Universidad Veracruzna

Inteligencia Artificial

---

Análisis de Algoritmos

# Actividad 3

Leonardo Flores Torres

21 de septiembre de 2022

**Programar un algoritmo de su elección (no tan sencillo) y analizarlo de la siguiente forma:**

1. Graficar el tiempo de ejecución en función de $N$,
2. sobre los mismos ejes graficar 2 cotas superiores y dos cotas inferiores,
3. repetir el punto 1 y 2 ejecutando el programa en otra computadora de distinto desempeño,
4. analizar los resultados y discutirlos. Escribir de la manera más completa las características de las 2 computadoras.

El algoritmo que se eligió fue computar un fractal, más específicamente, el fractal que se genera a partir del set de Mandelbrot. Las propiedades del fractal de Mandelbrot han sido estudiadas considerablemente, tan es así que en este trabajo se usa para estudiar los efector computacionales de implementar un algoritmo para generarlo.

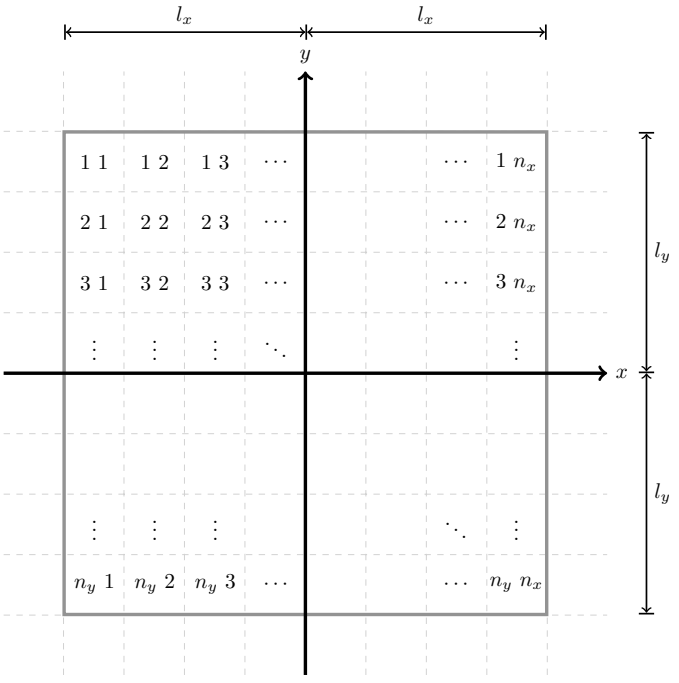



Figura 1: Discretización del espacio

El set de Mandelbrot se conoce como el conjunto de puntos generado a partir de

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

donde ambos $z$ y $c \in \mathbb{C}$. Pero no todos los puntos $z_i$ pertenecen al conjunto, solamente aquellos que $|z_i| \leq 2$, esto quiere decir que no es necesario computar todos los valores dentro de un dominio de cardinalidad infinita como lo es $\mathbb{C}$. Una manera útil de interpretar esta restricción es que solamente los números $z_i$ que estén dentro de un círculo centrado en el origen de radio 2 pueden pertenecer al conjunto.

Quisisera tomarme la libertad acerca de cómo fue que construí el algoritmo. Encontré varias fuentes en internet tales como un post en el sitio CodinGame, y en el sitio popular de tutoriales de `python`, RealPython. A pesar de incluir sus códigos faltaba la explicación que considero yo más importante, aquella que trata la discretización del espacio más allá de solamente definir una matriz de pixeles con ciertas dimensiones.

Primero se debe comenzar con una representación del espacio como se muestra en la figura 1 donde el grid tiene una anchura de $2l_x$, y una altura de $2l_y$. En los lenguajes de programación es común que al tratar con una imágen sus pixeles estén ordenados como se muestra en la imagen anteriormente mencionada. Esto asemeja a una matriz, sí, una matriz de pixeles donde cada entrada de la matriz tiene dos elementos importantes a considerar. Los índices de las entradas en la matriz, y el valor del color que cada entrada contiene, ya sea en RGB, HSV, u otro.

