Documentación de la experiencia (elaboración de fractales)

Carlos Manuel Carvajales Castrillo - 1001824814

En el proyecto, fueron elaborados en Python tres tipos de fractales:

- Fractales de Newton.
- Conjuntos de Julia.
- Sistemas iterados de funciones.

Además, se incluyeron dos fractales de Newton en 3 dimensiones.

Para estos fractales se utilizaron una serie de códigos generadores que, al insertar alguna función, mostraban fractales de distintas formas. Los colores se cambiaron a gusto aumentando o disminuyendo el número por el cual se estaba multiplicando la "*i*" de las iteraciones máximas para el modelo de color RGB en el caso de los fractales de Newton y los Conjuntos de Julia.

Fractales de Newton:

Para elaborar los fractales de Newton, se usó el método de Newton para números Reales, y el método de Newton para números complejos:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$ Para números reales.

$$x_{n+1} = x_n - a\left(\frac{f(x_n)}{f'(x_n)}\right)$$ Para números complejos

donde a es cualquier número complejo distinto de cero.

Primeramente, se procedió a importar los paquetes matplotlib.pyplot y PIL como factores principales para poder observar el fractal, y los paquetes numpy y sympy como factores secundarios en caso de que se usaran funciones propias de esos paquetes.

```
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np
import sympy as spp
```

Luego pasamos a definir la cantidad de pixeles o el "tamaño" de la imagen con imgx e imgy y definimos el modelo de color RGB para imgx e imgy. Posteriormente, se definió el rango para el cual se quiere ver el fractal, es decir, si se quiere ver más cerca o más lejos, teniendo en cuenta la distribución del plano cartesiano; luego, se colocó un valor de iteraciones máximo maxit; h como una herramienta que se usará en la derivada de la función para codificar el método de Newton y para ahorrar trabajo de procesamiento; y por último un épsilon eps que, en el método de Newton representa el "máximo error absoluto", de esta forma, cuando el valor de alguna iteración esté por debajo de este épsilon, se deja de iterar.

```
imgx=800
imgy=800
image=Image.new("RGB",(imgx,imgy))
xa=-5
xb=5
ya=-5
yb=5
maxit=202
h=1e-6
eps=1e-3
```

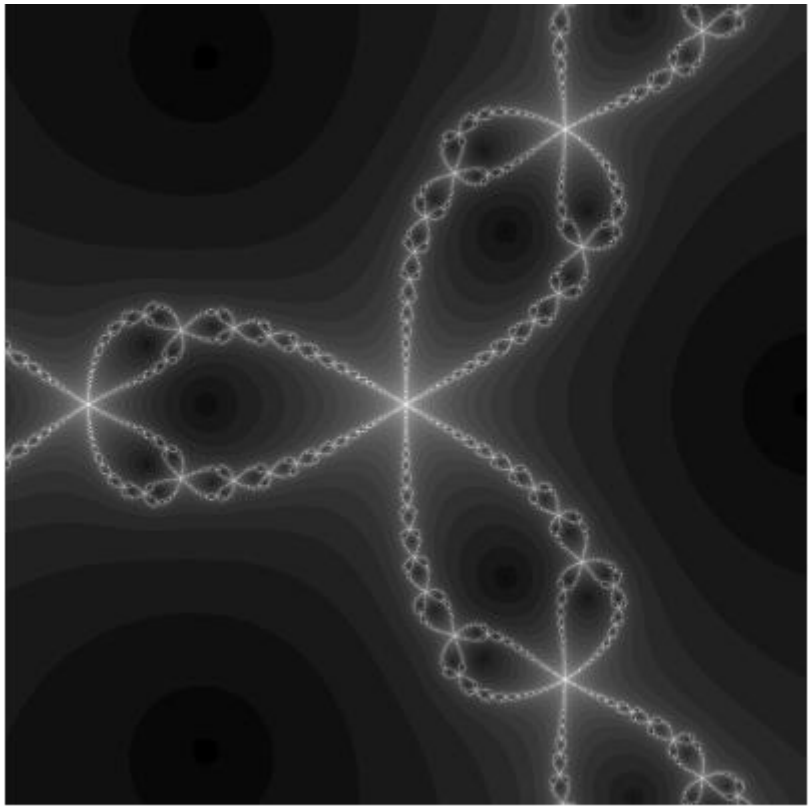
Ahora, se definió la función a ver en fractal de Newton:

```
def f(z):
    return z**3-1
```

Después se codificó el Método de Newton, y se codificó adecuadamente la distribución de los pixeles con su respectivo rango para visualizar el fractal.

```
for y in range (imgy):
    zy=y*(yb-ya)/(imgy-1)+ya
    for x in range (imgx):
        zx=x*(xb-xa)/(imgx-1)+xa
        z=complex(zx,zy)
        for i in range (maxit):
            dz=(f(z+complex(h,h))-f(z))/complex(h,h)
            z0=z-f(z)/dz
            if abs (z0-z)<eps:
                break
            z=z0
            r=i*8
            g=i*8
            b=i*8
            image.putpixel((x,y),(r,g,b))
```

Una vez hecho esto, sólo se escribió “image” en una celda aparte para obtener el siguiente fractal:



Note que, para este fractal, se usó el método de Newton para números reales. Sin embargo, para la elaboración de otros fractales se usó el método de Newton para números complejos. En concreto se utilizó:

$$a = \frac{1}{2} + \frac{1}{4}i$$

Y un formato de color para el modelo RGB de:

