Chaos Game simulations in Python

Indranil Ghosh
Jadavpur University, Physics Department
Email: indranilg49@gmail.com

December 29, 2017

Abstract

This paper simulates the *Chaos Game* to produce various fractal patterns like Sierpinski's triangle, Barnsley's fern and few restricted chaos game fractals. Chaos game is an algorithm to produce fractals using a *n-gon* and an initial point chosen randomly anywhere on the drawing platform. An iterative creation of a sequence of points thereafter produces the fractal. Here some of the few fractal patterns, each with a description, its own chaos game rule and python code to simulate its developement are listed. The programs are written in either Pygame or Pylab modules. Pygame is a cross-platform set of python modules to create video-games. One needs to separately install pygame and matplotlib for the modules to work, as they do not come ready with Python.

1 Introduction

The algorithm of *Chaos Game* was first deveploped by the british mathematician, ***Michael Barnsley*** around the year 1988, which produced some interesting fractal patterns (but not always). Generally, it refers the method of generating the attractor or the fixed point of an iterated function system(IFS). In the chaos game the fractal is produced (if any!) by iteratively creating a sequence of points, starting with an initial random point anywhere on the drawing platform. In the sequence, each point is the fraction of the distance between the previous point and a randomly chosen vertex (by rolling a dice, if human or by using a pseudo-random generator, if a computer) of the n-gon. Although the algorithm is quite simple, the patterns formed after continuous iterations, always exhibit infinte complexity and self similarity. Zooming a particular section of the fractal results in the same fractal pattern but with less density of points. If we go on zooming we notice the same pattern as the parent pattern with decreasing density. Mathematically the chaos game is carried out upto infinty which is not possible in reality for a human being or a computer! So, we create enough iterations, in a machine (though not upto infinity) to produce this effect of self-similarity. In these simulations with each iterations the fractal

pattern grows and becomes more clearer with time. If the n-gon is regular the fractal pattern formed is symmetric.

Contents

2 Sirepinski's Triangle

In constructing the sierpinski's triangle, also called Sierpinski's gasket, we consider 3 random vertices of a triangle and a random starting point. The distance factor **r** is $\frac{1}{2}$ (Distance factor is the fraction of the distance between a point plotted at the previous step and a randomly chosen vertex of the triangle). This triangle is one of the most famous examples of self similar sets which can be recursively subdivided into smaller equally shaped triangles. It is named after the Polish mathematician ***Wacław Sierpiński***.

Let us consider an example. Let A, B and C be the three random vertices of a triangle and St be the starting point, also chosen randomly on the drawing platform. In pygame, we use a drawing board having dimensions 800 X 800, i.e having a total of 640000 pixels. Generally if we play manually by rolling a die, we can come up with any random integer in the range [1, 6], with an equal probability of $\frac{1}{6}$. We design the game in such a way that if we come up with a face **1, 2**, we move towards the point **A** from the previous point and plot the a new point halfway between. Similarly if we come up with **3, 4** or **5, 6** we move halfway towards **B** or **C** respectively. This can be easily deduced from Figure 1. We have a starting random point and the first face we get after rolling a die is **5**. As a result we move halfway towards **C** as described by the rule and plot a point on the corresponding coordinate. The next face we get is **4** and so plot a point halfway between the last point and **B**. This process is continued for a large number of iterations that results in the formation of Figure 2.

