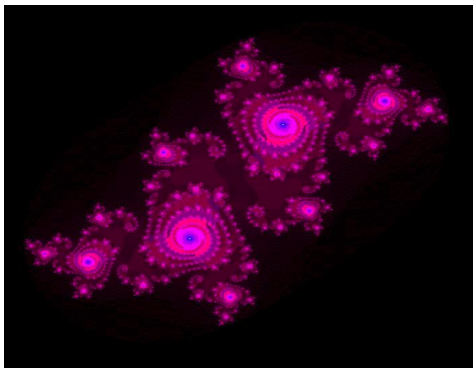
PROGRAMACIÓN FUNCIONAL

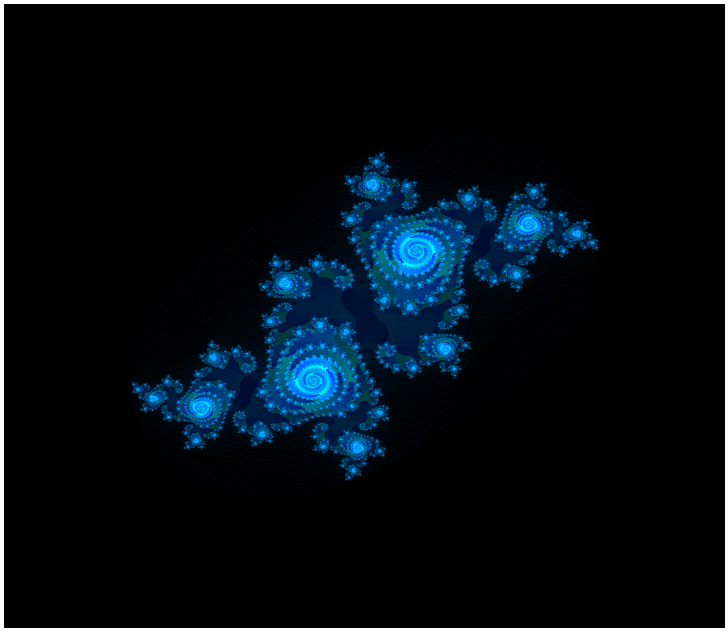
Construyendo Fractales con programación funcional

Janeth De Anda Gil



Introducción

Un fractal es un **objeto cuya estructura se repite a diferentes escalas.** Es decir, por mucho que nos acerquemos o alejemos del objeto, observaremos siempre la misma estructura.



Uno de los ejemplos más conocidos de fractales es probablemente el conjunto de mandelbrot, que fue propuesto en los años setenta, por Benoit Mandelbrot, pero no fue hasta una década más tarde cuando pudo representarse gráficamente con un ordenador.

Basandonos en el árticulo Composin Fractals, escrito por Mark P. Jones, se describira un programa escrito en Haskell que generan imágenes del conjunto de Mandelbrot . El desarrollo de este programa tiene una elegante construcción compositiva y hace uso de funciones de orden superior que nos permiten capturar patrones comunes de cómputo, particularmente en el caso de procesamiento de listas.

Otras ventajas del programa es que si se realizan pequeños cambios, se puede explorar una parte diferente del conjunto de Mandelbrot, se puede visualizar un tipo diferente de fractal o el procesamiento de la imágen resultante puede mostrarse con pixeles en una pantalla gráfica, cabe destacar que en el árticulo está última parte esta realizada con una librería que se usa para el sistema operativo Windows , en este trabajo se hace la aportación de realizar las imágenes en un sistema operativo de linux.

El conjunto de Mandelbrot es una colección de puntos , cada uno de los cuales es un par de números flotantes
type Point = (Float, Float)

Hay dos pasos en el procedimiento para determinar si dado un punto p es miembro del conjunto de Mandelbrot o no, en el primer paso, nosotros usamos

las coordenadas de p para construir una secuencia de puntos , *mandelbrot p*. En el segundo paso , nosotros examinamos los puntos en esta secuencia y, si todos ellos son "bastante cercanos" a el origen, entonces nosotros concluimos que p es un miembro del conjunto de Mandelbrot, pero si los puntos obtenidos se encuentran más y más lejos del origen , entonces nosotros podemos estar seguros que p no es miembro del conjunto de Mandelbrot.