$$r = i * 1$$

$$g = i * 12$$

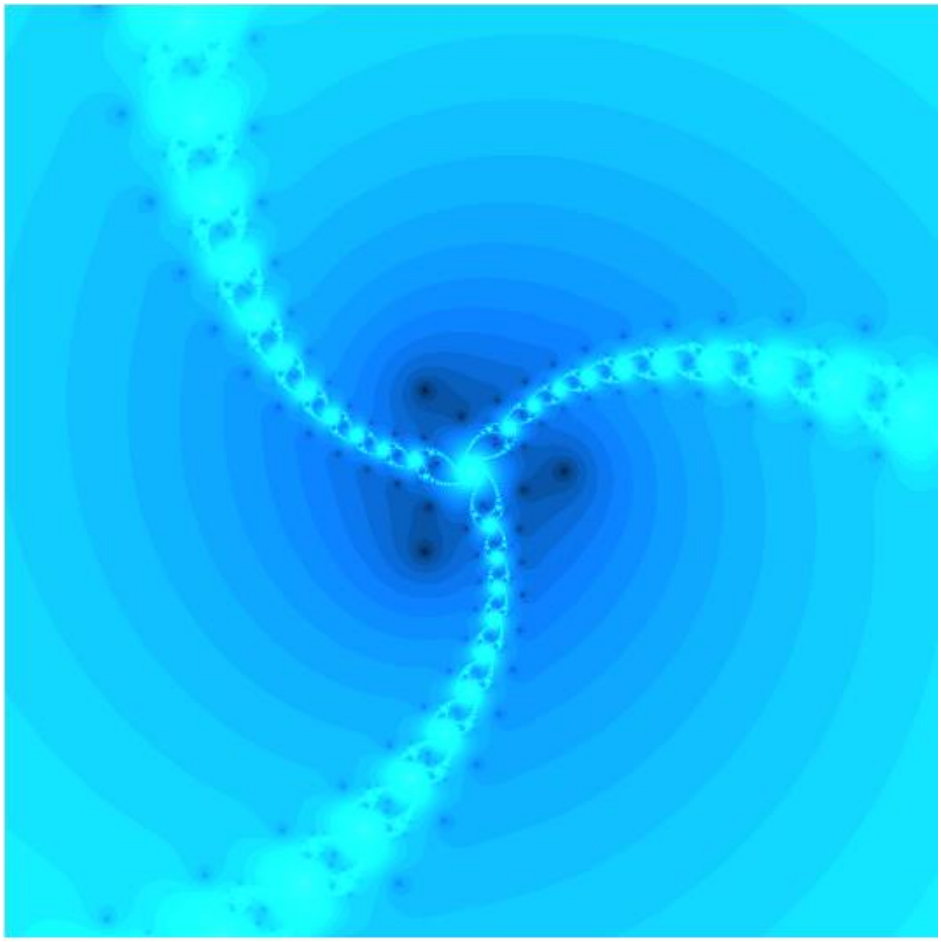
$$b = i * 24$$

Así:

```
for y in range (imgy):
    zy=y*(yb-ya)/(imgy-1)+ya
    for x in range (imgx):
        zx=x*(xb-xa)/(imgx-1)+xa
        z=complex(zx,zy)
        for i in range (maxit):
            dz=(f(z+complex(h,h))-f(z))/complex(h,h)
            z0=z-(complex((1/2),(1/4))*(f(z)/dz))
            if abs (z0-z)<eps:
                break
            z=z0
            r=i*1
            g=i*12
            b=i*24
            image.putpixel((x,y),(r,g,b))
image
```

Como resultado, tenemos el siguiente fractal de Newton:

Primer fractal de Newton:

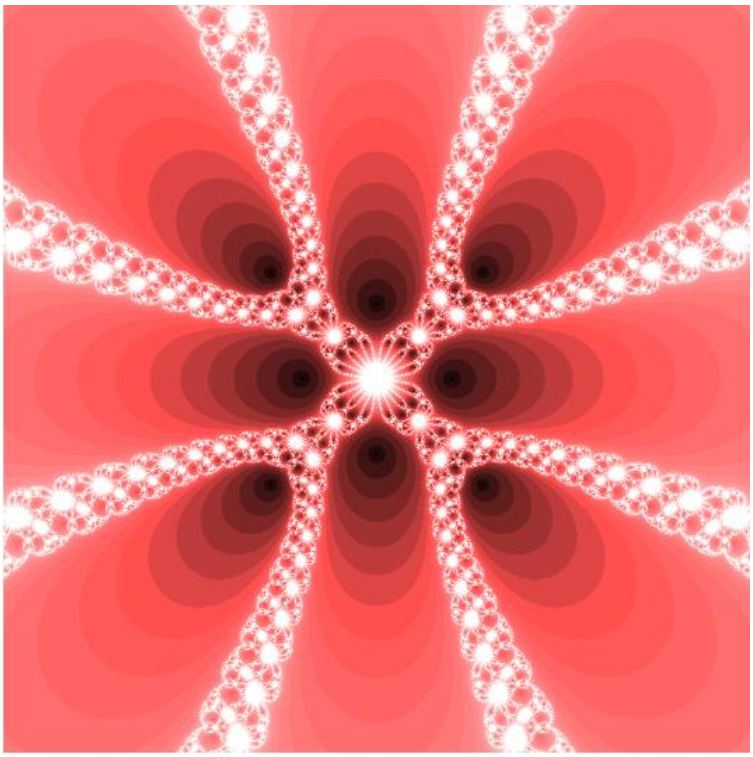


Este mismo concepto fue aplicado para los otros fractales realizados:

Segundo fractal de Newton:

```
xa=-5
xb=5
ya=-5
yb=5
maxit=202
h=1e-6
eps=1e-3

imgx=800
imgy=800
image=Image.new("RGB",(imgx,imgy))

def f(z):
    return z**8+15*z**4-16

for y in range (imgy):
    zy=y*(yb-ya)/(imgy-1)+ya
    for x in range (imgx):
        zx=x*(xb-xa)/(imgx-1)+xa
        z=complex(zx,zy)
        for i in range (maxit):
            dz=(f(z+complex(h,h))-f(z))/complex(h,h)
            z0=z-f(z)/dz
            if abs (z0-z)<eps:
                break
            z=z0
            r=i*32
            g=i*7
            b=i*7
            image.putpixel((x,y),(r,g,b))
image
```

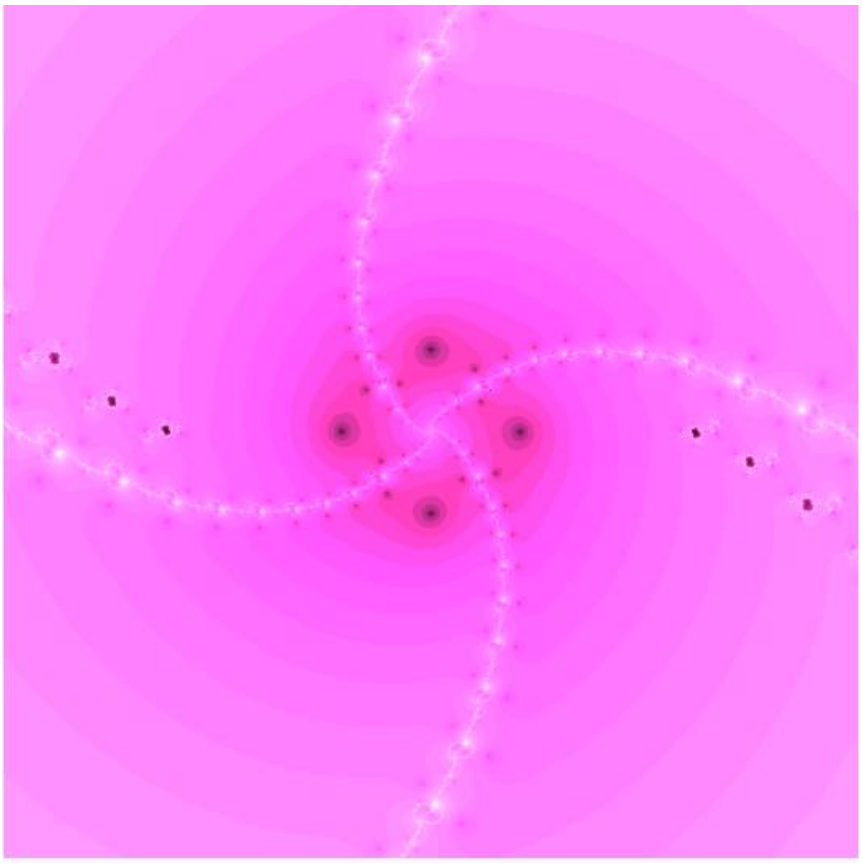
Fractal resultante:



Tercer fractal de Newton:

```
xa=-5
xb=5
ya=-5
yb=5
maxit=100
h=1e-6
eps=1e-3

imgx=800
imgy=800
image=Image.new("RGB",(imgx,imgy))

def f(z):
    return z**3/np.sin(z)

for y in range (imgy):
    zy=y*(yb-ya)/(imgy-1)+ya
    for x in range (imgx):
        zx=x*(xb-xa)/(imgx-1)+xa
        z=complex(zx,zy)
        for i in range (maxit):
            dz=(f(z+complex(h,h))-f(z))/complex(h,h)
            z0=z-(complex((1/2),(1/4))*(f(z)/dz))
            if abs (z0-z)<eps:
                break
            z=z0
            r=i*32
            g=i*8
            b=i*24
            image.putpixel((x,y),(r,g,b))
image
```

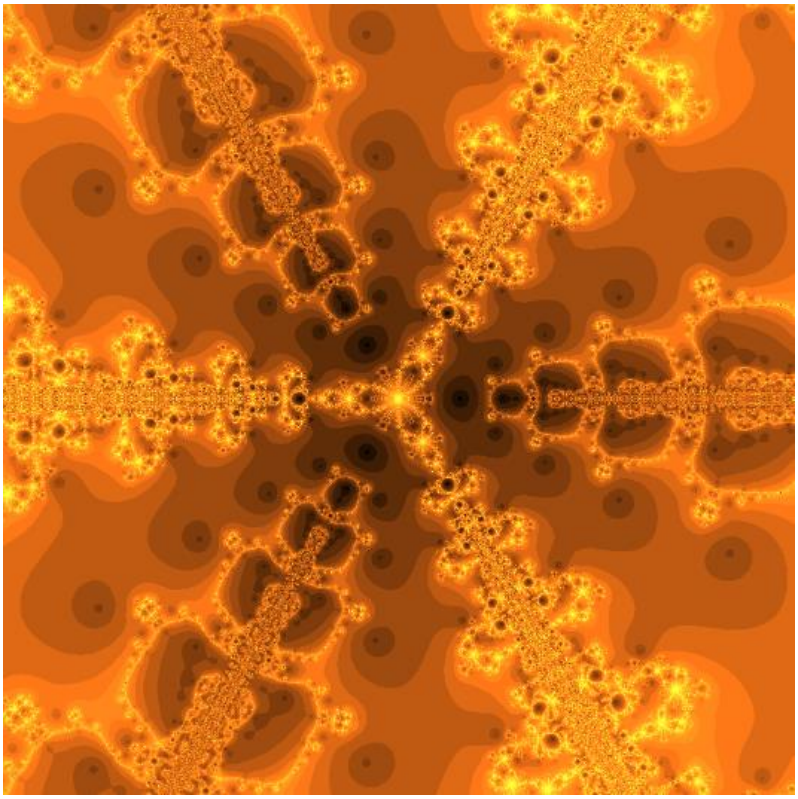
Fractal resultante:



Cuarto fractal de Newton:

```
xa=-5
xb=5
ya=-5
yb=5
maxit=20
h=1e-6
eps=1e-3

imgx=800
imgy=800
image=Image.new("RGB",(imgx,imgy))

def f(z):
    return np.tan(z**3)-1+z**3

for y in range (imgy):
    zy=y*(yb-ya)/(imgy-1)+ya
    for x in range (imgx):
        zx=x*(xb-xa)/(imgx-1)+xa
        z=complex(zx,zy)
        for i in range (maxit):
            dz=(f(z+complex(h,h))-f(z))/complex(h,h)
            z0=z-f(z)/dz
            if abs (z0-z)<eps:
                break
            z=z0
            r=i*32
            g=i*15
            b=i*3
            image.putpixel((x,y),(r,g,b))
image
```

Fractal resultante:

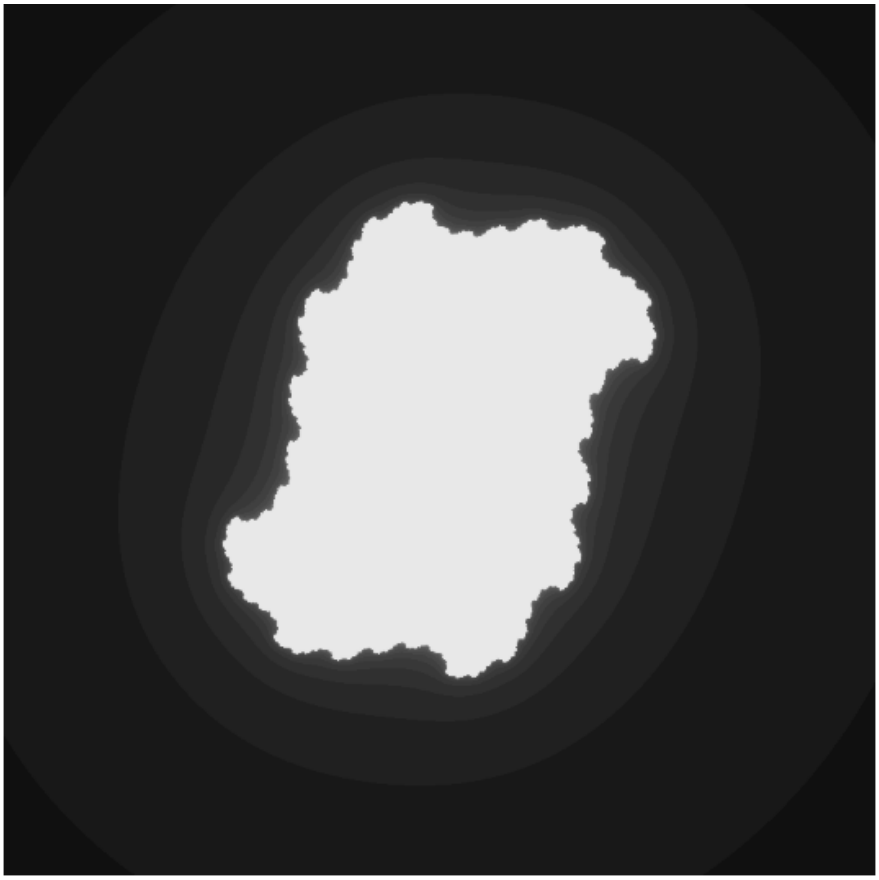


Conjuntos de Julia:

Para los conjuntos de Julia, se usaron los mismos paquetes que para la elaboración de los fractales de Newton, sin embargo, éste se caracteriza por la suma de un número complejo c a la función en cuestión:

```
xa=-2
xb=2
ya=-2
yb=2
maxit=30
def f(z):
    return z**2+complex(0.2,0.3)

for y in range (imgy):
    zy=y*(yb-ya)/(imgy-1)+ya
    for x in range (imgx):
        zx=x*(xb-xa)/(imgx-1)+xa
        z=complex(zx,zy)
        for i in range (maxit):
            z0=f(z)
            if abs(z)>1000:
                break
            z=z0
            r=i*8
            g=i*8
            b=i*8
            image.putpixel((x,y),(r,g,b))
```

Fractal resultante:

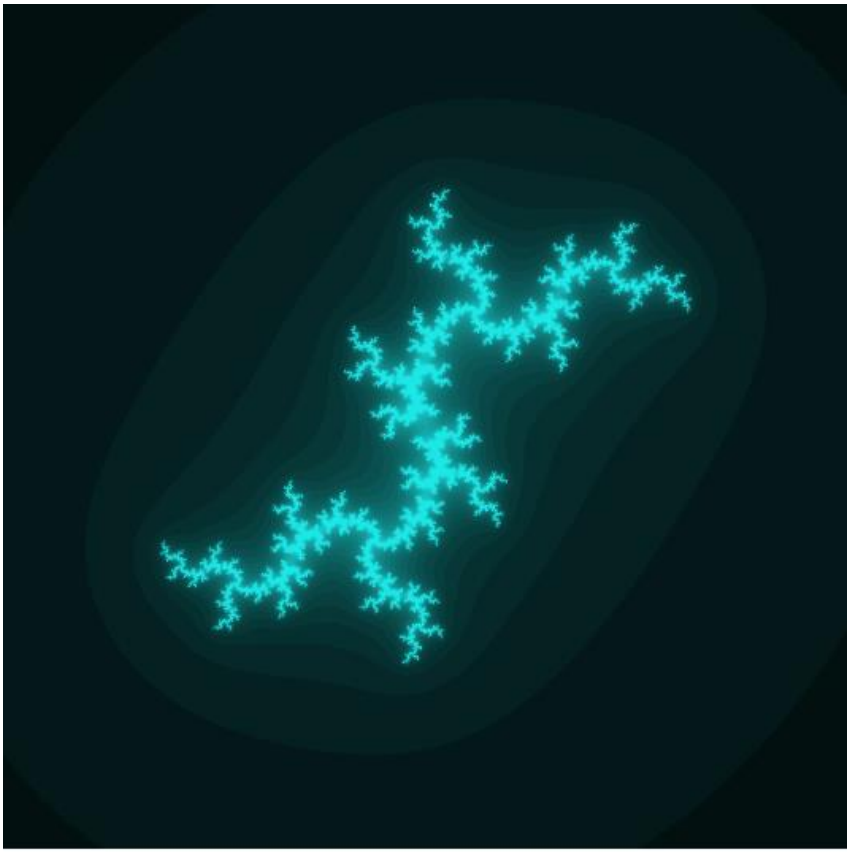


Así, los fractales generados se muestran a continuación:

Conjunto de Julia 1:

```
xa=-2
xb=2
ya=-2
yb=2
maxit=30

imgx=800
imgy=800
image=Image.new("RGB",(imgx,imgy))

def f(z):
    return z**2+complex(0,0.8)

for y in range (imgy):
    zy=y*(yb-ya)/(imgy-1)+ya
    for x in range (imgx):
        zx=x*(xb-xa)/(imgx-1)+xa
        z=complex(zx,zy)
        for i in range (maxit):
            z0=f(z)
            if abs(z)>1000:
                break
            z=z0
            r=i*1
            g=i*8
            b=i*8
            image.putpixel((x,y),(r,g,b))
image
```

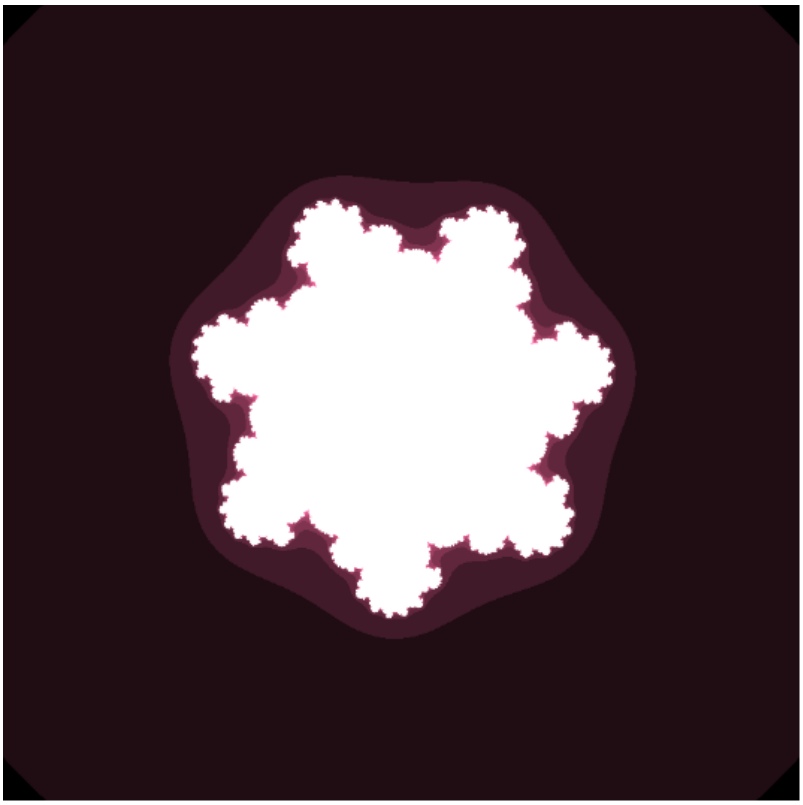
Fractal generado:



Conjunto de Julia 2:

```
xa=-2
xb=2
ya=-2
yb=2
maxit=30

imgx=800
imgy=800
image=Image.new("RGB",(imgx,imgy))

def f(z):
    return z**7-1+complex(np.cos(0.67),np.sqrt(0.5))

for y in range (imgy):
    zy=y*(yb-ya)/(imgy-1)+ya
    for x in range (imgx):
        zx=x*(xb-xa)/(imgx-1)+xa
        z=complex(zx,zy)
        for i in range (maxit):
            z0=f(z)
            if abs(z)>1000:
                break
            z=z0
            r=i*32
            g=i*13
            b=i*20
            image.putpixel((x,y),(r,g,b))
image
```

