trajectory end up being distinct. This means that when a curved trajectory forms an orbit, the orbit is generally not quite perfect – the curved trajectory is likely jittery, or precessing, or spiral-shaped, or all three.

Also, see Figures 6 - 8, where threshold equals 20.0.

These histograms in Figures 3 - 8 show that there are differences amongst the distributions where threshold equals 4.0, but that those differences largely go away where threshold equals 20.0.

These graphical (.PNG) histograms in Figures 3 - 8 were generated by using OpenCV.

9 Renderings of length, displacement, and magnitude-based fractals

It must be asked: are there new fractals to be discovered by limiting the length or displacement, like we traditionally do with magnitude $|Z|$? It turns out that the answer, where $Z = Z^2 + C$, is 'not really'. See Table 1 for renderings of the isosurfaces of the fractal sets. Where threshold equals 20.0 and length, displacement, and magnitude are all represented by similar distributions, it is found that these three properties also encode similar isosurfaces.

The following code shows how to generate fractals using the length, and displacement properties, instead of the traditional magnitude property:

```
// ... duplicate code omitted for brevity

float iterate_4d_length(vector< quaternion > &trajectory_points,
                        quaternion Z,
                        const quaternion C,
                        const short unsigned int max_iterations,
                        const float threshold)
```

```
{
    trajectory_points.clear();
    trajectory_points.push_back(Z);

    float length = 0.0f;
    float displacement = 0.0f;
    float magnitude = 0.0f;

    for (short unsigned int i = 0; i < max_iterations; i++)
    {
        Z = Z*Z + C;

        trajectory_points.push_back(Z);

        get_trajectory_properties(trajectory_points,
                                  length,
                                  displacement,
                                  magnitude);

        if (length >= threshold)
            break;
    }

    return length;
}

float iterate_4d_displacement(vector< quaternion > &trajectory_points,
                        quaternion Z,
                        const quaternion C,
                        const short unsigned int max_iterations,
                        const float threshold)
{
    trajectory_points.clear();
    trajectory_points.push_back(Z);

    float length = 0.0f;
    float displacement = 0.0f;
    float magnitude = 0.0f;

    for (short unsigned int i = 0; i < max_iterations; i++)
    {
        Z = Z*Z + C;

        trajectory_points.push_back(Z);

        get_trajectory_properties(trajectory_points,
                                  length,
                                  displacement,
                                  magnitude);

        if (displacement >= threshold)
            break;
    }

    return displacement;
}

// ... duplicate code omitted for brevity
```

10 Future research

Future research will include a look at different iterative equations, such as $Z = Z^5 + C$ or $Z = \sin(Z) + C * \sin(Z)$:

```
// For iterative equations like Z = sin(Z) + C*sin(Z)
quaternion sin(const quaternion &in)
{
    quaternion ret;

    const float mag_vector = sqrtf(in.y*in.y + in.z*in.z + in.w*in.w);

    ret.x = sin(in.x) * cosh(mag_vector);
    ret.y = cos(in.x) * sinh(mag_vector) * in.y / mag_vector;
    ret.z = cos(in.x) * sinh(mag_vector) * in.z / mag_vector;
    ret.w = cos(in.x) * sinh(mag_vector) * in.w / mag_vector;

    return ret;
}
```

Future research will also include a look at the sinuosity (length divided by displacement) of a trajectory.

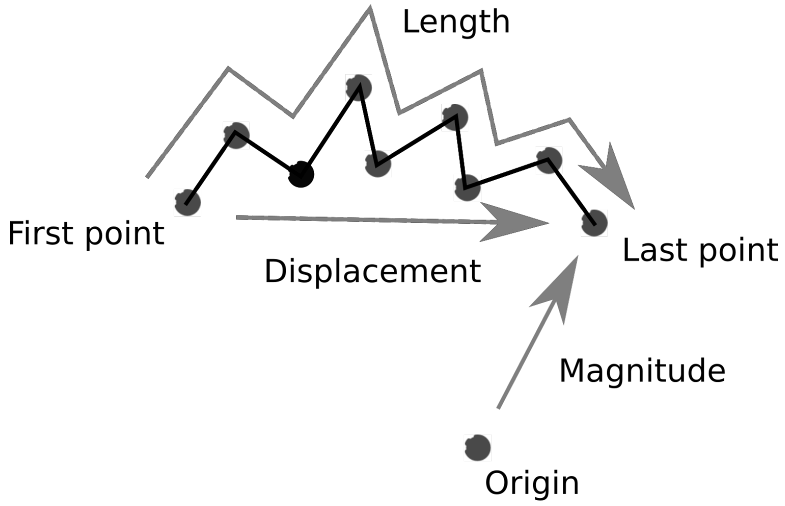


Figure 1: Length, displacement, and magnitude of a 2-dimensional (complex) trajectory.

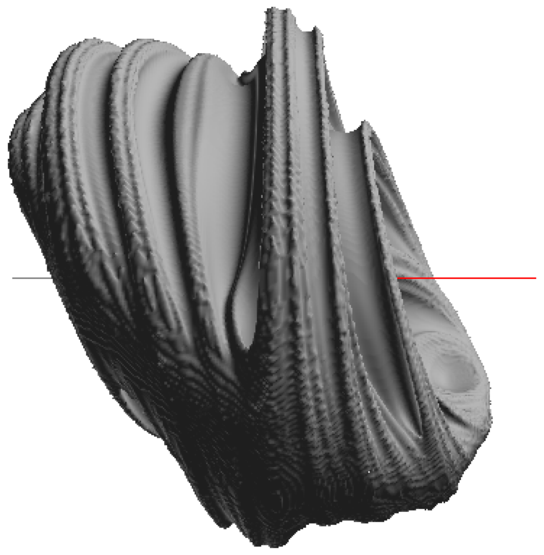


Figure 2: Isosurface of a quaternion fractal set, where threshold (isovalue) is equal to 4.0. The isosurface is a closed mesh (no cracks, no holes) that consists of a collection of triangles.

