

Boundary scanning and complex dynamics

A preprint version of a “Mathematical graphics” column from

Mathematica in Education and Research.

Mark McClure

mcmclur@unca.edu

Department of Mathematics
University of North Carolina at Asheville
Asheville, NC 28804

Abstract

Colorful images of the Mandelbrot set and Julia sets are typically drawn using an escape time algorithm. However, it is the boundary of such a set which is truly fractal. We can use a standard technique from image processing to highlight the boundary.

■ Mathematica Initializations

1. Introduction

The Mandelbrot set and its associated Julia sets are among the most well known mathematical images. There are many beautiful color images of these sets in [1]. Such images are generally produced using an escape time algorithm. This algorithm may be easily implemented with *Mathematica*, as we will show again here. In order to highlight the boundary of the set, we can use an additional image processing step as described in [2]. This boundary scanning technique, it turns out, benefits greatly from a very high level of resolution. As a result, we will use a Java implementation to speed up the escape time computations.

2. Complex dynamics

In complex dynamics, we study the iteration of a function $f : \mathbb{C} \rightarrow \mathbb{C}$. That is, given f and an initial complex valued input z_0 , we generate a sequence $\{z_0, z_1, z_2, \dots\}$, where $z_n = f(z_{n-1})$. Given z_0 , this sequence is called the *orbit* of z_0 under iteration of f . For example, here are the first few iterates of the point $z_0 = 1/2$ under the action of $f(z) = z^2$.

```
In[1]:= f[z_] := z^2;
        NestList[f, 1/2., 4]

Out[2]= {0.5, 0.25, 0.0625, 0.00390625, 0.0000152588}
```

Note that the orbit tends to 0. For this function, it is not difficult to see that the orbit of z_0 will tend to 0 if $|z_0| < 1$, while the orbit of z_0 will diverge to ∞ if $|z_0| > 1$.

■ Julia sets

In order to understand the global behavior of the dynamics of a function, we divide the complex plane into two regions. The *Fatou set*, F , is the set where the dynamics are stable in the sense that points close to one another have similar long term behavior. The *Julia set*, J , is defined to be the complement of the Fatou set and the dynamics of f are quite chaotic on J .

For example, our observations above suggest that the Fatou set for $f(z) = z^2$ should consist of two disjoint parts, the interior and the exterior of the unit circle. The dynamics right on the unit circle are much more complicated, however. If $|z_0| = 1$, then we may find points as close as we like to z_0 which tend to zero under iteration of f , and we may also find points as close as we like to z_0 that tend to ∞ under iteration of f . Thus the unit circle is the Julia set for this function. Note that the Julia set forms the boundary of the two components of the Fatou set.

More generally, we are interested in studying the iteration of functions of the form $f_c(z) = z^2 + c$. Although these ideas are more broadly applicable, it is this family of functions which lead to the Mandelbrot set. Furthermore, the necessary Java code is simplified by sticking to a single simple family. The escape time algorithm for f_c is based on the following lemma: Suppose we iterate any function of the form f_c starting from an initial value z_0 . Then, there are only two possibilities: either the orbit diverges to ∞ , or the orbit remains bounded by 2. Thus, the following code is a reasonable test to check if the complex number z_0 lies in the Julia set of the function f_c .

```
In[3]:= julia = Compile[{
    {c, _Complex}, {z0, _Complex}, {bail, _Integer}},
    Length[FixedPointList[#^2 + c &, z0, bail,
        SameTest -> (Abs[#] > 2 &)]]];
```

The code simply iterates f_c from the starting value z_0 at most *bail* times until the absolute value of the result exceeds 2. The result is the length of the resulting partial orbit. Note that the partial orbit will include the starting point, thus the return value may be as large as *bail* + 1. For example, the following computation suggests that 1.618 is in the Julia set of f_{-1} .

```
In[4]:= julia[-1, 1.618, 100]
```

```
Out[4]= 101
```

The following computation, however, shows that 1.6181 is not in the Julia set of f_{-1} .

```
In[5]:= julia[-1, 1.6181, 100]
```

```
Out[5]= 10
```

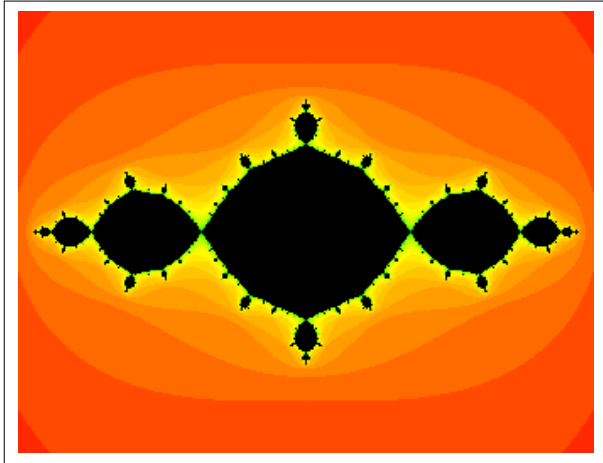
In fact, the golden ratio $\varphi \approx 1.61804$ is right on the boundary of the Julia set. This follows from the fact that it is a repulsive fixed point of the function.

We may generate an image of the Julia set of f_{-1} by generating a table of such values and passing the result to **ArrayPlot**.

```

data = Reverse[Table[julia[-1, a + b i, 100],
  {b, -1.3, 1.3, 0.01}, {a, -1.7, 1.7, 0.01}]];
ArrayPlot[data,
  ColorFunctionScaling → True,
  ColorFunction → (If[# == 1, RGBColor[0, 0, 0], Hue[#3/4]] &)]

```



Note that it is tempting to use the **NestWhileList** command in place of **FixedPointList** in the definition of **Julia**. This will execute significantly more slowly, however, since **NestWhileList** does not compile.

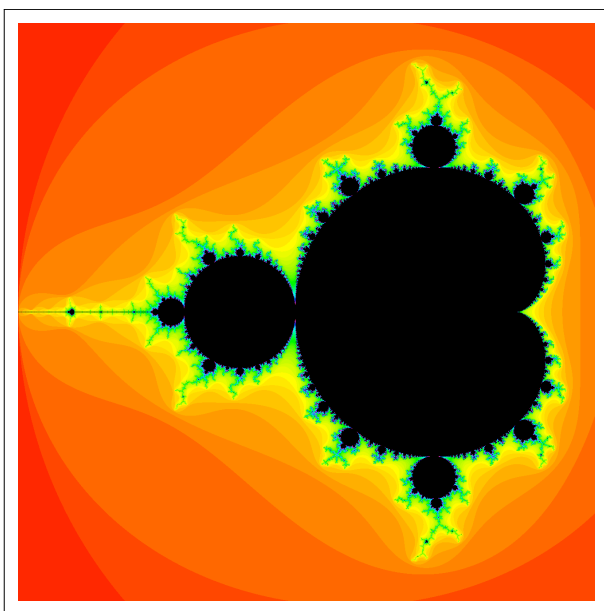
■ The Mandelbrot set

The Mandelbrot set may be thought of as an index of the Julia sets. Its definition is based on the following theorem: Suppose we iterate a function f_c from the starting value 0. Then there are two possibilities. Either, the orbit remains bounded by 2, in which case the Julia set of f_c is connected, or the orbit diverges to ∞ , in which case the Julia set is totally disconnected. We may use this to generate pictures of the Mandelbrot set in much the same way we did for Julia sets. However, we fix 0 as the single starting point and we investigate the behavior for a large grid of possible values of the parameter c .

```

mandel = Compile[{
  {c, _Complex}, {bail, _Integer}},
  Length[FixedPointList[#^2 + c &, 0, bail,
    SameTest -> (Abs[#] > 2 &)]]];
mandelData0 = Reverse[Table[mandel[a + b i, 100],
  {b, -1.3, 1.3, 0.002}, {a, -2, 0.6, 0.002}]];
ArrayPlot[mandelData0,
  ColorFunctionScaling -> True,
  ColorFunction -> (If[# == 1, RGBColor[0, 0, 0], Hue[#^(3/4)] &)]

```



Note that we used a very small step size of 0.002 in this example to generate a high resolution image. This will be particularly helpful when we pass the data to a boundary scanner.

3. Using Java

We will discuss the boundary scanning technique in the next section. As mentioned, this technique requires a high level of resolution to produce satisfactory results. The time complexity of the process is clearly quadratic with respect to the resolution, thus speed is of the essence. We can drastically increase the speed by doing the computations in Java and processing the results in *Mathematica*.

