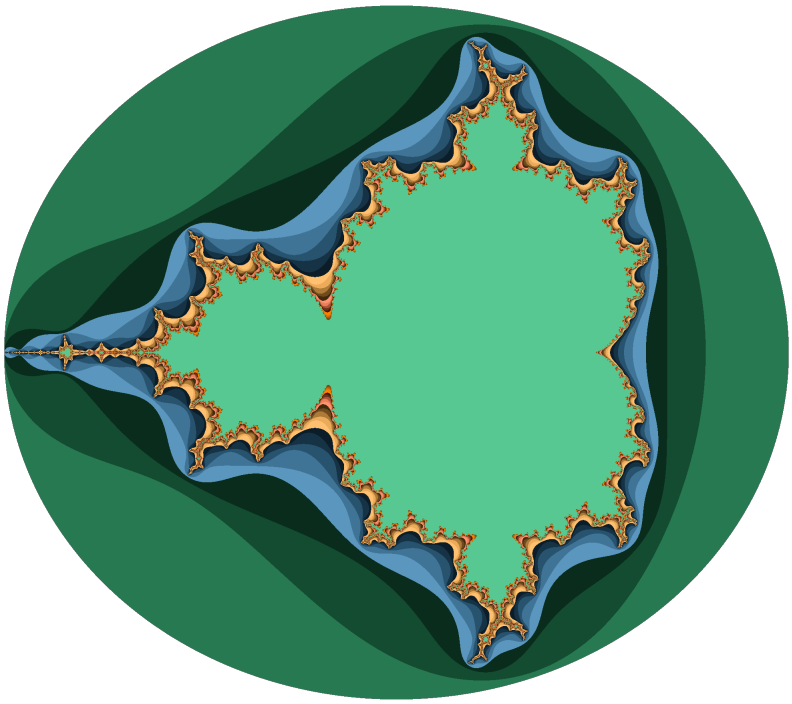
Graficación del Conjunto de Mandelbrot en Python

Andrs.ramos

March 2020



$$Z_{k+1} = Z_k^2 + C$$

Introducción

En el presente documento se aborda la graficación de fractales en python y se procede a dar algunos algoritmos base para la graficación del fractal del conjunto de Mandelbrot, así como algunas mejoras en los tiempos de ejecución y recomendaciones para hacer del código más funcional usando las librerías Matplotlib, Numpy, Numba y Pillow.

Conjunto de Mandelbrot

El conjunto mandelbrot es un conjunto de numeros complejos C que satisfacen la condición: $\lim_{n\to\infty} ||P_n(C)||$ no diverge.

Para entender más acerca de ésto, se define la siguiente ecuación recursiva.

$$Z_{k+1} = Z_k^2 + C; \quad Z_0 = 0; \quad C \in \mathbb{C}$$

Ahora, retomando la premisa de definir consistentemente el conjunto de Mandelbrot, se puede tomar la siguiente definición:

DEFINICIÓN.- Dado $C \in \mathbb{C}$, sea $P_n(C)$ la n-ésima iteración de la ecuación recursiva en el punto C, entonces si la norma de P_n se establece en una serie de valores finitos o en general, que permanezca acotada, entonces el punto C pertenece al conjunto de Mandelbrot

TEOREMA de acotamiento.- C pertenece al conjunto de Mandelbrot sí y sólo sí, se satisface que para toda para toda $n \geq 0$, $||P_n(C)|| \leq 2$

Algoritmos para graficación

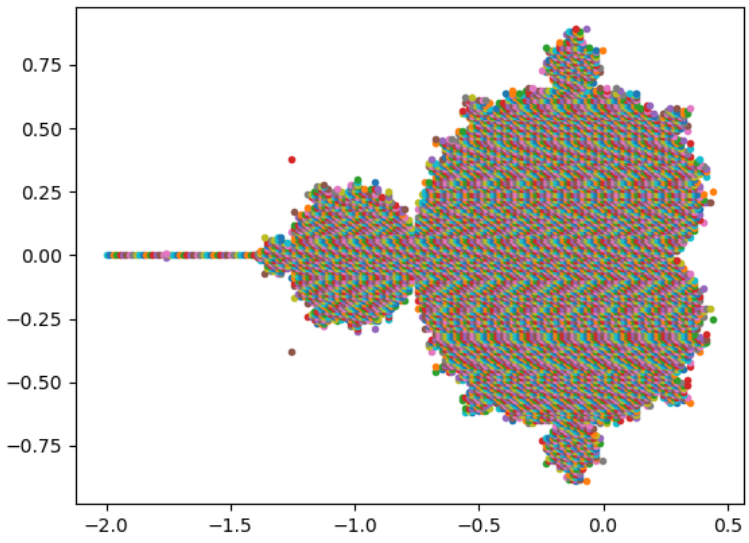
Nos enfocaremos en métodos y algoritmos que sean funcionales para la graficación a computadora del conjunto de mandelbrot.
Un algoritmo simple para graficar el fractal de mandelbrot es el siguiente:

Algoritmo I

1. Tomar un conjunto de puntos en el plano complejo, en un intervalo donde la norma de algunos de los puntos esté en la circunferencia crentrada en el origen de radio 2 (p ej $[-2, \frac{1}{2}]x[-i, i]$)
2. Iterar el punto sobre la ecuación recursiva un numero finito de veces (p ej. 20).
   A cada paso de la iteración calcular $||Z_n||$ y si es mayor que 2 para algún punto de la iteración dejar de iterar y asignar un 0 a dicho punto; en otro caso asignar a ese punto un 1.
3. Graficar dicho punto si su valor es 1
4. Tomar un nuevo punto en el intervalo (distinto al anterior) y aplicar los pasos 1 al 5 hasta que ya no haya más puntos

Para ejemplificar ésto se muestra la figura 1, que sigue del algorimo anterior

Figure 1: Algoritmo I: 120x120 puntos



Código I:

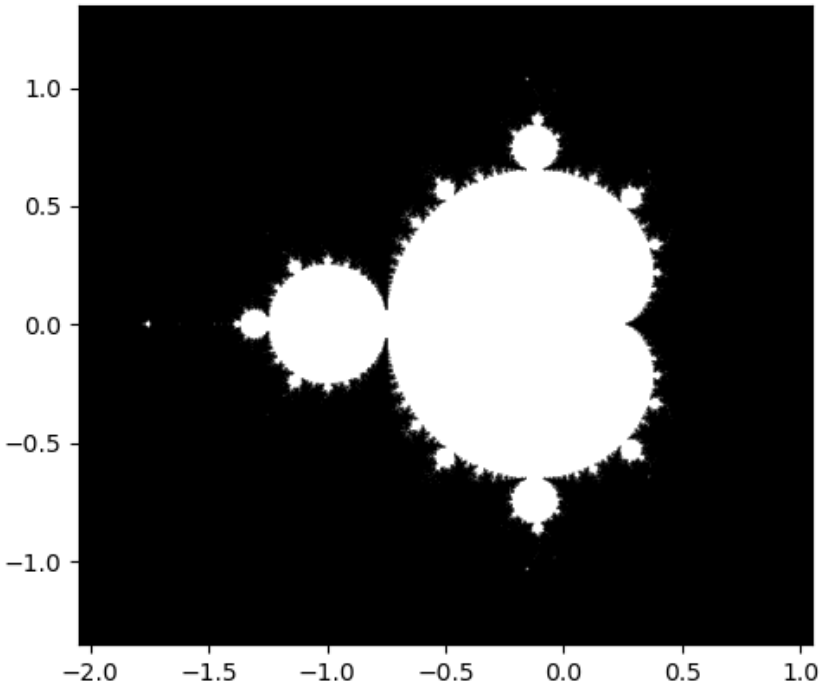
```
import matplotlib.pyplot as plt
def mandelbrot(c):
    z=c
    for i in range(20):
        z = z*z + c
        if abs(z)>2:
            return 0
    return 1
"---limites de la graficacion ---"
xlim=[-2,.7]
ylim=[-1,1]
"---Tamano de la imagen -----"
xdensity=100
ydensity=100


deltax=(xlim[1]-xlim[0])/xdensity
deltay=(ylim[1]-ylim[0])/ydensity

for i in range (ydensity):
    y0=ylim[0]+deltay*(i)
    for j in range(xdensity):
        x0=xlim[0]+deltax*(j)
        if mandelbrot(x0+y0*1j)==1:
            plt.plot(x0,y0,'.')
plt.show()
```

NOTAS DEL CÓDIGO I:
∗Puede usarse el comando linspace de la librería numpy para hacerse más fácil de manipular.
∗Grafica los puntos en varios colores, ya que no tiene un color especificado.
∗En python se puede poner el numero complejo haciendo $a + b * 1j$.