Se construye una función que nos ayuda a crear un punto de la secuencia

```
next::Point->Point->Point

next (u,v) (x,y)=(x*x-y*y+u,2*x*y+v)
```

Si se toma un punto p y otro punto z, entonces se pude aplicar la función next p repetidamente a z para generar una infinita secuencia como la que se muestra a continuación:

```
[z , next p z , next p (next p z ), next p (next p (next p z )), . . .
```

Para construir está secuencia se puede aplicar perfectamente la función iterate en Haskell, la cual se basa en una evaluación peresoza que permite la construcción de un conjunto de listas infinita. Para el conjunto de Manelbrot, se tomara el punto z como el origen (0,0) y nosotros construimos las secuencia correspondiente a un punto p usando la siguiente definición

```
mandelbrot ::Point->[Point]

mandelbrot p=iterate (next p) (0,0)
```

Ahora se debe determinar si los puntos correspondientes a la secuencia pertenecen al conjunto de Mandelbrot , entonces debemos verificar si si el punto (u,v) esta bastante cerca al origen. De hecho se dirá que el punto pertenece al conjunto si se encuentra a una distancia 10 del origen (este 10 es arbitrario), haciendo uso del teorema de pitagoras, esto es equivalente a $u^2+v^2<10$, que es lo mismo que $u^2+v^2<100$ por lo tanto, se define una función que indica si el elemento esta en el conjunto de mandelbrot

```
cerca::Point->Bool

cerca (u,v)=(u*u+v*v)<100
```

Después se verifica si todos los puntos en la correspondiente secuencia están lo suficientemente cerca al origen. Esto se puede expresar en Haskell facilmente con la función *all:* la cual acepta un predicado y una lista y devuelve verdadero si todos los elementos de la lista cumplen el predicado, debido a que la secuencia de números es infinita , se hace uso de la función take, que tomara cierto número de elementos ya que en vez de intentar determinar si todos los puntos en una secuencia se encuentran cerca del origen, solo se contaran cuantos de los puntos iniciales cuentan

con este criterio hasta un limite finito que en este caso será el número de colores que aplicaremos a nuestro fractal.

estaEnMandelbrot ::Int-> Point ->Bool

estaEnMandelbrot n p= all cerca (take n(mandelbrot p))

Los colores del fractal se escogeran de la siguiente manera, si el primer punto en una secuencia falla la prueba, entonces se desplegara este haciendo uso del primer color de nuestra paleta de colores, si la prueba falla cuando se está buscando el segundo punto, entonces se asigna el segundo color de la paleta sucesivamente. Para esto construimos la función chooseColor

chooseColor :: [color]->[Point]->color

chooseColor paleta =(paleta!!).length.take n.takeWhile cerca where n=length paleta-1

En esta función se toman los puntos que cumplan con que la función cerca sea verdadera , puede ocurir que la lista se haga de tamaño infinito, por lo tanto se tomaran solo n puntos (que son los elementos de la paleta), mide el tamaño de esta lista de puntos y toma el color que se encuentre en la posición de la longitud de la paleta.

Una imagen es un mapeo que asigna el valor de un color a cada punto

type Image color = Point ->color

Aquí el color es un tipo variable , el cual permite acomodar diferentes tipos de mecanismos de dibujo y paletas, en efecto el conjunto de mandelbrot puede ser pensado como una variable de tipo Image Bool, mapeando puntos que estan en el conjunto como verdaderos y dejando fuera los puntos Falsos.
Para obtener una imagen más colorida, nosotros podemos combinar una un generados de secuencias, es decir, un fractal, con una paleta de colores haciendo uso de la función chooseColor.

fracImage::(Point->[Point])->[color]->ImageFunction color

fracImage fractal paleta= chooseColor paleta.fractal

En la función de arriba chooseColor paleta se aplica a fractal (función) , en este caso se aplicara la función mandelbrot, y nos devuelve una función donde evalua un punto y devuele un color usando el chooseColor.