The essential Java methods are defined in the *quadraticIterator* class contained in the *SupplementaryFiles* folder distributed with this notebook. We can access those methods by first instantiating an object of the *quadraticIterator* class via the **JLink** package as follows.

```

In[11]:= Needs["JLink`"]
InstallJava[];
AddToClassPath[DirectoryName[ToFileName["FileName" /.
  NotebookInformation[EvaluationNotebook[]]]] <> "SupplementaryFiles"];
SetComplexClass["Complex"];
quadraticIterator = JavaNew["quadraticIterator"];

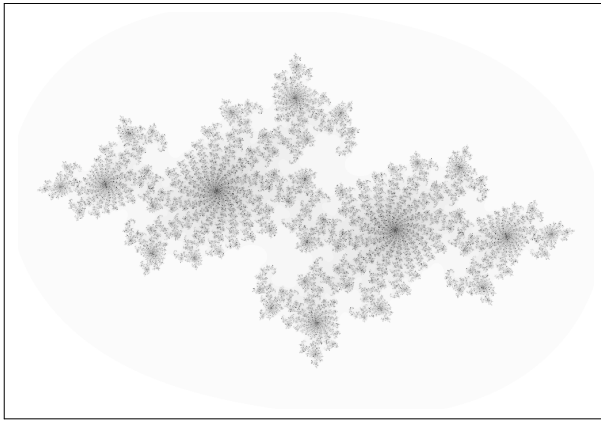
```

We can now access the two main methods *escapeTimes* and *criticalEscapeTimes* defined in this class. The general form of the *escapeTimes* method is as follows.

```
quadraticIterator@escapeTimes[c, z0Min, z0Max, bail, resRe, resIm];
```

This will return a **resRe**×**resIm** array of escape time values for the Julia set of f_c . The lower left hand corner of the corresponding rectangle in the complex plane is **z0Min** and the upper right hand corner is **z0Max**. Thus we may generate the image of an interesting Julia set as follows.

```
data = quadraticIterator@escapeTimes[  
  -0.73 + 0.216 i, -1.6 - 1.1 i, 1.6 + 1.1 i,  
  500, 1000, 688];  
ArrayPlot[Reverse[data]]
```



This is a nice example of the type of image we are interested in. However, no boundary scanning was actually necessary since the Julia set essentially is its own boundary.

There is a similar Java method called *criticalEscapeTimes* to generate images of the Mandelbrot set. The syntax is as follows.

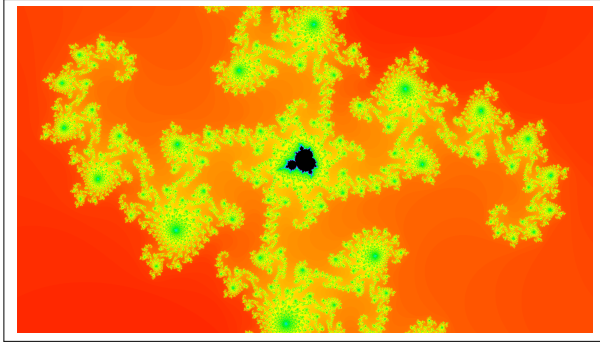
```
quadraticIterator@criticalEscapeTimes[cMin, cMax, bail, resRe, resIm];
```

The parameters are very similar to the *escapeTimes* method, but **cMin** and **cMax** define the corners of the image in the parameter plane and all iterations start at 0. Note that 0 is the critical point of f_c , hence the name *criticalEscapeTimes*. We can use this to create a nice zoom into the Mandelbrot set. Note that the bail out parameter is set to 1000, since we are zooming in fairly far. The resolution is again set high, since we will pass this data to a boundary scanner in the next section.

```

mandelData1 = quadraticIterator@criticalEscapeTimes[
  -0.7351 + 0.19696 i,
  -0.7348 + 0.19713 i,
  1000, 2000, 1132];
ArrayPlot[Reverse[mandelData1],
  ColorFunctionScaling -> True,
  ColorFunction -> (If[# == 1, RGBColor[0, 0, 0], Hue[#3/4]] &)]

