Presentation Notebook

May 11, 2016

In [1]:
```
%matplotlib notebook
%run escapeImages.py

from notebook.services.config import ConfigManager
cm = ConfigManager()

cm.update('livereveal', {
              'width': 1600,
              'height': 1080,
              'theme': 'serif',
              'transition': 'zoom',
              'start_slideshow_at': 'selected',
})

# Before presentation: run all image cells, everything else should be clea

"""
// Change top of notebook
$("body.notebook_app").css("top", "-100px")

// Simulate a click on the "Start Slideshow button
$("#start_livereveal").click()
"""
```

Out[1]:
```
'\n// Change top of notebook\n$("body.notebook_app").css("top", "-100px")\n
```

Pretty Pictures with Numpy
Making Fractals for Dynamical Systems Research using Numpy and Matplotlib
Evan Oman
www.evanoman.com
About Me

- Masters Degree in "Applied and Computational Mathematics" from UMD
- Currently a Systems/Algorithm Engineer at Black River Systems
  - Work on small research grants for Intelligence, Surveillance + Reconnaisance defense contracts
  - Mix of reading CS/Math papers and implementing prototypes

- Also part time Data Science consultant for financial data consulting company
  - Work with Apache Spark (in Scala), also making a Slackbot

Dynamical Systems

- Dynamical Systems is the study of the behavior of repeated maps of the form
  $z_n \mapsto z_{n+1} = f_{\alpha_1,\ldots,\alpha_n}(z_n)$
  where $f_{\alpha_1,\ldots,\alpha_n} : S \to S$
- The ultimate goal would be a characterization of the "long term" behavior of every point $x \in S$ as the parameters α_i vary
- My graduate research was focused on the family
  $z_n \mapsto f_{c,\beta}(z_n) = z_n^2 + c + \frac{\beta}{\bar{z_n}^2} = z_{n+1}$
  where $f_{c,\beta} : \mathbb{C} \to \mathbb{C}$
- This family is interesting because it is non-holomorphic blah, blah, blah…

Types of Orbits

- The orbit of a some point x under some map f is the set of points
  $\mathrm{Orb}(x, f) = \{x, f(x), f(f(x)) = f^2(x), \ldots\}$
- Orbits can be unbounded: $\forall K \in \mathbb{R}, \exists n \in \mathbb{N}$ such that $|f^n(x)| > K$
  - e.g. 2 under $f(x) = x^2$
- Orbits can be bounded ($\exists B$ such that $|f^n(x)| < B$, $\forall n \in \mathbb{N}$)
  - Fixed: 0 under $f(x) = x^2 : \{0, 0, \ldots\}$
  - Periodic: -1 under $f(x) = x^2 - 1 : \{-1, 0, -1, 0, \ldots\}$
  - Attracting: 1 under $f(x) = \frac{x}{2} : \{1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots\}$
  - Other: pre-periodic, pre-fixed, CHAOTIC

Escape Images

- So given some map, we want to be able to get an idea of what the long term behavior is for each of its points as parameters change
- One way to do this is to create an "escape image" which takes each point on the 2D plane as an initial condition (represented by a pixel) and records how long it took for the map, under that initial condition, to escape
- Here is an escape image you might have seen before:

```
In [2]: escImgParam(fn=lambda x,c: x*x + c, xmin=-2, xmax=.75, ymin=-1.25,
                    ymax=1.25)
```

```
<IPython.core.display.Javascript object>
```

```
&lt;IPython.core.display.HTML object&gt;
```

How to Make the Mandelbrot Set

- The Mandelbrot Set is a Parameter Space Escape Image for the map
  $f_c(z) = z^2 + c$
  such that $f : \mathbb{C} \to \mathbb{C}$
- "Parameter Space" means that each pixel is representing a value for the parameter $c \in \mathbb{C}$
- So what we need to do is to iterate the map $f_c(0)$ some number of times for each pixel (value for c) in our plane and determine how "fast" that point escapes
- We then color that pixel with the relative amount of time it took for that point to escape
  - We iterate on the value 0 for technical reasons (it's special)
  - In this case, "escape" means $|f_c^n(0)| \geq 2$.
  - Thus we are going to color each pixel for the first n such that $|f_c^n(0)| > 2$
- For this talk I am just using a spectral colormap

How to Make the Mandelbrot Set

- Suppose we wanted to make a 3×3 Mandelbrot Set on the region $[-2, 2] \times [-2, 2]$:

How to Make the Mandelbrot Set

- Now we perform 100 iterations and color each "pixel" with how many iterations it took to get beyond a radius of 2

$$\left|f_{c=(-2,2i)}(0)\right| = |0 * 0 + (-2, 2 * i)| = \sqrt{(-2)^2 + (2)^2}$$
$$= \sqrt{8} = 2\sqrt{2} \geq 2 \Rightarrow 1 \text{ iteration}$$
$$\left|f_{c=(0,2i)}(0)\right| = |0 * 0 + (0, 2i)| = \sqrt{(0)^2 + 2^2}$$
$$= \sqrt{4} = 2 \geq 2 \Rightarrow 1 \text{ iteration}$$
$$\left|f_{c=(2,2i)}(0)\right| = |0 * 0 + (2, 2i)| = \sqrt{2^2 + (2)^2}$$
$$= \sqrt{8} = 2\sqrt{2} \geq 2 \Rightarrow 1 \text{ iteration}$$
$$\left|f_{c=(-2,0i)}(0)\right| = |0 * 0 + (-2, 0i)| = \sqrt{(-2)^2 + (0)^2}$$
$$= \sqrt{4} = 2 \geq 2 \Rightarrow 1 \text{ iteration}$$
$$\left|f_{c=(0,0i)}(0)\right| = |0 * 0 + (0, 0)| = \sqrt{(0)^2 + (0)^2}$$
$$= 0 = \left|f^{100}_{c=(0,0i)}(0)\right| < 2 \Rightarrow 100 \text{ iterations} \ldots \text{ et cetera}$$