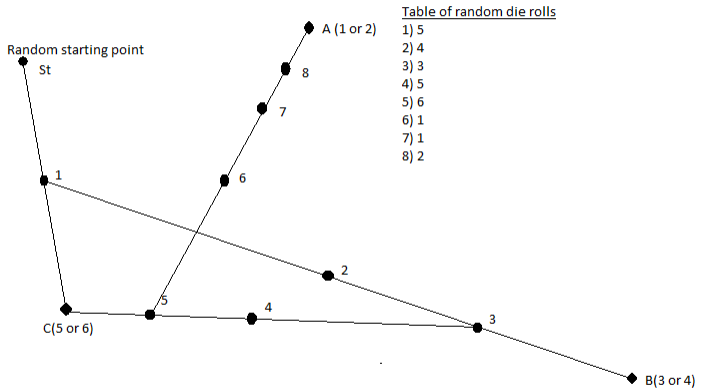

Figure 1: scatter plot of the first few iterations of the Sierpinski's Triangle

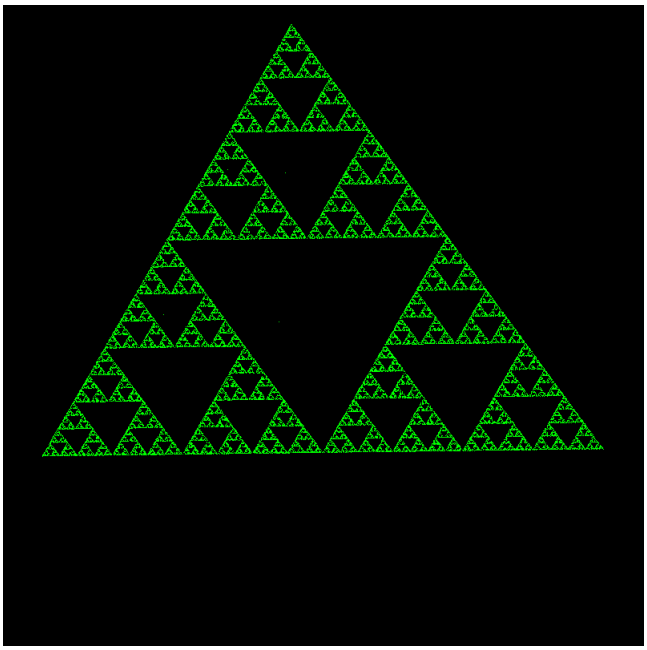

Figure 2: Sierpinski's Triangle

Surely the above mentioned algorithm is tremendously time and energy consuming for a human being to carry out for a large number of iterations. So pylab! To simulate a die roll, we take the help of a pseudo-random generator that generates a random number in the range [1, 6]. Then we carry out the iterations with the help of a while loop to notice the Sierpinski triangle grow with time. At first we choose 4 random points by clicking on arbitrary positions on the drawing surface. Among these points the first 3 points are **A**, **B** and **C** respectively followed by the starting point **St**. Nothing happens until we choose the 4 points on the drawing surface. As soon as the fourth point is chosen, the fractal pattern starts to grow. We do not set any upper-bound in the while loop, for the process to continue forever! But we can surely terminate the program by clicking the close button on the top-right corner of the screen. A point to be noted is that, if we click anywhere on the drawing platform after the process has started, the program will terminate. Listed below is the python program to simulate the fractal growth.

```
import random, pygame, sys
from pygame.locals import *

#set up the window
DISPLAYSURF=pygame.display.set_mode((800, 800))
pygame.display.set_caption('Sierpinski Triangle')

#set up the colors
BLACK=(0, 0, 0)
GREEN=(0, 255, 0)

i=0
#run the loop
while True:
    for event in pygame.event.get():
        if event.type==QUIT:
            pygame.image.save(DISPLAYSURF, "Sierpinski1.png")
            pygame.quit()
            sys.exit()
        elif event.type==MOUSEBUTTONUP:
            i+=1
            if i==1:
                A=(event.pos[0], event.pos[1])
                pygame.draw.circle(DISPLAYSURF, GREEN, A, 0, 0)
            elif i==2:
                B=(event.pos[0], event.pos[1])
                pygame.draw.circle(DISPLAYSURF, GREEN, B, 0, 0)
            elif i==3:
                C=(event.pos[0], event.pos[1])
```

```
                pygame.draw.circle(DISPLAYSURF, GREEN, C, 0, 0)
            elif i==4:
                St=(event.pos[0], event.pos[1])
                pygame.draw.circle(DISPLAYSURF, GREEN, St, 0, 0)
            else:
                pygame.quit()
                sys.exit()
    if i==4:
        x=random.randint(1, 6)
        if x==1 or x==2: St=((St[0] + A[0])/2, (St[1] + A[1])/2)
        elif x==3 or x==4: St=((St[0] + B[0])/2, (St[1] + B[1])/2)
        else: St=((St[0] + C[0])/2, (St[1] + C[1])/2)
        pygame.draw.circle(DISPLAYSURF, GREEN, St, 0, 0)
    pygame.display.update()
```

N.J.A Sloane's Online Encyclopedia of Integer Sequences also mentions this pattern as a triangle of integer sequence, read by rows, formed by reading Pascal's Triangle Mod 2. See A047999 and A001317.

