1 Fractals and Blue Screens

1.1 A Very Brief Introduction to Chaos

In a real physical system, equal starting conditions means equal ending conditions. Of course, equal starting conditions are impossible (2nd law of Thermodynamics), so it's a theoretical concept.

Many real physical systems, however, obey the strong causality theory: similar starting conditions give similar ending conditions. If I drop a ball from about the same place, it hits the ground in about the same place every time.

Chaos is when the strong causality theory fails. Consider the dropping ball example. If I drop it onto a smooth, flat surface then it bounces straight back up every time. What happens if I drop it onto a traffic cone? Small changes in the starting position of the ball now have a big influence on where it goes.

What kinds of systems are chaotic? Pretty much any system with some kind of nonlinear feedback loop can be chaotic. Linear systems are much better behaved, in general.

- A dropping faucet is chaotic: how much fell in the last drop affects the next drop.
- Weather is chaotic.
- Traffic is chaotic.

Attractors are dynamic systems that tend towards certain stable states (like dry weather/rainy weather), but may switch between them unpredictably.

Lorentz Attractor

$$\begin{aligned} x' &= a(y - x) \\ y' &= bx - y - xz \\ z' &= xy - cz \end{aligned} \tag{1}$$

In the above differential equations, a, b, c are constants, and x, y, z represent parameters of the system, which was used by Lorentz to model weather. Each time step, the equations provide new values for the change in the parameters x', y', z', which then update the parameter values. Interesting values for a, b, c are 10, 28, and $8/3$, respectively. Plotting the x and y parameters over time on a screen from $(-30, -30)$ to $(30, 80)$ produces an interesting picture.

Other interesting attractors include the Verhults attractor and the Henon attractor.

Verhulst Attractor ($k = 1.6$)

$$p_{n+1} = p_n + 0.5k(3p_n(1 - p_n) - p_{n-1}(1 - p_{n-1}) \tag{2}$$

Henon Attractor ($a = 7/5, b = 3/10, (x_0, y_0) = (0, 0)$)

$$\begin{aligned} x_{n+1} &= y_n - ax_n^2 + 1 \\ y_{n+1} &= bx_n \end{aligned} \tag{3}$$

1.2 Julia and Mandelbrot Sets

The simplest attractor, called a Julia set, was proposed by Beniot Mandelbrot.

$$z_{n+1} = z_n^2 - c \tag{4}$$

Both z and c are complex numbers in (4), and the system has two attractors: zero and infinity.

- Once the system enters the infinity attractor it never comes back.
- It can be shown that if, during any iteration, $||z|| > 2$ then the system is part of the infinity attractor.

Each unique value of c creates a different Julia set. A Julia set is defined as the set of start values for z that stay in the zero attractor. Complex numbers map nicely into an image because they represent points on a plane. Each point on the plane represents a different starting value for the attractor.

- A complex number $z = x + jy$, where j is the square root of -1.
- The Mandelbrot equations must be coded in two parts, the real and imaginary parts of z_{n+1}.

$$\begin{aligned} x_{n+1} &= x_n^2 - y_n^2 - c_x \\ y_{n+1} &= 2x_n y_n - c_y \end{aligned} \tag{5}$$

- The length of a complex number is given by its Euclidean distance.

$$||z|| = \sqrt{x^2 + y^2} \tag{6}$$

The **Mandelbrot Set** is the set of all Julia sets at the point $(0, 0)$. The visualization of the Mandelbrot set is a plane, and each point on the plane represents a different Julia set (unique c). The iteration is always started at $(x, y) = (0, 0)$.

1.3 Representing Julia and Mandelbrot Sets

For both Julia and Mandelbrot sets there are two values that vary over a plane. At each point we can iterate the Mandelbrot equations to identify if the system is in the zero or infinity attractor.

- For Julia sets, the initial start value of the system is given by the plane location and c is constant.
- For the Mandelbrot set, the initial start value of the system is $(0, 0)$ and c is given by the plane location.