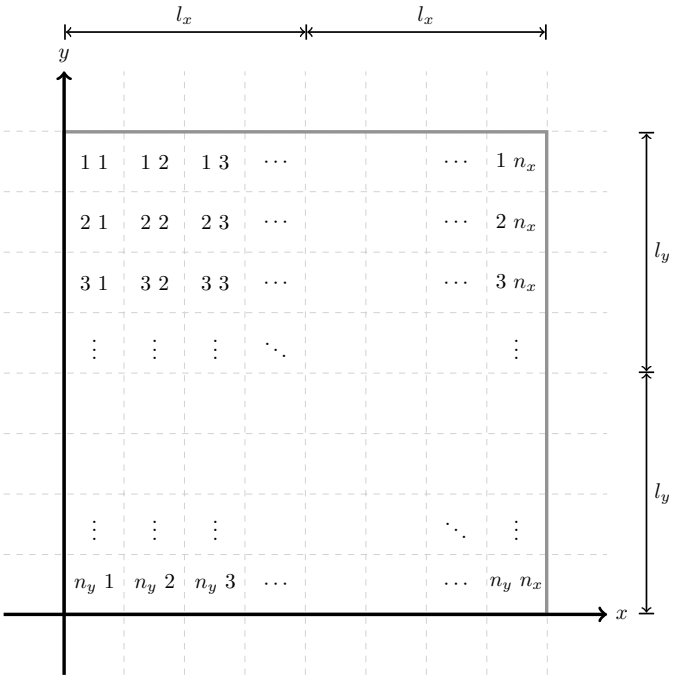



Figura 2: Traslación aplicada al espacio discretizado.

En este punto todavía es algo confuso sobre cómo trabajar con los índices de la matriz para generar las coordenadas necesarias para realizar el cómputo de $z_i$. Es importante clarificar que el eje-$y$ es el eje de la componente imaginaria de $z_i$, $Im(z_i)$, mientras que en el eje-$x$ se encuentran las componentes reales $Re(z_i)$ ¿Que tal si se hace una traslación de las coordenadas para que un vértice de nuestro espacio discretizado coincida con el origen de coordenadas? La traslación elegida fue $x' = x - l_x$, y $y' = y - l_y$. Nótese que por la restricción inicial $|z_i| \leq 2$ se deriva que $l_x \leq 2$ y $l_y \leq 2$.

Después de la traslación, figura 2, pareciera ser mas intuitivo cómo debe uno moverse en el espacio para moverse entre vertices de pixeles. Por ejemplo, si quisiera moverme al pixel en la primera fila y tercera columna hay dos

opciones. Si comenzara a moverme en el vértice superior izquierdo de mi espacio podría moverme una distancia $3\Delta x$ y ninguna en $y$ lo que me posicionaría tocando al pixel $[1,\ 3]$ en su vértice superior derecho, o podría moverme $2\Delta x$ y $\Delta y$ para encontrarme en el vértice inferior izquierdo del mismo pixel. Nuestra intuición va bien encaminada, solamente necesita de un paso más.

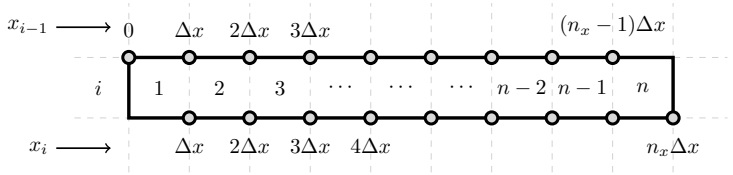



Figura 3: Movimiento entre los vértices de pixeles en una fila cualquiera dentro de la imagen.

No se ha mencionado porque ha sido ilustrado en las figuras 1 y 2 pero los pixeles representan un movimiento desde un vértice a otro vértice de tamaños $\Delta x$ o $\Delta y$ dependiendo si nos movemos a lo largo del eje-$x$ o del eje-$y$. Donde,

$$\Delta x = \frac{2l_x}{n_x} \ ,$$
$$\Delta y = \frac{2l_y}{n_y} \ ,$$

y $n_x$, $n_y$, representan el número de divisiones del espacio lo cual es quivalente al número de pixeles en el que se divide la imagen a lo largo de los ejes.

