Complex Codomain Coloring, the Mandlebrot Set, and Primes

Jake Looney
Undergrad at the University of Tennessee Knoxville

January - March, 2022

Contents

1 Motivation

Let $f : \mathbb{C} \to \mathbb{C}$ be a function from the complex plane to itself. The standard and practical way to visually represent f is with *domain coloring*, whereby you take a region on the complex plane, apply the function to a grid a points within that region, and color each pixel based on the output of those points. This, in effect, shows you where "each point goes" after applying the function. However, what if you want to see where "each point is from"? To do that, we use *codomain coloring*, which is the concern of this paper. Codomain coloring works by taking a large set of points, applying the function to each point, and then storing each output point with its corresponding input point. Then, you select a region on the plane, divide that region into a grid, and color each pixel based on the *input point* that is nearest to the pixel. In metaphor, it is like giving each point a name, having those points move around the plane, and then coloring pixels based off the name of the closest point.

There's not many reasons to do this: it goes against the standard, it's computationally intensive, it's slow as all get out, it's blocky, it's imprecise, and it only shows one point for each pixel. However, despite this, I did discover some pretty interesting results while shoving functions into it.

2 Code explanation

There is an attached GitHub repository containing the code. You can view it here: `https://github.com/6a6c/jraph`

The function `createPicture` takes a map of pixel indices and complex points to output a bitmap image. It mainly utiliizes a bitmap implementation I stole from one of my classes, but importantly it colors points in the image based on their argument and magnitude:

```
z = fit->second;
mag = abs(z);
ar = arg(z);

pixarray[fit->first].red = min(5*mag, 255.0);
pixarray[fit->first].blue = min(40*ar, 255.0);
pixarray[fit->first].green = min(100*mag, 255.0);
```

This can be changed in order to alter the colorscheme.

There are three main functions that are used to create these graphs. The function `makeFunction` creaates a multimap of doubles corresponding to output magnitudes and a pair of complex points corresponding to input and output points of a function. To do this, it applies a function to a region of complex points and adds them to the multimap:

```
for( i = -1 * pts; i < pts; i++){
  for( j = -1 * pts; j < pts; j++){

    z = complex<double>(aCoeff * i * i * i, aCoeff * j * j * j);
```

```
    output = func(z);

    // if(abs(output) > 2) continue;

    function->insert(make_pair(abs(output), make_pair(output, z)));
  }
}
```

The points selected to be input points for the function are scaled based on a cubic function. This means that there are more points around 0 (the interesting parts) and less points further out. The coefficient on the cubic as well as the number of points can be changed. The output is set to the `func` function, which can be altered to change the function. Note, the commented out line, which will ignore points that diverge to infinity when calculating the Mandlebrot function. This will help keep the map small and runtime down.

The function `makeMap` is used to create the maps that are passed to `createPicture`. To do this, it is passed a multimap created by `makeFunction` as well as the boundaries of a region on the complex plane. It inserts points into the map based on their preimage by passing points to `preImage` based on the region.

```
for(i = 0; i < w * h; i++){
  complex<double> z(realLeft + ((i\%w) * ((realRight - realLeft)/w)),
                        imagTop - ((i/w) * ((imagTop - imagBot)/h)));

  complex<double> pre = preImage(z, function);

  ret->insert(make_pair(i, pre));

}
```

`preImage` works by first isolating a range of the function multimap with magnitudes within a certain ammount of the image point z. Then, it iterates through this range and finds the point with the least distance from z. The input point that corresponds to this output point (that is, second part of the pair in the multimap).

```
double mag = abs(z);
double d1, d2, best;
map< double, pair< complex<double>, complex<double> >
    >::const_iterator low, high, fit, bit;
size_t i = 0;

low = func->lower_bound(mag-(ep * mag));
high = func->upper_bound(mag+(ep * mag));

bit = low;
best = 100000;
for(fit = low; fit != high; fit++){
  d1 = sqrt( pow(z.real() - fit->second.first.real(), 2) +
```

```
        pow(z.imag() - fit->second.first.imag(), 2) );

  if(d1 < best) {
    best = d1;
    bit = fit;
  }

}

return bit->second.second;
```

The result, after calling `preImage` on all the points for the picture, is a preimage point for each point in the codomain.

