

Julia Set, Parallelization, & Compilation

Jesse Bettencourt

Quick Intro

After the class on the Mandelbrot set visualization I attempted to implement some Julia set and other iterated fractal set visualizations. *Mathematica* has a built in function `JuliaSetPlot` which is already extremely efficient at visualizing these iterated fractal sets because it implements the more efficient method of orbit detection, instead of escape time algorithms. However, it's limited in that it only accepts rational functions. However pretty and efficient the built in *Mathematica* function is, as a learning exercise I felt it was important to continue to try to implement my own, and experiment with methods of improving efficiency.

Mathematica's Built In Function

```
? JuliaSetPlot
```

```
Manipulate[
```

```
  JuliaSetPlot[z^2 + realc + complexc * I, z],  
  {{realc, 0.3}, 0, 1}, {{complexc, 0.53}, 0, 1}]
```

`JuliaSetPlot[f, z]` plots the Julia set of the rational function f of the variable z .

`JuliaSetPlot[c]` plots the Julia set of the function $f(z) = z^2 + c$. >>

Some Different Attempts By Me

In order to be competitive with the built in function, which will outperform my best attempt on most function, I learned how to compile the *Mathematica* code to C with the `Compile` function. Here are some different attempts for a computing the Julia set escape times at a complex number:

```

(*Attempt 1*)
(*This is very similar to our in-class Mandelbrot implementation. *)
Clear[JuliaC]
JuliaC = Compile[{{x, _Real}, {y, _Real}, limit, {cx, _Real}, {cy, _Real}},
  Module[{z, count = 0},
    z = x + I y;
    While[Abs[z] < 2.0 && count ≤ limit,

      (*Iterated Function*)
      z = z^2 + (cx + I * cy);

      ++count]; count],
  CompilationTarget → "C",
  RuntimeOptions → "Speed"
];

(*Attempt 2*)
(*This implementation is very similar to the first attempt,
except I wanted to see if Compile made better use of directly input
Complex numbers z and c instead of attempt 1 which input the components
of both. It seemed that this actually made performance slightly worse,
though I still prefer it for clarity.*)
(*The other thing to note is that here I've changed the function we are
iterating. Particularly interesting is we can now iterate over non-
rational functions. Because of Mathematica's built in implementation methods,
only rational functions were permitted, so this is very exciting.*)
JuliaComplex = Compile[{{Z, _Complex}, {c, _Complex}},
  Module[{count = 0, z = Z, limit = 250},
    While[
      Abs[z] < 2.0 && count ≤ limit,

      (*Iterated Function*)
      z = Tan[z^3 + c];

      ++count]; count],
  CompilationTarget → "C",
  RuntimeOptions → "Speed",
  RuntimeAttributes → {Listable},
  Parallelization → True
];

```


Here's my best attempt

I like this attempt because it is a lot more *Mathematica* in flavour. Namely, I left the procedural approach of the While loop and instead used the function FixedPointList which generates a list of the outputs of a function iteratedly reapplied until it either reaches a fixed point defined by the criterion SameTest or if it reaches the maximum number of iterations 250.

Try playing with the function which is being iterated!

```

Julia = Compile[{{z, _Complex}, {c, _Complex}},
  Length[FixedPointList[
    (*****ITERATED FRACTAL FUNCTION*****)
    Tan[#^3 + c] &,
    (*****)
    z, 250,
    SameTest -> (Abs[#] > 2 &)]],
  RuntimeAttributes -> {Listable},
  Parallelization -> True,
  CompilationTarget -> "C"]
    
```

CompiledFunction [ Argument count: 2
Argument types: {_Complex, _Complex}]

Parallelization

Initially I was using my newly compiled functions with the DensityPlot function, which takes my function and applies it to on a grid over my specified plot range. However, I realized that this was not optimal because I could be computing the Julia set for multiple points at once if I made use of multiple computing kernels. *Mathematica* makes it extremely easy to parallelize these sorts of tasks, so I learned how to make use of those features here.

First things first is to initialize some Kernels with the availability of CPU cores. Depending on the power of your CPU and the number of cores you have, your benefits for parallelization may vary.

```

LaunchKernels [];
Kernels []
    
```

Now, instead of generating the Julia set inside the DensityPlot function, we will first create a table with all of our values and then plot it after with ListDensityPlot. When we generate our table of values, we evaluate at complex coordinates (x,y) and Increment along the real and complex axis by the value of the variable **resolution**, as this parameter corresponds with the quality of our plot.

Further, to make use of parallelization and the Kernels we've initialized, instead of the vanilla Table

function, we use `ParallelTable`, which automatically spreads the computations over our cores for us.

```
resolution = 0.005;
julialist = ParallelTable[-Julia[x + I y, 2.0625 + 0.15 I ],
  {y, -2, 2, resolution}, {x, -2, 2, resolution}];
```

```
ListDensityPlot[julialist, Frame → False, PlotRangePadding → None]
```

Wow, that was pretty slow though.... Which is a shame. We've optimized the actual computation of the set, only to have to wait for the density list to plot. Well, consider that we just want to visualize the set, and don't need all of the bells and whistles of having a density plot graphics object, which is expensive but feature-rich. Instead, we can use `Image`!

(Note that I've tried many methods here, including `ArrayPlot`, `Graphics@Raster`, and a couple others. `Image` is by far the fastest I've come across)

```
Colorize[Image[Rescale[-julialist]], ColorFunction → "StarryNightColors"]
```

```
julialist2 =
  ParallelTable[-Julia[x + I y, 0.5 + 0.05 I ], {y, -2, 2, 0.005}, {x, -2, 2, 0.005}];
```

```
Colorize[Image[Rescale[julialist2]], ColorFunction → "SunsetColors"]
```

```
julialist3 =
  ParallelTable[-Julia[x + I y, 0.1 + 0.5 I ], {y, 0.5, 2, 0.001}, {x, 0, 1.5, 0.001}];
```

```
Colorize[Image[Rescale[julialist3]], ColorFunction → "StarryNightColors"]
```

Bonus: Playing with Time (uses *Mathematica's* `JuliaSetPlot`)

```
animatelist = ParallelTable[
  JuliaSetPlot[z^2 + (0.01 + 0.1 Sin[θ] I) / z^3, z,

  Frame → False,
  PlotRange → {{-1.2, 1.2}, {-1.2, 1.2}},
  PlotRangePadding → None],
```

```
{θ, 0, 2 Pi, 0.1}];
```

```
ListAnimate[Image /@ animatelist]
```