Por lo regular en los dispositivos las imágenes se producen mediante la especificación de colores para puntos en un espacio rectangular acotado por filas y columnas.

type Grid a = [[a]]

Por ejemplo una imagen con r filas y c columnas puede ser descrita como una lista de longitud r , donde cada renglon es una lista de longitud c.

La siguiente tarea es construir está malla, cada imagen cubre un rango de valores de puntos, los cuales se describen por las coordenas (xmin,ymin) de el punto en la esquina inferior izquierda, y las coordenadas (xmax, ymax) de la esquina superior derecha. Entonces construimos una malla con estas coordenadas.

Cada segmento del valor mínimo al máximo debe de estar dividido en tramos iguales por lo que se construira la siguiente función

for :: Int->Float-> Float ->[Float]

for n min max =take n [min, min + delta ..] where delta = (max - min)/fromIntegral (n — 1)
Se debe dividir la malla en los segmentos que se desean en filas y en columnas, para esto hacemos una lista de compresión la cual va a contener puntos (x,y) donde x representa cada uno de los tramo de la división del segmento de las columnas y y cada uno de los tramos de la división del segmento de las filas.

grid :: Int -> Int ->Point -> Point-> Grid Point

grid c r (xmin , ymin) (xmax , ymax)= [[(x , y) | x<- for c xmin xmax] | y<- for r ymin ymax]

Dado un punto en la rejilla podemos aplicarle un color, ahora para toda la rejilla entonces se crea la función sample, la cual hace un doble map ya que la malla es una lista de listas

sample :: Grid Point-> ImageFunction color->Grid color

sample points image = map (map image) points

Ya tenemos la mayoría de las piezas para armar nuestra imagén fractal, por lo que ahora las juntaremos en una función llamada draw

draw ::Grid Point-> (Point ->[Point])-> [color]->(Grid color ->image)->image

draw points fractal palette render = render (sample points (fracImage fractal palette))

En este caso fracImage regresa image(que convierte los puntos a colores) y sample regresa la malla de colores a partir de una malla de puntos y del image color; render convierte la malla de colores a un objeto (imagen donde los colores pueden ser caracteres o colores , etc)

Fractales con caractéres

Es posible producir figuras atractivas para el conjunto de Mandelbrot usando solo caracteres, el primer paso es definir la paleta de caracteres

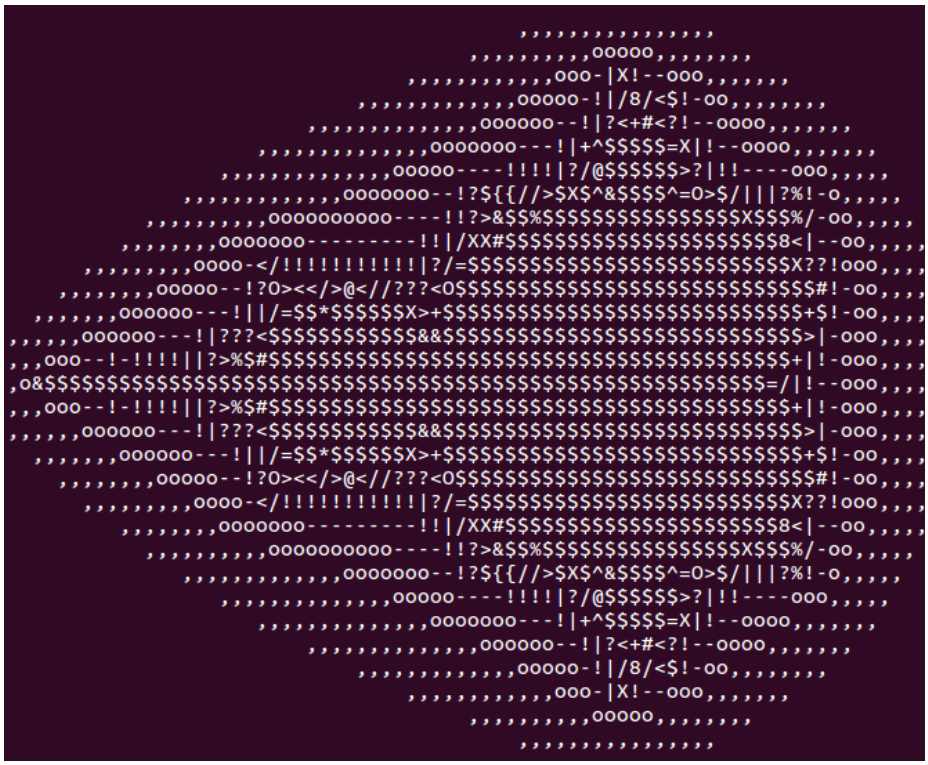
```
charPalette :: [Char]

charPalette= "     ,o-!|?/<>X+={^O#%&8*$"
```