3 That's cool, but what does it look like?

For reference, it's important to see what the unaltered complex plane looks like:

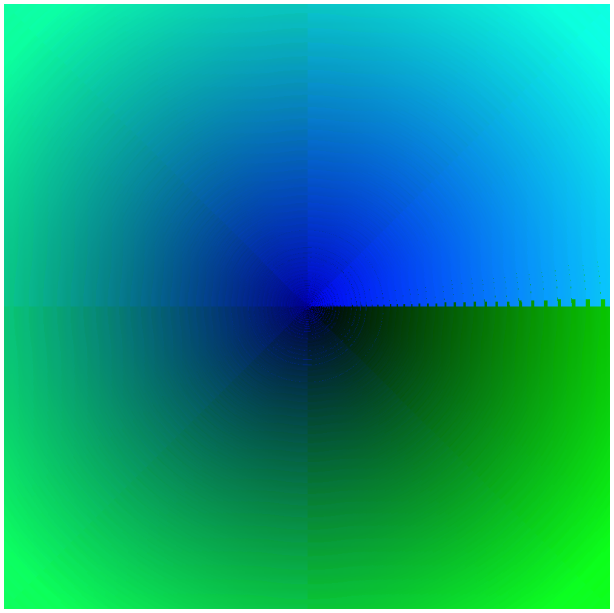

Figure 1: $f(z) = z$

This image was obtained by using $f(z) = z$ as the function through the codomain grapher. The real and imaginary axes runs directly through the middle. Because all points in the domain this function equal their outputs in the codomain, this graph can be used as the domain for other functions.

Also, notice here some of the limitations in this method in the artifacts the graph produces. Near the positive real axis, there are some instances of the green "jumping up" into

the first quadrant.

Here's another example of the codomain grapher:

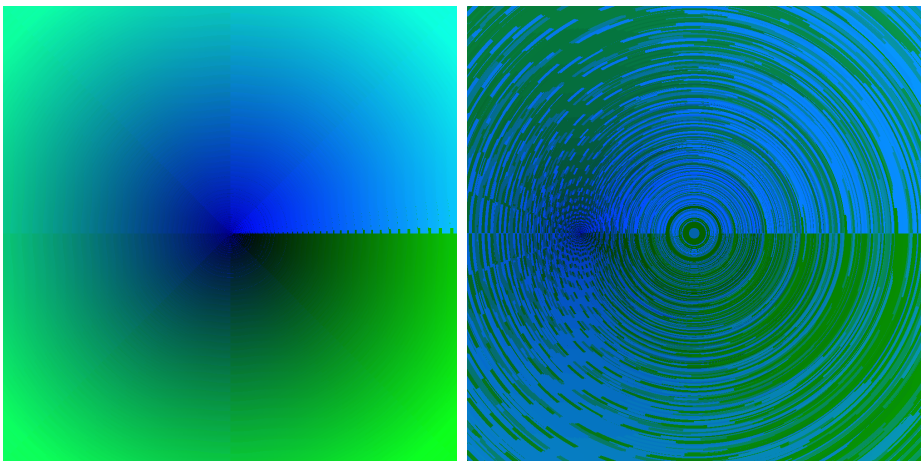

Figure 2: The complex plane before and after applying $f(z) = z^3 - 1$

This is the function $f(z) = z^3 - 1$. You can think of these two pictures as the points in the left "hopping over" to the points on the right. Compare this function with it's corresponding domain graph:

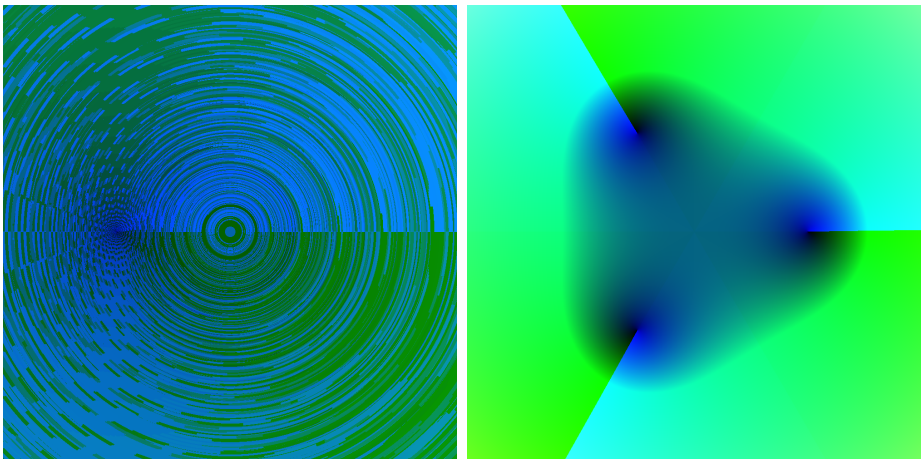

Figure 3: The codomain graph(left) and domain graph(right) of $f(z) = z^3 - 1$

Both graphs show the same interval on the complex plane, but one shows where points in that interval go and one shows some points that end up in that interval. Notice, the key word there, codomain coloring shows *some* of the points in an interval, not all.

This highlights the biggest practical restriction with this method of graphing: points in the codomain only display one preimage point despite potentially having more preimage points. For intance, we can clearly see from the domain graph that this function has 3 roots. However, in the codomain graph, each pixel can only be colored once, so 0 only shows the value of the root $z = \frac{-1+i\sqrt{3}}{2} = -0.5000 + 0.8660i$. The other two roots are not visible in the

codomain graph.

I hope these examples illustrate how to interpret these graphs. After trying this with a handful of functions, the obvious next step was:

4 Jamming the Mandlebrot function in this

To compute the Mandlebrot set, one only needs to use points with a magnitude to less than two to genereate the function. Anypoints with a magnitude of greater than two will diverge to infinity and will not be apart of the set. So, to genereate the collection of Mandlebrot graphs, I set `pts` to 2000 and `aCoeff` to 10^{-9}

In addition to this, images shown are on the inteveral $-0.5 \leq Re(z), Im(z) \leq 0.5$. This is because this is the most interesting parts, while the area outside this is mainly concentric circles.

Let's take a look at the first one of these I tried, using 500 iterations before bailing out of the Mandlebrot function:

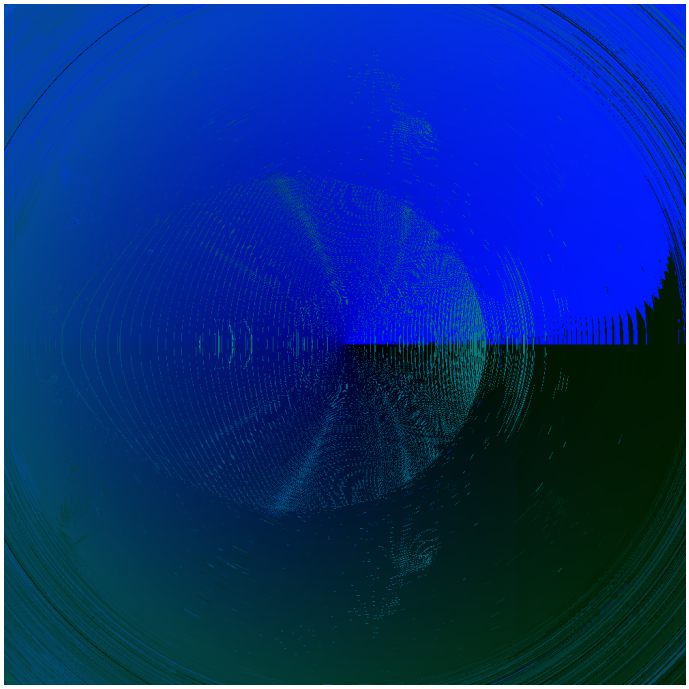

Figure 4: 500 iterations

Ain't that something! I was really suprised at the detail that shows up and the shape that points take.

It feels very similar to the Mandlebrot shape, yet completely different. It kinda reminds me of the Buddhabrot set, which makes sense as in a sense, this is one "snippet" of a Buddhabrot drawing.

I almost walked away from it but then I remebered that points in iterative functions

iterate. So I tried changing the number of iterations:

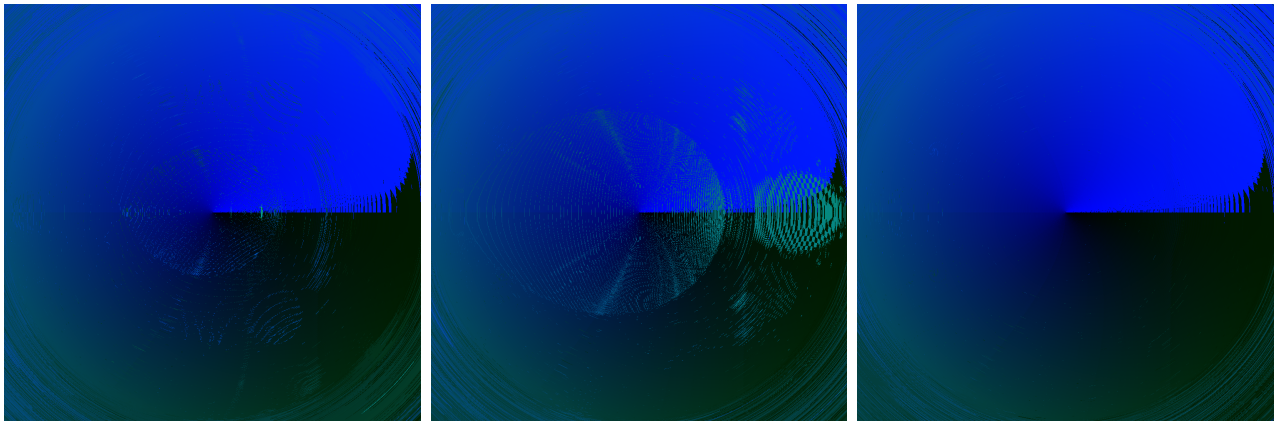

Figure 5: LtR: 501 iterations, 502 iterations, and 503 iterations

We see some similar results. 501 has some detail but is nothing like 500. 502 looks exactly like 500 but with an extra little bit on the right of it. 503 has basically no detail however. Wait a second...

5 Hey, why are the prime numbers here?

So, over the course of about a day and a half, I had my computer calculate the graphs for the first 1000 iterations of the Mandlebrot function. However, after about 100 iterations, all prime number iterations have strikngly little detail compared to composite iterations and the primes under 100. Further, iterations where one of the prime factors is a prime greater than 100 have less detail. This really confused me for a good couple of weeks. Let me explain why:
So, points in the Mandlebrot set will have one of three properties:

1. The point will move to a limit point as the function,
2. The point will be a part of a cycle, or
3. The point moves around chaotically

The last one is confusing but luckily accounts for a negligable number of points, so we can effectively ignore it.

Points with the first property are the unchanging blue gradient behind all the green and turquoise points.

The points that cycle are the green and turquoise points. They are the interesting ones for us, so we should go into more detail:

Points in Mandlebrot set will cycle if they are in the auxillary bulbs of the set (that is, they are not in the main cardioid). More specifically, a point will be in a n-cycle where n is determined by what bulb the point is in:

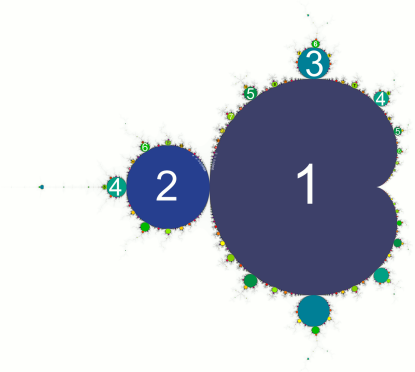

Figure 6: From `https://en.wikipedia.org/wiki/Mandelbrot_set`

This explains why every even iteration looks mostly the same: they all display points from the bulb immediately to the left of the cardioid. However, here we also see what's confusing about the lack of prime numbers: there should points in an n-cycle for any natural number n. So, why is this?

Well, indeed, if we had infinite computational power, we would see points for every iteration. However, we started with only a couple million points. The number of points that are part of n-cycle for suffiecently large n is too small to notice because the bulbs take up less and less area as n grows.

So, the primes over 100 don't show up because our sample of points was too small. If we were to add more and more points in the map, we would see more and more detail in later iteations of the function.

6 Conclusion

I hope reading through this writeup and looking at the graphs generated was enjoyable and worthwhile. Again, practical uses for this method of graphing functions is extremely limited. That said, working through creating this and investigating it have been extremely insightful for me and have helped me to better understand complex functions.