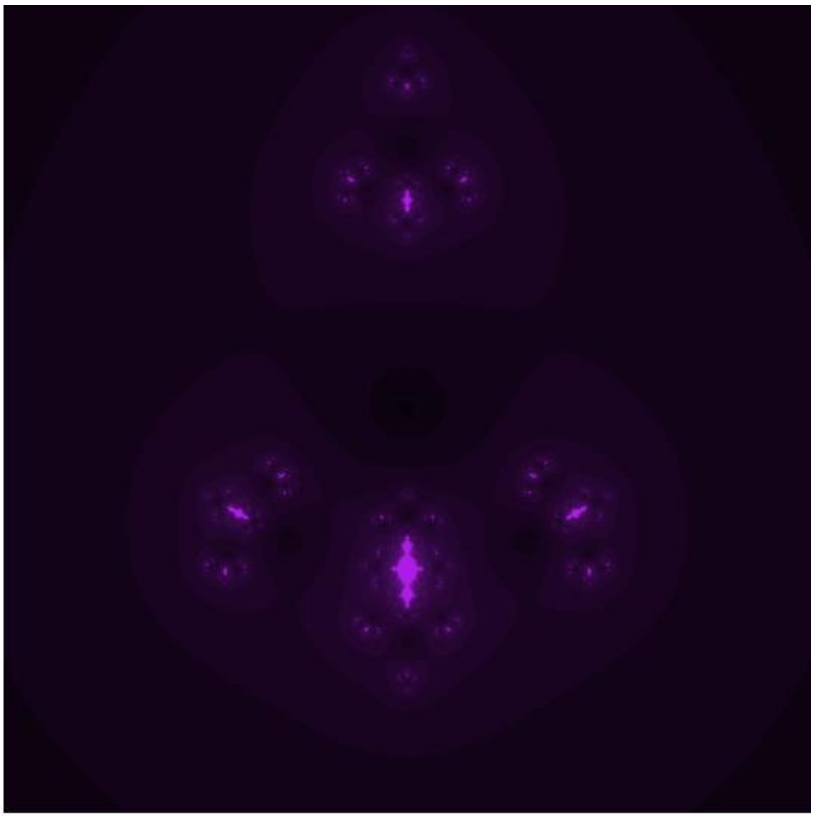
Fractal generado:



Conjunto de Julia 3:

```
xa=-2
xb=2
ya=-2
yb=2
maxit=30

imgx=800
imgy=800
image=Image.new("RGB",(imgx,imgy))

def f(z):
    return complex(1/z,z**2)+complex(0,0.9)

for y in range (imgy):
    zy=y*(yb-ya)/(imgy-1)+ya
    for x in range (imgx):
        zx=x*(xb-xa)/(imgx-1)+xa
        z=complex(zx,zy)
        for i in range (maxit):
            z0=f(z)
            if abs(z)>1000:
                break
            z=z0
            r=i*6
            g=i*1
            b=i*8
            image.putpixel((x,y),(r,g,b))
image
```

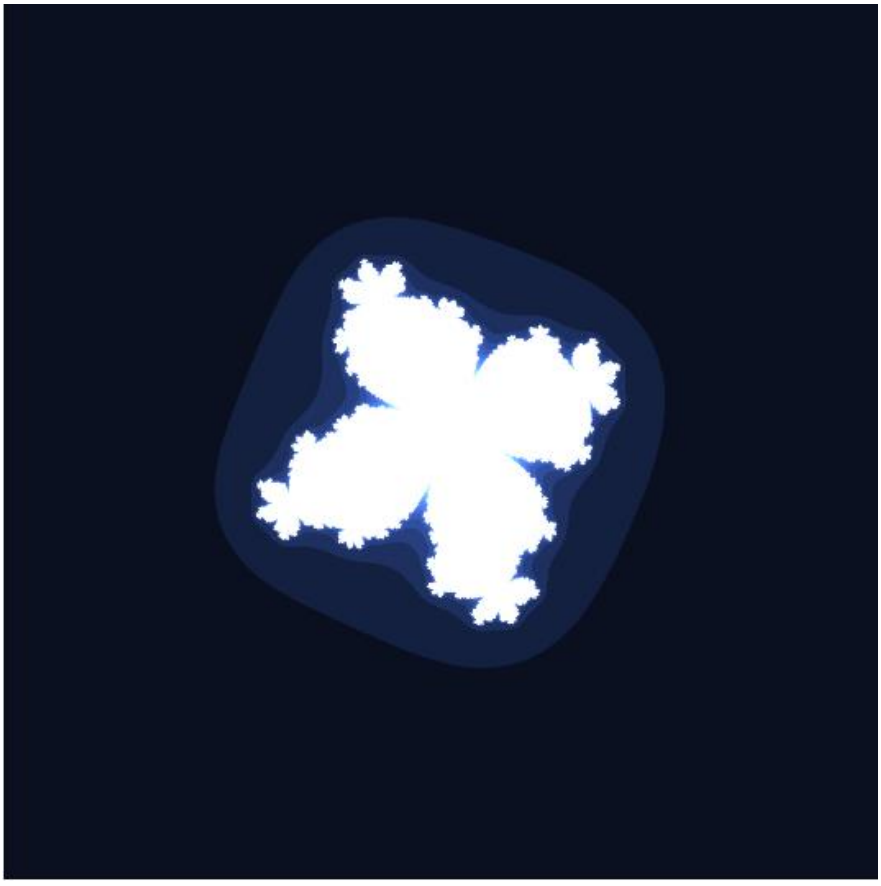
Fractal generado:



Conjunto de Julia 4:

```
xa=-2
xb=2
ya=-2
yb=2
maxit=30

imgx=800
imgy=800
image=Image.new("RGB",(imgx,imgy))

def f(z):
    return complex(3*z**5,z**3/z**2-1)+complex(0,0.9)

for y in range (imgy):
    zy=y*(yb-ya)/(imgy-1)+ya
    for x in range (imgx):
        zx=x*(xb-xa)/(imgx-1)+xa
        z=complex(zx,zy)
        for i in range (maxit):
            z0=f(z)
            if abs(z)>1000:
                break
            z=z0
            r=i*10
            g=i*16
            b=i*32
            image.putpixel((x,y),(r,g,b))
image
```

Fractal generado:

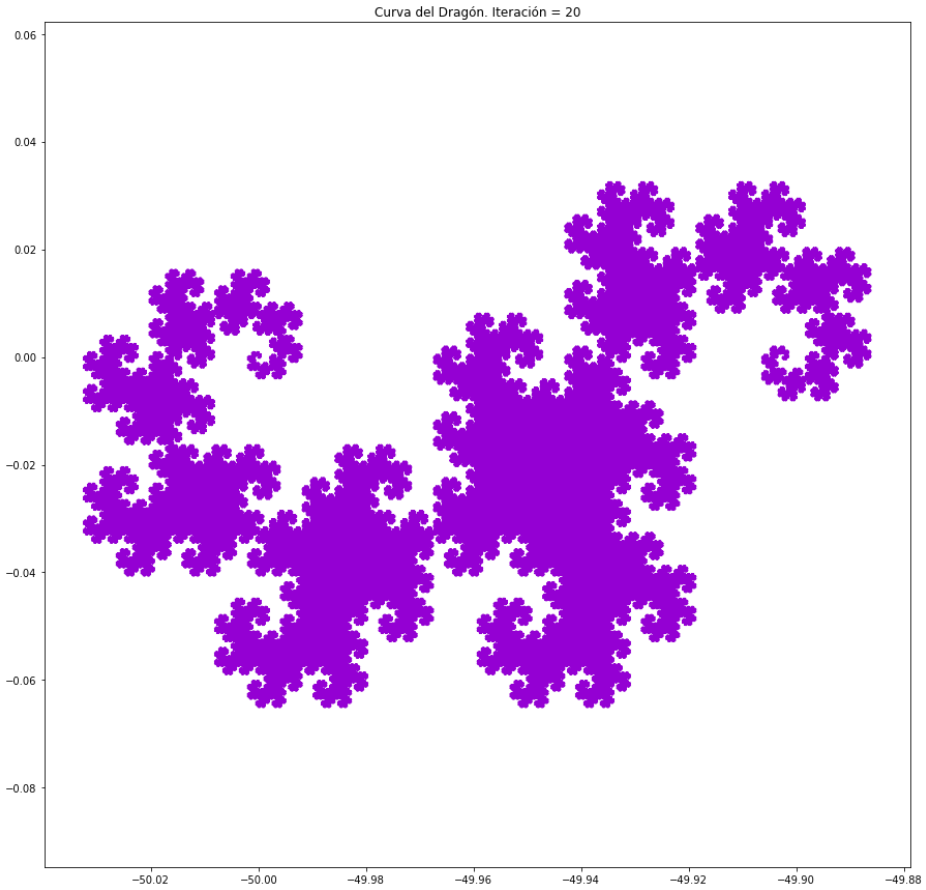


Sistemas Iterados de Funciones

Los sistemas de funciones iteradas son conjuntos de n transformaciones afines contractivas. Normalmente se utilizan dos tipos de algoritmos, el algoritmo determinista y el algoritmo aleatorio.

Los fractales generados por un algoritmo determinista se muestran a continuación:

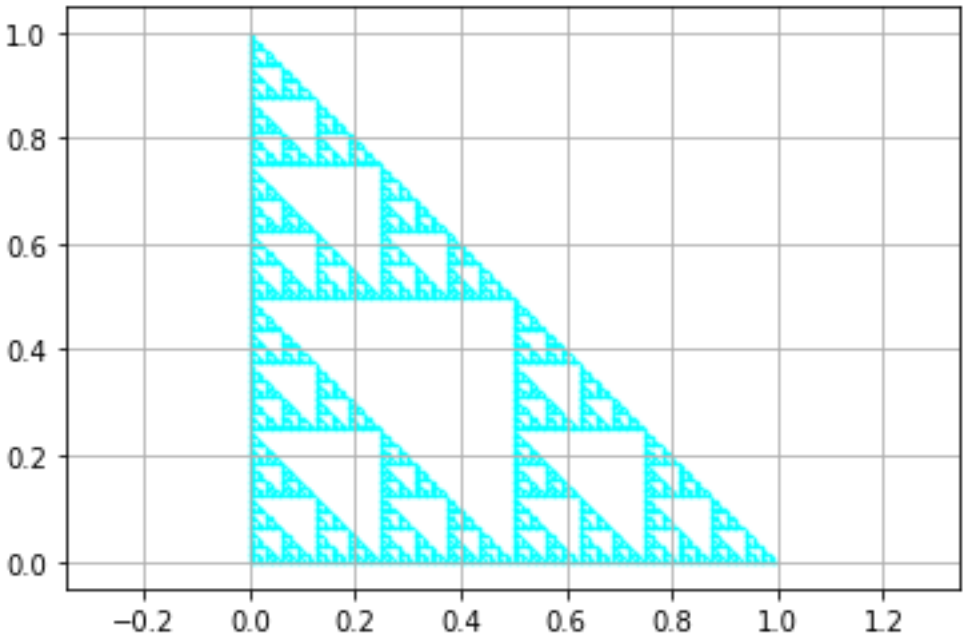
Curva del dragón:



El código utilizado para la construcción de este fractal se muestra a continuación:

```
import matplotlib.pyplot as plt
import argparse
import math

def Dragon(level, initial_state, trgt, rplcmnt, trgt2, rplcmnt2):
    state = initial_state

    for counter in range(level):
        state2 = ''
        for character in state:
            if character == trgt:
                state2 += rplcmnt
            elif character == trgt2:
                state2 += rplcmnt2
            else:
                state2 += character
        state = state2
    return state

totalwidth=100
iterations = 20

plt.figure(figsize=(15,15))

points = dragon(iterations,totalwidth,(-totalwidth/2,0))
plt.plot([p[0] for p in points], [p[1] for p in points], '-',color='darkviolet')
plt.axis('equal')

plt.title("Curva del Dragón. Iteración = 20")

plt.show()
```

Triángulo de Sierpinski:

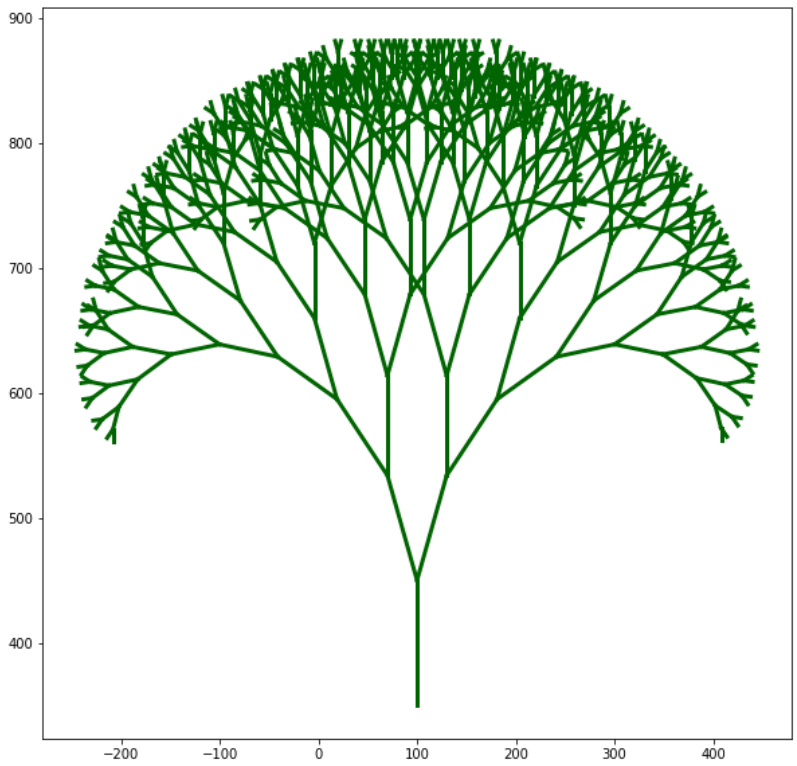


El código utilizado para la construcción de este fractal se muestra a continuación:

```
fig=plt.figure()
ax=plt.gca()
Tri=np.array([[0,0],[1,0],[0,1],[0,0]])

def transafin(M,t,x):
    y=M@x+t
    return y

transafin([[0.5,0],[0,0.5]],[0,0],Tri[1])

Tri=np.array([[0,0],[1,0],[0,1],[0,0]])
tritrans=np.array([transafin([[0.5,0],[0,0.5]],[0,0],i) for i in Tri])
tritrans2=np.array([transafin([[0.5,0],[0,0.5]],[0,0.5],i) for i in Tri])
tritrans3=np.array([transafin([[0.5,0],[0,0.5]],[0.5,0],i) for i in Tri])

Tri=np.concatenate((tritrans,tritrans2,tritrans3))

Tri=np.array([[0,0]])
for i in range(8):
    tritrans=np.array([transafin([[0.5,0],[0,0.5]],[0,0],i) for i in Tri])
    tritrans2=np.array([transafin([[0.5,0],[0,0.5]],[0,0.5],i) for i in Tri])
    tritrans3=np.array([transafin([[0.5,0],[0,0.5]],[0.5,0],i) for i in Tri])
    Tri=np.concatenate((tritrans,tritrans2,tritrans3))
plt.scatter(Tri.transpose()[0],Tri.transpose()[1],color='aqua',s=0.2)
ax.set_xticks(np.arange(-0.2,1.4,0.2))
ax.set_yticks(np.arange(-0.2,1.4,0.2))
plt.grid()
ax.axis("equal")
```

Los fractales generados por un algoritmo aleatorio se muestran a continuación:

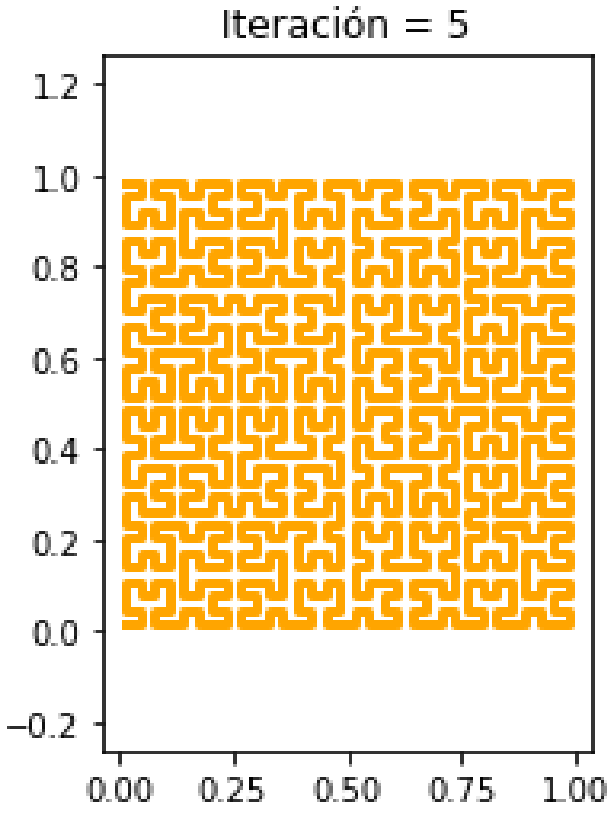
Árbol binario:



El código utilizado para la construcción de este fractal se muestra a continuación:

```
import matplotlib.pyplot as plt
import math


def dibujaTree(x1, y1, angle, depth):

    if depth:
        x2 = x1 + int(math.cos(math.radians(angle)) * depth * 10.0)
        y2 = y1 + int(math.sin(math.radians(angle)) * depth * 10.0)
        plt.plot([x1,x2],[y1,y2],'-',color='darkgreen',lw=3)
        dibujaTree(x2, y2, angle - 20, depth - 1)
        dibujaTree(x2, y2, angle + 20, depth - 1)


plt.figure(figsize=(10,10))
depth = 10
dibujaTree(100, 350, 90, depth)

plt.show()
```

Curva de Hilbert:



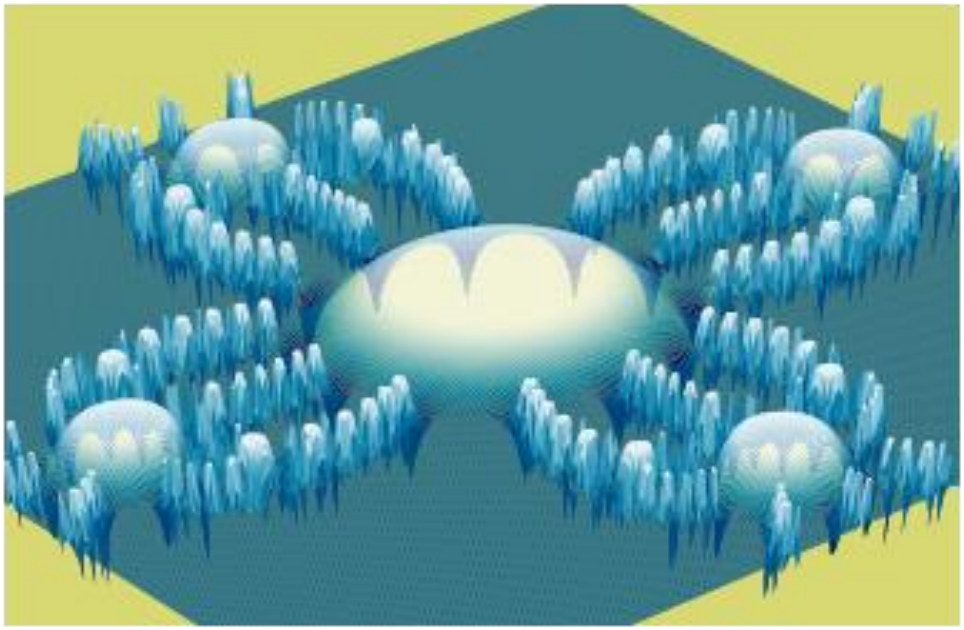
El código utilizado para la construcción de este fractal se muestra a continuación:

```
import sys, math
import numpy as np
import matplotlib.pyplot as plt

def hilbert(x0, y0, xi, xj, yi, yj, n,points):
    if n <= 0:
        X = x0 + (xi + yi)/2
        Y = y0 + (xj + yj)/2
        points.append((X,Y))
    else:
        hilbert(x0,               y0,               yi/2, yj/2, xi/2, xj/2, n - 1,points)
        hilbert(x0 + xi/2,        y0 + xj/2,        xi/2, xj/2, yi/2, yj/2, n - 1,points)
        hilbert(x0 + xi/2 + yi/2, y0 + xj/2 + yj/2, xi/2, xj/2, yi/2, yj/2, n - 1,points)
        hilbert(x0 + xi/2 + yi,   y0 + xj/2 + yj,  -yi/2,-yj/2,-xi/2,-xj/2, n - 1,points)
        return points


a = np.array([0, 0])
b = np.array([1, 0])
c = np.array([1, 1])
d = np.array([0, 1])

iterations = 5

plt.subplot(1,2,2).set_title("Iteración = 5")

points = hilbert(0.0, 0.0, 1.0, 0.0, 0.0, 1.0, iterations,[])
plt.plot([p[0] for p in points], [p[1] for p in points], '-',lw=3,color='orange')

plt.axis('equal')
```