Retomando el ejemplo anterior, las dos maneras de comenzar el movimiento entre vértices se ilustra en la figura 3 para una fila cualquiera de pixeles dentro de la imagen, lo mismo aplica para el movimiento para una columna cualquiera dentro de pixeles. Ahora los índices que se tenian en un inicio nos dicen algo, dependiendo del índice será la posicion en la que estaremos con movimientos de longitus $\Delta x$,

$$x_{i-1} = (i-1)\Delta x \ ,$$
$$x_i = i\Delta x \ ,$$

donde $i = 1, 2, 3, \ldots, N$. Además, $N$ puede ser $n_x$ ó $n_y$ dependiendo de la dirección del movimiento. Lo mismo es cierto en la dirreción del eje-$y$. Ahora, si tomamos $x_i$ y $x_{i-1}$ podemos encontrar el punto medio, aquel que se encuentra justo en el centro del pixel

$$\begin{aligned} x_c^i &= \frac{x_i + x_{i-1}}{2} = \frac{i\Delta x(i-1)\Delta x}{2} \ , \\ &= \Delta x \frac{(2i-1)}{2} = \frac{2l_x}{2}\frac{(2i-1)}{2} \ , \\ x_c^i &= \frac{l_x}{n}(2i-1) \ . \end{aligned}$$

De igual manera,

$$y_c^i = \frac{l_x}{n}(2i-1) \ .$$

De aquí que podemos acceder a cualquier elemento dentro de nuestra matriz de pixeles partiendo de sus índices para obtener sus coordenadas $(x_c^i, y_c^i)$ ¿Recuerda usted la traslación hecha anteriormente? Tenemos que regresar a nuestro sistema de coordenadas iniciales, esto se obtiene al movernos de regreso una distancia $l_x$ en el eje-$x$, y una distancia $l_y$ sobre el eje-$y$,

$$\begin{aligned} x_i' &= x_c^i - l_x \ , \\ y_i' &= y_c^i - l_y \ . \end{aligned} \tag{1}$$

El haber estudiado el problema de antemano permite concentrar los esfuerzos en realizar la implementación de manera más adecuada, sencilla, y elegante. Aunque los requisitos de la presente tarea no incluían la explicación y desarrollo de la construcción del algoritmo preferí hacerlo para proveer una noción mas clara de lo que este hace, y esclarecer el módulo que se adjunta en el apéndice el cuál está escrito en `julia`.

El módulo no se encuentra documentado pero espero que el algoritmo hable por sí mismo. Se incluyen tres funciones para computar los colores del fractal, la única diferencia importante entre ellas es el color que da como resultado la imagen final.

El extracto de código que se muestra a continuación es una representación del modo de trabajo que se lleva en `julia` usando el REPL[1], asemeja un ambiente de trabajo y ejecución de comandos en la terminal.

```
julia> ns = 10:10:1600;                    # valores que puede tomar N
julia> reps = 20;                          # numero de repeticiones
julia> timings = zeros(length(ns), 2);     # arreglo bidimensional
julia> for rep in 1:reps
            for (index, n) in enumerate(ns)
                time = @elapsed mf.fractalCMap(n, n, maxiter=100)
                if rep == 1
                    timings[index, 1] = n
                end
                timings[index, 2] += time
            end
        end
julia> timings[:,2] = timings[:,2] / reps;  # promedio de tiempo
julia> timings = vcat([0 0], timings);      # agregar entrada extra para el tiempo cero
julia> nvalues = timings[:,1];              # lista de iteraciones
julia> time = timings[:,2];                 # lista de tiempos
```

Para graficar el tiempo de ejecución se hicieron 20 repeticiones indicadas por `reps`, y se definió una variable `ns` para guardar el conjunto de valores que puede tomar $N = 10,\ 20,\ 30,\ \ldots,\ 1600$. La variable `timings` guarda en la primera columna el valor de $N$, mientras que en la segunda columna guarda el tiempo $t(N)$ que le toma al algoritmo computar el fractal. Por cada iteración del loop se suman los tiempos $t(N)$ a sus respectivas entradas, y al final toda la columna de tiempos se divide entre la cantidad de repeticiones `reps` lo que resulta en tiempos promedio $\tilde{t}(N)$. Se usaron los tiempos promedio ya que procesos activos y en ejecución en los ordenadores pueden generar cambios en los tiempos de cómputo del fractal.

Los tiempos de ejecución en el ordenador `diannao` se pueden observar en la figura 4 donde las dos líneas continuas representan las cotas superiores, mientras que las líneas no continuas son las pertenecientes a las cotas inferiores. Se tuvo que utilizar un factor de escalamiento para los cuatro casos ya que los tiempos eran muy pequeños incluso para el caso en el que $N = 1600$ donde el tamaño del fractal corresponderia a una imagen de $1600 \times 1600$ pixeles con un número de iteraciones máximo de 100.

