Generating a Mandelbrot Image

Programming II - Elixir Version

Johan Montelius

Spring Term 2018

Introduction

In this exercise you will implement a Mandelbrot set generator, or rather an image generator. You should do some reading so that you understand the basics of what the Mandelbrot set is and why it can generate some beautiful images; this text only contains a minimal explanation.

1 Mandelbrot

The Mandelbrot set is defined as the set of complex numbers c for which the sequence z_n does not approach infinity. The value z_n is defined as follows:

$$\begin{aligned} z_0 &= 0 \\ z_{n+1} &= z_n^2 + c \end{aligned}$$

If you remember how to do the square of a complex numbers you know everything there is to know to start:

$$(x + yi)^2 = x^2 - y^2 + 2xyi$$

How do we know if a complex number $(x+yi)$ belongs to the Mandelbrot set? We could of course start to compute z_n for higher values and see if we approach infinity but that would of course take a very long time (to say the least).

An observation saves us from spending the rest of our lives computing z_n: *if $|z_n| > 2$ then there is no turning back, z_n will only increase in size.* The absolute value of a complex number is of course:

$$|(x + yi)| = \sqrt{x^2 + y^2}$$

We still do not want to compute forever; if the number actually does belong to the Mandelbrot set we will of course never hit the threshold.

Therefore, we set an upper limit n that will be the maximum *depth* of our computation.

So given a maximum value of n, we can for any complex number c say if it *definitely does not belong to* or if it *could possibly belong to* the Mandelbrot set. In the case where we know for sure that the number does not belong to the set we also have a value i which was the point where $|z_i| > 2$. This value i is the color we need to generate a beautiful Mandelbrot image.

1.1 Complex numbers

Since we are going to work with complex numbers we might as well start by implementing a module to handle these. Let's make it simple and represent a complex number as a tuple with its real and imaginary values. Create a module `Cmplx` that exports the following functions:

- `new(r, i)` : returns the complex number with real value r and imaginary i
- `add(a, b)` : adds two complex numbers
- `sqr(a)` : squares a complex number
- `abs(a)` : the absolute value of a

You might want to use the sqrt/1 function exported from the Erlang :math module when calculating the absolute value. You call Erlang modules like any module but the Erlang modules all have atoms as names.

```
> :math.sqrt(42)
```

The Complex module implements an abstract data type; the internals of how complex numbers are represented should not be visible outside of the module. We of course know, but we should not make use of this knowledge.

1.2 The Brot module

For no reason at all we will call our first module `Brot`, it will implement the computation of the i value given a complex value c. We must of course give it a maximum iteration *depth* or we risk getting stuck in an infinite computation.

Implement a function `mandelbrot/2` that, given the complex number c and the maximum number of iterations m, return the value i at which $|z_i| > 2$ or 0 if it does not for any $i < m$ i.e. it should always return a value in the range $0..(m-1)$.

```
def mandelbrot(c, m) do
  z0 = Cmplx.new(..., ...)
  test(0, z0, c, m)
end
```

The `test/4` function should of course test if we have reached the maximum iteration, in which case it returns zero, or if the absolute value of `z` is greater than 2, in which case it returns `i`. Make sure that you use the functions exported from the `Cmplx` module.

Do some test to see that the function works:

- `Brot.mandelbrot(Cmplx.new(0.8, 0), 30)`
- `Brot.mandelbrot(Cmplx.new(0.5, 0), 30)`
- `Brot.mandelbrot(Cmplx.new(0.3, 0), 30)`
- `Brot.mandelbrot(Cmplx.new(0.27, 0), 30)`
- `Brot.mandelbrot(Cmplx.new(0.26, 0), 30)`
- `Brot.mandelbrot(Cmplx.new(0.255, 0), 30)`

What is happening? Which values could possibly belong to the Mandelbrot set - how sure are you? Do some more testing, why stop at thirty iterations? Try fifty!

1.3 The printer

Before carrying on we should make sure that we can generate an image. You should have the file `PPM` module that will write the final image to a file. Make sure that you can use this module and that you know where files are located when created.