Fractales 3D:

Los fractales elegidos para elaborar en 3D, fueron los fractales de Newton. Estos se muestran a continuación:

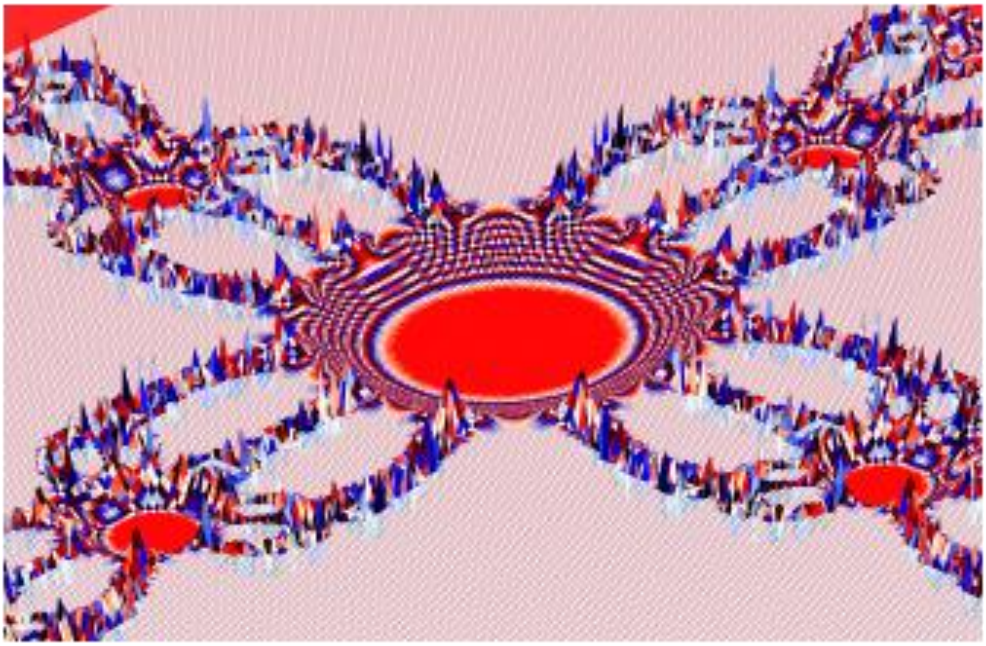
Primer fractal en 3D:



El código utilizado y los pasos realizados se definen de la siguiente forma:

```
import matplotlib.pyplot as plt # importar librería matplotlib
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm # Módulo de importación de mapas de color
import numpy as np # Importar librería numpy

fig = plt.figure() # Establecer entorno de figura 3D
ax = fig.add_subplot(111, projection='3d')
ax.view_init(azim=-130,elev=45) # establecer la orientación de la vista
ax.dist = 4.3 # establecer la distancia del punto de vista
ax.set_facecolor([.85,.85,.45]) # establecer color de fondo

n = 8 # establecer número de ciclos
dx = 0.0 # establecer el cambio inicial de parámetros en x
dy = 0.0 # establecer el cambio inicial de parámetros en y
L = 1.0 # establecer lado del área cuadrada
M = 300 # establecer el número lateral de píxeles

def f(Z): # definir la amortiguación de escala de la función de profundidad
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M) # matriz de la variable x
y = np.linspace(-L+dy,L+dy,M) # matriz de la variable y
X,Y = np.meshgrid(x,y) # cuadrícula de área cuadrada
Z = X + 1j*Y # área compleja del plano

for k in range(1,n+1): # ciclo de recursividad
    ZZ = Z - (Z**4 + 1)/(4*Z**3)
    Z = ZZ
    W = f(Z)

ax.set_xlim(dx-L,dx+L) # establecer límites de eje x
ax.set_zlim(dy-L,dy+L) # establecer límites de eje y
ax.set_zlim(-2.5*L,2*L) # establecer límites de eje z
ax.axis("off") # no mostrar ejes
ax.plot_surface(X, Y, -W, rstride=1, cstride=1, cmap="ocean") # hacer trazado de contorno
plt.show() # Mostrar figura
```

Segundo fractal en 3D:



El código utilizado y los pasos realizados se definen de la siguiente forma:

```
import matplotlib.pyplot as plt # importar librería matplotlib
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm # Módulo de importación de mapas de color
import numpy as np # Importar librería numpy

fig = plt.figure() # Establecer entorno de figura 3D
ax = fig.add_subplot(111, projection='3d')
ax.view_init(azim=-140,elev=45) # establecer la orientación de la vista
ax.dist = 3.0 # establecer la distancia del punto de vista
ax.set_facecolor([1.0,.15,.15]) # establecer color de fondo

n = 8 # establecer número de ciclos
dx = 0.0 # establecer el cambio inicial de parámetros en x
dy = 0.0 # establecer el cambio inicial de parámetros en y
L = 1.3 # establecer lado del área cuadrada
M = 300 # establecer el número lateral de píxeles

def f(Z):  # definir la amortiguación de escala de la función de elevación
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M) # matriz de la variable x
y = np.linspace(-L+dy,L+dy,M) # matriz de la variable y
X,Y = np.meshgrid(x,y) # cuadrícula de área cuadrada
Z = X + 1j*Y # área compleja del plano

for k in range(1,n+1): # ciclo de recursividad
    ZZ = Z - (Z**4 + 1)/(4*Z**3)
    Z = ZZ
    W = f(Z)

ax.set_xlim(dx-L,dx+L) # establecer límites de eje x
ax.set_zlim(dy-L,dy+L) # establecer límites de eje y
ax.set_zlim(-3.5*L,4*L) # establecer límites de eje z
ax.axis("off") # no mostrar ejes
ax.plot_surface(X, Y, W, rstride=1, cstride=1, cmap='flag') # hacer trazado de contorno
plt.show() # Mostrar figura
```