Para ambos casos, tanto para el cómputo con `diannao` y con `hongdiannao`, el comportamiento del tiempo $t$ respecto a $N$ se comporta de manera cuadrática. Si se mira el módulo incluído en el apéndice se podrá observar que el algoritmo incluye dos `for` loops principales en los que se accede a la informacion de una matriz inicialmente definida con pixeles al negro `RGB(0, 0, 0)` por lo que este es un comportamiento esperado. Se calculan las coordenadas

[1] REPL es un acrónimo para Read-Eval-Print loop.

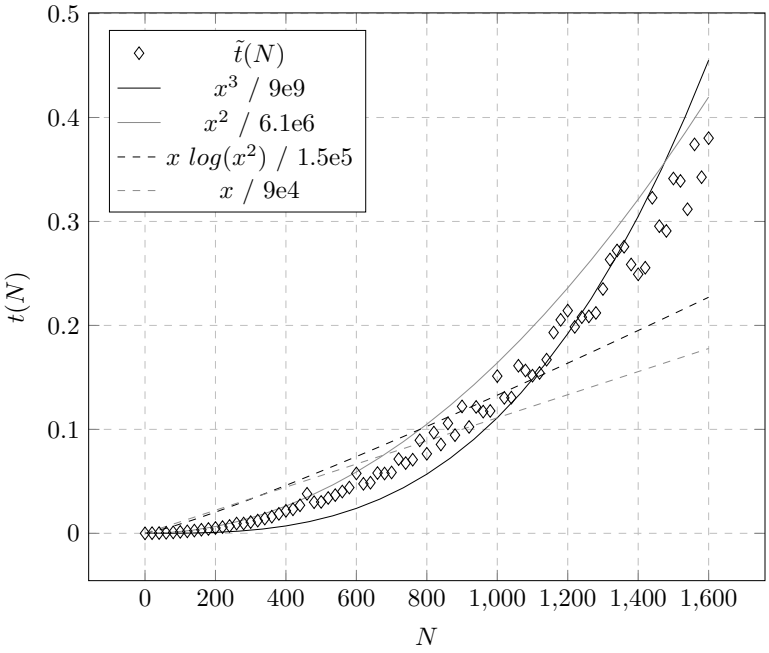



Figura 4: Tiempos de ejecución usando `diannao`.

anteriormente mencionadas $c = (x', y')$ dependiendo del índice del elemento al que se accede en la matriz, y para acceder a estos elementos se itera mediante dos `for` loops, uno para las columnas y otro para las filas.

De manera similar, los tiempos de ejecución usando `hongdiannao` se muestran en la figura 5. Claramente hay una diferencia notable en la magnitud de los tiempos medidos entre los dos ordenadores. Esto es evidenciable al mirar las especificaciones técnicas de ambos las cuales están incluidas más adelante. Las características de las computadoras

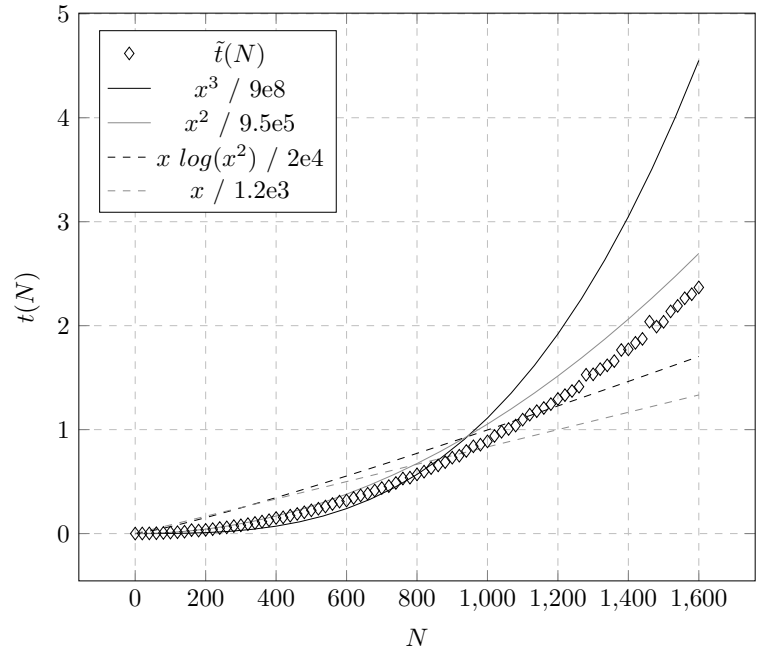



Figura 5: Tiempos de ejecución usando `hongdiannao`.

usadas, `diannao` y `hongdiannao`, se muestran en la figuras 6 y 7, respectivamente. Ambos son ordenadores propios, `hongdiannao` es una laptop adquirida en el año 2014 mientras que `diannao` es un equipo adquirido en el 2019. La diferencia en sus componentes, principalmente CPU's y medios de almacenamiento, es notoria. Mientras que `hongdiannao` tiene un disco duro que data del 2014, no se ha cambiado (y posiblemente se hayan deteriorado sus velocidades de lectura y escritura), `diannao` usa un SSD (solid state drive). El SSD de `diannao` y su CPU i7 le proporcionan mejor rapidez en comparación con el otro ordenador para esta tarea.

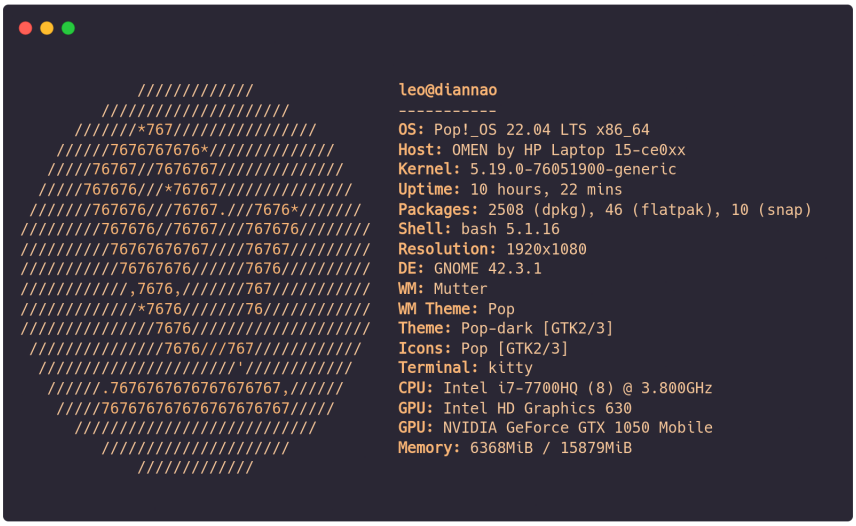



Figura 6: Características del ordenador `diannao`.

