VISUALIZING JULIA SETS

NATHAN A. SCHINE

ABSTRACT. Julia sets are one of the most widely known families of fractals. First developed by Gaston Julia in the early 20th century, this family of fractals has many different and beautiful exemplars. Beyond their aesthetic appeal, Julia sets stand apart fromother fractals due to the extraordinarily simple method required to produce them. Furthermore, the Julia sets are intimately related to the highly famous Mandelbrot set. This paper demonstrates a method for producing Julia sets using Mathematica and then quickly overviews their relationship with the Mandelbrot set.

CONTENTS

1. TO PRODUCE A JULIA SET

1.1. **Definition.** The Julia set is defined as the set of all complex numbers, z, which are bounded under repeated iteration of the complex quadratic polynomial

$$z_{n+1} = z_n^2 + c \tag{1.1}$$

1.2. **A More Useful Description.** Consider two complex planes: one in which a given point is named c, and the other in which a given point is named z. Pick a value for c. This parameter determines which Julia set will be produced. Now move to the second complex plane. On every z value, apply the complex quadratic polynomial Equation.(1.1) lets say 100 times. For some z, the final value will be enormous; for others it will be quite small. The Julia set is just the set of all z for which these final values are small.

Date: January 26, 2010.

The author would like to thank Professor Frederick Strauch for teaching the coding language and debugging the code. The author would also like to thank Julian Hess for his help reviewing the code and offering suggestions.

1.3. **An Example.**

- Pick $c = 1$
- Pick $z = 1$
- Apply Equation(1.1)
  Output: $1, 1+i, 3i, -9+i, 80-17i, 6111-2719i, 29951360-33231617i, ...$
  since this is not bounded, $1+0i$ is not in the Julia set.

- Pick another z. $z = .5 + .25i$
- Apply Equation(1.1)
  Output:$i, -1+i, -i, -1+i, i, -1+i, -i, -1+i, i, ...$
  since this is bounded, $0+i$ is in the Julia set.

- Repeat for all points in the complex plane

2. IDEAS INTO CODE

2.1. **First Ideas.** The author first decided to use the Mathematica package for the visualization of the Julia sets. The author decided to make the entire program as one or multiple defined functions that then would be called to produce the images. Three primary functions identified as necessary to produce the images.

(1) Produce a matrix containing all complex points to be considered.
(2) Iteratively apply Equation(1.1) to a point.
(3) Apply function 2 to the complex plane produced by function 1 and isolate and plot all points whose values are less than or equal to 2.

These ideas were successfully translated into Mathematica code as three Modules. The program produced images of the points within the Julia set.

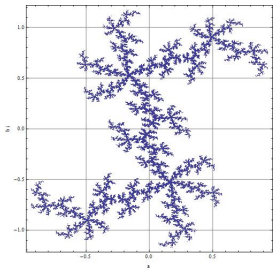
2.2. **First Successful Program.** The first program written that successfully produced an image of a Julia set were direct applications of the above three functions. complexpts[] makes a matrix of the complex plane, juliapoint[] iteratively applies Equation(1.1) to a point, and Julia[] applies juliapoint[] to the matrix produced by complexpts[] and then plots the points appropriate points from the result.

```
complexpts[list1_, list2_] :=
 Module[{L1 = list1, L2 = list2},
  cpts = Table[0, {j, 1, Length[L2]}, {i, 1, Length[L1]}];
  For[j = 1, j <= Length[L2], j++,
   For[i = 1, i <= Length[L1], i++,
    cpts[[j]][[i]] = L1[[i]] + I*L2[[j]]
    ]
   ]
  ]

juliapoint[z_, c_, n_] := Module[{ztemp = z},
  For[k = 0, k < n, k++,
   ztemp = ztemp^2 + c;
   If[ztemp > 10^100,
    Break[]
    ]
   ];
  zfinal = ztemp
  ]
```

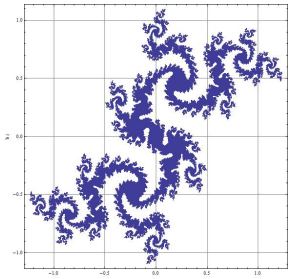
```
Julia[creal_, cimaginary_, realextent_, realstep_, imaginaryextent_,
  imaginarystep_] :=
 Module[{cre = creal, cim = cimaginary,
   reex = realextent, \[CapitalDelta]re = realstep,
   imex = imaginaryextent, \[CapitalDelta]im = imaginarystep},
  c = cre + I*cim;
  reals = Range[-reex, reex, \[CapitalDelta]re];
  imaginaries = Range[-imex, imex, \[CapitalDelta]im];
  jtrue = Table[{0, 0}, {k, 1, Length[reals]*Length[imaginaries]}];
  jvalues =
   Table[{0, 0}, {j, 1, Length[imaginaries]}, {i, 1, Length[reals]}];
  complexpts[reals, imaginaries];
  jvalues = juliapoint[cpts, c, 40];
  count = 1;
  For[j = 1, j <= Length[imaginaries], j++,
   For[i = 1, i <= Length[reals], i++,
    If[Abs[jvalues[[j]][[i]]] <= 2,
     jtrue[[count]] = {Re[cpts[[j]][[i]]], Im[cpts[[j]][[i]]] };
     count += 1
     ]
    ]
   ];
  ListPlot[jtrue, PlotStyle -> PointSize[0.001], Axes -> False,
   Frame -> True, FrameLabel -> {"a", "b \[ImaginaryI]"},
   GridLines -> Automatic, AspectRatio -> 1]
  ]
```

This program produced images like Figures First Attempt 1 and 2 below. While this actually does create the desired image, it requires around at least 250,000 points per square unit to make a reasonably nice image which shows multiple levels of branching with relatively few disconnected points. The program also produces overflow errors; the Break[] command in juliapoint[] does not work correctly. This causes the program also to be relatively inefficient. It doesn't actually need to iteratively evaluate any points once $zfinal$ is greater than 2. A better program could be written.

First Attempt $c = .417 - .355i$



First Attempt $c = -.04 - .684i$



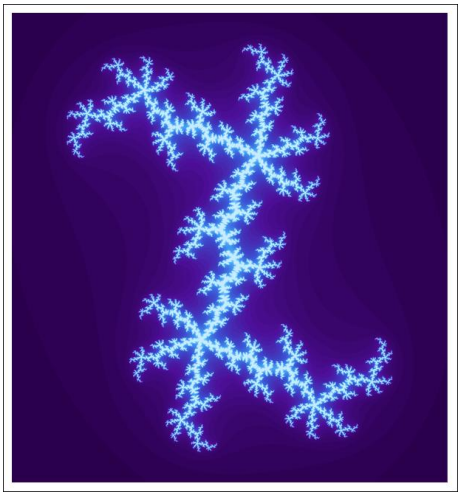
2.3. **A Nice Output.** The author wished to make a nicer and faster output than that which was produced on the first attempt. Rather than just displaying the points which were in the set, the new desire became to color all the points based on how quickly they increased beyond 2. All the points that never did so would be colored similarly and would be in the fractal; those points that did increase past two would produce an interesting and complicated pattern of colors. This new goal, however, required substantial revision of the code.

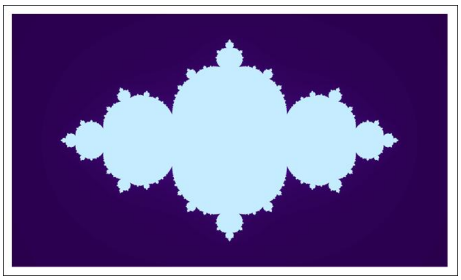
Mathematica allowed the author to evaluate juliapoint[cpts,c,40], where cpts is the matrix containing all the complex points to be evaluated. Mathematica interpreted this as evaluating juliapoint[] for each cell in the matrix and then putting the results into another matrix of the same dimensions. This simplicity is not retained in the new code since the goal is not simply to find the value of zfinal after a given number of iterations and then test if it is less than or equal to 2. Rather, the program must output when zfinal surpasses 2. This leads to the incorporation and modification of juliapoint[] directly into the the code of Julia[]

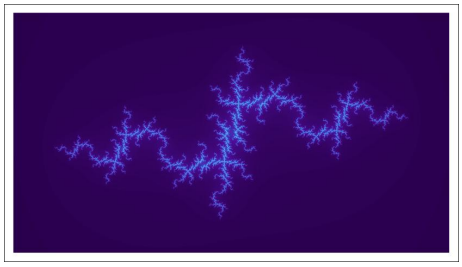
The second program:

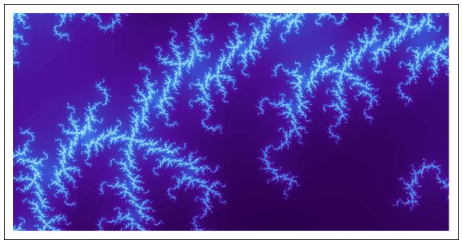
```
complexpts[list1_, list2_] :=
 Module[{L1 = list1, L2 = list2},
  cpts = Table[0, {j, 1, Length[L2]}, {i, 1, Length[L1]}];
  For[j = 1, j <= Length[L2], j++,
   For[i = 1, i <= Length[L1], i++,
    cpts[[j]][[i]] = L1[[i]] + I*L2[[j]]
    ]
   ]
  ]

Julia[creal_, cimaginary_, realextent_, realstep_, imaginaryextent_,
  imaginarystep_, centerpointreal_, centerpointimaginary_] :=
 Module[{cre = creal, cim = cimaginary,
   reex = realextent, \[CapitalDelta]re = realstep,
   imex = imaginaryextent, \[CapitalDelta]im = imaginarystep,
   recpt = centerpointreal, imcpt = centerpointimaginary},
  c = cre + I*cim;
  reals = Range[recpt - reex, recpt + reex, \[CapitalDelta]re];
  imaginaries = Range[imcpt - imex, imcpt + imex, \[CapitalDelta]im];
  jvalues =
   Table[0, {j, 1, Length[imaginaries]}, {i, 1, Length[reals]}];
  complexpts[reals, imaginaries];
  For[j = 1, j <= Length[imaginaries], j++,
   For[i = 1, i <= Length[reals], i++,
    zfinal = cpts[[j]][[i]];
    For[k = 1, k <= 100, k++,
     If[Abs[zfinal] >= 2,
      jvalues[[j]][[i]] = (k - 1);
      Break[]
      ];
     If[k == 40 \[And] Abs[zfinal] < 2,
      jvalues[[j]][[i]] = 100
      ];
     zfinal = zfinal^2 + c
     ]
    ]
   ];
  ArrayPlot[jvalues, ColorFunction -> ColorData["DeepSeaColors"]]
  ]
```