Realmente éste no parece una grafica competente del fractal, sino apenas una visualizacion punto por punto, no podemos incrementar demasiado la resolucion, por que el tiempo de ejecucion se eleva demasiado, ya que graficar punto por punto es una tarea que requiere bastantes recursos.
Ahora otra forma más optimizada de usar este algoritmo es con imshow de la libreria matplotlib, que requiere una matriz donde las cordenadas de los pixeles coinciden con los elementos de la matriz, es decir pasarle a las computadora los la posicion del pixel con su respectivo valor 1 o 0 y no graficar punto por punto, sino mandar los pixeles e imshow asigna un color negro al 0 y un color blanco al 1

En la figura 2 podemos ver a más detalle el fractal del conjunto de mandelbrot

Figure 2: Algoritmo I: 800x800 puntos



Código II:

```
import numpy as np
import matplotlib.pyplot as plt

def mandelbrot(c):
    z=c
    for i in range(50):
        z = z*z + c
        if abs(z)>2:
            return 0
    return 1

#inicio=t.time()
"------limites de graficacion y  densidad de pixeles ------"
x=np.linspace(-2,1,2000)
y=np.linspace(-1,1,2000)

"-----graficacion -----"
sx=np.size(x)
sy=np.size(y)
a=np.empty((sy,sx))
for i in range(sy):
    for j in range(sx):
        a[i,j]=mandelbrot(x[j]+y[i]*1j)
p=plt.imshow(a,cmap='Greys',extent=[-2,1,-1,1])
plt.show()
```

Bajo la misma idea, es momento de graficar sobre una paletta de color el fractal del conjunto de Mandelbrot, pero primero una breve explicación de como introducir una paleta de color a un gráfico. Primero debemos tener una paleta de color, que no es mas que tener conjunto de colores por código, ya sea RGB, RGBA o CMYK u otro, aqui unicamente usaremos RGB y RGBA.
RGB es un código para los colores que viene dado en ternas, así el primer elemento es la cantidad de rojo, el segundo la cantidad de verde, y el tercero una cantidad de azul en una escala de 0 a 255, una imagen 24-bits png es una imagen RGB donde cada pixel puede tener una gamma de 2^{24} colores. Para el RGBA los colores vienen dados en cuartetos de datos donde los tres primeros son iguales a los del RGB pero se añade la transparencia de color al ultimo elemento

del cuarteto, también en una escala de 0 a 255, donde 0 corresponde al transparente y 255 al totalmente opaco.

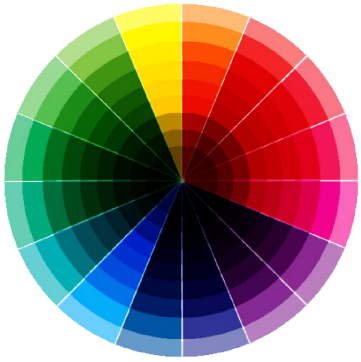
Ahora el problema es saber como escoger nuestra paleta de color y como aplicarla a nuestro gráfico, para eso tenemos el siguiente algoritmo:

Algoritmo II

1. Seleccionar una paleta de color, ya sea alguna paleta por defecto (p ej. "Viridis" en python) o personalizarla aqui sugiero hacer lo siguiente:

   (a) Tomar una paleta básica de unos 4 o 5 colores

   (b) Degradar el color hacia un tono mas oscuro otras 4 o 5 veces, (moverse de afuera hacia dentro del circulo de color) en total tendremos unos 16-25 colores en la paleta.



   $*$ la razón de escoger un color mas oscuro cada vez es que parezca una sombra en la curva de nivel de la imagen.

2. Tomar un conjunto de puntos en el plano complejo, en un intervalo donde la norma de algunos de los puntos esté en la circunferencia crentrada en el origen de radio 2 (p ej $[-2, \frac{1}{2}]x[-i, i]$)