To visualize the set you need to decide how to color the image. One way is to just make it one color if it's in the set and another color if it's not. Since we also know how many iterations it took for the system to rise above the infinity attractor threshold, we can use that information as well. Some possibilities include:

- If the iteration goes over the threshold on an even iteration make it one color, otherwise make it the other color.
- Make the zero attractor one color and blend the zero and infinity attractor colors based on how many iterations it takes to go over the threshold.

1.4 Coordinate Systems

The Mandelbrot and Julia sets exist on the continuous complex plane. To visualize them, we would like to specify a rectangle in the complex plane by defining a lower left coordinate (x_0, y_0) and the upper right coordinate (x_1, y_1). The complex plane coordinate system defines x as positive to the right and y as positive going up. An image is defined in terms of its pixel size and the y values in an image go down.

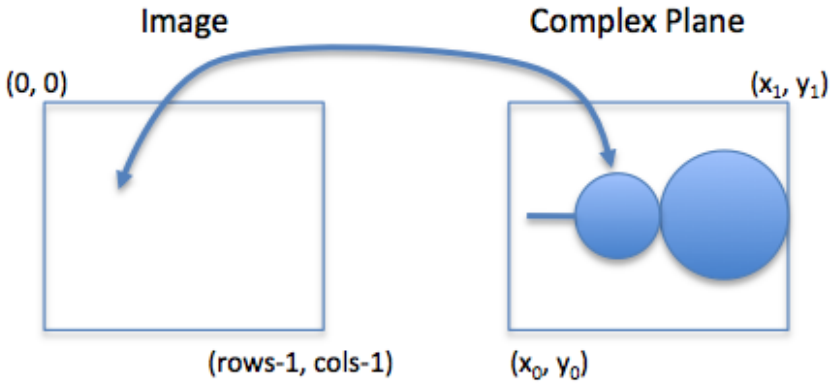

Figure 1: Transformation from one coordinate system to another.

What we need is a way to transform one coordinate system into the other. The question we need to answer is: for a given pixel (i, j) what is the corresponding (x, y) location on the complex plane?

To transform the image into its complex plane representation we have to do three steps:

1. Scale the image to be the same size as the complex plane
2. Invert the y-axis
3. Translate so the rectangles align

The scale factor to convert pixel location (i, j) into the complex plane scale is given by the following.

$$\begin{aligned} s_{\text{cols}} &= \frac{x_1 - x_0}{\text{cols}} \\ s_{\text{rows}} &= \frac{y_1 - y_0}{\text{rows}} \end{aligned} \tag{7}$$

To invert the y axis, we just multiply it by -1.

To translate so the rectangles align, we need the point $(0, 0)$ to move to the upper left of the rectangle on the complex plane.

$$\begin{aligned} t_x &= x_0 \\ t_y &= y_1 \end{aligned} \tag{8}$$

Combining all of these together, we get the following.

$$\begin{aligned} x &= s_{\text{cols}} j + t_x \\ y &= -s_{\text{rows}} i + t_y \end{aligned} \tag{9}$$

1.5 Algorithm

You should now have an idea how to calculate what location to test on the complex plane given a pixel and how to test it. The complete algorithm is given in pseudo-code below. The example below takes in two pairs of floats to represent (x_0, y_0) and (x_1, y_1).

```
Image *image_mandelbrot(float x0, float y0, float x1, float y1, int rows) {
  Image *im;

  // calculate the number of columns  cols = (x1 - x0) * rows / (y1 - y0)

  // allocate an image that is rows by cols

  // for each pixel in the image (i, j)

    // calculate (x, y) given (i, j)
    // this corresponds to cx and cy in the Mandelbrot equation

    // set zx and zy to (0, 0)

    // for some number of iterations up to N (e.g. 100)

      // iterate the Mandelbrot equation

      // if the length of z is greater than 2.0,
      // store the number of iterations and break

    // color pixel (i, j)

  // return the image
  return(im);
}
```