Koch Snowflakes simulation using Python C++ Mixed Programming

Javid Ibrahimov
Mathematics, Computer, and Information Sciences
Mississippi Valley State University
Itta Bena, USA
javid.ibrahimov@mvsu.edu

***Abstract* - Fractals are very common in math and in the natural world and are used extensively in industry and science. It is well known that C++ is an object-oriented language and efficient for computation, and Python is a powerful computer language. In this project, we will use the speed of computing of C++ and Python graphics packages to simulate snowflake fractals.**

Keywords – snowflake fractals, object-oriented, Python, C++

I. INTRODUCTION

Fractals are fascinating mathematical objects that exhibit self-similarity at different scales. They have been studied for decades in the fields of mathematics, physics, computer science, and other areas of science. Fractal geometry has many applications in real-world problems, such as in computer graphics, image compression, and signal processing.

Snowflakes are a beautiful example of natural fractals that are formed from the unique combination of temperature and humidity in the atmosphere. Studying the structure of snowflakes can provide insights into the physics of crystal growth and can be used better to understand the behavior of fluids in extreme conditions.

In this project, the aim is to create a simulation of snowflakes using the Koch Snowflake and Minkowski Sausage algorithms, famous fractal patterns that resemble the different shapes of a snowflake. Our approach involves using Python and C++ programming languages. Since C++ is an object-oriented language, efficiency, and readability are better than Python. Also, since Python is easier and better than C++ for visualization, we take advantage of these two languages to implement the algorithms and visualize the results.

Overall, this project aims to demonstrate the beauty and complexity of fractal geometry.

II. BACKGROUND OF FRACTALS AND APPLICATIONS

A. Importance of Fractals

Fractals are fascinating mathematical objects that have a wide range of applications in various fields of study. From art and design to science and technology, fractals have become an important tool for understanding complex systems and phenomena.

One of the most notable applications of fractals is in computer graphics. Fractal algorithms have revolutionized how images and animations are created, allowing for detailed and intricate designs that were previously difficult to achieve. Fractal images are created by repeating simple patterns at different scales, resulting in complex and beautiful structures that are inherently self-similar. The scalability of fractal images means they can be resized without losing quality, making them ideal for large-format prints or displays.

Fractals are also used for data compression, enabling large amounts of data to be compressed without losing any information. This technique is particularly useful in the compression of images and videos, significantly reducing file sizes without sacrificing quality. Fractal compression algorithms analyze the data and identify self-similarities or repeating patterns, which can be encoded using a smaller amount of data. This makes fractal compression an efficient and effective method for data storage and transmission.

In signal processing, fractals are used to analyze and manipulate signals such as sound and radio waves. Fractal analysis can reveal hidden patterns and structures within a signal, making it useful in areas such as speech recognition, image processing, and even financial forecasting. By understanding the underlying structure of a signal, it is possible to extract important information that may not be apparent otherwise.

B. Application of Fractals for Image Compression

Fractals are mathematical patterns that repeat themselves at various scales, which makes them perfect for digital image compression. Fractal compression methods find patterns in photos and save them as equations that can be used to accurately

reproduce the original image. Smaller file sizes produced by this approach enable faster network transmission without sacrificing crucial information.

Fractal compression's ability to compress images without noticeably losing information is one of its main benefits. Conventional picture compression algorithms, like JPEG, might result in a loss of quality since they discard certain information to minimize the file size. Fractal compression, on the other hand, recreates the image using mathematical equations, preserving crucial aspects while minimizing the amount of data required to represent the image.

Fractal compression techniques locate patterns at various scales and repeat them, significantly reducing the amount of information required to represent an image. These patterns might be straightforward, like a repeating form, or more intricate, like a texture. Fractal compression can achieve significant levels of compression while keeping image quality by employing these patterns to represent the image.

Fractal compression results in smaller files, which has various advantages, including quicker network transmission. This is crucial in applications like satellite imaging, medical imaging, and video streaming, where file size and transmission speed are crucial. These applications frequently employ fractal compression since it speeds up the transmission of huge files without significantly reducing image quality.

C. Application of Fractals for Antenna Design

A potential technique in the area of wireless communication is fractal antenna designs. Because of their capacity to enhance antenna performance and reduce size, they are frequently employed in applications such as mobile devices, Wi-Fi networks, and satellite communications.

The performance of wireless communication systems can be greatly enhanced by the use of fractal geometry in antenna design. Due to their large bandwidth and ability to operate at numerous frequencies, fractal antennas are well-suited for a variety of wireless communication requirements. This contrasts with conventional antenna designs, which could have a small bandwidth and take up a lot of room.

Because fractal antennas rely on self-similarity, the size of the antenna has no effect on how well it performs. This characteristic is especially helpful in small devices like smartphones and wearable technology, where size is an important consideration.

The ability of fractal antenna designs to achieve wide bandwidth is one of their main advantages. The capacity of conventional antennas to broadcast and receive signals across a large frequency range is constrained by the fact that they frequently function at a single frequency. Fractal antennas, on the other hand, may function at numerous frequencies, which makes them perfect for applications with high bandwidth needs like Wi-Fi networks.

In satellite communication systems, where the antenna must be built to function in a range of different frequency bands, fractal antennas are also helpful. The antenna can be created to work at many frequencies utilizing fractal geometry, enabling more effective communication.

III. ALGORITHMS AND IMPLEMENTATIONS

Two common fractals are the Koch Snowflake and the Minkowski Sausage.

The Koch Snowflake is recursively used by the following algorithm involves starting with a segment and dividing into 3 equal parts recursively adding smaller equilateral triangles to each of its sides. The degree used to form each equilateral triangle is 60 degrees. At each iteration, three new equilateral triangles are added to each of the three sides of the original triangle, resulting in a total of 12 new triangles each time. The process can be repeated indefinitely, creating a more complex shape with each iteration. The resulting shape is a fractal known as the Koch Snowflake. The Koch Snowflake is an example of a self-similar fractal, meaning it appears similar at different scales. This property makes it an ideal tool for implementing fractals in various applications, such as antenna design and image compression.

The Minkowski Sausage method involves starting with a straight line and repeatedly adding squares to each of its sides. The degree used to form each square is 90 degrees. At each iteration, four new squares are added to each of the four sides of the original line, resulting in a total of 16 new squares each time. The process can also be repeated indefinitely, creating a more complex shape with each iteration. The resulting shape is a fractal known as the Minkowski Sausage. The Minkowski Sausage is an example of a space-filling fractal, meaning that it can fill a given area without leaving any gaps. This property makes it useful for implementing fractals in applications such as data compression and generating random numbers.

A. Algorithm of Koch Snowflake

The algorithm is the following:

1. Starts with the segment P_0P_4.
2. Divide P_0P_4 into three equal parts to get points P_1, P_3.
3. Rotate the segment P_1P_3 through $\frac{\pi}{3}$ to get point P_2.
4. Repeat 1 for $P_0P_1, P_1P_2, P_2P_3, P_3P_4$.

Now, we can easily calculate expressions P_1, P_2, P_3 in terms of P_0 and P_4. In fact

$$P_1 = \frac{2}{3}P_0 + \frac{1}{3}P_4$$

$$P_2 = P_1 + \frac{c}{3}(P_4 - P_0)$$

$$P_3 = \frac{1}{3}P_0 + \frac{2}{3}P_4$$

where $c = \exp\left(\frac{\pi i}{3}\right)$.

B. Algorithm of Minkowski Sausage

1. Starts with the segment P_0P_8.
2. Divide P_0P_8 into four equal parts to get P_1, P_4, P_7.
3. Rotate P_1P_4 through $\frac{\pi}{2}$ to get point P_2.
4. Rotate P_4P_7 through $\frac{\pi}{2}$ to get point P_3
5. Rotate P_4P_7 through $-\frac{\pi}{2}$ to get point P_5.
6. Rotate P_7P_4 through $-\frac{\pi}{2}$ to get point P_6.
7. Repeat 1 for $P_0P_1, P_1P_2, P_2P_3, P_3P_4, P_4P_5, P_5P_6, P_6P_7, P_7P_8$.

$$P_1 = \frac{3}{4}P_0 + \frac{1}{4}P_8$$

$$P_4 = \frac{2}{4}P_0 + \frac{2}{4}P_8$$

$$P_7 = \frac{1}{4}P_0 + \frac{3}{4}P_8$$

$$P_2 = P_1 + (P_4 - P_1)i$$

$$P_3 = P_2 - (P_4 - P_1)i$$

$$P_5 = P_4 - (P_7 - P_4)i$$

$$P_6 = P_5 - (P_7 - P_4)i$$

C. Implementation of Koch Snowflake

Now we use the C++ STL library to implement the algorithm:

```
vector<complex<double>> KochFlake::koch_flakes()
{
   vector<complex<double>> new_segment;
   complex<double> point[5];

   point[0] = c1;
   point[4] = c2;
   point[1] = 2.0/3*point[0] + 1.0/3*point[4];
   point[3] = 1.0/3*point[0] + 2.0/3*point[4];
   point[2] = point[1] + const3*(point[3] - point[1]);

   for(int n = 0; n < 5; n++)
  {
      new_segment.push_back(point[n]);
   }
   return new_segment;

vector<complex<double>>
generate_koch(vector<complex<double>> segments)
{
   int size = segments.size();
   vector<complex<double>> new_segments, tmp;
   complex<double> c1, c2;

for (int n = 0; n < size-1; n++) {
      c1 = segments[n];
      c2 = segments[n+1];
      KochFlake min = KochFlake(c1, c2);
      tmp = min.koch_flakes();
      new_segments.insert(new_segments.end(),
tmp.begin(), tmp.end());
   }

   return new_segments;
}
```

D. Implementation of Minkowski Sausage

Similarly we have the following C++ code:

```
vector<complex<double>> KochFlake::koch_Quadratic()
{
   vector<complex<double>> new_segment;

   complex<double> point[9];

   point[0] = c1;
   point[8] = c2;
   point[1] = 3.0/4*point[0] + 1.0/4*point[8];
   point[4] = 2.0/4*point[0] + 2.0/4*point[8];
   point[7] = 1.0/4*point[0] + 3.0/4*point[8];
   point[2] = point[1] + 1i*(point[4] - point[1]);
   point[3] = point[2] + (point[4] - point[1]);
   point[5] = point[4] - 1i*(point[7] - point[4]);
   point[6] = point[5] + (point[7] - point[4]);

   for(int n = 0; n < 9; n++) {

      new_segment.push_back(point[n]);
   }

   return new_segment;
}

vector<complex<double>>
generate_kochQuadratic(vector<complex<double>>
segments) {
   int size = segments.size();
   vector<complex<double>> new_segments, tmp;
   complex<double> c1, c2;

   for (int n = 0; n < size-1; n++) {
      c1 = segments[n];
      c2 = segments[n+1];
      KochFlake min = KochFlake(c1, c2);
      tmp = min.koch_Quadratic();
      new_segments.insert(new_segments.end(),
tmp.begin(), tmp.end());
   }

   return new_segments;
`
```

E. Python code to run the program

We use Python package cppyy to link C++ code with matplotlib library:

```
import cppyy
import numpy as np
import matplotlib
matplotlib.use('TkAgg')
from matplotlib import pyplot as plt
import time
from cppyy.gbl.std import vector, pair, complex
cppyy.include("koch.h")
cppyy.load_library("libkoch.so")

#from cppyy.gbl import generate_new_segments,
generate_snowflakes, generate_cesaro_fractal,
generate_nflakes
from cppyy.gbl import generate_koch,
generate_kochQuadratic

plt.xlim(-3, 3)
plt.ylim(-3, 3)

c1 = 0+0.j
c2 = 1+0.j
c3 = 1-1j
c4 = -1j
segment1 =[c1,c2,c3,c4,c1]
level = 6
for i in range(level):
  #segment1 = generate_koch(segment1)
    segment1 = generate_kochQuadratic(segment1)
    tmp1 = []

    size = len(segment1)

    for i in range(size):
        tmp1.append(segment1[i])

    tmp1 = np.array(tmp1)

    plt.plot(tmp1.real,tmp1.imag)

    plt.axis((-0.3 , 1.2 , -0.4 , 1.2))
    plt.draw()
    plt.pause(0.5)
    plt.clf()
    plt.gca().set_aspect('equal')


plt.show()
```


ACKNOWLEDGMENT

The author expresses his sincere appreciation to the Department of Mathematics, Computer and Information Sciences at Mississippi Valley State University for their invaluable support and resources during this project. Special thanks to Dr. Xiaoqin Wu for his guidance and expertise, and to department chair Latonya Garner-Jackson for her unwavering support.

REFERENCES

[1] Texas Instruments Europe, "An Introduction to Fractal Image Compression," literature number BPRA065, Oct. 1997..

[2] C. P. Baliarda, "The Fractal Antennas Gallery," IEEE Antennas and Propagation Magazine, vol. 40, no. 1, pp. 42-57, Apr. 1998, doi: 10.1109/74.660941.

[3] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," in Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, May/June 2007, doi: 10.1109/MCSE.2007.55.

[4] J. Shlens, "Cppyy: Efficient Python bindings for C++," 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Brighton, United Kingdom, 2019, pp. 6606-6610, doi: 10.1109/ICASSP.2019.8683184.

[5] S. Van der Walt, S. C. Colbert and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," Computing in Science & Engineering, vol. 13, no. 2, pp. 22-30, Mar.-Apr. 2011, doi: 10.1109/MCSE.2011.37.

[6] G. Powell, "The Vector C++ Library: A Parallel Programming Model with High-Level Syntax," in Proceedings of the 2012 International Conference on High Performance Computing & Simulation (HPCS), Madrid, Spain, 2012, pp. 477-485, doi: 10.1109/HPCSim.2012.6266975.

[7] C. S. Oliveira, G. A. Pagot, and A. M. A. Carvalho, "Complex C++: A C++ Library for Complex Number Arithmetic," in Proceedings of the 2017 IEEE International Symposium on Circuits and Systems (ISCAS), Baltimore, MD, 2017, pp. 1-4, doi: 10.1109/ISCAS.2017.8050673.