Analysis of *Chaos Game* simulations with Pygame

Indranil Ghosh

Jadavpur University, Department of Physics

indranilg49@gmail.com — +91 7550 860 174

Abstract

This project is based on the dynamic simulations of various Chaos Game algorithms to produce fractal patterns, with the help of Pygame. Patterns to be discussed are Sierpinskis Triangle, Barnsleys fern and few restricted chaos game fractals. The Chaos game makes use of a random process to produce visualizations of self-similar fractal patterns on a plane. In this project some of the few fractal patterns, each with a description , its own chaos game rule and Pygame codes, to simulate its development are listed. Now, Pygame is a cross-platform set of python modules to create interactive video-games. These Pygame simulators make the visualizations of the pattern-generation grow with time, and very beautiful to look at. I also discuss some of the norms to be followed while using Pygame and its applications in scientific programming.

Introduction

The algorithm of *Chaos Game* was first developed by the british mathematician, **Michael Barnsley** around the year 1988, which produced some interesting fractal patterns. Generally, it refers to the method of generating the fixed point (attractor) of an iterated function system(IFS). Using the algorithm, the pattern is produced 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 infinite complexity and self similarity. Zooming a particular section of the fractal, results in the same pattern but with less density of points.

- When the simulations are carried out, the fractal patterns grow and become clearer with time, with each iterations. If the n-gon is regular, the pattern formed is symmetric.

Sierpinski's Triangle

Coined by the Polish mathematician **Waclaw Sierpinski**, this triangle needs 3 random vertces and a random starting point to start with the simulation. The distance factor **r** is $\frac{1}{2}$. Let A, B and C be the 3 random vertices of a triangle and St be the starting point, also chosen randomly on the drawing platform. Playing with rolling a dice, we come up with any random integer in the range [1, 6] with an equal probability $\frac{1}{6}$. We design the game in such a way that, if we come up with a face **1 or 2**, we move towards point **A** from the previous point and plot a new point halfway between. SImilarly for faces **3 or 4** or **5 or 6**, we move halfway towards **B** or **C** respectively. The process continued for a large number of iterations results in Figure 1.

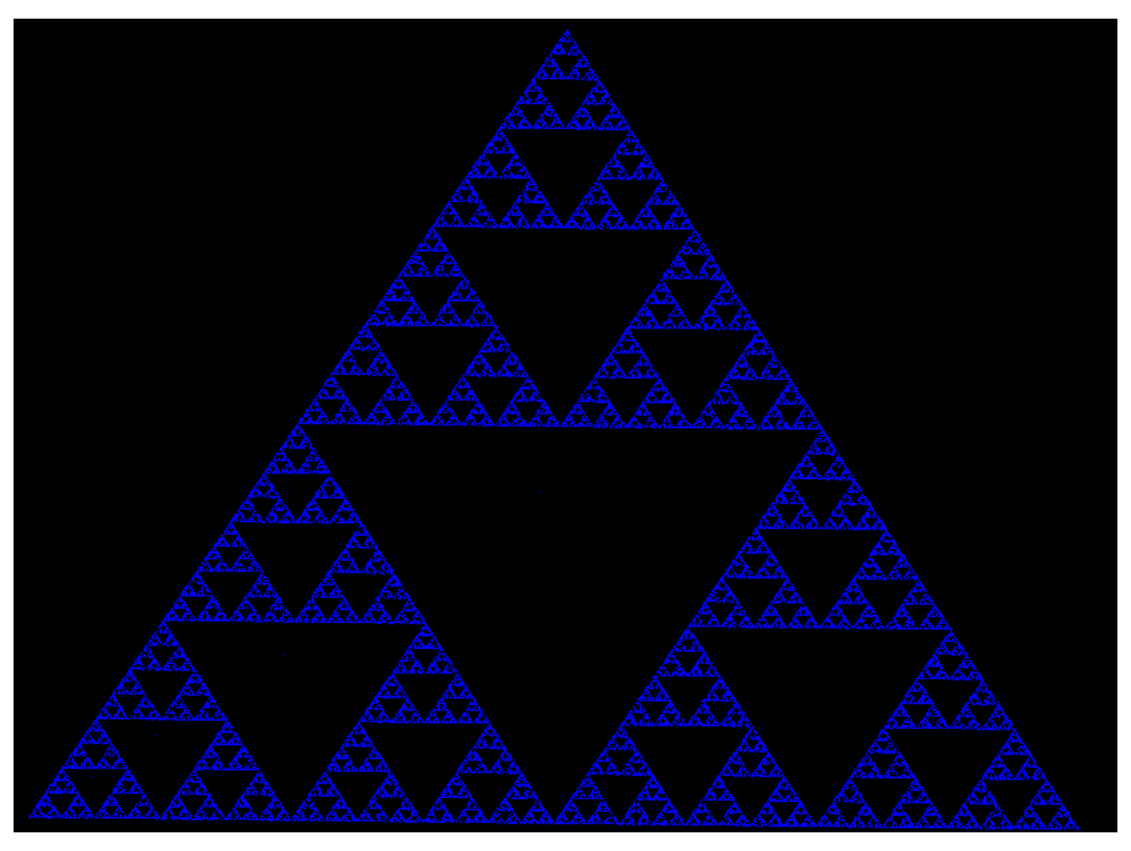

Figure 1: Simulation of Sierpinski's Triangle

Code snippet

```
import random, pygame, sys
from pygame.locals import∗
#set up the window
DISPLAYSURF=pygame.display.set_mode((800, 800))
#set up the colors
BLACK=(0, 0, 0)
BLUE=(0, 0, 255)
i=0
while True:
    for event in pygame.event.get():
        if event.type==QUIT:
            pygame.image.save(DISPLAYSURF, ”Sierpinski.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,BLUE,A,0,0)
            elif i==2:
                B=(event.pos[0], event.pos[1])
                pygame.draw.circle(DISPLAYSURF,BLUE,B,0,0)
            elif i==3:
                C=(event.pos[0], event.pos[1])
                pygame.draw.circle(DISPLAYSURF,BLUE,C,0,0)
            elif i==4:
                St=(event.pos[0], event.pos[1])
                pygame.draw.circle(DISPLAYSURF,BLUE,St,0,0)
            else:
                pygame.quit()
                sys.exit()
    if i==4:
        x=random.randint(1, 6)
        if x in [1, 2]: St=((St[0]+A[0])//2, (St[1]+A[1])//2)
        elif x in [3, 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, BLUE, St, 0, 0)
    pygame.display.update()
```

- The Hausdorff Dimension of Sierpinski's Triangle is 1.5849.

number of vertices=5, r=$\frac{1}{2}$

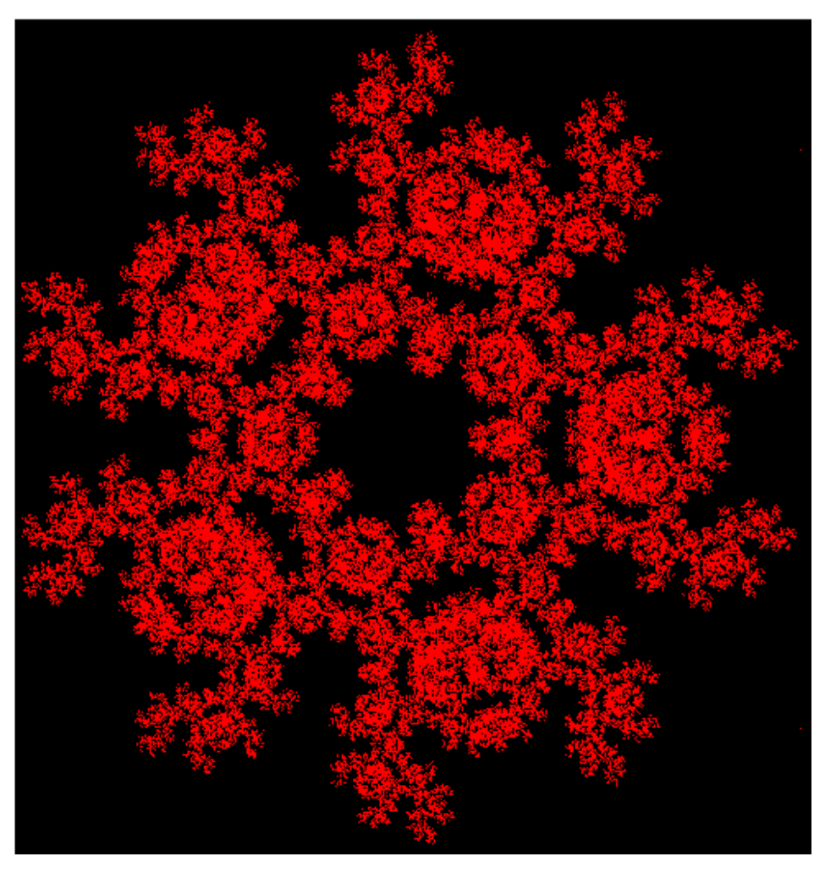

Figure 2: A restricted Pentagon

number of vertices=6, r=$\frac{1}{3}$

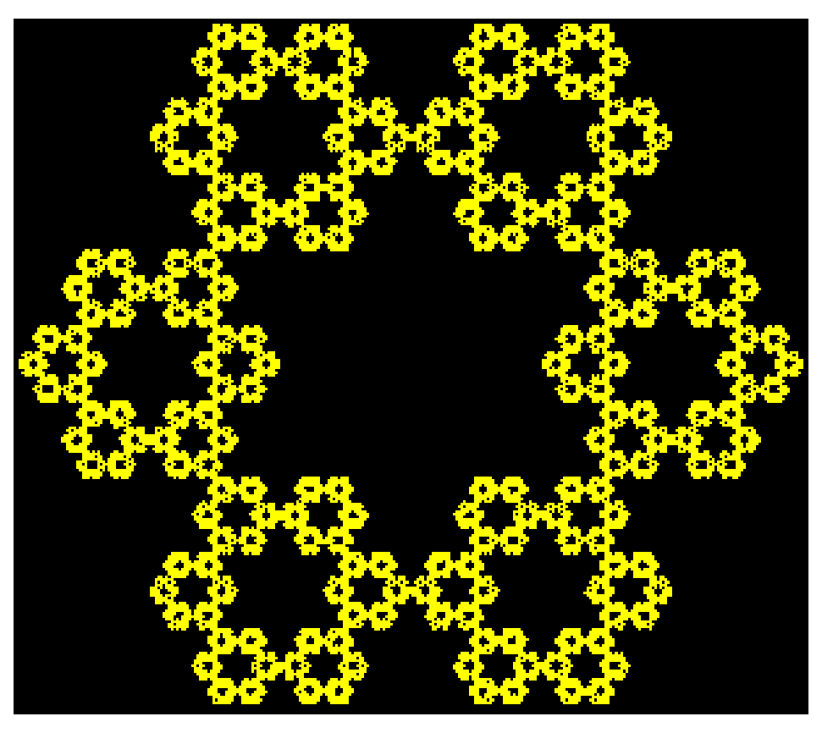

Figure 3: A Hexagon

- The Hausdorff Dimension of this pattern is 1.6309.

Barnsley's fern

- Devised by Michael Barnsley, the computer code that simulates this pattern is also an example of IFS (iterated Function System).
- The algorithm developed by Barnsley also follows from the *collage theorem.*
- To construct the leaf, we need these four affine transformations:

$$f_1(x,y) = \begin{bmatrix} 0.00 & 0.00 \\ 0.00 & 0.16 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{1}$$

$$f_2(x,y) = \begin{bmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix} \tag{2}$$

$$f_3(x,y) = \begin{bmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix} \tag{3}$$

$$f_4(x,y) = \begin{bmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 0.44 \end{bmatrix} \tag{4}$$

- In simulating the growth, the first point is drawn at the origin ($X_0 = 0, Y_0 = 0$) and the successive points are iteratively plotted by randomly choosing one of the above four transformations.
- f_1 transformation is chosen 1% of the times and maps the base of the stem of the leaf, f_2 is chosen 85% of the times and maps the smaller leaflets, f_3 and f_4 are each chosen 7% of the times and maps the largest left-handed and the largest right-handed leaflet respectively.