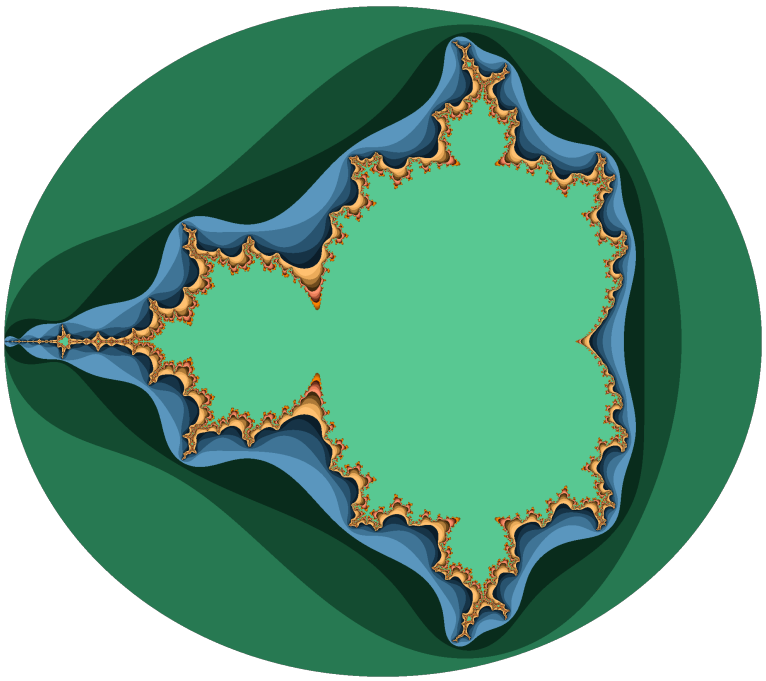
3. Iterar el punto sobre la ecuación recursiva según colores tenga en la paletta de color.
   A cada paso de la iteración calcular $||Z_i||$ y si es mayor que 2 para algún punto de la iteración dejar de iterar y asignar el valor de i a dicho punto; en otro caso asignar a ese punto un 0.

4. Graficar dicho pixel con el color correspondiente en la escala de la paleta de color (p ej.- asignar al 1 el primer color en la paleta de color, el 2 al segundo y así sucesivamente)

5. Tomar un nuevo punto en el intervalo (distinto al anterior) y aplicar los pasos 1 al 5 hasta que ya no haya más puntos

Ahora como resultado del algortmo II podemos ver la siguiente figura:

Figure 3: Mandelbrot 1960x2080



Código III:

```
from PIL import Image    #pkg pillow para graficar
import time as t         #time para medir el tiempo de ejecucion
from numba import jit    #pkg numba para reducir el tiempo de ejecuci
#paleta colores
colores=[
(88, 200, 146 ),(63, 164, 115),(39, 121, 82 ),
(20, 76, 49  ),(9, 44, 28  ),
(90, 150, 190 ),(63, 116, 150),(40, 83, 111 ),
(22, 51, 70  ),(11, 29, 41 ),
(255, 192, 112),(235, 171, 90),(174, 122, 56),
(110, 74, 29 ),(64, 42, 14 ),
(255, 149, 112),(235, 127, 90),(174, 86, 56 ),
(110, 50, 29 ),(64, 26, 14 ),
(255, 132, 0  ),(243, 127, 3 ),(203, 109, 9 ),
(158, 87, 12 ),(124, 69, 10)]

@jit     #jit permite una pseudo compilacion de las funciones
def mandelbrot(c):
    z=0
    for i in range(maxIt):
        z = z*z + c
        if abs(z)>2:
            return i
    return 0
# limites de graficacion y densidad de pixeles
xlim = (-2,1)
ylim = (-1.5,1.5)
sx = 1960
sy = 2080

maxIt = len(colores)  # iteraciones maximas permitidas


inicio=t.time()
image = Image.new("RGBA", (sx, sy))
for y in range(sy):
    zy = y * (ylim[1] - ylim[0]) / (sy - 1) + ylim[0]
    for x in range(sx):
```

```
            zx = x * (xlim[1] - xlim[0]) / (sx - 1) + xlim[0]
            if ((zx+0.5)/1.5)**2 + (zy/1.25)**2 <=1:
            #si esta dentro de la elipse grafica el punto
                i=mandelbrot(zx + zy * 1j)
                image.putpixel((x, y), colores[i])
            else:
            #si esta fuera de la elipse que sea transparente
                image.putpixel((x, y), (0,0,0,0))
fin=t.time()
image.show()
```

