~~Curious Case of Complex Codomain Coloring~~ getting high and finding primes

Jake Looney
Undergrad at the University of Tennessee Knoxville

January 2022

Contents

1 Motivation

I ain't gonna lie, I was really high when I wrote the code for this and was trying to just make a domain coloring grapher. I had seen a couple of pictures on Wikipedia using domain coloring and saw the term so I decided to just have a go at it. I was thinking that "domain coloring" would make sense to color the domain and then apply the function. Of course, normal domain coloring works the other way, where you apply the function then color based on where the points land. So, what resulted is the other direction of the usual way to do this: points are colored based on their preimage. This requires some working around in order to work with general functions. There's not many reasons to do this: it goes against the standard, its computationally intensive, slow as shit, blocky, imprecise, and only shows one point in the preimage per point in the image. However, despite this, I did discover something pretty interesting while shoving fun functions into it. But first, an explanation of the code.

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 mainily utiliizes a bitmap implementation I stole from class, 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, but I have no idea what the hell LAB color is so I will have to come back to it.

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 = complex<double>(0,0);
    k=0;
    while(k < mandoIters && abs(output) < 2){
      output = pow(output, 2) + z;
```

```
      k++;
    }
    if(abs(output) > 2) output = complex<double>(INFINITY, 0);

    //output = sin(z);
    //output = (((z * z) + complex<double>(-1,0))
    //           * pow((z + complex<double>(-2, -1)), 2))
    //            /(z*z + complex<double>(2,2));
    //output = ((z * z * z) + complex<double>(-1, 0));
    //output = z;

    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. Currently, the function being applied in this code is the Mandlebrot function, with the ability to change the number of iterations (see section 4). However, other example functions are commented out, and the comments can be changed to change the function applied.

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 do the pictures look like?

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

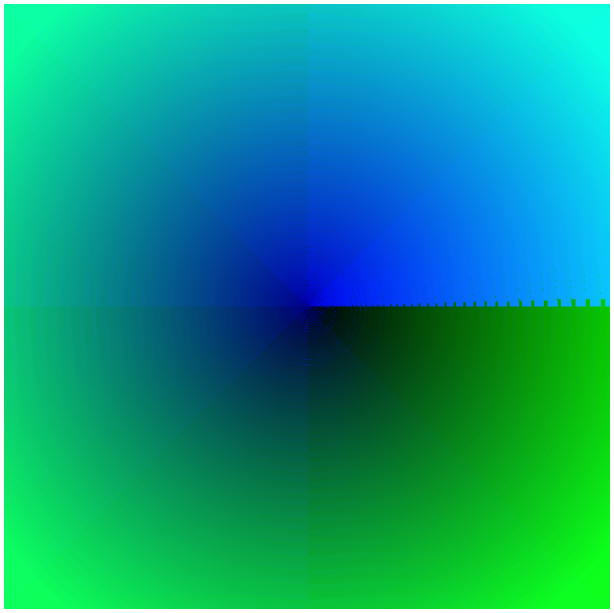

Figure 1: $f(z) = z$

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. ¡Find out why this is and describe it¿
Here's another example of the codomain grapher:

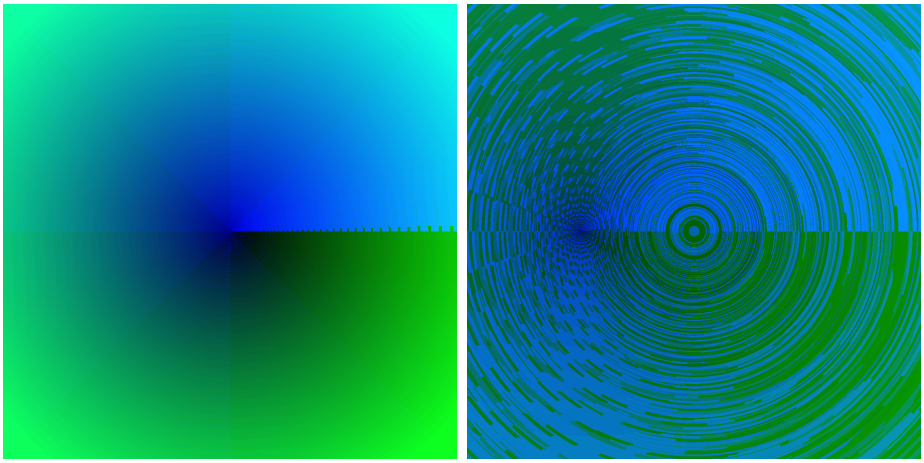

Figure 2: The complex plane before and after applying $f(z) = z^3 - 1$

This is the function $f(z) = z^3 - 1$. Compare this function with it's corresponding domain graph:

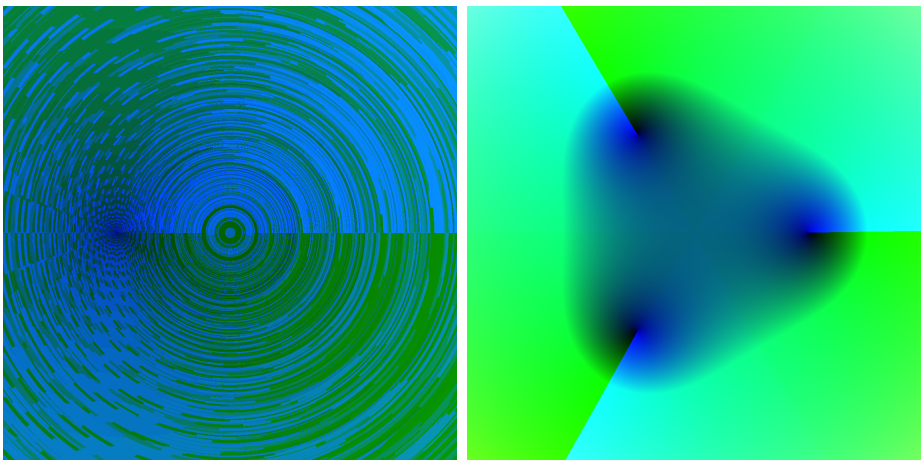

Figure 3: The codomain graph(left) and domain graph(right) of $f(z) = z^3 - 1$

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. 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, as each point

can only be colored once, 0 only shows the value of the root $z = \frac{-1+i\sqrt{3}}{2} = -0.5000 + 0.8660i$, omitting the other two roots.

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

4 Jamming the Mandlebrot function in this

To graph the Mandlebrot with the codomain grapher, 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[*citationneeded*]. So, to genereate the collection of Mandlebrot graphs, I set pts to 2000 and the aCoeff to 10^{-8} (which is actually one order of magnitude higher than it should've been and I literally didn't notice until right now while I writing this there is a lot of points that are completely useuless because they just diverge away because of fucking course they do what was I thinking god bless i should re run the entire program and maybe also write it in a way that shortens running time god bless)

In addition to this, most image generated are on the interval $-0.5 \leq Re(z), Im(z) \leq 0.5$. This is because this is the pretty and interesting part and everything else is just kinda circles.

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

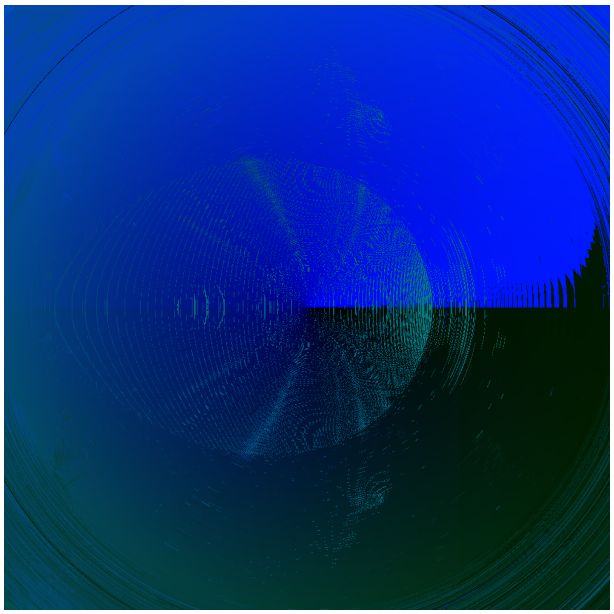

Figure 4: 500 iterations

Wowie ain't that cool! Lets do that again but this time, we'll edit the interval and image size so that it makes a good desktop wallpaper:

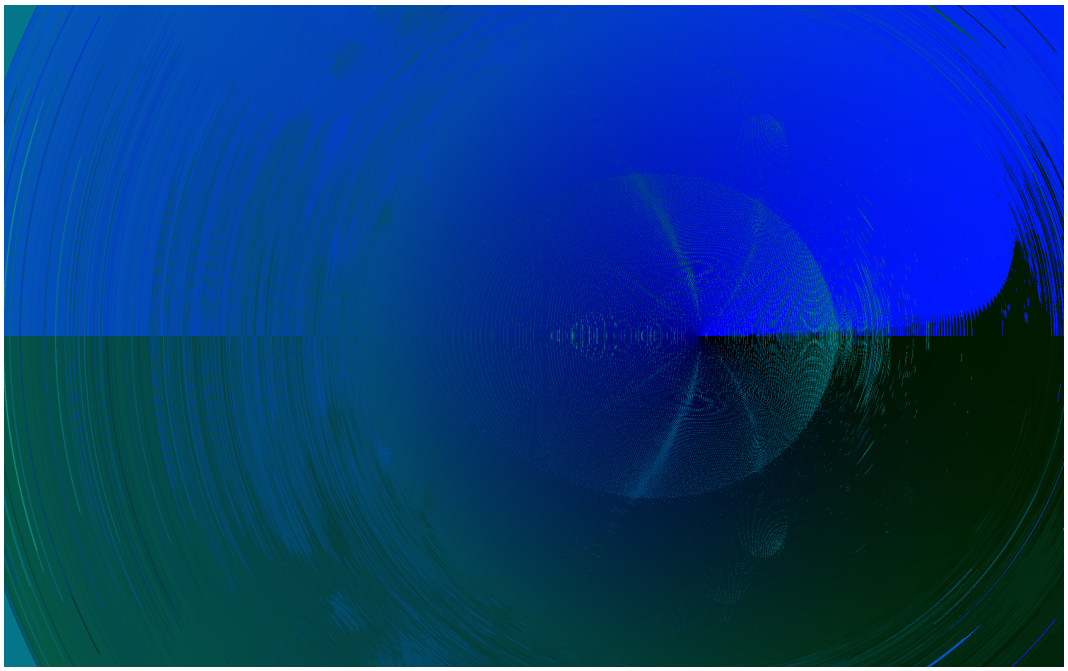

Figure 5: A bitching wallpaper, send it to your colleagues!

Wow that's nice. I was really suprised at the detail that shows up. I almost walked away from it but then I remebered that points in iterative functions iterate. So I tried changing the number of iterations:

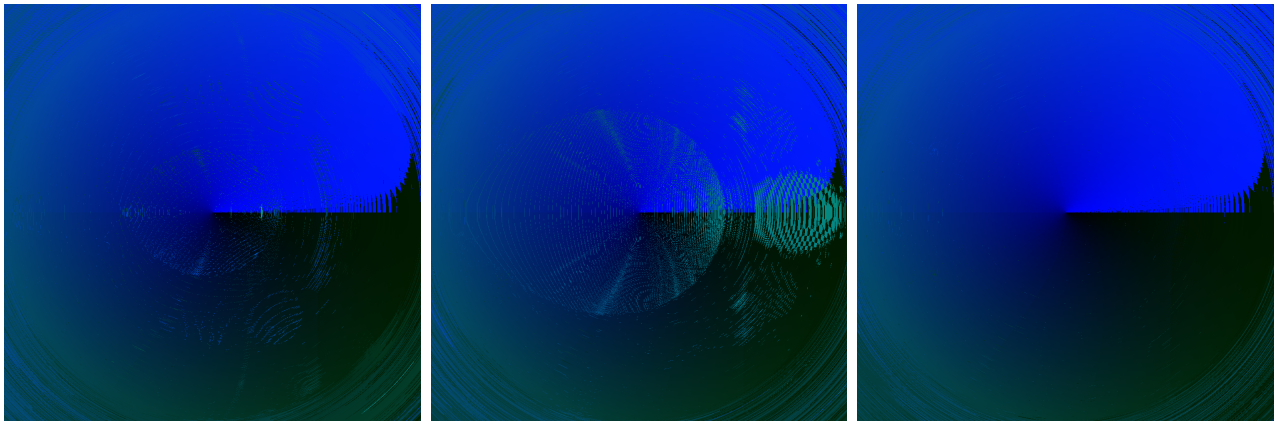

Figure 6: LtR: 501 iterations, 502 iterations, and 503 iterations

Oh wow that's cool. 501 has some detail but is nothing like 500. 502 looks exactly like 500 but with an extra little thingy on the right of it. 503 has basically no detail though. Wait a second...

5 Ding dong, get the door, its the motherfucking prime numbers!

Yooooooooooo!!!!!!!