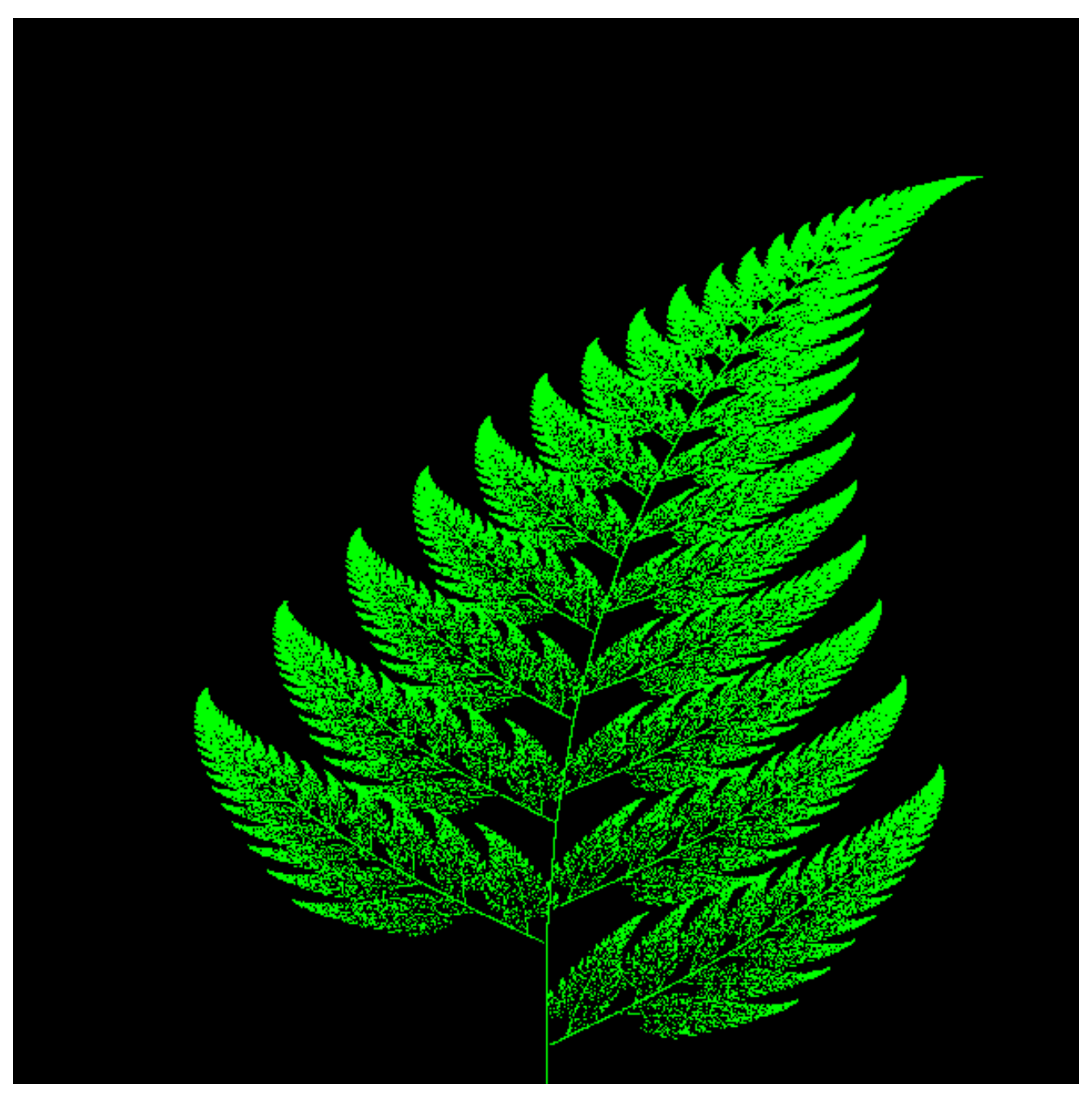

Figure 4: Barnsley's Fern, type: ***Black Spleenwort***

Code Snippet

```
GREEN=(0, 255, 0)

X=[0.0]
Y=[0.0]
i=0

while True:
    r=random.random()
    if r<=0.02:
        X+=[0.0, ]
        Y+=[0.16∗Y[i], ]
    elif r<=0.86:
        X+=[0.85∗X[i] + 0.04∗Y[i], ]
        Y+=[−0.024∗X[i] + 0.85∗Y[i] + 1.6, ]
    elif r<=0.93:
        X+=[0.20∗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
```