Quickest NumPy Intro

- NumPy is a versatile vector library with lots of cool features:

```
In [14]: ex = np.array([1,2,3,4])

         # element-wise adition
         print(ex + 2)
         # vector ops
         print(ex * ex)
```

```
[3 4 5 6]
[ 1  4  9 16]
```

```
In [16]: # logical indexing
         print(ex > 2)
         print(ex)
         print(ex[ex > 2])
```

```
[False False  True  True]
[1 2 3 4]
[3 4]
```

```
In [5]: # Same as the following list comprehension:
        def myLog(x, mask):
            return [x_i[0] for x_i in zip(x, mask) if x_i[1]]

        print(myLog(ex, ex > 2))
```

```
[3, 4]
```

How to Make the Mandelbrot Set

- So lets walk though the computation of a 5×5 image
- Credit for Numpy approach: Dan Goodman

```
In [17]: # Set all of the initial variables
         fn = lambda x,c: x*x + c
         critPoint = 0
         escapeRad = 2.0
         n = 5
         m = 5
         itermax = 100
         xmin = -2
         xmax = 2
         ymin = -2
         ymax = 2
```

How to Make the Mandelbrot Set

```
In [18]: # makes an n*m grid of integers (for indexing)
         ix, iy = np.mgrid[0:n, 0:m]

         # n evenly spaced points between xmin and xmax
         x = np.linspace(xmin, xmax, n)[ix]
         y = np.linspace(ymin, ymax, m)[iy]
```

```
        c = x + complex(0, 1) * y
        del x, y  # save a bit of memory, we only need z

        # Now lets look inside c:
        print(c[0,0])
        print(c[1,1])
        print(c[2,2])
        print(c[3,3])
        print(c[4,4])
```

```
(-2-2j)
(-1-1j)
0j
(1+1j)
(2+2j)
```

```
In [19]: # Create array of critical points (in this case 0)
        img = np.zeros(c.shape, dtype=int)

        # Now we flatten these maps
        ix.shape = n * m
        iy.shape = n * m
        c.shape = n * m

        # Check contents of c
        print(c)
```

```
[-2.-2.j -2.-1.j -2.+0.j -2.+1.j -2.+2.j -1.-2.j -1.-1.j -1.+0.j -1.+1.j
 -1.+2.j  0.-2.j  0.-1.j  0.+0.j  0.+1.j  0.+2.j  1.-2.j  1.-1.j  1.+0.j
  1.+1.j  1.+2.j  2.-2.j  2.-1.j  2.+0.j  2.+1.j  2.+2.j]
```

```
In [11]: # Now apply our function (f_c) to all of the points in c
        z = fn(critPoint, np.copy(c))

        for i in range(itermax):

            # If there aren't any values left, stop iterating
            if not len(z):
                break  # all points have escaped

            # Perform an iteration of our map
            z = fn(z, c)

            # A logical vector specifying the points which have escaped
            rem = abs(z) > escapeRad

            # Store the iteration, i, at which the rem points escaped
```

```
            img[ix[rem], iy[rem]] = i + 1

            # Now rem represents the points which have remained bounded
            rem = ~rem

            # Filter out all escaped values
            z, ix, iy, c = z[rem], ix[rem], iy[rem], c[rem]

        # sets those points which have not yet escaped to itermax + 1 (which is to
        #  to escape, or didn't)
        img[img == 0] = itermax + 1
```

```
In [12]: print(img.T)
```

```
[[  1   1   1   1   1]
 [  1   2 101   1   1]
 [101 101 101   2   1]
 [  1   2 101   1   1]
 [  1   1   1   1   1]]
```

```
In [ ]: # Reverses the colormap for aesthetic reasons
        img = abs(itermax - img)

        # create a new figure
        fig = plt.figure()

        # Does the coloring according to the colormap
        image = plt.imshow(img.T, origin='lower left', interpolation="none")
        image.set_cmap("nipy_spectral")
```

```
In [13]: # Now we can make the image higher resolution and we get the image we expe
         escImgParam(fn = lambda x,c: x*x + c)
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

Other Maps

- The way I have written this this, you can give `escImgParam` any lambda
- Burning Ship: $z \mapsto (|\operatorname{Re}(z)| + i\,|\operatorname{Im}(z)|)^2 + c$

```
In [20]: def bShip(z,c): return (abs(np.real(z)) + abs(np.imag(z))*complex(0,1))**2

         escImgParam(fn=bShip)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
In [21]: escImgParam(fn=bShip, xmin=-1.8, xmax=-1.5, ymin=-.15, ymax=.15, interp="b
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Conclusion

- The NumPy logical indexing allows Python to perform these operations at almost `C/C++` speeds without the need for parallelization
- Lambdas make the Python code extremely flexible for just about any Dynamical System
- Tools used:
  - RISE: https://github.com/damianavila/RISE
  - NumPy: http://www.numpy.org/
  - Matplotlib: http://matplotlib.org

Questions?
Escape Image for my Research
My research function: $f_{c,\beta}(z) = z^2 + c + \frac{\beta}{\bar{z}^2}$

```
In [29]: r = pow((2/2)*abs(.001), 1.0/(2+2))

         #The set of critical points is a circle with the radius given by above, fo
         critPoint = r*complex(math.cos(0), math.sin(0))

         escImgParam(fn=lambda x,c: x*x + c + .001/(np.conj(x)*np.conj(x)), critPoi
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```