3 Sierpinski's Pentagon

3.1 Type I [n=5, r=$\frac{3}{8}$]

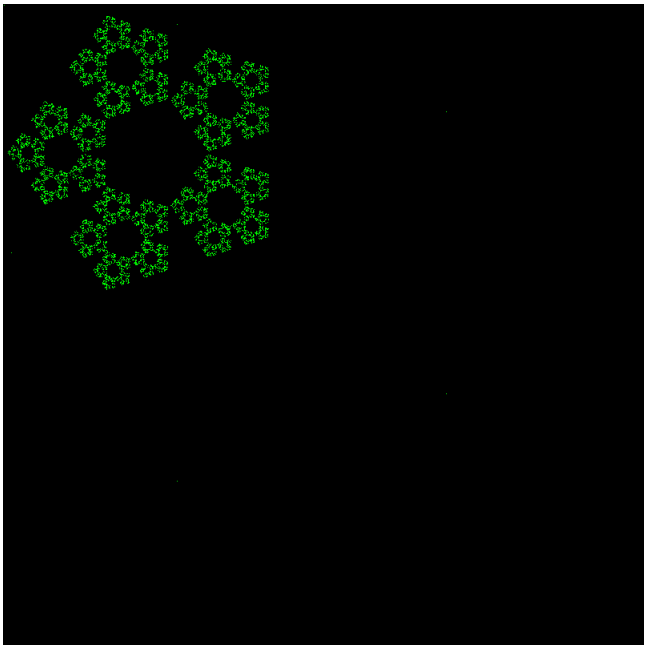

Figure 3: Sierpinski's Pentagon Type **I**

To construct the above fractal pattern, we need to follow almost the same algorithm as of Sierpinski's triangle, with slight alterations. Here we have number of vertices **n** = 5, of a regular pentagon along with a random starting point and the distance factor **r**= $\frac{3}{8}$. We chose a regular pentagon to get a symmetric fractal pattern. In the code we arbitrarily select five vertices that make up a regular pentagon (obviously one can change the coordinates of these vertices) and a random starting point. Listed below is the python program that simulates this fractal growth.

```
import random, pygame, sys
from pygame.locals import *

#set up the window
DISPLAYSURF=pygame.display.set_mode((800, 800), 0, 32)
pygame.display.set_caption('Sierpinski_pentagon_type_I')

#set up the colors
GREEN=(0, 255, 0)
```

```
#set up the points for sierpinski's pentagon Type I
A=(217, 25)
B=(553, 134)
C=(553, 486)
D=(217, 595)
E=(10, 310)
St=(1, 1)

pygame.draw.circle(DISPLAYSURF, GREEN, A, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, B, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, C, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, D, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, E, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, St, 0, 0)

#run the loop
while True:
    x=random.randint(1, 5)
    if x==1 : St=((St[0] + A[0])*3/8, (St[1] + A[1])*3/8)
    elif x==2 : St=((St[0] + B[0])*3/8, (St[1] + B[1])*3/8)
    elif x==3 : St=((St[0] + C[0])*3/8, (St[1] + C[1])*3/8)
    elif x==4 : St=((St[0] + D[0])*3/8, (St[1] + D[1])*3/8)
    else: St=((St[0] + E[0])*3/8, (St[1] + E[1])*3/8)
    pygame.draw.circle(DISPLAYSURF, GREEN, St, 0, 0)

    for event in pygame.event.get():
        if event.type==QUIT:
            pygame.image.save(DISPLAYSURF, "pentagon.png")
            pygame.quit()
            sys.exit()
    pygame.display.update()
```

3.2 Type II [n=5, r=$\frac{1}{2}$, restricted]

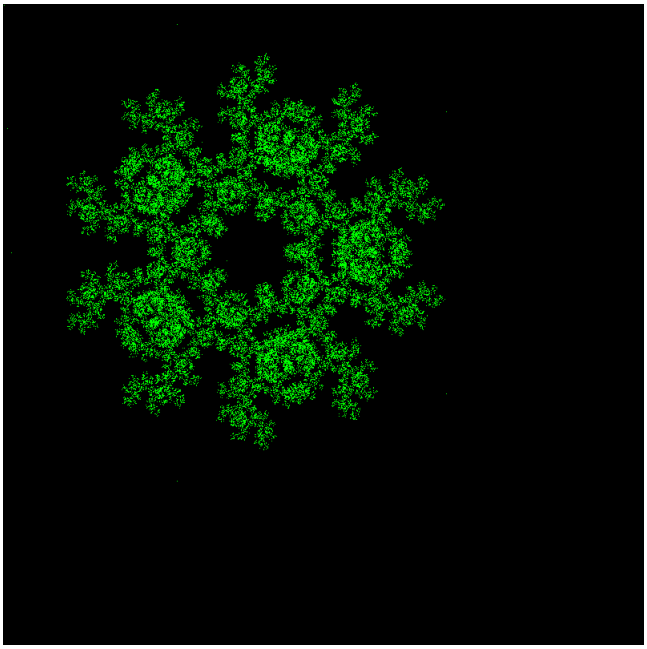

Figure 4: Sierpinski's Pentagon Type **II**

The fractal pattern we see above is an example of a "resctricted chaos game". Like type I, this pattern uses 5 vertices of a regular pentagon along with a random starting point, but with distance factor **r**=$\frac{1}{2}$. This pattern has a keyword: restriction because, to plot this, although we allow a point to be plotted midway towards a random vertex, the currently chosen vertex must be different than the previous one. In the python code, we use the same vertices and the starting point we used for type I. Listed below is the python program that simulates this fractal growth

```
import random, pygame, sys
from pygame.locals import *

#set up the window
DISPLAYSURF=pygame.display.set_mode((800, 800), 0, 32)
pygame.display.set_caption('Sierpinski_pentagon_type_II')

#set up the colors
GREEN=(0, 255, 0)

#set up the points for sierpinski's pentagon Type II
```

```
A=(217, 25)
B=(553, 134)
C=(553, 486)
D=(217, 595)
E=(10, 310)
St=(1, 1)

pygame.draw.circle(DISPLAYSURF, GREEN, A, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, B, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, C, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, D, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, E, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, St, 0, 0)

prev=random.randint(1, 5)
#run the loop
while True:
    l=range(1, 6)
    l.remove(prev)
    x=random.choice(l)
    if x==1:St=((St[0] + A[0])/2, (St[1] + A[1])/2)
    elif x==2:St=((St[0] + B[0])/2, (St[1] + B[1])/2)
    elif x==3:St=((St[0] + C[0])/2, (St[1] + C[1])/2)
    elif x==4:St=((St[0] + D[0])/2, (St[1] + D[1])/2)
    else:St=((St[0] + E[0])/2, (St[1] + E[1])/2)
    prev=x
    pygame.draw.circle(DISPLAYSURF, GREEN, St, 0, 0)

    for event in pygame.event.get():
        if event.type==QUIT:
            pygame.image.save(DISPLAYSURF, "pentagon2.png")
            pygame.quit()
            sys.exit()
    pygame.display.update()
```

4 Sierpinski's Square[n=4, r=$\frac{1}{2}$, restricted]

In this case we always consider the number of vertices **n**=4 and distance factor **r**=$\frac{1}{2}$. Without any restrictions we get a random distribution of points all over the drawing platform, without any generation of interesting fractal pattern. So we introduce various restrictions to observe some captivating fractal images.

4.1 Type I

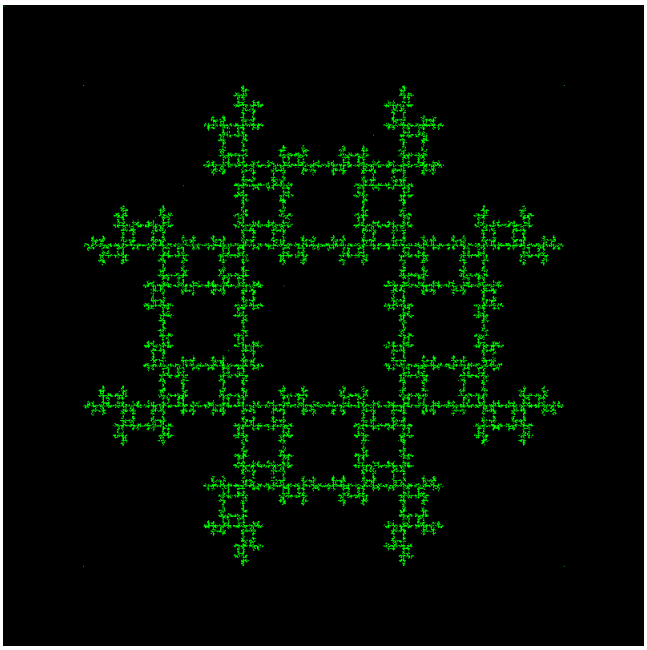

Figure 5: Sierpinski's Square Type **I**

Like type II of Sierpinski's pentagon, we follow the same rule except that the number of vertices to start with is 4 and we will have a regular square. The simulation randomly chooses a vertex and plots a point midway between the preveiously chosen point and this vertex, but this vertex must be different from the last vertex chosen. This continues for a large number of iterations and results in the formation of Figure 5. Listed below is the python code.

```
import random, pygame, sys
from pygame.locals import *

#set up the window
DISPLAYSURF=pygame.display.set_mode((800, 800), 0, 32)
pygame.display.set_caption('Sierpinski_square_Type_I')

#set up the colors
GREEN=(0, 255, 0)

#set up the points for sierpinski's square Type I
A=(100, 100)
B=(700, 100)
C=(700, 700)
```

```
D=(100, 700)
St=(1, 1)

pygame.draw.circle(DISPLAYSURF, GREEN, A, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, B, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, C, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, D, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, St, 0, 0)

prev=random.randint(1, 4)
#run the loop
while True:
    l=range(1, 5)
    l.remove(prev)
    x=random.choice(l)
    if x==1:St=((St[0] + A[0])/2, (St[1] + A[1])/2)
    elif x==2:St=((St[0] + B[0])/2, (St[1] + B[1])/2)
    elif x==3:St=((St[0] + C[0])/2, (St[1] + C[1])/2)
    else:St=((St[0] + D[0])/2, (St[1] + D[1])/2)
    prev=x
    pygame.draw.circle(DISPLAYSURF, GREEN, St, 0, 0)

    for event in pygame.event.get():
        if event.type==QUIT:
            pygame.image.save(DISPLAYSURF, "square.png")
            pygame.quit()
            sys.exit()
    pygame.display.update()
```

4.2 Type II

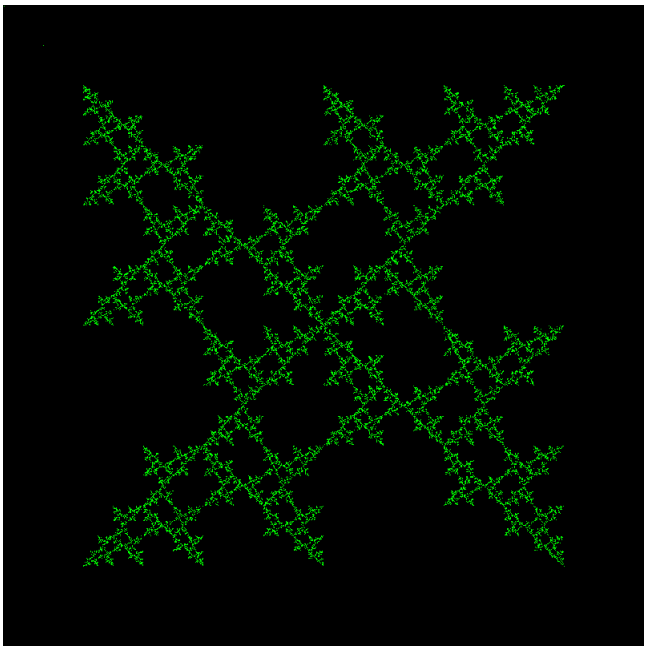

Figure 6: Sierpinski's Square Type **II**

To construct the above type of sierpinski's square we plot 4 vertices of a regular square at first and a random starting point. Then in the simulation, we randomly chose a vertex in each iteration and plot a new point midway between the previously chosen point and the current vertex, but with the restriction that the current vertex should not be 1 place away in the anti-clockwise direction from the previously chosen vertex.In the code, we chose the same vertices and the starting point as that of type I. Listed below is the python code to simulate the growth of the fractal.

```
import random, pygame, sys
from pygame.locals import *

#set up the window
DISPLAYSURF=pygame.display.set_mode((800, 800), 0, 32)
pygame.display.set_caption('Sierpinski square Type II')

#set up the colors
GREEN=(0, 255, 0)

#set up the points for sierpinski's square Type II
A=(100, 100)
```

```
B=(700, 100)
C=(700, 700)
D=(100, 700)
St=(1, 1)

pygame.draw.circle(DISPLAYSURF, GREEN, A, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, B, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, C, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, D, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, St, 0, 0)

prev=random.randint(1, 4)
#run the loop
while True:
    l=range(1, 5)
    l.remove(prev)
    x=random.choice(l)
    if x==1:
        St=((St[0] + A[0])/2, (St[1] + A[1])/2)
        prev=4
    elif x==2:
        St=((St[0] + B[0])/2, (St[1] + B[1])/2)
        prev=1
    elif x==3:
        St=((St[0] + C[0])/2, (St[1] + C[1])/2)
        prev=2
    else:
        St=((St[0] + D[0])/2, (St[1] + D[1])/2)
        prev=3
    pygame.draw.circle(DISPLAYSURF, GREEN, St, 0, 0)


    for event in pygame.event.get():
        if event.type==QUIT:
            pygame.image.save(DISPLAYSURF, "square2.png")
            pygame.quit()
            sys.exit()
    pygame.display.update()
```

Note that, in the restriction, if instead of not allowing the vertex lying 1 place away in the anti-clockwise direction from the previously chosen vertex, we do not chose the vertex 1 place away in the clockwise dorection, we get the same image as Figure 6.

4.3 Type III

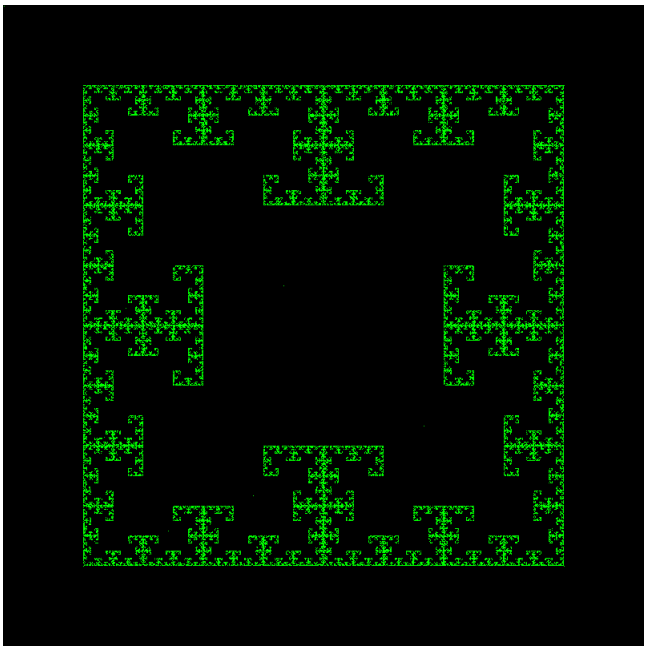

Figure 7: Sierpinski's Square Type **III**

To construct this above figure, we select 4 vertices of a regular rectangle and a random starting point as usual. Then with each iteration, we select a random vertex and plot a point midway between the previous point and the current chosen vertex, with the restriction that this vertex must not be 2 places away from the previously chosen vertex. In the pygame code, we keep every thing same with slight alterations inside the while loop. Listed below is the main body of the loop we need, to generate Figure 7.

```
while True:
    l=range(1, 5)
    l.remove(prev)
    x=random.choice(l)
    if x==1:
        St=((St[0] + A[0])/2, (St[1] + A[1])/2)
        prev=3
    elif x==2:
        St=((St[0] + B[0])/2, (St[1] + B[1])/2)
        prev=4
    elif x==3:
        St=((St[0] + C[0])/2, (St[1] + C[1])/2)
        prev=1
```

```
else:
    St=((St[0] + D[0])/2, (St[1] + D[1])/2)
    prev=2
pygame.draw.circle(DISPLAYSURF, GREEN, St, 0, 0)


for event in pygame.event.get():
    if event.type==QUIT:
        pygame.image.save(DISPLAYSURF, "square3.png")
        pygame.quit()
        sys.exit()
pygame.display.update()
```

5 Sierpinksi's Hexagon

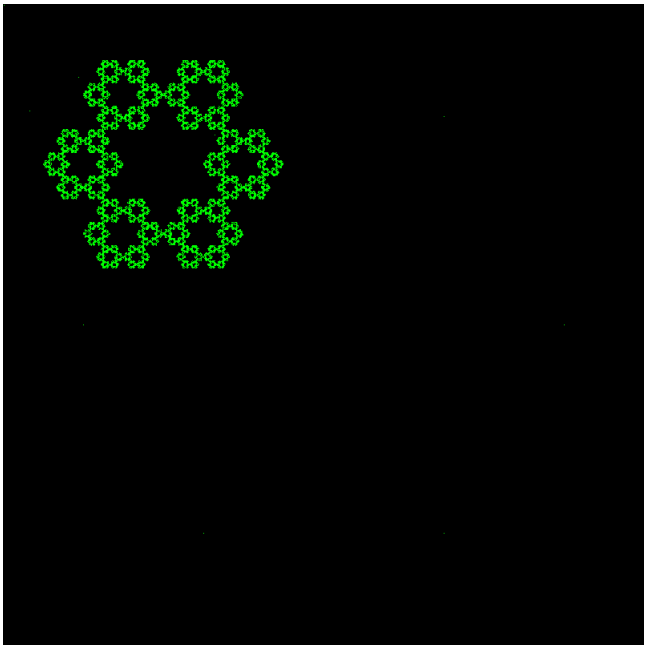

Figure 8: Sierpinski's Hexagon

To construct Figure 8, we choose 6 vertices of a regular hexagon and a random starting point. here, **n**=6 and distance factor **r**=$\frac{1}{3}$. Next, with each iteration, we plot the points and simulate the frcatal pattern. Listed below is the python code.

```
import random, pygame, sys
```

```
from pygame.locals import *

#set up the window
DISPLAYSURF=pygame.display.set_mode((800, 800), 0, 32)
pygame.display.set_caption('Sierpinski_hexagon')

#set up the colors
GREEN=(0, 255, 0)

#set up the points for sierpinski's hexagon
A=(250, 140)
B=(100, 400)
C=(250, 660)
D=(550, 660)
E=(700, 400)
F=(550, 140)
St=(1, 1)

pygame.draw.circle(DISPLAYSURF, GREEN, A, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, B, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, C, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, D, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, E, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, F, 0, 0)
pygame.draw.circle(DISPLAYSURF, GREEN, St, 0, 0)

#run the loop
while True:
    x=random.randint(1, 6)
    if x==1: St=((St[0] + A[0])/3, (St[1] + A[1])/3)
    elif x==2: St=((St[0] + B[0])/3, (St[1] + B[1])/3)
    elif x==3: St=((St[0] + C[0])/3, (St[1] + C[1])/3)
    elif x==4: St=((St[0] + D[0])/3, (St[1] + D[1])/3)
    elif x==5: St=((St[0] + E[0])/3, (St[1] + E[1])/3)
    else: St=((St[0] + F[0])/3, (St[1] + F[1])/3)
    pygame.draw.circle(DISPLAYSURF, GREEN, St, 0, 0)

    for event in pygame.event.get():
        if event.type==QUIT:
            pygame.image.save(DISPLAYSURF, "hex.png")
            pygame.quit()
            sys.exit()
    pygame.display.update()
```

6 Barnsley's fern

Like Sierpinski's gasket, this fern is another one of the basic fractal patterns, and named after the same person, Michael Barnsley we talked about earlier. The computer code that creates this pattern is also an example of IFS (iterated function system), like the ones we have beeen talking throughout. The Barnsley's fern shows, how one can built graphically beautiful structues from iterative use of mathematical formulas. The fern code developed by Barnsley also follows from the *collage theorem*. Barnsley's fern uses four affine transformations, each of which looks like this:

$$f(x,y) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

Here **a** to **f** are the coefficients used in the affine transformations. We show three types of fractal leaves, their respective structues, affine transformations and Pygame programs to simulate their growth.

6.1 Type *Black Spleenwort*

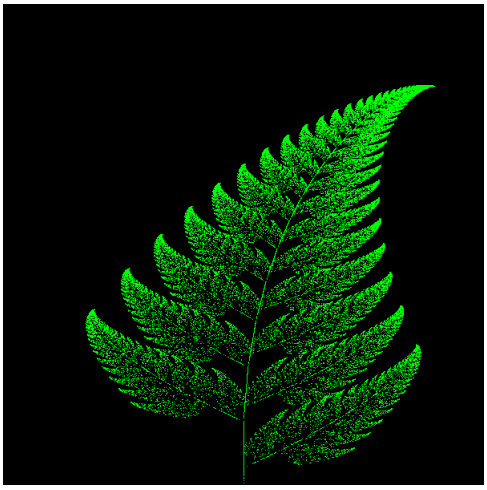

Figure 9: Barnsley's Fern type: ***Black Spleenwort***

To construct the fractal leaf, we require these transformations:

$$f_1(x,y) = \begin{bmatrix} 0.00 & 0.00 \\ 0.00 & 0.16 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$f_2(x,y) = \begin{bmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix}$$

$$f_3(x,y) = \begin{bmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix}$$

$$f_4(x,y) = \begin{bmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 0.44 \end{bmatrix}$$