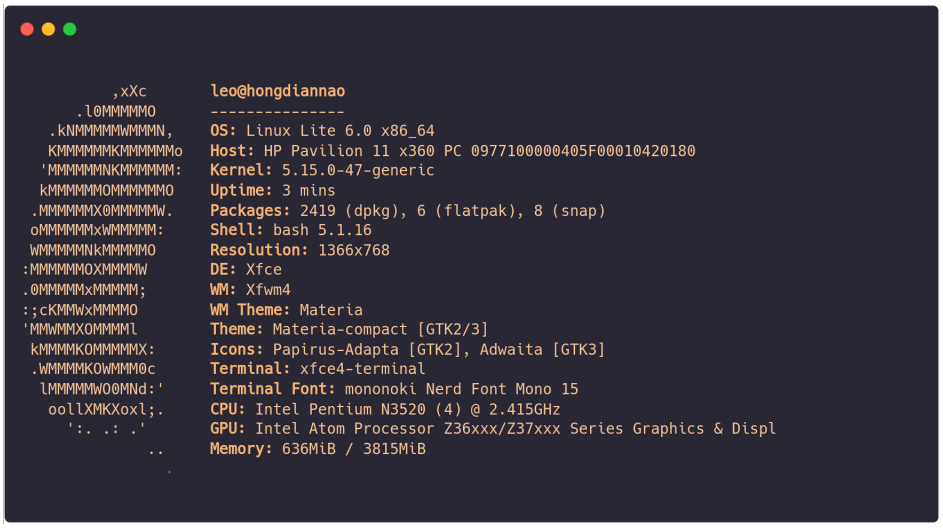
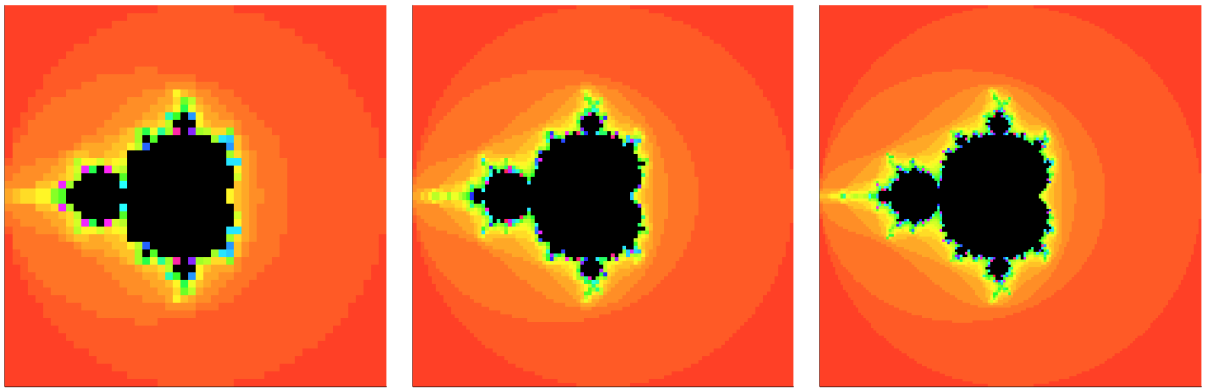
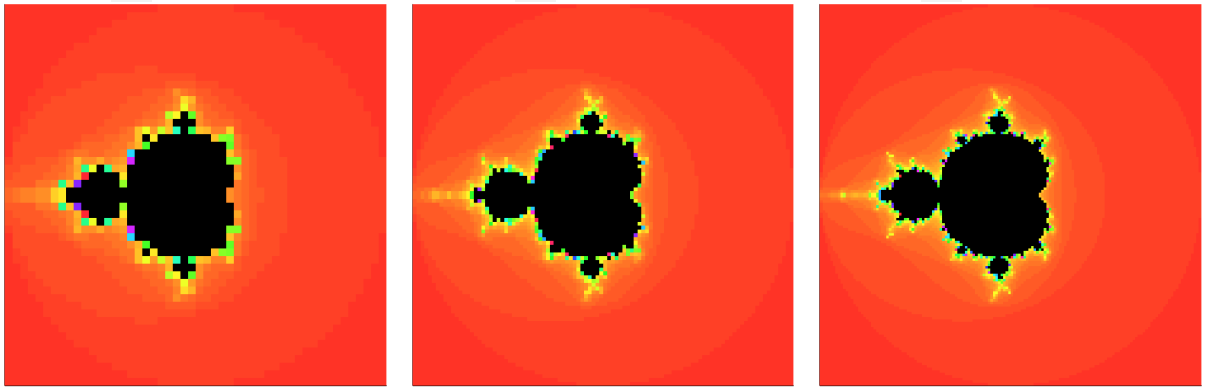



Figura 7: Características del ordenador `hongdiannao`.

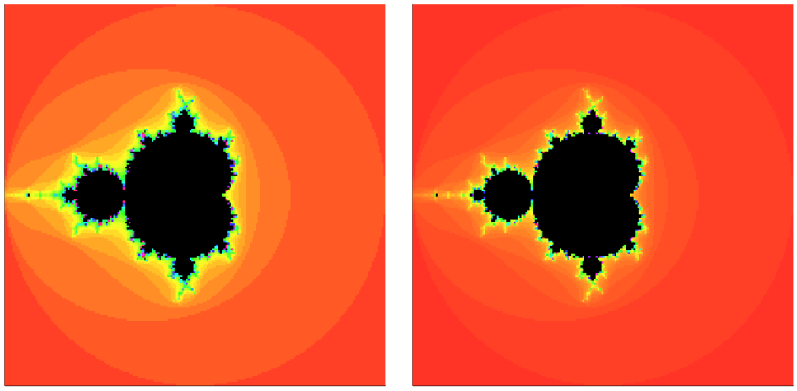
Por último quisiera mencionar que `julia` no es un lenguage para hacer scripts, similar al modo de trabajo usando `python`. Por ello fue que se adjunto al inicio de esta tarea una demostración sobre como se usó el módulo en el apéndice para computar los tiempos requeridos en ambas computadoras. Se anexaron como demostración imagenes de fractales al final del trabajo, mientras más alto es el número máximo de iteraciones se obtiene una mejor coloración, y aumenta la definición de la imagen mientras más fino es el mallado. Las zonas que asemejan nubes de color son zonas en donde el mismo número de iteraciones se obtuvo.



