Almond bread

Praanto Samadder

Spring ~~2018~~ 2024 | 26 February 2024 | 08:01

I found the problem PDF from 2018. Took 1 minute to see if this year's problem is identical to 2018's version of the problem.

1 The `Brod` module

For no other reason, we will call our module `Brod` instead of `Brot`. The `mandelbrot/2` method is called recursively. On our first call, we take in `m` to specify the max iteration and we decrement this every iteration.

We use `Cmplx.abs/2` to check if the absolute of our complex number is greater than 2, in which case, we can stop checking and return the number of iterations it took us to get here.

```
def mandelbrot(_, _, _, 0) do 0 end
def mandelbrot(z, c, m, dci) do
  case Cmplx.abs(z) > 2 do
    true ->
      m - (dci - 1)
    false ->
      mandelbrot(Cmplx.add(c, Cmplx.sqr(z)), c, m, dci - 1)
  end
end
def mandelbrot(c, m) do mandelbrot(0, c, m, m) end
```

2 The `Cmplx` module

The `Cmplx` module was implemented to handle complex numbers. For our purposes, complex numbers can be represented using tuples.

```
c = {real, imaginary}
```

2.1 Adding and squaring two complex numbers

Trivial implementation that can be achieved by summing the real and imaginary parts (separately). On the instance that we receive 0 as a parameter, for `sqr/1` we return `{0, 0}`.

```
def sqr(0) do
  {0, 0}
end
def sqr({r, b}) do
  {r * r - b * b, 2 * r * b}
end
```

2.2 Absolutes a complex number

`abs/1` always returns a number. As instructed, I make use of the `:math.sqrt/1` module.

```
def abs(0) do
  0
end
def abs({r, i}) do
  :math.sqrt(r * r + i * i)
end
```

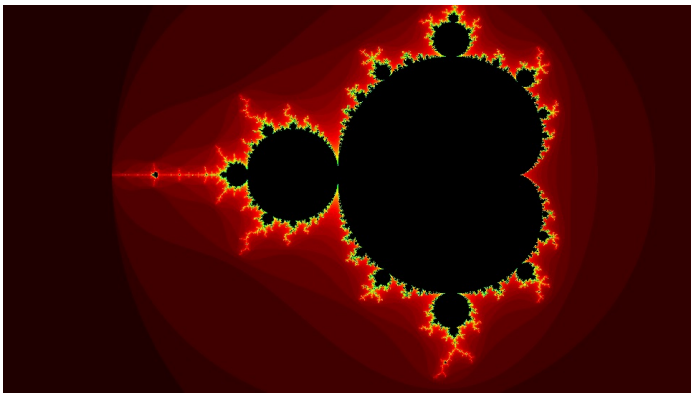
3 The `Mandel` module

Now it is time for us to recursively create a Mandelbrot set. I generate our set row-wise: meaning we generate a row left to right, then move to the next line and generate another row. I do this by first defining a function, `gen_width/2` that takes in the `width` and `height` for our entire set, a row variable to keep track of what rows we have made, the `trans/2` function and a `depth` parameter. We make generous use of our `Brod` and `Colour` modules to create a 2-dimensional array. Each element in this array represents a pixel in our final image.

```
def rows(_, 0, _, _, row) do row end
def rows(width, height, trans, depth, row) do
  new_row = gen_width(width, height, [], trans, depth)
  rows(width, height - 1, trans, depth, row ++ [new_row])
end
def gen_width(0, _, r, _, _) do r end
def gen_width(width, height, r, trans, depth) do
  complex = trans.(width, height)
  i = Brod.mandelbrot(complex, depth)
  colour = Colour.convert(i, depth)
  gen_width(width - 1, height, [ colour | r], trans, depth)
end
```

4 And finally...

And finally, we have our image generated using the `PPM` module.



5 Playing around with the colours

We can modify our `Colour` module and/or change the depth and starting values to change the colours and/or change the position of our fractal in the image. For instance, with the following parameters,

```
def demo() do
  small(-3, 1.420, 1.69)
end
```

we can generate the following image: