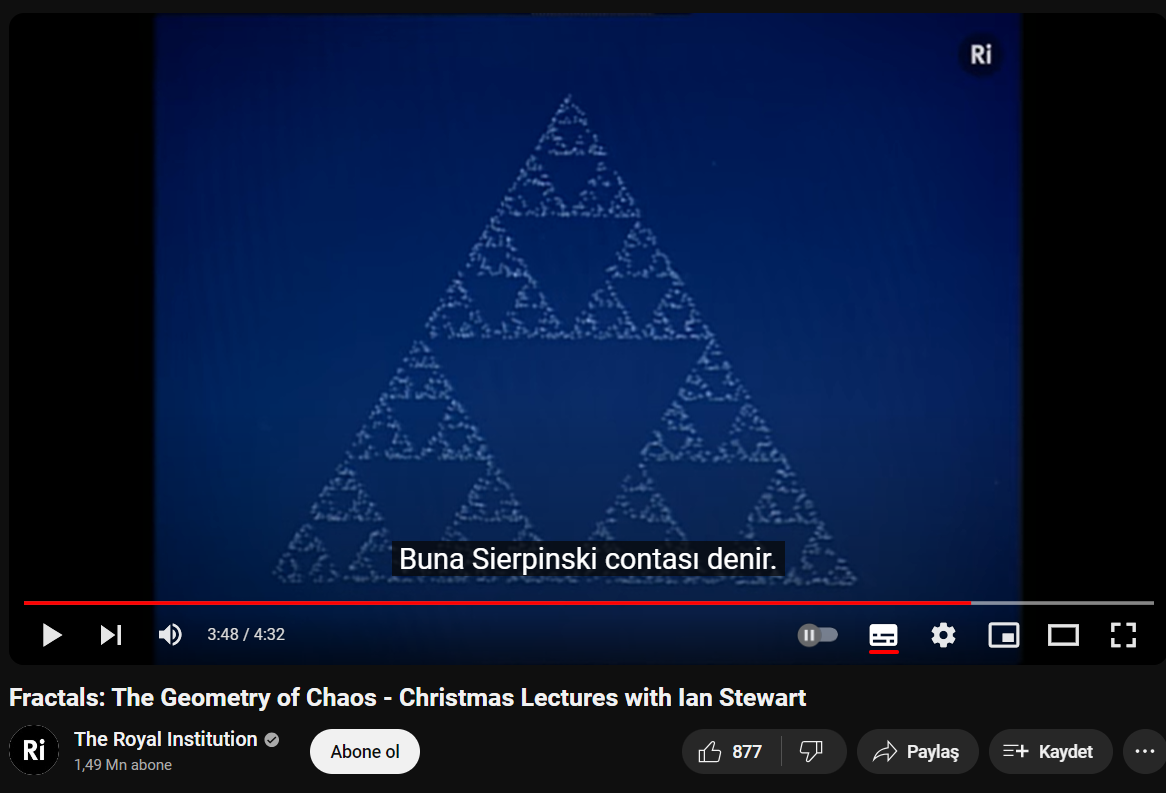
Fraktallar kendini tekrar eden örüntülerdir diyebiliriz. Örneğin kar, yıldırım yaprakların lifleri vs. Diğer bir deyişle fraktallar kaosun ya da kaotik yapıların geometrisidir.

Görüntü sıkıştırma, kanserli hücrelerin tespiti, petrol analizi gibi birçok alanda kullanılabilmektedir.



Sierpinski contası tipik bir fraktal örneğidir. Matematiksel olarak buradaki boşluklar sonsuza kadar devam edebilir.

**Fraktal**; [matematikte](https://tr.wikipedia.org/wiki/Matematik), çoğunlukla kendine benzeme veya oransal kırılma özelliği gösteren karmaşık geometrik şekillerin ortak adıdır. Fraktallar, klasik, yani [Öklid (Euklides)](https://tr.wikipedia.org/wiki/%C3%96klid) geometrideki kare, daire, küre gibi basit şekillerden çok farklıdır. Bunlar doğadaki, Öklid'çi geometri aracılığıyla tanımlanamayacak pek çok uzamsal açıdan düzensiz olguyu ve düzensiz biçimi tanımlama yeteneğine sahiptir. Fraktal terimi parçalanmış ya da kırılmış anlamına gelen [Latince](https://tr.wikipedia.org/wiki/Latince) "fractus" sözcüğünden türetilmiştir. İlk olarak 1975'te Polonya asıllı matematikçi [Benoit B. Mandelbrot](https://tr.wikipedia.org/wiki/Benoit_B._Mandelbrot) tarafından ortaya atılan kavram, yalnızca matematik değil [fiziksel kimya](https://tr.wikipedia.org/wiki/Fiziksel_kimya), fizyoloji ve [akışkanlar mekaniği](https://tr.wikipedia.org/wiki/Ak%C4%B1%C5%9Fkanlar_mekani%C4%9Fi) gibi değişik alanlar üzerinde önemli etkiler yaratan yeni bir geometri sisteminin doğmasına yol açmıştır.

Fraktalların belirleyici bir özelliği, *fraktal boyut* olarak adlandırılan matematiksel bir parametrelerinin olmasıdır. Bu parametrenin bütünüyle geçerli ve basit bir tanımı yoktur. Mandelbrot bu parametreyi [Haussdorf](https://tr.wikipedia.org/wiki/Hausdorff_uzay) boyutu ile denk tutmaktadır. Fraktal boyut, Öklid'çi şekillerin topolojik boyutlarına eşit, fraktallar için topolojik boyutlarından büyüktür. Örneğin [Cantor](https://tr.wikipedia.org/wiki/Cantor_teoremi) kümesinin fraktal boyutu �=log⁡2/log⁡3∼0.6309>0D = log2/log3~0.6309>0,, [topolojik](https://tr.wikipedia.org/wiki/Topoloji) boyutu ise ��=0 Dt=0'dır.[[1]](https://tr.wikipedia.org/wiki/Fraktal#cite_note-Mandelbrot83-1):14-15

Kendisinin tam bir kopyasını daha küçük boyutlarda içeren fraktallar için fraktal boyutu ve *kendine benzerlik boyutu* değerleri aynıdır. Bir şekil kendisine benzeyen �n kadar kopyadan oluşuyor ve her bir kopya özgün şekle göre, uzunluk olarak, 1/�1/m büyüklüğünde ise, bu şeklin kendine benzeme boyutu log⁡�/log⁡�logn/logm ile verilir. Yukarıda örnek olarak verilen Sierpinski üçgeni, kendine benzeyen �=3n=3 kopyadan oluşmuş, her bir kopya da özgün şeklin yarısı (�=2m=2) uzunluğundadır; dolayısıyla Sierpinski üçgenin fraktal boyutu �=log⁡3/log⁡2∼1.585D=log3/log2~1.585'tir.

(Wikipedia)

Fraktallar, farklı ölçeklerde kendine benzeyen sonsuz karmaşık desenlerdir. Örneğin bir ağaç gövdesi daha küçük dallara bölünür. Bunlar da daha küçük dallara bölünür ve bu şekilde devam eder.

Fraktalları programlı olarak üreterek basit şekilleri karmaşık tekrar eden desenlere dönüştürebiliriz.

Bu makalede, bazı temel A-Seviyesi geometri ve biraz programlama bilgisi kullanarak Python'da nasıl etkileyici fraktallar oluşturabileceğimizi araştıracağım.

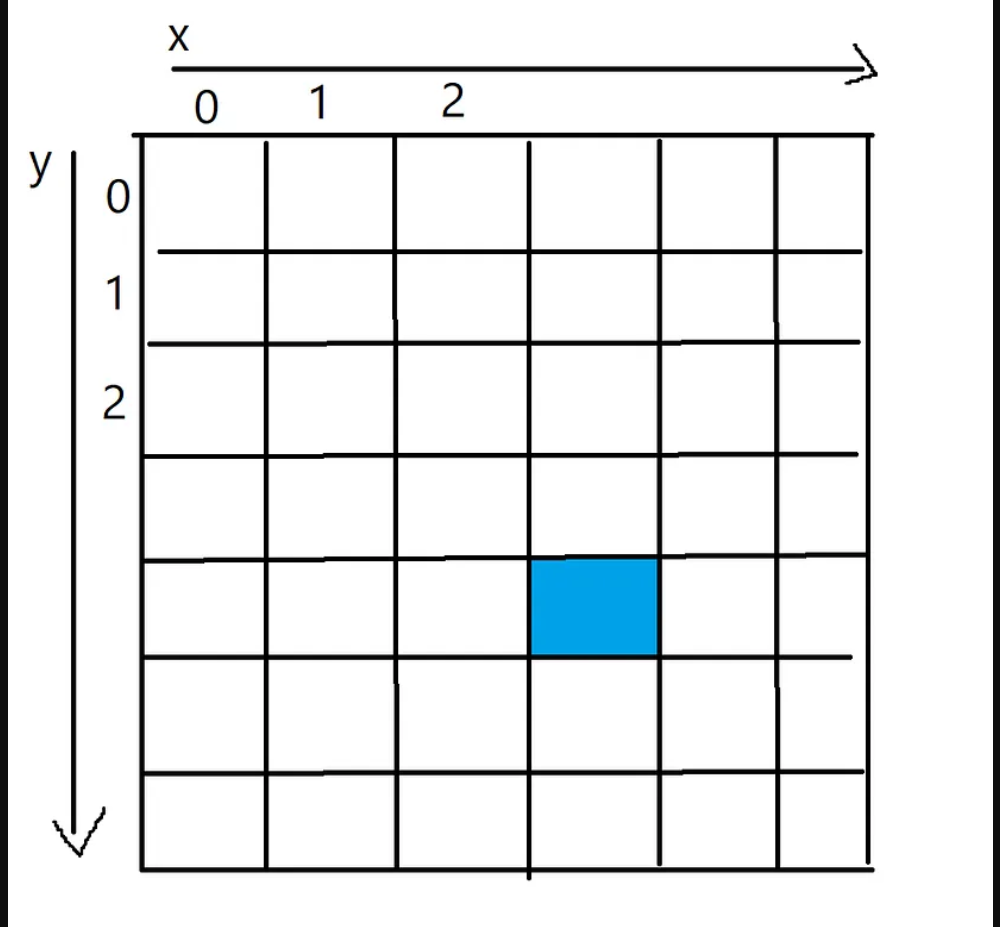
Fraktallar veri biliminde önemli bir rol oynamaktadır. Örneğin fraktal analizde, veri kümelerinin fraktal özellikleri, altta yatan süreçlerin yapısının anlaşılmasına yardımcı olmak için değerlendirilir. Ayrıca fraktal üretimin merkezinde yer alan yinelenen algoritma, ikili arama algoritmasından yinelenen sinir ağlarına kadar çok çeşitli veri problemlerine uygulanabilmektedir.

Eşkenar üçgen çizebilen bir program yazmak istiyorum. Üçgenin her iki tarafında biraz daha küçük, dışa bakan bir üçgen çizebilmelidir. Bu süreci istediğim kadar tekrarlayabilmeli, umarım bazı ilginç modeller ortaya çıkarabilirim.

Bir görüntüyü temsil etme

Bir görüntüyü iki boyutlu bir piksel dizisi olarak temsil edeceğim. Piksel dizisindeki her hücre, o pikselin rengini (RGB) temsil edecektir.

Bunu başarmak için, piksel dizisini oluşturmak için NumPy kütüphanelerini ve onu kaydedebileceğimizbir görüntüye dönüştürmek için Pillow kütüphanelerini kullanabiliriz.



Mavi pikselin x değeri 3 ve y değeri 4'tür ve piksellere[4][3] benzer bir 2 boyutlu diziden erişilebilir.

Bir çizgi çiz

Artık kodlama zamanı!

Öncelikle iki koordinat kümesini alıp aralarına çizgi çizebilecek bir fonksiyona ihtiyacım var.

Aşağıdaki kod, iki nokta arasında enterpolasyon yaparak ve her adımda piksel dizisine yeni pikseller ekleyerek çalışır. Bu işlemi bir satırın piksel piksel renklendirilmesi gibi düşünebilirsiniz.

Daha uzun kod satırlarını sığdırmaya yardımcı olmak için her kod parçacığında '\' devam karakterini kullandım.

Bir üçgen çiz

Artık iki nokta arasına doğru çizebilen bir fonksiyonum var, sıra ilk eşkenar üçgeni çizmeye geldi.

Bir üçgenin merkez noktası ve kenar uzunluğu verildiğinde, kullanışlı formülü kullanarak yüksekliği hesaplayabiliriz: h = ½(√3a).

Şimdi bu yüksekliği, merkez noktasını ve kenar uzunluğunu kullanarak üçgenin her köşesinin nerede olması gerektiğini bulabilirim. Daha önce yaptığım plot\_line fonksiyonunu kullanarak her köşe arasına bir çizgi çizebiliyorum.

Fraktalın oluşturulması

Sahne ayarlandı. Python'da ilk fraktalımı oluşturmak için ihtiyacım olan hemen hemen her şey hazır. Ne kadar heyecan verici!

Ancak bu son adım tartışmasız en zor olanıdır. Üçgen fonksiyonumuzun sahip olduğu her kenar için kendisini çağırmasını istiyorum. Bunun işe yaraması için, yeni küçük üçgenlerin her birinin merkez noktasını hesaplayabilmem ve onları bağlı oldukları tarafa dik olacak şekilde doğru şekilde döndürebilmem gerekiyor.

Döndürmek istediğim koordinatlardan merkez noktamızın uzaklığını çıkararak ve ardından bir çift koordinatı döndürmek için formülü uygulayarak, üçgenin her köşesini döndürmek için bu işlevi kullanabiliriz.

Artık bir üçgeni döndürebiliyorum, odak noktamı ilk üçgenin her iki yanında daha küçük yeni bir üçgen çizmeye çevirmeliyim.

Bunu başarmak için, her bir kenar için, kenar uzunluğunun küçültülmüş\_yan\_by parametresi tarafından azaltıldığı yeni bir üçgenin dönüşünü ve merkez noktasını hesaplamak üzere Draw\_triangle fonksiyonunu genişlettim.

Yeni üçgenin merkez noktasını ve dönüşünü hesapladıktan sonra, yeni, daha küçük üçgeni mevcut çizginin merkezinden dışarı çizmek için Draw\_triangle'ı (kendisi) çağırır. Bu daha sonra daha küçük bir üçgen için başka bir merkez noktaları ve dönüş kümesini hesaplayan aynı kod bloğuna ulaşacaktır.

Draw\_triangle işlevimiz artık çizmek istediğimiz üçgenlerin maksimum derinliğine ulaşana kadar kendisini çağıracağından buna yinelenen algoritma denir. Bu kaçış cümlesinin olması önemlidir, çünkü aksi takdirde fonksiyon teorik olarak sonsuza kadar yinelenmeye devam eder (ancak pratikte çağrı yığını çok büyür ve bu da yığın taşması hatasına neden olur)!

İşte python ile yaptığım fraktalın kodu:

import numpy as np

from PIL import Image

import math

# Plots a line between two coordinates

# Parameter list:

# from\_coordinates  -  [y,x] point within the array of pixels to begin plotting the line

# to\_coordinates    -  [y,x] point within the array to plot the line to

# thickness         -  how thick we want the line to be

# colour            -  [r,g,b] colour we want the line to have

# pixels            -  3d array representing the pixels e.g pixels[row][column][colour]

def plot\_line(from\_coordinates, to\_coordinates, thickness, colour, pixels):

    # Figure out the boundaries of our pixel array

    max\_x\_coordinate = len(pixels[0])

    max\_y\_coordinate = len(pixels)

    # The distances along the x and y axis between the 2 points

    horizontal\_distance = to\_coordinates[1] - from\_coordinates[1]

    vertical\_distance = to\_coordinates[0] - from\_coordinates[0]

    # The total distance between the two points, used to calculate how far we want to step each time we look for a new pixel to colour in

    distance = math.sqrt((to\_coordinates[1] - from\_coordinates[1])\*\*2 + (to\_coordinates[0] - from\_coordinates[0])\*\*2)

    step = round(distance)

    # How far we will step forwards along the x and y axis each time we colour in a new pixel

    horizontal\_step = horizontal\_distance/step

    vertical\_step = vertical\_distance/step

    # At this point, we enter the loop to draw the line in our pixel array

    # Each iteration of the loop will add a new point along our line

    for i in range(step):

        # These 2 coordinates are the ones at the center of our line

        current\_x\_coordinate = round(from\_coordinates[1] + (horizontal\_step\*i))

        current\_y\_coordinate = round(from\_coordinates[0] + (vertical\_step\*i))

        if (current\_x\_coordinate > 0 and current\_x\_coordinate < max\_x\_coordinate and current\_y\_coordinate > 0 and current\_y\_coordinate < max\_y\_coordinate):

            pixels[current\_y\_coordinate][current\_x\_coordinate] = colour

        # Once we have the coordinates, we draw a 'point' (a square) around the coordinates of size 'thickness'

        for x in range (-thickness, thickness):

            for y in range (-thickness, thickness):

                x\_value = current\_x\_coordinate + x

                y\_value = current\_y\_coordinate + y

                if (x\_value > 0 and x\_value < max\_x\_coordinate and y\_value > 0 and y\_value < max\_y\_coordinate):

                    pixels[y\_value][x\_value] = colour

# Draws a triangle, then calls itself for each outer facing edge it has

# Parameter list:

# center            -  [y,x] point within the array of pixels where the center of the triangle lies

# side\_length       -  length of each side of the triangle

# degrees\_rotate    -  how many degrees we wish it to be rotated about the center

# thickness         -  how thick we want the edges to be

# colour            -  [r,g,b] colour we want each line to have

# pixels            -  3d array representing the pixels e.g pixels[row][column][colour]

# shrink\_side\_by    -  fraction we wish each side to be shrunk by with each iteration

# max\_depth         -  maximum number of iterations

def draw\_triangle(center, side\_length, degrees\_rotate, thickness, colour, pixels, shrink\_side\_by, iteration, max\_depth):

    # The height of an equilateral triangle is, h = ½(√3a) where 'a' is the side length

    triangle\_height = side\_length \* math.sqrt(3)/2

    # The top corner

    top\_corner = [center[0] - triangle\_height/2, center[1]]

    # Bottom left corner

    bottom\_left\_corner = [center[0] + triangle\_height/2, center[1] - side\_length/2]

    # Bottom right corner

    bottom\_right\_corner = [center[0] + triangle\_height/2, center[1] + side\_length/2]

    if (degrees\_rotate != 0):

        top\_corner = rotate\_coordinate\_around\_point(top\_corner, center, degrees\_rotate)

        bottom\_left\_corner = rotate\_coordinate\_around\_point(bottom\_left\_corner, center, degrees\_rotate)

        bottom\_right\_corner = rotate\_coordinate\_around\_point(bottom\_right\_corner, center, degrees\_rotate)

    lines = [[top\_corner, bottom\_left\_corner], [top\_corner, bottom\_right\_corner], [bottom\_left\_corner, bottom\_right\_corner]]

    line\_number = 0

    # Draw a line between each corner to complete the triangle

    for line in lines:

        line\_number += 1

        plot\_line(line[0], line[1], thickness, colour, pixels)

        # Draw some new triangles

        if (iteration < max\_depth and (iteration < 1 or line\_number < 3)):

            gradient = (line[1][0] - line[0][0]) / (line[1][1] - line[0][1])

            new\_side\_length = side\_length\*shrink\_side\_by

            # Center of the line of the traingle we are drawing

            center\_of\_line = [(line[0][0] + line[1][0]) / 2, (line[0][1] + line[1][1]) / 2]

            new\_center = []

            new\_rotation = degrees\_rotate

            # Amount we need to rotate the traingle by

            if (line\_number == 1):

                new\_rotation += 60

            elif (line\_number == 2):

                new\_rotation -= 60

            else:

                new\_rotation += 180

            # In an ideal world this would be gradient == 0, but due to floating point division

            # we cannot ensure that this will always be the case

            if (gradient < 0.0001 and gradient > -0.0001):

                if (center\_of\_line[0] - center[0] > 0):

                    new\_center = [center\_of\_line[0] + triangle\_height \* (shrink\_side\_by/2), center\_of\_line[1]]

                else:

                    new\_center = [center\_of\_line[0] - triangle\_height \* (shrink\_side\_by/2), center\_of\_line[1]]

            elif gradient != 0:

                # Calculate the normal to the gradient of the line we're going to draw a new triangle on

                difference\_from\_center = -1/gradient

                # Calculate the distance from the center of the line that the center of our new traingle will be

                distance\_from\_center = triangle\_height \* (shrink\_side\_by/2)

                # Calculate the size of the x axis, from the center of our line to the center of our new triangle

                x\_length = math.sqrt((distance\_from\_center\*\*2)/(1 + difference\_from\_center\*\*2))

                # Figure out which way around the x direction needs to go

                if (center\_of\_line[1] < center[1] and x\_length > 0):

                    x\_length \*= -1

                # Now calculate the y length and direction of the axis

                y\_length = x\_length \* difference\_from\_center

                # Offset the center of the line with our new x and y values

                new\_center = [center\_of\_line[0] + y\_length, center\_of\_line[1] + x\_length]

            draw\_triangle(new\_center, new\_side\_length, new\_rotation, thickness, colour, pixels, shrink\_side\_by, iteration+1, max\_depth)

# Rotates 'coordinate' around the 'center\_point'

# Paramater list:

# coordinate    -  [y,x] point to rotate

# center\_point  -  [y,x] point to rotate around

# degrees       -  degrees to rotate by

def rotate\_coordinate\_around\_point(coordinate, center\_point, degrees):

    # Subtract the point we are rotating around from our coordinate to remove the offset from 0,0

    x = (coordinate[0] - center\_point[0])

    y = (coordinate[1] - center\_point[1])

    # Python's cos and sin functions take radians instead of degrees

    radians = math.radians(degrees)

    # Calculate our rotated points

    new\_x = (x \* math.cos(radians)) - (y \* math.sin(radians))

    new\_y = (y \* math.cos(radians)) + (x \* math.sin(radians))

    # Add back our offset we subtracted at the beginning to our rotated points and return

    return [new\_x + center\_point[0], new\_y + center\_point[1]]

# Define the size of our image

pixels = np.zeros( (10000,10000,3), dtype=np.uint8 )

# Draw 4 fractals

draw\_triangle([3000,3000], 2000, 0, 0, [255,200,0], pixels, 1/2, 0, 9)

draw\_triangle([3000,7000], 1200, 0, 0, [165, 242, 243], pixels, 2/3, 0, 9)

draw\_triangle([7000,3000], 1000, 0, 0, [124,252,0], pixels, 2.28/3, 0, 9)

draw\_triangle([7000,7000], 800, 0, 0, [203, 195, 227], pixels, 2.5/3, 0, 9)

# Turn our pixel array into a real picture

img = Image.fromarray(pixels)

# Show our picture, and save it

img.show()

img.save('Fractal.png')

kaynak = <https://towardsdatascience.com/creating-fractals-in-python-a502e5fc2094>

Kodun Açıklaması:

1. **import numpy as np**: NumPy kütüphanesini içeri aktarır ve kısaltma olarak "np" olarak kullanır.
2. **from PIL import Image**: PIL kütüphanesinden sadece Image modülünü içeri aktarır. Bu, görüntü oluşturma ve işleme işlevleri sağlar.
3. **import math**: Python'ın matematiksel işlevlerini içeren math modülünü içeri aktarır.
4. **def plot\_line(from\_coordinates, to\_coordinates, thickness, colour, pixels):**: Bir fonksiyon tanımlar. Bu fonksiyon, iki koordinat arasında bir çizgi çizer. Parametreler şunlardır:
   * **from\_coordinates**: Başlangıç koordinatları (y,x) dizisi.
   * **to\_coordinates**: Bitiş koordinatları (y,x) dizisi.
   * **thickness**: Çizgi kalınlığı.
   * **colour**: Çizgi rengi (r,g,b) dizisi.
   * **pixels**: Çizimi gerçekleştireceğimiz pikselleri içeren 3 boyutlu dizi.
5. **max\_x\_coordinate = len(pixels[0])** ve **max\_y\_coordinate = len(pixels)**: Piksel dizisinin boyutlarını alarak, x ve y koordinatlarının maksimum değerlerini hesaplar.
6. **horizontal\_distance = to\_coordinates[1] - from\_coordinates[1]** ve **vertical\_distance = to\_coordinates[0] - from\_coordinates[0]**: İki nokta arasındaki yatay ve dikey mesafeyi hesaplar.
7. **distance = math.sqrt((to\_coordinates[1] - from\_coordinates[1])\*\*2 + (to\_coordinates[0] - from\_coordinates[0])\*\*2)**: İki nokta arasındaki mesafeyi hesaplar (Pitagoras teoremi kullanılarak).
8. **step = round(distance)**: Çizginin ne kadar adımla oluşturulacağını belirler. Mesafeyi yuvarlar.
9. **horizontal\_step = horizontal\_distance/step** ve **vertical\_step = vertical\_distance/step**: Her adımda yatay ve dikey yönde ilerleme miktarını hesaplar.
10. **for i in range(step):**: Çizgiyi oluşturmak için bir döngü başlatır.
11. **current\_x\_coordinate = round(from\_coordinates[1] + (horizontal\_step\*i))** ve **current\_y\_coordinate = round(from\_coordinates[0] + (vertical\_step\*i))**: Her adımda yeni bir nokta belirler.
12. **if (current\_x\_coordinate > 0 and current\_x\_coordinate < max\_x\_coordinate and current\_y\_coordinate > 0 and current\_y\_coordinate < max\_y\_coordinate):**: Geçerli koordinatların piksel dizisinin sınırları içinde olup olmadığını kontrol eder.
13. **pixels[current\_y\_coordinate][current\_x\_coordinate] = colour**: Çizgiyi piksel dizisine ekler.
14. **for x in range (-thickness, thickness):** ve **for y in range (-thickness, thickness):**: Belirtilen kalınlıkta bir kare çizmek için döngüler başlatır.
15. **pixels[y\_value][x\_value] = colour**: Kalınlık boyunca bir kare çizgisini piksel dizisine ekler.
16. **triangle\_height = side\_length \* math.sqrt(3)/2**: Üçgenin yüksekliğini hesaplar. Bir eşkenar üçgenin yüksekliği, yarım taban uzunluğu çarpı √3/2 formülüyle bulunur.
17. **top\_corner = [center[0] - triangle\_height/2, center[1]]**: Üçgenin en üst köşesinin koordinatlarını hesaplar. Yüksekliğin yarısını merkezden çıkarır ve x koordinatını sabitler.
18. **bottom\_left\_corner = [center[0] + triangle\_height/2, center[1] - side\_length/2]** ve **bottom\_right\_corner = [center[0] + triangle\_height/2, center[1] + side\_length/2]**: Üçgenin sol alt ve sağ alt köşelerinin koordinatlarını hesaplar. Üçgenin yüksekliğinin yarısıyla merkezden uzaklaşır ve aynı zamanda yan uzunluğun yarısı kadar yatayda kayar.
19. **if (degrees\_rotate != 0):**: Dönme açısının sıfır olmadığını kontrol eder. Eğer sıfır değilse, üçgenin köşelerini döndürür.
20. **lines = [[top\_corner, bottom\_left\_corner], [top\_corner, bottom\_right\_corner], [bottom\_left\_corner, bottom\_right\_corner]]**: Üçgenin kenarlarını temsil eden köşe çiftlerini içeren bir liste oluşturur.
21. **line\_number = 0**: Kenar sayacını başlatır.
22. **for line in lines:**: Üçgenin her kenarı için bir döngü başlatır.
23. **line\_number += 1**: Her bir kenarın sırasını artırır.
24. **plot\_line(line[0], line[1], thickness, colour, pixels)**: **plot\_line()** fonksiyonunu kullanarak üçgenin kenarlarını çizer.
25. **if (iteration < max\_depth and (iteration < 1 or line\_number < 3)):**: Üçgenin daha fazla tekrar edilebilir olup olmadığını kontrol eder. Eğer en fazla derinlikte değilse ve iterasyon 1'den küçükse veya kenar sayısı 3'ten küçükse, daha fazla üçgen çizilir.
26. **gradient = (line[1][0] - line[0][0]) / (line[1][1] - line[0][1])**: Kenarın eğimini hesaplar.
27. **new\_side\_length = side\_length\*shrink\_side\_by**: Yeni üçgenin yan uzunluğunu hesaplar. Yan uzunluğu küçültmek için belirtilen oranla çarpar.
28. **center\_of\_line = [(line[0][0] + line[1][0]) / 2, (line[0][1] + line[1][1]) / 2]**: Üçgenin çizildiği kenarın ortasını hesaplar.
29. **new\_center = []** ve **new\_rotation = degrees\_rotate**: Yeni üçgenin merkez koordinatlarını ve dönme açısını tanımlar.
30. **if (line\_number == 1):**, **elif (line\_number == 2):**, **else:**: Üçgenin döndürme açısını belirler. İlk kenar için 60 derece ekler, ikinci kenar için 60 derece çıkarır, üçüncü kenar için 180 derece ekler.
31. **if (gradient < 0.0001 and gradient > -0.0001):**: Kenarın eğiminin çok yaklaşık sıfıra eşit olup olmadığını kontrol eder. Bu, kenarın neredeyse dikey olduğunu ve doğru bir normal hesaplamasının mümkün olmadığını gösterir.
32. **if (center\_of\_line[0] - center[0] > 0):**: Kenarın merkezi, üçgenin merkezinden sağ taraftaysa:

* **new\_center = [center\_of\_line[0] + triangle\_height \* (shrink\_side\_by/2), center\_of\_line[1]]**: Yeni üçgenin merkezini, kenarın ortasından biraz daha yükseğe kaydırır.
* **else: new\_center = [center\_of\_line[0] - triangle\_height \* (shrink\_side\_by/2), center\_of\_line[1]]**: Kenarın merkezi, üçgenin merkezinden sol taraftaysa, yeni üçgenin merkezini kenarın ortasından biraz daha düşük bir yere kaydırır.

1. **elif gradient != 0:**: Kenarın eğimi sıfıra eşit değilse:

* **difference\_from\_center = -1/gradient**: Kenarın eğimine dik olan bir çizgiyi hesaplar.
* **distance\_from\_center = triangle\_height \* (shrink\_side\_by/2)**: Yeni üçgenin merkezine olan mesafeyi hesaplar.
* **x\_length = math.sqrt((distance\_from\_center\*\*2)/(1 + difference\_from\_center\*\*2))**: Yeni üçgenin merkezinin kenarın ortasına olan yatay mesafesini hesaplar.
* **if (center\_of\_line[1] < center[1] and x\_length > 0):**: Kenarın merkezi, üçgenin merkezinin sol tarafında ve x uzunluğu pozitifse:
  + **x\_length \*= -1**: x uzunluğunu negatife çevirir.
* **y\_length = x\_length \* difference\_from\_center**: Yeni üçgenin merkezinin kenarın ortasına olan dikey mesafesini hesaplar.
* **new\_center = [center\_of\_line[0] + y\_length, center\_of\_line[1] + x\_length]**: Yeni üçgenin merkezini hesaplar.