Para desplegar la imágen se hace uso se IO(), en la siguiente función putStr imprime en pantalla un salto de línea cada que encuentra un \n y unlines devuelve un String que concatena los elementos de una lista de String separados por un \n

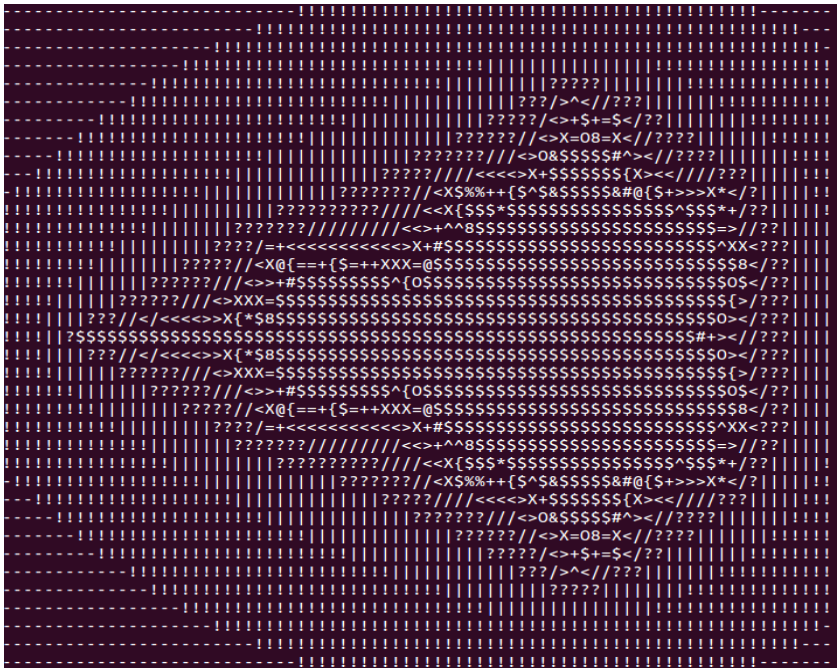
```
charRender :: Grid Char->IO()

charRender=putStr.unlines
```

Haciendo uso de la paleta y usando la función Mandelbrot podemos contruir la siguiente figura

```
figure1 = draw points mandelbrot charPalette charRender where points = grid
79 37 (-2.25, -1.5) (0.75, 1.5)
```



Podemos obtener diferentes figuras jugando con la paleta de caracteres



Gracias a como están escritas las funciones, se puede de manera muy sencilla implementar la función de julia, la única diferencia es que con el conjunto de mandelbrot el punto que se dejaba fijo era el (0,0) y se varia el primer parametro y en el conjunto de julia, el punto fijo es el primer parametro y se varia el punto inicial

julia::Point->Point->[Point]

julia c= iterate (next c)

figure2 = draw points (julia (0.32, 0.043)) charPalette charRender where points = grid 79 37 (-1.5, -1.5) (1.5, 1.5)

Podemos obtener diferentes figuras jugando con la paleta de caracteres

Fractales con colores

Otra forma de representar imágenes fractales es para mostrar estos utilizando alta resolución graficos, con un píxel de color para cada punto en la cuadrícula de entrada. Se puede modificar el código para dibujar imágenes como esta dado sólo unas pocas primitivas simples: una paleta de colores.

