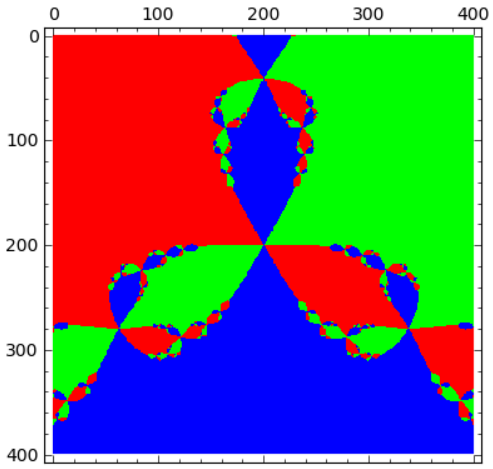
Generating and Plotting Fractals with Sage

Alex Carney

December 12, 2013



Introduction

Fractals are self repeating patterns that can occur when a dynamical system has unstable fixed points. For example if we use Newton's approximation method (see section 1) to find solutions to the equation $z^3 - 1 = 0$, then depending on the starting value we use with Newton's method, the root that we converge to could be different. In regions where this system is unstable, then small changes in the starting value can dramatically influence which root we end up converging to, so if we plot these starting values assigning them a colour corresponding to the root that they converge to we will be able to see the pattern that it generates and the resulting fractal.
So how will we plot this using Sage? Well this is basically a three step process:

1. For a given starting value apply Newton's method to it for a few iterations.
2. Test it to see which root it is closest to and assign it an integer that corresponds to the root.
3. Finally use the data we generated and use it to construct a matrix that will represent the fractal and use Sage's *matrix_plot* function to generate an image that represents the fractal.

1 Newton's Approximation

So in the previous section you saw me mention Newton's approximation method, in this section I will briefly outline what this and how it relates to plotting a fractal.

Newton's approximation is a numerical method that can be used to find roots of equations, it is an iterative formula given by:

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

However if you use this and your equation has multiple roots you have no control over which root this method will find and two similar starting values could lead you to a completely different roots. When this happens you actually end up with a dynamical system and certain regions can become increasingly sensitive to your initial value when you run this approximation.

It's this property that generates the fractal, if we run this dynamcial system using a grid of cordinates on the Argand plane as starting values and we assign the result an integer value corresponding to which root it converged to. However due to the time it would take to compute the root exactly for each point we will only run the system for a few iterations since Newton's approxmiation converges very quickly[4] and the points will get close enough for us to decide which root it was converging to. Here is my implementation of this in Sage:

```
def newtonApprox(f, f_dash, x_0 ,n):
    ans = x_0

    for i in xrange(0,n):
        if f_dash(ans) == 0:
           return ans
        ans = ans - (f(ans)/f_dash(ans))

    return CDF(ans)
```

Due to limited space I have ommitted the code comments, please refer to [2] for full details regarding this implementaion.

2 Testing the Root

So once we have ran a point through the approximation we need to determine which root a particular point was converging to. Since we are dealing with complex numbers this is quite easy, since taking the modulus of number gives you it's distance from the origin and taking the modulus of the difference gives you the distance between those 2 points. So if we do this for each root for every point we test and get the minimum value out of them all, then we know which point we were converging to.

Here is the implementation in Sage:

```
def testRoot(f, z, solns):
    test = [float((z - solns[0].rhs()).abs()),
        float((z - solns[1].rhs()).abs()), (z - solns[2].rhs()).abs()]

    return test.index(min(test))
```

To save on space I have omitted the code comments so please refer to [2] where this implementation is explained in full.

3 Plotting the fractal

The above two functions provide us with all the data we need to plot a fractal in Sage the final step is to create a couple of functions that will process a grid of points on the Argand plane and output a list of lists that will be used in the matrix constructor when we get Sage to plot it. The first is a function that will let us create a grid of points to test of arbitary size and resolution, unfortanately the code for this has had to be omitted due to limited space but please refer to [2] for the full details.

Finally the last function is the one that ties all these together and gives us the final list containing rows each represented by a list and the elements of this list will be of integer values from the set $\{0, 1, 2\}$, each of these correspnd to a unique root to the equation $z^3 - 1 = 0$ and will be represented by a separate color when it is plotted by sage.

```
def fractalGrid(grid_coords):
    f(x) = x^3-1
    f_dash = diff(f,x)
    soln = solve(f,x)
    rows = []
    for i in grid_coords:
        columns = []
        for j in i:
            columns.append(testRoot(f,newtonApprox(f,f_dash,j,5),soln))
        rows.append(columns)
    return rows
```

Once again please refer to [2] for full details this implementation.

Now all that's left are three short commands to get Sage to plot the fractal and generate an image file for us.

```
grid = compileGrid(-1,1,-1,1,0.0005)
F = matrix(fractalGrid(grid))
F.plot(cmap='brg_r')
```

The rest of the magic is now performed by Sage it will go through each element returned by *fractalGrid* and construct a matrix, then when the *matrix_plot* function is called it will go through each item and assign it a colour based on the colour map specified and generate an image file containing the fractal.

4 Final thoughts and considerations

What I have just outlined previously is a working method to plot fractals, however there many improvements to be made, first of all because of how the *testRoot* is coded this method will only work with equations with three roots. The function needs to be generalised to cope with an arbritary number of roots.

This method is also very computationally intensive and uses a lot of memory, it took around 30 minutes to generate the image you saw at the start of this document, and I frequently ran into issues where I would run out of memory mid computation. I have already started to address this by refactoring my code to use generators [1] whenever possible since they use hardly any memory and provide a small speed increase[3]. However at this time the code that uses generators will only generate a simplistic version of what you saw at the start of this document, missing out on alot of detail even when high resolution grids are used.

Another way to speed up this process is to try and minimse the amount of computation the computer must perform. Notice how in the image above there are large areas of a single colour? Well there is a lot of

wasted computation in those areas when it's "obvious" which colour it's going to end up with. Since this is a dynamical system there are a number of tests we can do. There is the stability test where you can see for which values of a certain parameter the system becomes increasingly sensitive to changes to the starting value, which would allow us to focus the computations on the sensitive areas and only perform a few checks in the more stable regions. There is also the attractor test, where you can find regions for which starting values in those regions will converge towards the root.

So as you can see there are plenty of elements to this method which I can investigate further and try to improve them in the future.

References

[1] Alex Carney. Newton's fractal generator method.sagews.

[2] Alex Carney. Newton's fractal list method.sagews.

[3] The Python Community. Generators. `https://wiki.python.org/moin/Generators`.

[4] The Wikipedia Community. Newton's method. `http://en.wikipedia.org/wiki/Newton's_method#Description`.