```



4. Boundary scanning

As described in chapter 45 of [2], many interesting image processing operations can be achieved via convolution with a kernel. Suppose we have a large two dimensional matrix, which might represent color values for an image. A kernel is a, typically much smaller, two dimensional matrix which is used to process the large matrix via convolution. The easiest way to describe convolution is to investigate the formula in a small, but arbitrary case.

```

In[20]:= ListConvolve[ $\begin{pmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{pmatrix}$ ,  $\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$ ] // MatrixForm

```

```

Out[20]//MatrixForm=

```

$$\begin{pmatrix} a_{22} k_{11} + a_{21} k_{12} + a_{12} k_{21} + a_{11} k_{22} & a_{23} k_{11} + a_{22} k_{12} + a_{13} k_{21} + a_{12} k_{22} \\ a_{32} k_{11} + a_{31} k_{12} + a_{22} k_{21} + a_{21} k_{22} & a_{33} k_{11} + a_{32} k_{12} + a_{23} k_{21} + a_{22} k_{22} \end{pmatrix}$$

If, for example, each $k_{ij} = 1/4$, then each possible 2 by 2 block in the larger matrix will be replaced by the average of the values in the block. This can be used to create a blur effect in an image.

Now suppose we have a very large matrix representing an array of gray levels in an image. We would like a kernel which detects boundaries in that image. Such a kernel, suggested in [2], is

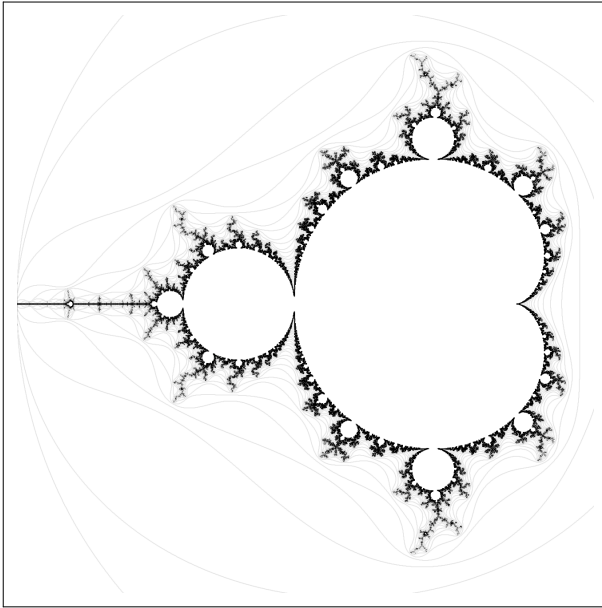
$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$

Note that in an essentially monochromatic region (i.e. the values are very close to one another), convolution with the kernel will return values close to zero. Values far from zero only arise near the boundaries. Let's apply this kernel to **mandelData0** which we used to generate our first image of the Mandelbrot set.

```

kernel = {
  {1, 1, 1},
  {1, -8, 1},
  {1, 1, 1}
};
convolvedData = ListConvolve[kernel, mandelData0];
ArrayPlot[Abs[convolvedData],
  ColorFunctionScaling -> True,
  ColorFunction -> (GrayLevel[(1 - #)30] &)]