(a) `nx` = `ny` = 50; `maxiter` = 50. (b) `nx` = `ny` = 100; `maxiter` = 50. (c) `nx` = `ny` = 150; `maxiter` = 50.

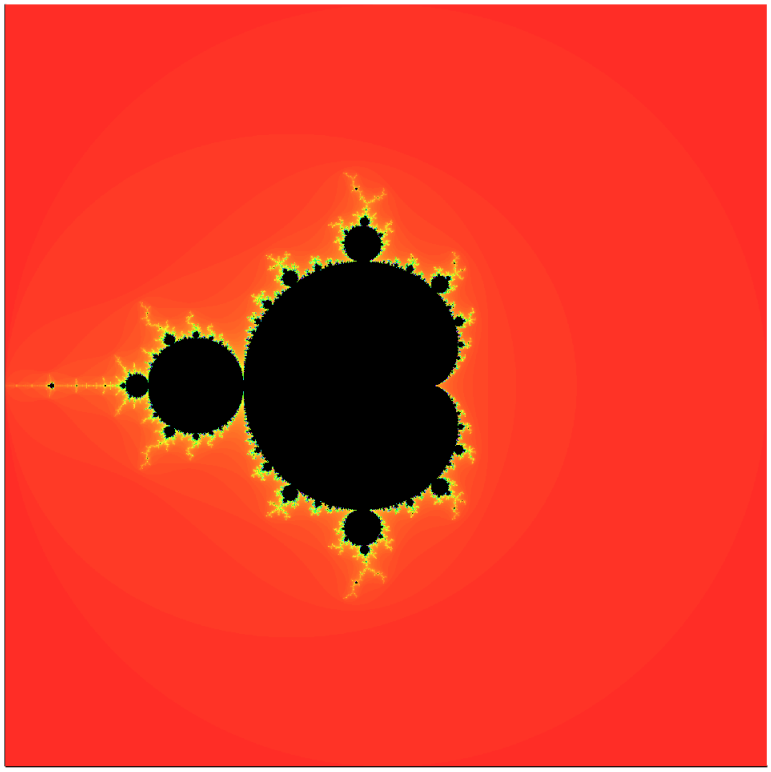


(d) `nx` = `ny` = 50; `maxiter` = 100. (e) `nx` = `ny` = 100; `maxiter` = 100. (f) `nx` = `ny` = 150; `maxiter` = 100.