1. **draw\_triangle(new\_center, new\_side\_length, new\_rotation, thickness, colour, pixels, shrink\_side\_by, iteration+1, max\_depth)**: Yeni üçgenin çizimini yapar.
2. **def rotate\_coordinate\_around\_point(coordinate, center\_point, degrees):**: Bir fonksiyon tanımlar. Bu fonksiyon, belirli bir merkez etrafında belirli bir açıyla bir noktayı döndürür.
3. **x = (coordinate[0] - center\_point[0])** ve **y = (coordinate[1] - center\_point[1])**: Noktayı döndürmek için gereken işlemi basitleştirmek için noktanın merkez etrafında olan konumunu hesaplar.
4. **radians = math.radians(degrees)**: Dereceleri radyanlara dönüştürür.
5. **new\_x = (x \* math.cos(radians)) - (y \* math.sin(radians))** ve **new\_y = (y \* math.cos(radians)) + (x \* math.sin(radians))**: Noktayı döndürmek için gerekli matematiksel işlemi gerçekleştirir.
6. **return [new\_x + center\_point[0], new\_y + center\_point[1]]**: Döndürülmüş noktanın koordinatlarını döndürür.
7. **pixels = np.zeros( (10000,10000,3), dtype=np.uint8 )**: 10.000x10.000 boyutlarında bir piksel dizisi oluşturur.
8. **draw\_triangle()** fonksiyonunu kullanarak 4 farklı fraktal çizilir. Her bir fraktalın merkezi, yan uzunluğu, dönme açısı, kalınlığı, rengi, pikselleri içeren dizi, yan uzunluğu küçültme oranı, iterasyon sayısı ve maksimum derinlik gibi parametreler belirtilir.
9. **img = Image.fromarray(pixels)**: Piksel dizisini bir görüntüye dönüştürür.
10. **img.show()**: Oluşturulan görüntüyü gösterir.
11. **img.save('Fractal.png')**: Oluşturulan görüntüyü "Fractal.png" adıyla kaydeder.

Fraktallar, kaotik sistemlerin analizinde önemli bir rol oynar. Kaotik sistemler, belirli bir matematiksel model tarafından tanımlanan ancak zamanla karmaşık ve öngörülemeyen davranışlar sergileyen sistemlerdir. Fraktallar, bu karmaşıklığı ve öngörülemezliği temsil etmek için kullanılabilir ve kaotik sistemlerin dinamiklerini anlamak için güçlü bir araç olabilir. İşte fraktalların kaotik sistemlerin analizinde kullanımının bazı yolları:

Attractorların Karakterizasyonu: Kaotik sistemler genellikle belirli bir çekiciye veya "attractor"a sahiptir. Bu attractorlar, belirli bir fraktal geometriye sahip olabilirler. Fraktal boyutlar ve karmaşıklık ölçüleri gibi fraktal özellikler kullanılarak bu attractorlar karakterize edilebilir.

**Bifurkasyon Analizi**: Kaotik sistemlerdeki belirli bir parametre değişikliği, sistemde belirli bir noktada belirgin bir değişikliğe neden olabilir. Bu noktalar bifurkasyon noktaları olarak adlandırılır. Fraktal analiz, bu bifurkasyon noktalarının nasıl dağıldığını ve nasıl bir kalıp izlediğini inceleyebilir.

**Özyinelemeli Yapıların İncelenmesi**: Kaotik sistemlerde, sistem davranışı belirli bir ölçekte özyinelemeli olabilir. Fraktal analiz, sistem davranışının bu özyinelemeli yapılarını tanımlayabilir ve anlayabilir.

**Doğrusal Olmayan Dinamiklerin İncelenmesi**: Kaotik sistemlerin doğrusal olmayan dinamikleri, sistemdeki küçük değişikliklerin zamanla büyük sonuçlara yol açabileceği anlamına gelir. Fraktal analiz, bu doğrusal olmayan dinamiklerin nasıl işlediğini ve sistemin zamanla nasıl evrildiğini ortaya çıkarabilir.

**Belirsizlik ve Hassasiyet Analizi**: Kaotik sistemlerdeki belirsizlik ve hassasiyet, başlangıç koşullarındaki küçük değişikliklerin sonuçları üzerinde büyük etkilere yol açabilir. Fraktal analiz, bu belirsizlikleri ve hassasiyetleri tanımlayabilir ve sistemin davranışını daha iyi anlamak için kullanılabilir.

Kaynak(chatGpt)