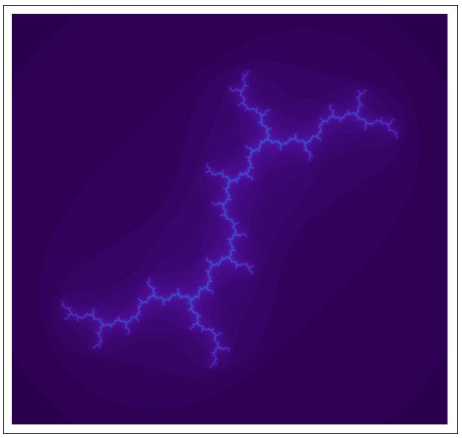
This program successfully creates a much nicer output without any error messages. The color scheme is arbitrary; the author picked "Deep Sea Colors" because it looked neat. Some of the produced images are included below.

Nice Output $c = .417 - .355i$ 

Nice Output $c = -.7$ 

Nice Output $c = -1 + .3i$ 

Blow-up of $c = -1 + .3i$ centered at $(.9, 0)$ 

Nice Output $c = i$ 

3. Julia sets and Mandelbrot

Julia sets and the Mandelbrot set are intimately related. They are defined in analogous ways and have many shared properties. Specific regions of Julia sets even look almost identical to the Mandelbrot Set. The Mandelbrot set can defined as the set of all c such that the Julia set for that parameter c is completely continuous. This means that for any c outside the Mandelbrot set, the Julia set is scattered, sometimes with well defined but separated lobes, more often just with disconnected points. These sets are sometimes refered to as Julia dust. The most common definition of the Mandelbrot set is just an inversion of the definition of the Julia set:

Take the complex quadratic polynomial Equation(1.1) and let $z_0 = 0$. Then, a point c is in the Mandelbrot set if Equation.(1.1) yields a bounded result. Basically, the roles of z and c are reversed, with z always equal to 0.

Only a very slight manipulation of the previous code is needed to produce the Mandelbrot set:

```
complexpts[list1_, list2_] :=
 Module[{L1 = list1, L2 = list2},
  cpts = Table[0, {j, 1, Length[L2]}, {i, 1, Length[L1]}];
  For[j = 1, j <= Length[L2], j++,
   For[i = 1, i <= Length[L1], i++,
    cpts[[j]][[i]] = L1[[i]] + I*L2[[j]]
    ]
   ]
  ]

mandelbrot[realextent_, realstep_, imaginaryextent_, imaginarystep_,
  centerpointreal_, centerpointimaginary_] :=
 Module[{reex = realextent, \[CapitalDelta]re = realstep,
   imex = imaginaryextent, \[CapitalDelta]im = imaginarystep,
   recpt = centerpointreal, imcpt = centerpointimaginary},
  reals = Range[recpt - reex, recpt + reex, \[CapitalDelta]re];
  imaginaries = Range[imcpt - imex, imcpt + imex, \[CapitalDelta]im];
  jvalues =
   Table[0, {j, 1, Length[imaginaries]}, {i, 1, Length[reals]}];
  complexpts[reals, imaginaries];
  For[j = 1, j <= Length[imaginaries], j++,
   For[i = 1, i <= Length[reals], i++,
    c = cpts[[j]][[i]];
```

```
  zfinal = 0;
  For[k = 1, k <= 100, k++,
   If[Abs[zfinal] >= 2,
    jvalues[[j]][[i]] = (k - 1);
    Break[]
    ];
   If[k == 40 \[And] Abs[zfinal] < 2,
    jvalues[[j]][[i]] = 100
    ];
   zfinal = zfinal^2 + c
   ]
  ]
 ];
ArrayPlot[jvalues, ColorFunction -> ColorData["DeepSeaColors"]]
]
```

The familiar output of this code is included below:

Mandelbrot Set