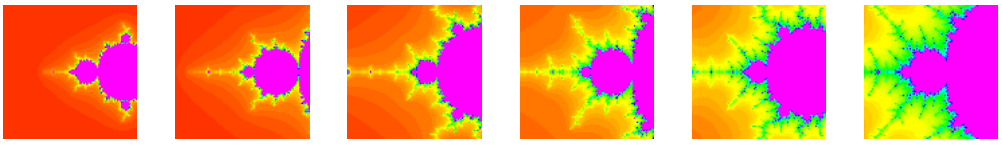
7 Fractals

We continue the series of practicals on bitmap images, now turning to fractal images such as the Mandelbrot set shown at the start of Practical 4. The 'fractal' nature comes from self-similarity: the whole image of the Mandelbrot set is closely replicated at finer and finer scales. The following series of images starts at the same scale as shown in Practical 4, only shifted a little to the left, and zooms in by a factor of two at each step on the bulb on the left; that bulb as its own little bulb on the left, which in turn has an even smaller bulb on the left, and so ad infinitum.



The Mandelbrot set is defined as follows: For a given complex number c, we define an infinite sequence $z_0, z_1, z_2 \ldots$, starting with $z_0 = 0$:

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

We say that the given number c is in the Mandelbrot set if the magnitude of the z_i values in this sequence is bounded by some constant, and outside the set if the magnitude grows unboundedly. For example, with $c = 1$ the sequence starts $0, 1, 2, 5, 26 \ldots$ and grows unboundedly, so $c = 1$ is not in the Mandelbrot set; but with $c = -1$, the sequence oscillates $0, -1, 0, -1 \ldots$ with magnitude at most 1, and so $c = -1$ is in the set.

We can represent the equation for computing the next element of the sequence by the function *next*:

```
next :: CF → CF → CF
next c z = z * z + c
```

1. Define a function to compute the trajectory of values for given c as an infinite list:

   ```
   mandelbrot :: CF → [CF]
   ```

   For example, *mandelbrot* $1 = [0, 1, 2, 5, 26 \ldots]$. We will call a function like *mandelbrot* a 'trajectory function'.

   ```
   mandelbrot c = iterate (next c) 0
   ```

It is in general impossible to tell whether a given point is in the Mandelbrot set: some cases are clearcut, but in general the trajectory might wander around indefinitely without either obviously converging or obviously diverging, in which case it is impossible to decide. But for the purposes of making pretty pictures, it suffices to approximate. We can take the first 'few' (say, 200) elements of the trajectory, and look to see whether they are all 'fairly close' to the origin (say, with magnitude at most 100). If these first few elements are all fairly close, we consider the point to be in the set; if not, we consider it to be outside the set.

2. Define a function

   $fairlyClose :: CF \to Bool$

   to approximate whether a point has not yet diverged.

   $$fairlyClose\ z = magnitude\ z < 100$$

3. Define a function

   $firstFew :: [CF] \to [CF]$

   to take the first few elements of an infinite sequence

   $$firstFew = take\ aFew$$
   $$aFew :: Int$$
   $$aFew = 200$$

4. Hence define a function

   $approximate :: (CF \to [CF]) \to Image\ Bool$

   that takes a trajectory function like *mandelbrot* and yields a boolean image, black for the points c whose trajectory is (approximately) bounded, and white for the points whose trajectory is (approximately) unbounded. For example,

   ```
   pbmRender "test.pbm"
     (sample (grid 100 100 ((-2.25) :+ (-1.5)) (0.75 :+ 1.5))
             (approximate mandelbrot))
   ```

samples the approximate Mandelbrot set to produce the following bitmap image:

approximate traj = *all fairlyClose* ∘ *firstFew* ∘ *traj*

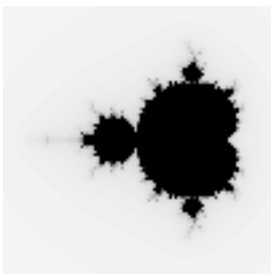
5. We can produce slightly more sophisticated pictures by asking a more sophisticated question: not just 'does the trajectory diverge?', but 'how quickly does the trajectory diverge?'. Define a function

   fuzzy :: (*CF* → [*CF*]) → *Image Float*

   that takes a trajectory function like *mandelbrot* and produces a greyscale image: a point *c* for which the first 'few' elements of the trajectory remain 'fairly close' should be black, as before, and a point *c* that is already beyond 'fairly close' should be white; but in between, points whose trajectories take some number of steps greater than zero but less than a 'few' to diverge should be some corresponding shade of grey. For example,

   ```
   pgmRender "test.pgm" 255
     (sample (grid 100 100 ((−2.25) :+ (−1.5)) (0.75 :+ 1.5))
             (fuzzy mandelbrot))
   ```

   yields this greyscale image:



```
fuzzy traj z = 1 − fromIntegral (steps z) / fromIntegral aFew
  where steps = length ∘ takeWhile fairlyClose ∘ firstFew ∘ traj
```

6. We can produce prettier pictures still by colouring the images. There is still only one dimension of information in the pixels, so the three-dimensional colouring is not adding any information (although it can make details easier to see); this is called *pseudo-colouring*, and you may have seen it also in physical images such as heatmaps and relief maps. We define a palette of colours:

```
rgbPalette :: [RGB]
rgbPalette =
  [RGB i 0 15 | i ← [15,14..0]] ++    -- purple to blue
  [RGB 0 i 15 | i ← [0..15]]    ++    -- blue to cyan
  [RGB 0 15 i | i ← [15,14..0]] ++    -- cyan to green
  [RGB i 15 0 | i ← [0..15]]    ++    -- green to yellow
  [RGB 15 i 0 | i ← [15,14..0]]       -- yellow to red
```

   Now define a function that will take a greyscale image such as that of the fuzzy Mandelbrot set above, and make a pseudo-colour image from it.

```
ppmView :: [RGB] → Grid Float → Grid RGB
```

   (Hint: you already did most of the work in an earlier exercise.) For example,

```
ppmRender "test.ppm" 15 (ppmView rgbPalette
  (sample (grid 100 100 ((−2.25) :+ (−1.5)) (0.75 :+ 1.5))
          (fuzzy mandelbrot)))
```

   produces the pseudo-coloured Mandelbrot image at the start of Practical 4.

```
ppmView = paletteView
```

7. Another kind of fractal image is defined using the same equation

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

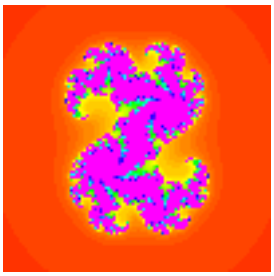
as before, but this time using a fixed constant for c and starting with z_0 being the candidate point rather than the origin. Define a function

$$julia :: CF \to CF \to [CF]$$

so that *julia c* is the trajectory function for a given constant c. For example, for $c = 0.32 :+ 0.043$ we can use

```
ppmRender "test.ppm" 15 (ppmView rgbPalette
  (sample (grid 100 100 ((-1.5) :+ (-1.5)) (1.5 :+ 1.5))
          (fuzzy (julia (0.32 :+ 0.043)))))
```

to yield this so-called Julia set:



```
julia c = iterate (next c)
```