In simulating the fractal growth, the first point is drawn at the origin ($X_0 = 0$, $Y_0 = 0$) and the next points are iteratively plotted by randomly choosing one of the above four transformations. After continuing for a large number of iterations the fractal pattern starts to pop up and the beauty of mathematical simulations is revealed.

• f_1: This transformation is chosen 1% of the times and maps the base of the stem of the leaf

• f_2: This transformation is chosen 85% of the times and maps the succesive smaller leaflets

• f_3: This transformation is chosen 7% of the times and maps the largest left-hand leaflet

• f_4: This transformation is chosen 7% of the times and maps the largest right-hand leaflet

This pattern is very popular among mathematicians, scientists, graphic designers and even video game developers. In designing video games, if a developer needs to simulate leaves, this technique is used often. In our code, we first need to remember that in a pygame surface the top-left corner is coordinated $(0, 0)$ and the the value of Y-axis increases downwards. We also need to remember that, pygame does not allow float values. So to get a convinient simulation, we need to map the pygame coordinates to a new coordinate sytem that suits our needs. Listed below is the pygame program that that plots the fractal pattern. Also here we use a 600 X 600 drawing surface.

```
import random, pygame, sys
from pygame.locals import *

#set up the window
DISPLAYSURF=pygame.display.set_mode((600, 600))
pygame.display.set_caption("Barnsley's fern: Black Spleenwort")

#set up the colors
GREEN=(0, 255, 0)

#set up the initial points
```

```
X=[0.0]
Y=[0.0]
i=0
#run the loop
while True:
    r=random.uniform(0, 1) #choosing a float between 0 and 1
    if r<=0.01:
        X+=[0.0, ]
        Y+=[0.16*Y[i], ]
    elif r>0.01 and r<=0.86:
        X+=[0.85*X[i] + 0.04*Y[i], ]
        Y+=[-0.04*X[i] + 0.85*Y[i] + 1.6, ]
    elif r>0.86 and r<=0.93:
        X+=[0.2*X[i] - 0.26*Y[i], ]
        Y+=[0.23*X[i] + 0.22*Y[i] + 1.6, ]
    else:
        X+=[-0.15*X[i] + 0.28*Y[i], ]
        Y+=[0.26*X[i] + 0.24*Y[i] + 0.44, ]
    pygame.draw.circle(DISPLAYSURF, GREEN, (int(X[i]*90 + 300),
600 - int(Y[i]*50)), 0, 0)
    i+=1

    for event in pygame.event.get():
        if event.type==QUIT:
            pygame.image.save(DISPLAYSURF, "fern.png")
            pygame.quit()
            sys.exit()
    pygame.display.update()
```

6.2 Type *Thelypteridaceae*

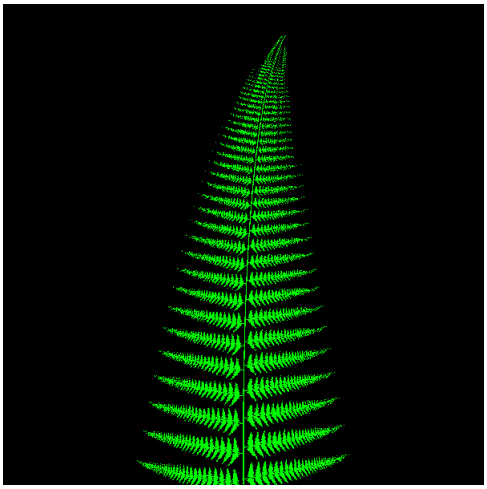

Figure 10: Barnsley's Fern type: ***Thelypteridaceae***

To construct the fractal leaf, we require these transformations:

$$f_1(x,y) = \begin{bmatrix} 0.00 & 0.00 \\ 0.00 & 0.25 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ -0.4 \end{bmatrix}$$

$$f_2(x,y) = \begin{bmatrix} 0.95 & 0.005 \\ -0.005 & 0.93 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.002 \\ 0.5 \end{bmatrix}$$

$$f_3(x,y) = \begin{bmatrix} 0.035 & -0.2 \\ 0.16 & 0.04 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.09 \\ 0.02 \end{bmatrix}$$

$$f_4(x,y) = \begin{bmatrix} -0.04 & 0.2 \\ 0.16 & 0.04 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.083 \\ 0.12 \end{bmatrix}$$

- f_1: This transformation is chosen 2% of the times
- f_2: This transformation is chosen 84% of the times
- f_3: This transformation is chosen 7% of the times
- f_4: This transformation is chosen 7% of the times