Se implemento una librería que se puede usar en linux llamada juicyPixel y en el árticulo mencionado se implementa una paleta de colores utilizando las facilidades proporcionadas por las librerías de gráficos Hugs(Reid , 2001) pero esta librería es para el sistema operativo Windows, a continuación se muestra la implementación

```
rgbPalette ::[RGB]
graphicsWindow :: Int → Int → IO Window
setPixel :: Window → Int → Int → RGB → IO ()
```

Los tipos anteriores reflejan esta herencia en el uso de los colores RGB y el tipo de ventana para una ventana gráfica, además de dibujar la imagen, el siguiente código crea una ventana, esperando que el usuario presione una tecla cuando haya visto el resultado y entonces se cerrara la ventana

```
rgbRender :: Grid RGB → IO ()
rgbRender g = do w ← graphicsWindow (length (head g)) (length g) sequence
[setPixel w x y c | (row ,y) ← zip g [0..], (c, x ) ← zip row [0..]]
getKey w
closeWindow w
```

Ahora explicaremos como se implemento el programa para generar imagenes fractales con una paleta de colores haciendo uso de la librería juicyPixel.

Necesitaremos una paleta de colores, es decir una lista de colores RGB

```
paletaRGB :: [PixelRGB8]
```

Se puede formar una paleta de colores, investigando los colores y colocandolos en una lista, pero en este caso preferi usar colores degradados para que la imagen tenga colores que combinaran, entonces se hace una función de degragado

```
degradado :: PixelRGB8 -> PixelRGB8 -> Int -> [PixelRGB8]
degradado (PixelRGB8 x y z) (PixelRGB8 x2 y2 z2) pasos = [PixelRGB8 (x+
(division x2 x pasos)*(fromIntegral i)) (y+(division y2 y
pasos)*(fromIntegral i)) (z+(division z2 z pasos)*(fromIntegral i)) | i <-
[0..pasos]]

division :: Pixel8 -> Pixel8 -> Int -> Pixel8

division x y pasos= round ((fromIntegral x-fromIntegral y) / (fromIntegral
pasos))
```

La función degradado acepta un color RGB inicial y un color RGB final y el número de colores a usar, donde el tipo de datos PixelRGB8 acepta tres datos x,y, z que significan x tono de rojo , y tono de verde y z tono de azul, entonces, si queremos n colores, dividimos el tramo de color inicial a final entre n colores suponiendo esto es c, entonces nuestro color inicial aumentara de c en c hasta llegar al color final
Ahora si creamos una paleta de colores

```
paletaRGB = degradado (PixelRGB8 0 0 0) (PixelRGB8 100 175 225) 200
```

Para crear una imágen se usa writePng seguido de la dirección donde se desea que se guarde la imagen, el panel de colores y el ancho y largo de la imágen

```
imageCreator :: String -> Int -> Int -> (Int->Int->PixelRGB8) -> IO ()
imageCreator path width height pixelRenderer= writePng path $ generateImage
pixelRenderer width height
```

Para crear una imágen se usa writePng seguido de la dirección donde se desea que se guarde la imagen, el panel de colores y el ancho y largo de la imágen