NOTAS.
∗Nos apoyamos del comando Image de la librería Pillow, primero iniciamos la imagen con un numero pixeles, luego le decimos que le asigne un color de la paleta de colores a cada pixel
∗Se escoge el sistema de color RGBA para que la imagen no tenga bordes, mas que los del fractal y se vea mejor, ya que si fuese sólo RGB tendría un cuadro blanco alrededor
∗La paleta de color la podemos ingresar como RGB e image le asigna un valor de transparencia 255 por defecto
∗Se puede hacer una modificación a este código muy sencilla para crear un gif animado muy bonito y vistoso.

```
imagenes=[]
for k in range(len(colores)):
    img = Image.new("RGBA", (sx, sy),(255,255,255,255))
    for y in range(sy):
        zy = y * (ylim[1] - ylim[0]) / (sy - 1) + ylim[0]
        for x in range(sx):
            zx = x * (xlim[1] - xlim[0]) / (sx - 1) + xlim[0]
            z = zx + zy * 1j
            i=mandelbrot(z,k+1)
            img.putpixel((x, y), colores[i])
            if zx**2 + zy**2 >4:
                img.putpixel((x,y), (255,255,255,255))
    imagenes.append(img)
imagenes[0].save('mandelbrotgif.gif',save_all=True,
                 append_images=imagenes[1:],
                 optimize=False, duration=180, loop=0)
```

Tenemos otra opción que es usar una vez más Imshow de la

librería matplotlib, ahora como en el algoritmo I aplicamos la librería, la aplicamos seleccionando una paleta de color de matplotlib, como lo podemos buscar en su pagina de documentación de la librería. Aun que ésta opción no es mi favorita, es muy buena para ir probando paletas de colores de python e ir jugando con los parametros Imshow y ver como afecta a la imagen.
Jugando con los parámetros podemos tener imagenes como estas:

Figure 4: Mandelbrot 1080x1080, "viridis", norm = powerNorm(.3)