Listed below is the python code to simulate the frcatal growth:

```
import random, pygame, sys
from pygame.locals import *

#set up the window
DISPLAYSURF=pygame.display.set_mode((600, 600),)
pygame.display.set_caption("Barnsley's fern: Thelypteridaceae")

#set up the colors
GREEN=(0, 255, 0)

#set up the initial points
X=[0.0]
Y=[0.0]
i=0
#run the loop
while True:
    r=random.uniform(0, 1) #choosing a float between 0 and 1
    if r<=0.02:
        X+=[0.0, ]
        Y+=[0.25*Y[i] - 0.4, ]
    elif r>0.02 and r<=0.86:
        X+=[0.95*X[i] + 0.005*Y[i] - 0.002, ]
        Y+=[-0.005*X[i] + 0.93*Y[i] + 0.5, ]
    elif r>0.86 and r<=0.93:
        X+=[0.035*X[i] - 0.2*Y[i] - 0.09, ]
        Y+=[0.16*X[i] + 0.04*Y[i] + 0.02, ]
    else:
        X+=[-0.04*X[i] + 0.2*Y[i] + 0.083, ]
        Y+=[0.16*X[i] + 0.04*Y[i] + 0.12, ]

    pygame.draw.circle(DISPLAYSURF, GREEN, (int(X[i]*90 + 300),
 600 - int(Y[i]*80)), 0, 0) # Plotting the points after required transformations
    i+=1

    for event in pygame.event.get():
        if event.type==QUIT:
            pygame.image.save(DISPLAYSURF, "fern2.png")
            pygame.quit()
            sys.exit()
    pygame.display.update()
```

6.3 Type *Leptosporangiate*

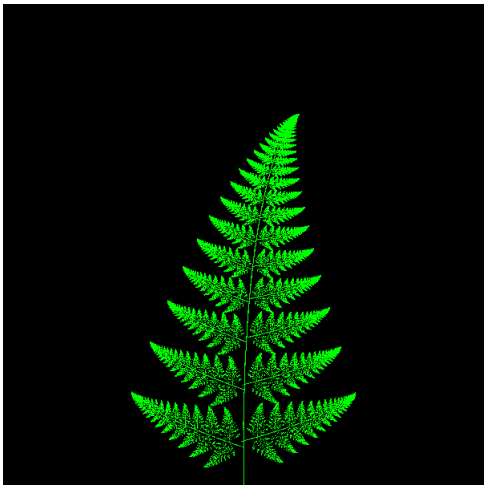

Figure 11: Barnsley's Fern type: ***Leptosporangiate***

To construct the fractal leaf, we require these transformations:

$$f_1(x,y) = \begin{bmatrix} 0.00 & 0.00 \\ 0.00 & 0.25 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ -0.14 \end{bmatrix}$$

$$f_2(x,y) = \begin{bmatrix} 0.85 & 0.02 \\ -0.02 & 0.83 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.0 \end{bmatrix}$$

$$f_3(x,y) = \begin{bmatrix} 0.09 & -0.28 \\ 0.3 & 0.11 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 0.6 \end{bmatrix}$$

$$f_4(x,y) = \begin{bmatrix} -0.09 & 0.28 \\ 0.3 & 0.09 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.0 \\ 0.7 \end{bmatrix}$$

- f_1: This transformation is chosen 2% of the times
- f_2: This transformation is chosen 84% of the times
- f_3: This transformation is chosen 7% of the times
- f_4: This transformation is chosen 7% of the times

Listed below is the code written in pylab to simulate the fractal growth:

```
import random, pygame, sys
from pygame.locals import *

#set up the window
DISPLAYSURF=pygame.display.set_mode((600, 600),)
pygame.display.set_caption("Barnsley's fern: Leptosporangiate")

#set up the colors
GREEN=(0, 255, 0)

#set up the initial points
X=[0.0]
Y=[0.0]
i=0
#run the loop
while True:
    r=random.uniform(0, 1) #choosing a float between 0 and 1
    if r<=0.02:
        X+=[0.0, ]
        Y+=[0.25*Y[i] - 0.14, ]
    elif r>0.02 and r<=0.86:
        X+=[0.85*X[i] + 0.02*Y[i], ]
        Y+=[-0.02*X[i] + 0.83*Y[i] + 1.0, ]
    elif r>0.86 and r<=0.93:
        X+=[0.09*X[i] - 0.28*Y[i], ]
        Y+=[0.3*X[i] + 0.11*Y[i] + 0.6, ]
    else:
        X+=[-0.09*X[i] + 0.28*Y[i], ]
        Y+=[0.3*X[i] + 0.09*Y[i] + 0.7, ]
    pygame.draw.circle(DISPLAYSURF, GREEN, (int(X[i]*90 + 300),
600 - int(Y[i]*80)), 0, 0) # Plotting the points after required transformations
    i+=1

    for event in pygame.event.get():
        if event.type==QUIT:
            pygame.image.save(DISPLAYSURF, "fern3.png")
            pygame.quit()
            sys.exit()
    pygame.display.update()
```

Articles referred: Wikipedia and Wolfram Mathworld.