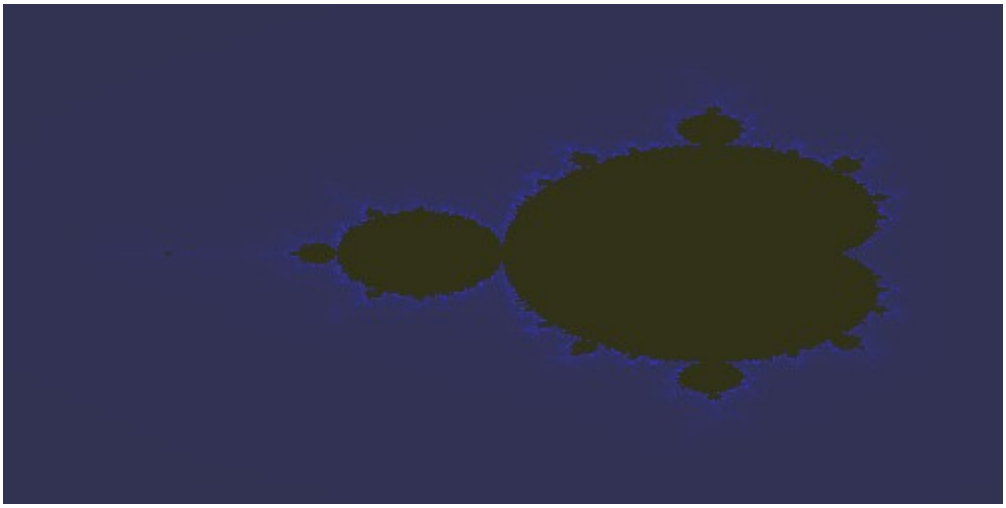
```
imageCreator :: String -> Int -> Int -> (Int->Int->PixelRGB8) -> IO ()
imageCreator path width height pixelRenderer= writePng path $ generateImage
pixelRenderer width height
```

Ahora creamos una imagen dada una malla de pixeles

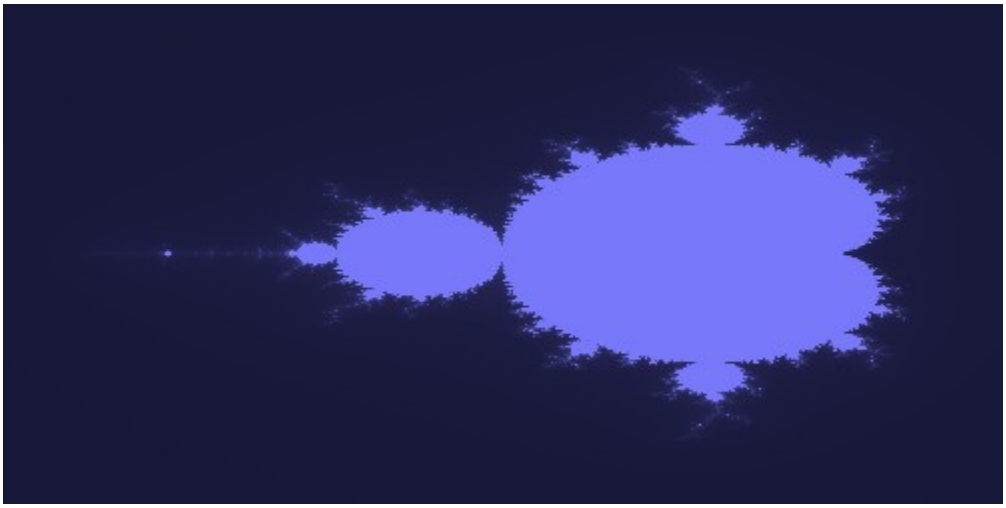
```
rgbRender :: Grid PixelRGB8 -> IO()
rgbRender grid = imageCreator "fractal_j.png" (length (grid!!0)) (length
grid) pixelRenderer
               where pixelRenderer x y = (grid!!y)!!x
```

Ahora hagamos una imágen fractal de Mandelbrot utilizando los siguientes parametros para el panel de colores

```
paletaRGB = degradado (PixelRGB8 50 50 80) (PixelRGB8 100 120  200) 200

figure3 = draw points mandelbrot paletaRGB rgbRender where points = grid
500 250 (-2.25, -1.5) (0.75, 1.5)
```

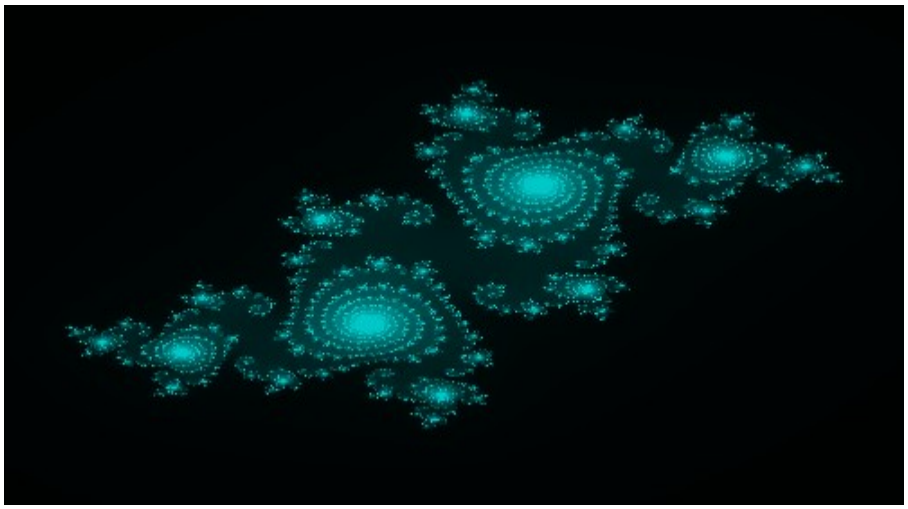


```
paletaRGB = degradado (PixelRGB8 20 20 50) (PixelRGB8 100 100  200) 100
```



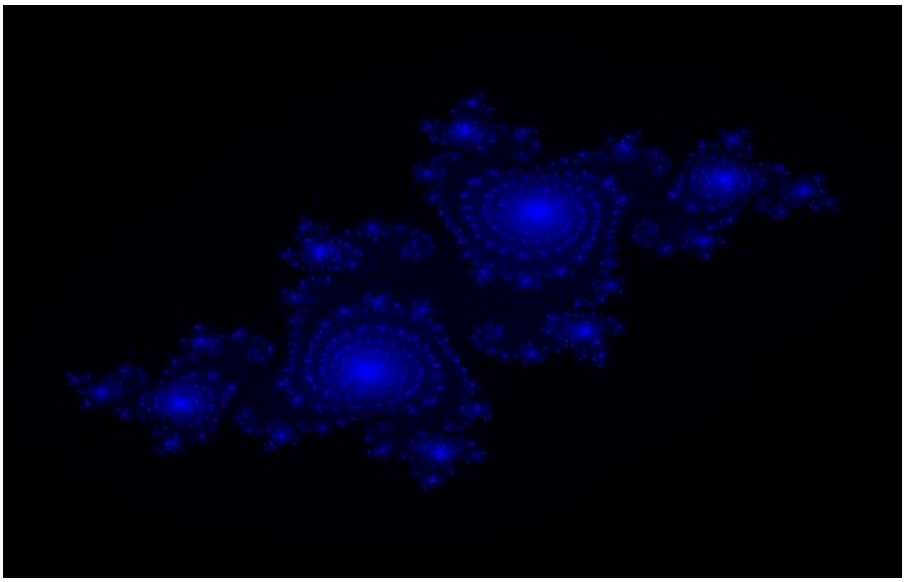
Ahora hagamos una imágen fractal de Julia utilizando los siguientes parametros para el panel de colores

```
figure4 = draw points (julia (-0.194, 0.6557)) paletaRGB rgbRender where
points = grid 500 250 (-1.5, -1.5) (1.5, 1.5)

paletaRGB = degradado (PixelRGB8 0 0 0) (PixelRGB8 100 175 225) 200
```



Para la siguiente paleta

```
paletaRGB = degradado (PixelRGB8 0 0 0) (PixelRGB8 100 100 220) 250
```