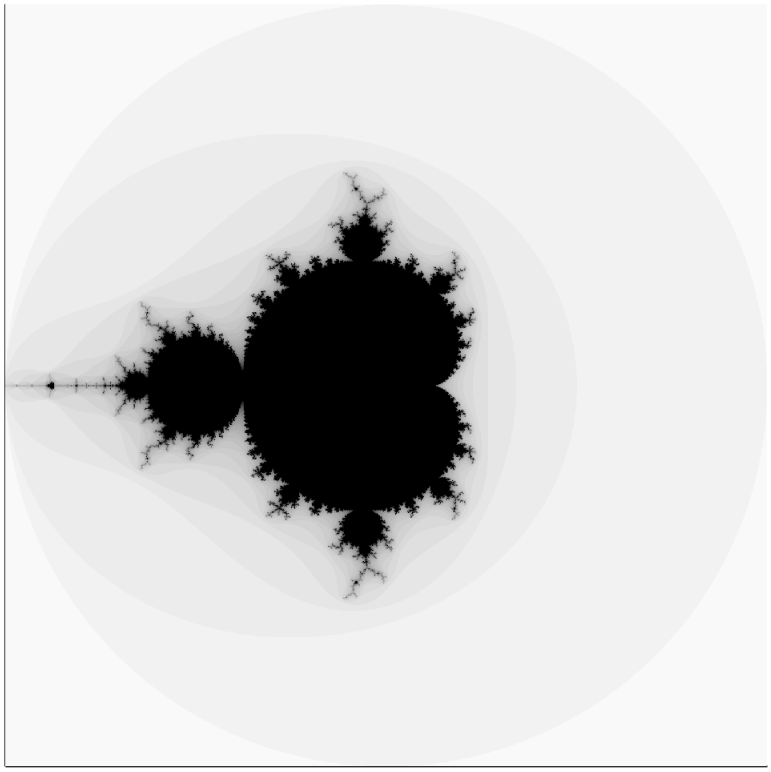


(g) `nx` = `ny` = 200; `maxiter` = 50. (h) `nx` = `ny` = 200; `maxiter` = 100.

Figura 8: Ejemplos de fractales que se pueden obtener usando el módulo desarrollado. Se anexan sus condiciones iniciales.



(a) `nx` = `ny` = 1600; `maxiter = 200`.



(b) `nx` = `ny` = 1600; `maxiter = 40`.

Figura 9: Fractales con mayor definición.

## Referencias

[1] Mandelbrot set. (20 Septiembre 2022). En Wikipedia. https://en.wikipedia.org/wiki/Mandelbrot_set

[2] Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. SIAM Review, 59(1), 65–98. https://doi.org/10.1137/141000671

[3] Vladimir, R. F. H. (21 Septiembre 2022). Análisis de Algoritmos. Universidad Veracruzna.

## Apéndice

```
17  module MandelbrotFractal
18
19  #==============================
20  Required libraries
21  ==============================#
22
23  using Images
24  using Plots
25
26  #==============================
27  Image operations
28  ==============================#
29
30  function imageSave(path, img)
31      save(path, img)
32  end
33
34  function visualize(rgbmap)
35      return plot(rgbmap, ticks=false)
36  end
37
38  #==============================
39  Fractal genereration
40  ==============================#
41
42  function mandelbrot(c, maxiter)
43      z = 0
44      n = 0
45      while abs(z) <= 2 && n < maxiter
46          z = z*z + c
47          n += 1
48      end
49      return n
50  end
51
52  function canvasRGB(height, width)
53      return zeros(RGB, height, width)
54  end
```

```
55
56  function canvasHSV(height, width)
57      return zeros(HSV, height, width)
58  end
59
60  xc(i, lx, nx) = lx * (2*i - 1) / nx
61
62  function fractalGrays(nx, ny; lx=2, ly=2, maxiter=80)
63      cv = canvasRGB(ny, nx)
64
65      for row in 1:ny
66          y = xc(row, ly, ny) - ly
67
68          for col in 1:nx
69              x = xc(col, lx, nx) - lx
70              c = x + y * 1im
71              m = mandelbrot(c, maxiter) / maxiter
72
73              pixel = 1 - m ▷ RGB
74              cv[row, col] = pixel
75          end
76      end
77      return cv
78  end
79
80  function fractalColors(nx, ny; lx=2, ly=2, maxiter=80)
81      cv = canvasHSV(ny, nx)
82
83      for row in 1:ny
84          y = xc(row, ly, ny) - ly
85
86          for col in 1:nx
87              x = xc(col, lx, nx) - lx
88              c = x + y * 1im
89              m = mandelbrot(c, maxiter) / maxiter
90
91              hue = 360 * m           # H between 0 and 360 (color wheel)
92              sat = 0.85              # S between 0 and 1 (saturation)
93              val = m < 1 ? 1 : 0     # V between 0 and 1 (brightness)
94              cv[row, col] = HSV(hue, sat, val)
95          end
96      end
97      return cv
98  end
99
100 function fractalCMap(nx, ny; lx=2, ly=2, maxiter=80, cname="Oranges")
101     cv = canvasRGB(ny, nx)
102
103     cmap_divs = 200
104     cmap = colormap(cname, cmap_divs, logscale=true) ▷ reverse
105
106     for row in 1:ny
107         y = xc(row, ly, ny) - ly
108
```

```
        for col in 1:nx
            x = xc(col, lx, nx) - lx
            c = x + y * 1im
            m = mandelbrot(c, maxiter) / maxiter

            # To select a color of the colormap
            cv[row, col] = cmap[ ceil(m * cmap_divs) ▷ Int ]
        end
    end
    return cv
end

end # module MandelbrotFractal
```