```

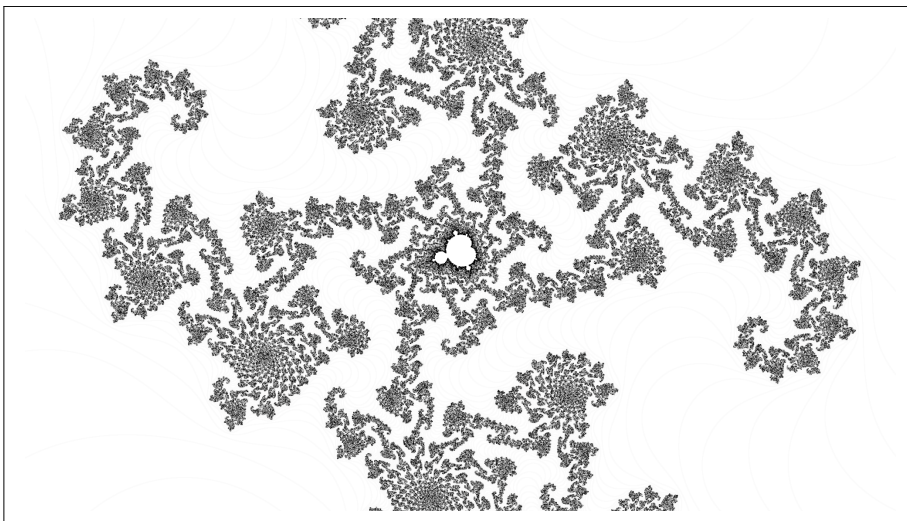


Next, we apply the technique to the zoom of the Mandelbrot set we generated using the Java code.

```

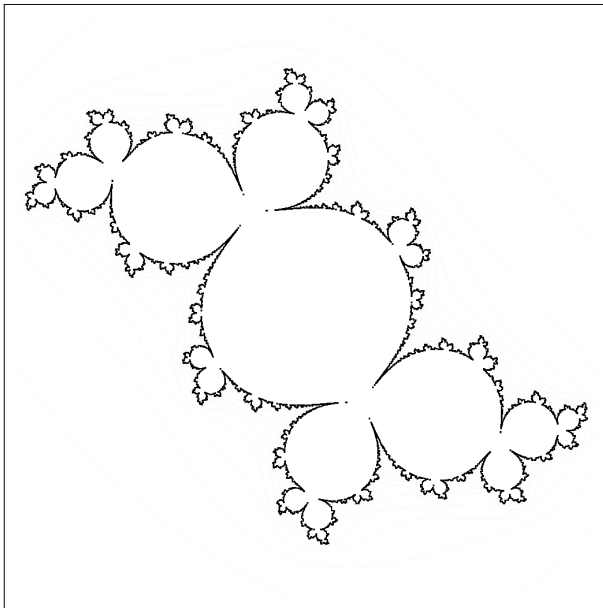
convolvedData = ListConvolve[kernel, Reverse[mandelData1]];
ArrayPlot[Abs[convolvedData],
  ColorFunctionScaling -> True,
  ColorFunction -> (GrayLevel[(1 - #)30] &)]

```

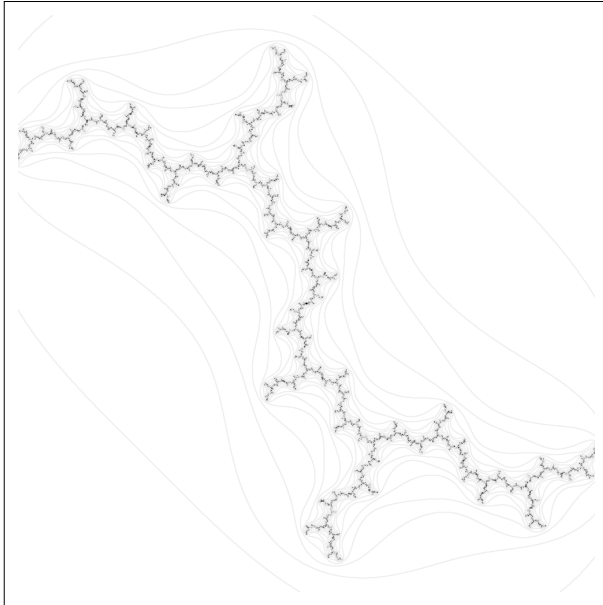


Finally, we look a couple more interesting Julia sets.

```
data = Reverse[quadraticIterator@escapeTimes[  
  -0.125 + 0.64952 i, -1.3 - 1.3 i, 1.3 + 1.3 i,  
  1000, 1000, 1000]];  
convolvedData = ListConvolve[kernel, data];  
ArrayPlot[Abs[convolvedData],  
  ColorFunctionScaling -> True,  
  ColorFunction -> (GrayLevel[(1 - #)30] &)]
```




```
data = Reverse[quadraticIterator@escapeTimes[  
  i, -1.3 - 1.3 i, 1.3 + 1.3 i,  
  1000, 1000, 1000]];  
convolvedData = ListConvolve[kernel, data];  
ArrayPlot[Abs[convolvedData],  
  ColorFunctionScaling -> True,  
  ColorFunction -> (GrayLevel[(1 - #)200] &)]
```



References

- [1] H. Peitgen and P. Richter. *The Beauty of Fractals: images of complex dynamical systems*. Springer, NY, 1986.
- [2] J. Glynn and T. Gray, The Beginner's Guide to *Mathematica* Version 4. Cambridge University Press, NY, 2000.