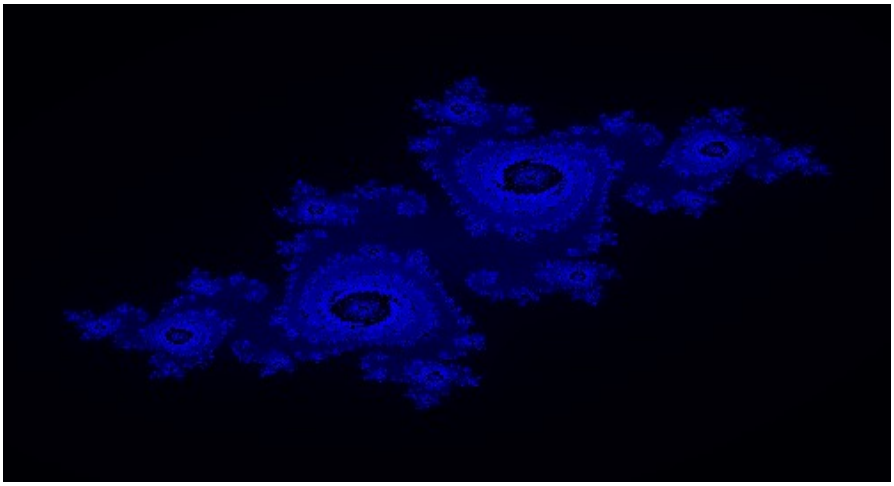


Cambiando la degradación para que el color azul sea más visible, es decir el dato que representa al color azul aumentara dos veces más rápido

```
degradado (PixelRGB8 x y z) (PixelRGB8 x2 y2 z2) pasos =
  [PixelRGB8 (x+(division x2 x pasos)*(fromIntegral i))
             (y+(division y2 y pasos)*(fromIntegral i))
             (z+(division z2 z pasos)*(fromIntegral j)) | i <- [0..pasos],j
<- [0,2..pasos*2]]
```

Y con los siguientes datos como paleta se tiene

```
paletaRGB = degradado (PixelRGB8 0 0 0) (PixelRGB8 100 100 100) 100
```

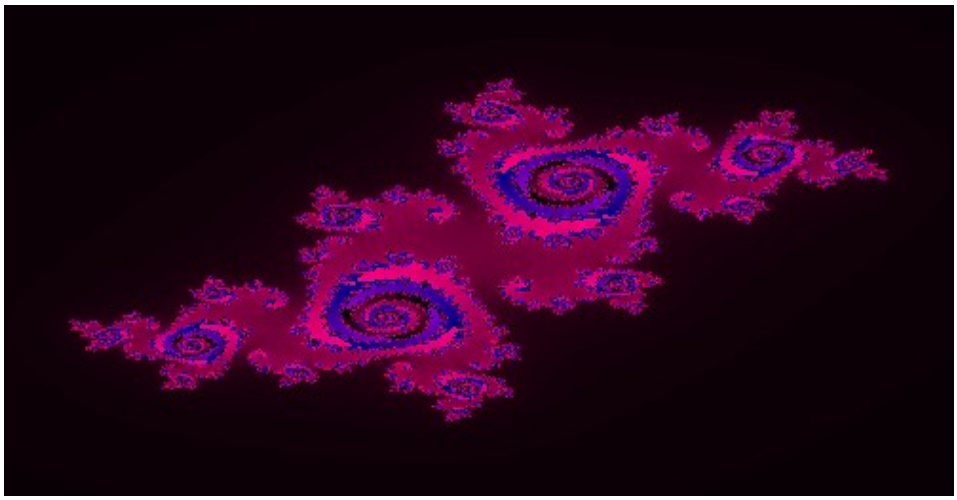


Si no solo aumentamos los tramos al doble del dato que representa al color azul, si no también el del color rojo

```
degradado (PixelRGB8 x y z) (PixelRGB8 x2 y2 z2) pasos =
  [PixelRGB8 (x+(division x2 x pasos)*(fromIntegral j))
             (y+(division y2 y pasos)*(fromIntegral i))
             (z+(division z2 z pasos)*(fromIntegral j)) | i <- [0..pasos],j
<- [0,2..pasos*2]]
```

y con la siguiente paleta de colores

```
paletaRGB = degradado (PixelRGB8 0 0 0) (PixelRGB8 190 100 100) 100
```



y con la siguiente paleta de colores

paletaRGB = degradado (PixelRGB8 0 0 0) (PixelRGB8 100 100 190) 100