Yet Another Fern

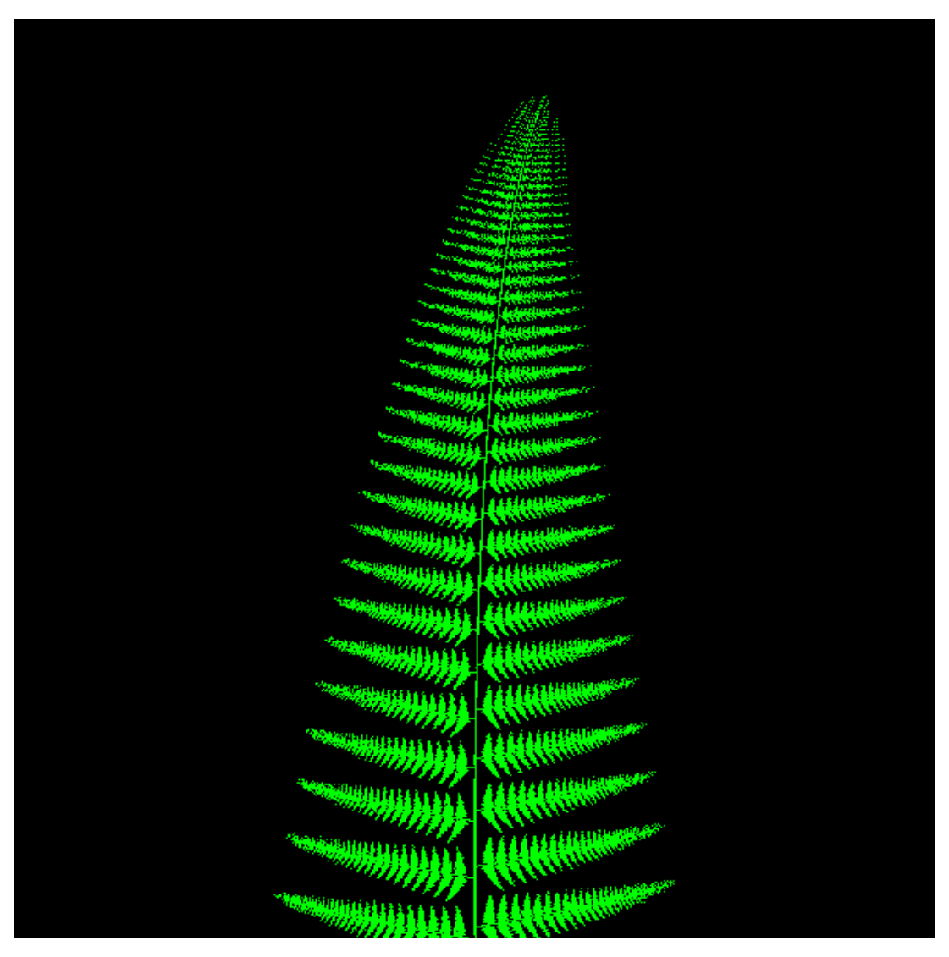

Figure 5: Barnsley's Fern, type: ***Thelypteridaceae***

Conclusions

- In our code, we need to keep in mind that the top left corner is coordinated (0, 0) in a pygame drawing surface and the value of Y-axis increases downwards.
- Pygame does not allow floating point values. So, to get a convinient simulation, we need to map the pygame coordinates to a new coordinate system that suits our needs.
- The dynamic simulations of fractals have turned out to be useful in scientific applications ranging from computer graphics, image compression, mathematical modeling, in video game industries, etc.

Acknowledgement

I would like to express my special thanks of gratitude to my Professors, Dr. Debasish Lohar and Dr. Subhankar Ray who helped me with useful suggestions.

References

[1] Rubin H. Landau, Manuel Jose Paez and Christian C. Bordeianu, *"Computational Physics"*, New York: Wiley, 2007

[2] Al Sweigart, *"Making Games with Python and Pygame"*, https://github.com/indrag49/Fractals-pygame

[3] Wolfram Mathworld, *"Chaos Game"*, http://mathworld.wolfram.com/ChaosGame.html

[4] Wikipedia, *"Barnsley's Fern"*, https://en.wikipedia.org/wiki/Barnsleyfern

[5] Indranil Ghosh, *"Fractals-pygame"*, https://github.com/indrag49/Fractals-pygame