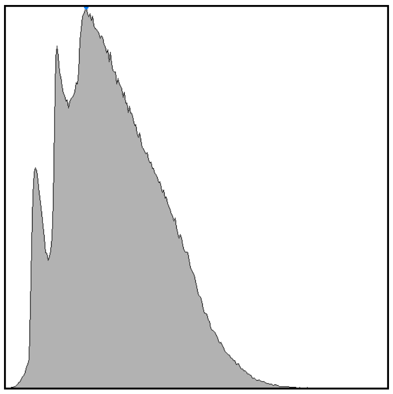


Figure 3: Lengths of $Z = Z^2 + C$, for all points (both those points in the set as well as those points not in the set). Threshold equals 4.0. For this histogram the maximum length is 36.7, mode: 7.7.

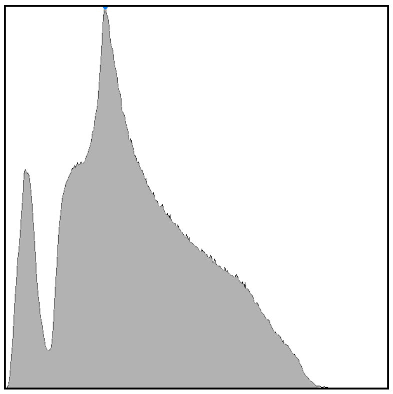


Figure 4: Displacements of $Z = Z^2 + C$, for all points. Threshold equals 4.0. For this histogram the maximum displacement is 18.6, mode: 4.8.

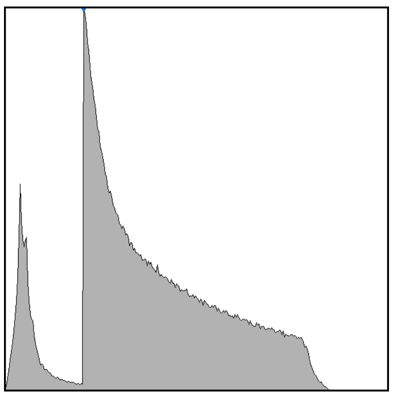


Figure 5: Magnitudes of $Z = Z^2 + C$, for all points. Threshold equals 4.0. For this histogram the maximum magnitude is 16.7, mode: 3.4.

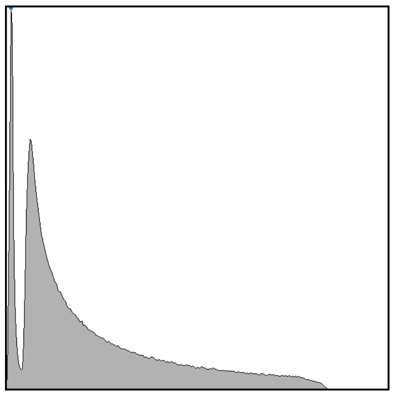


Figure 6: Lengths of $Z = Z^2 + C$, for all points. Threshold equals 20.0. For this histogram the maximum length is 453.6, mode: 4.5.

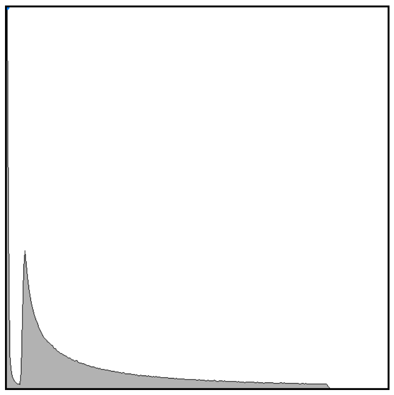


Figure 7: Displacements of $Z = Z^2 + C$, for all points. Threshold equals 20.0. For this histogram the maximum displacement is 402.4, mode: 0.0.

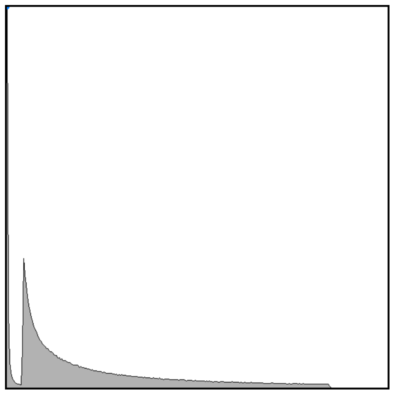


Figure 8: Magnitudes of $Z = Z^2 + C$, for all points. Threshold equals 20.0. For this histogram the maximum magnitude is 400.6, mode: 0.0.

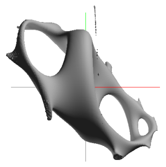
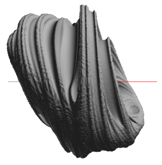
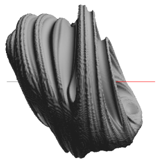
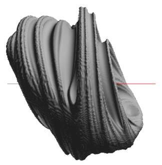
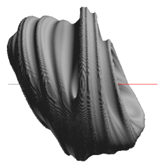
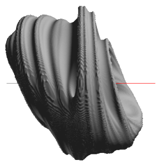
Threshold	Length	Displacement	Magnitude
4.0			
20.0			

Table 1: Length, displacement, and magnitude-based isosurfaces (length, or displacement, or magnitude remains below the threshold during iteration), where threshold equals 4.0 or 20.0.