The API to the module is:

- `write(name, image)`: where the name is the name (possibly full path name) of the file and the image is a list of rows where each row is a list of tuples `{:rgb, r, g, b}` (each value being in the range 0..255).

So once we know that it is working we can carry on to produce some images.

1.4 Colors

We create one module (that in the end will be the one that you want to play with the most), the `Color` module. This module should export a function `convert(depth, max)` that given a depth on a scale from zero to max gives us a color,

The conversion of depth information to *RGB values* can of course be done in many different ways and the one presented here is only for inspiration.

Let's assume that we have a depth of a point d, with the maximum possible depth being m (or rather the maximum depth is $m - 1$ since we return 0 if we reach the maximum depth). We could create five sections that divides the range 0 to m. Divide d by m and so that you have a fraction f. Then multiply this fraction by five to generate a floating point a from 0 to (not inluding) 5. Now truncate the value to give you an integer x from 0 to 4 (this is the section) and generate an offset y that is the truncated value of $255 * (a - x)$.

The two values x and y will now be used to give you an RGB value. You can use the following transformation:

```
case x do
  0 -> {:rgb, y, 0, 0}
  1 -> {:rgb, 255, y, 0}
  2 -> {:rgb, 255 - y, 255, 0}
  3 -> {:rgb, 0, 255, y}
  4 -> {:rgb, 0, 255 - y, 255}
end
```

What colors does this correspond to? Does it look anything like a rainbow? Close to a rainbow? The mapping from depth to colors is one thing that one can play with, its not at all given that the colors should be chosen base only on the depth, one might even want to know the distribution of depths in the whole image or reuse colors at different depths.

1.5 Computing the set

So we know how to find the depth of a complex number so why not try to compute the depth at all points in a rectangular plane. We create a module `Mandel` that should calculate an image. The function that will be our interface to the module looks like this:

```
def mandelbrot(width, height, x, y, k, depth) do
  trans = fn(w, h) ->
    Cmplx.new(x + k * (w - 1), y - k * (h - 1))
  end
```

```
  rows(width, height, trans, depth, [])
end
```

What is happening here? We want to generate an image of the size *Width by Height*. The upper left corner of this image is the point $x + yi$ and the offset between two points is k. This means that the first pixels of the upper row should correspond to the "depth" of $x + yi$, $(x + k) + yi$, $(x + 2k) + yi$ etc and that the second row starts with $x + (y - k)i$.

To help the Mandelbrot generator from keeping track of this we simply provide a function that does the work. The *trans* function will take a pixel position (w, h) and return a complex number that is the one we should compute the depth of. It is better to do this here and then we could more easily change the function rather than passing all the necessary information in arguments.

Now the *rows* function should return a list of rows, where each row is a list of colors. Each item in a row corresponds to a pixel at (w, h) and the color is computed by:

- generating the complex number that corresponds to the pixel
- calculate the depth of this value
- convert the depth to a color

The only tricky issue is to generate the rows in "correct" order, it is easy to generate the image up side down or mirrored. In the end it does not mean very much but try to get it right.

When you can generate an image, write it to a file using the `PPM` module. This code would hopefully give you a first look at the Mandelbrot set.

```
def demo() do
  small(-2.6, 1.2, 1.2)
end

def small(x0, y0, xn) do
  width = 960
  height = 540
  depth = 64
  k = (xn - x0) / width
  image = Mandel.mandelbrot(width, height, x0, y0, k, depth)
  PPM.write("small.ppm", image)
end
```

2 Carrying on

Generate a nice looking image, you will find the most interesting things close to the edge of the black set. This is where the fractals start to spin out of control and the beauty of the Mandelbrot set is found. It's amazing that so much information could be hidden in a function this simple.

$$\begin{aligned} z_0 &= 0 \\ z_{n+1} &= z_n^2 + c \end{aligned}$$

Could we speed up the calculations? Are there any operations that are of unnecessary complexity? Can you include an image in your report (you would probably have to convert it to png)?