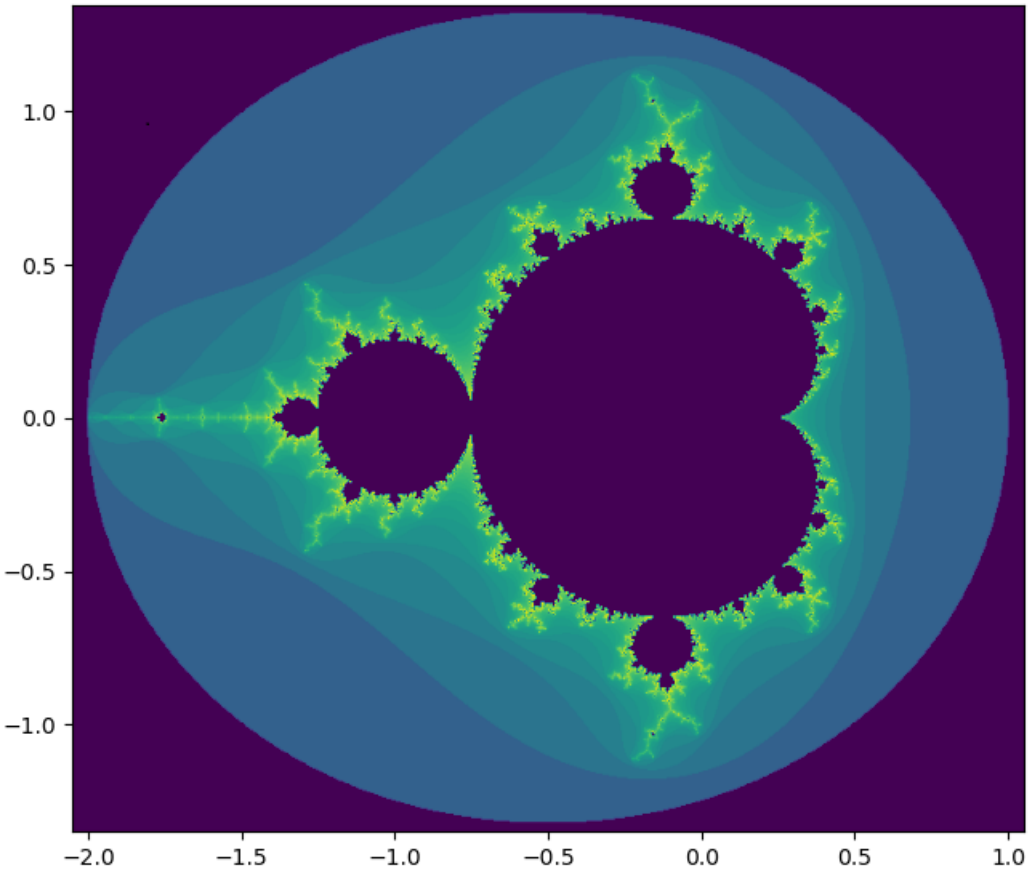


Figure 5: Mandelbrot 1080x1080, "viridis", norm=defalut

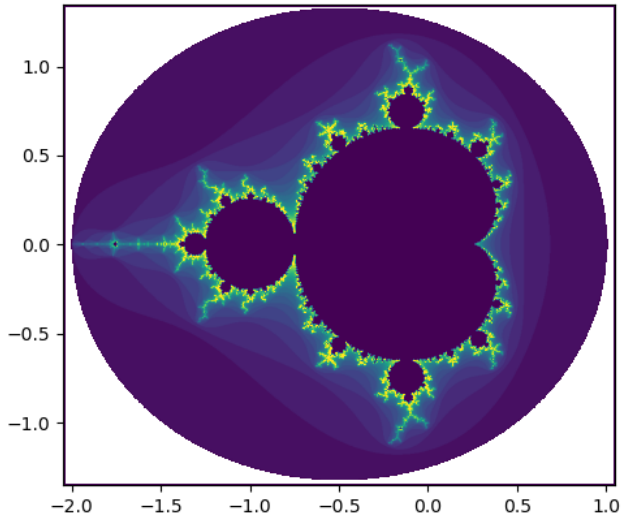
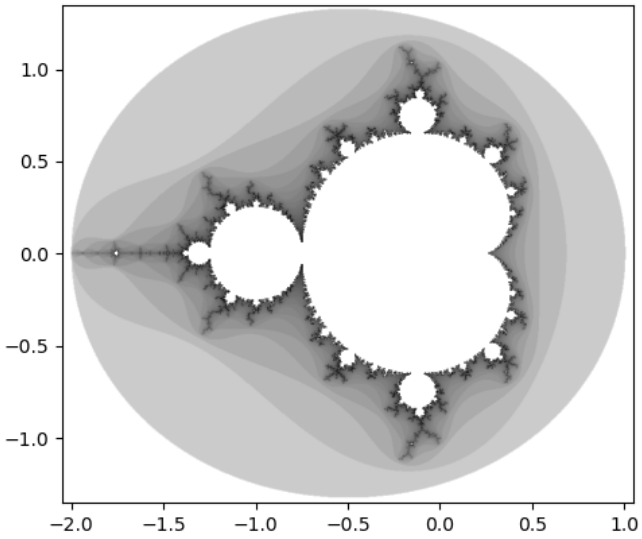


Figure 6: Mandelbrot 1080x1080, "Greys", norm=defalut



El código del que preceden dichas imagenes no es más que una modificación al Código II.
Código IV:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
import time as t
from numba import jit

@jit
def mandelbrot(c):
    z=c
    for i in range(50):
        z = z*z + c
        if abs(z)>2:
            return 0
    return 1

inicio=t.time()
"-----limites de graficacion y  densidad de pixeles -----"
x=np.linspace(-2.05,1.05,2000)
y=np.linspace(-1.35,1.35,2000)

"-----graficacion -----"
sx=np.size(x)
sy=np.size(y)
a=np.empty((sy,sx))
for i in range(sy):
    for j in range(sx):
        a[i,j]=mandelbrot(x[j]+y[i]*1j)
norm = colors.PowerNorm(0.3)
p=plt.imshow(a,cmap='Greys',
norm=norm,extent=[-2.05,1.05,-1.35,1.35])
plt.show()
plt.savefig("mandelbrotfractal.png")
final=t.time()
print("tiempo : ",final-inicio)
```

Python no es caracterizado por ser el lenguaje más rápido, o con el mejor sistema de graficación, sin embargo tiene muchas caracter-

isticas que lo hacen un lenguaje muy potente. Si queremos mejorar la rapidez de ejecución del código podemos apoyarnos de JIT de la librería Numba, que mediante una pseudo- compilación puede mejorar la el tiempo en el que se ejecuta una función, aun que JIT está un poco restringido a que funciones se puede usar, podemos usarlo sin pproblemas a la funcioón que más se ejecuta en todos los códigos vistos la funcion $mandelbrot(c)$, que se ejecuta una vez por pixel y ejecuta la ecuación recursiva de 1 a 20,25 veces o mas.