The Chaos Game

The chaos game is a random process that can be used to create deterministic results. Consider the following example: Pick a random point in the triangle and repeat: Pick a random corner (A, B, or C). Move to another point halfway to that corner. Try this in Python or Netlogo. What happens?

The Chaos Game you described is a fascinating process that uses simple rules to generate complex patterns, often resulting in fractal structures like the Sierpinski Triangle. The algorithm's simplicity belies the depth and beauty of the patterns it can create.

Let's implement this in Python to observe the outcomes. We will simulate the Chaos Game by:

Defining a triangle using three vertices (corners A, B, and C).

Randomly choosing a starting point inside this triangle.

Iteratively pick one of the triangle's vertices randomly, then move the current point halfway towards this vertex.

Plotting the points to observe the pattern that emerges.

Here's the Python code to accomplish this:

Python code

import matplotlib.pyplot as plt

import random

# Define the vertices of the triangle

A = (0, 0)

B = (1, 0)

C = (0.5, 0.866)  # This is the height of an equilateral triangle

# Start with a random point inside the triangle

current\_point = (random.uniform(0, 1), random.uniform(0, 0.866))

# Number of iterations

iterations = 10000

# Function to move halfway towards a chosen vertex

def move\_halfway(point, vertex):

    return ((point[0] + vertex[0]) / 2, (point[1] + vertex[1]) / 2)

# Store points to plot

points = []

for \_ in range(iterations):

    choice = random.choice([A, B, C])

    current\_point = move\_halfway(current\_point, choice)

    points.append(current\_point)

# Unpacking the list of points for plotting

x, y = zip(\*points)

# Create the plot

plt.figure(figsize=(6, 6))

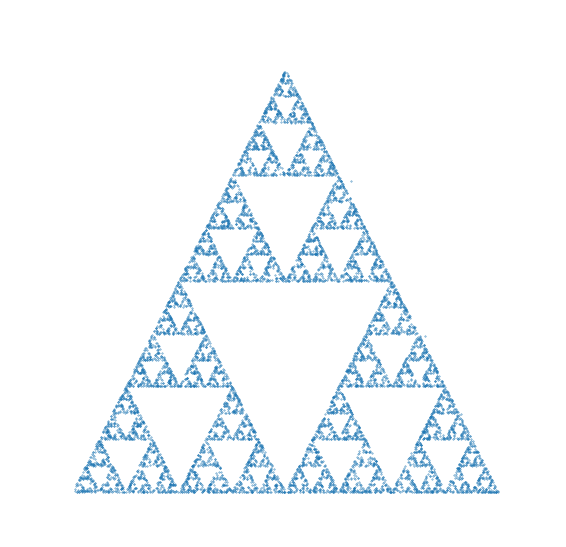
plt.scatter(x, y, s=0.1)  # Use a small size for each point

plt.axis('off')  # Turn off the axis

plt.show()

This code will generate a plot of points that, after a sufficient number of iterations, will resemble the Sierpinski Triangle, demonstrating how deterministic patterns emerge from seemingly random processes. The **plt.axis('off')** command removes the axes for a cleaner presentation of the fractal pattern.

You can adjust the **iterations** variable to increase or decrease the number of points plotted, which may affect the clarity and definition of the Sierpinski Triangle pattern. More iterations will generally result in a clearer fractal.



Using the Python **turtle** module, we can visualize the Chaos Game to generate the Sierpinski Triangle using turtle graphics. This will provide a more dynamic and interactive visualization as the process unfolds.

**Converting Python Code to NetLogo**

NetLogo is an agent-based programming language that is ideal for simulating complex systems. Here is how to translate the Python Chaos Game into NetLogo:

1. **Define the Setup and Go Procedures**:
   * setup will initialize the model.
   * go will iterate to create the Sierpinski Triangle.
2. **Create Buttons for Control**:
   * A "Setup" button to initialize the model.
   * A "Go" button to run the simulation.

**NetLogo Code**

netlogo

Copy code

; Define the vertices of the triangle

globals [A B C current-point]

to setup

clear-all

; Define vertices of the triangle

set A (list 0 0)

set B (list 100 0)

set C (list 50 86.6) ; Height of an equilateral triangle with side length 100

; Start with a random point inside the triangle

set current-point (list (random-float 100) (random-float 86.6))

reset-ticks

end

to go

; Choose a random vertex

let choice one-of (list A B C)

; Move halfway towards the chosen vertex

set current-point move-halfway(current-point, choice)

; Create a point at the current position

create-turtles 1 [

setxy item 0 current-point item 1 current-point

set size 0.1

set color black

pen-down

]

tick

end

to-report move-halfway [point vertex]

let new-x ((item 0 point) + (item 0 vertex)) / 2

let new-y ((item 1 point) + (item 1 vertex)) / 2

report (list new-x new-y)

end

**Setting Up NetLogo Interface**

1. **Create Buttons**:
   * Add a button for setup with the label "Setup".
   * Add a button for go with the label "Go" and set it to "Forever".
2. **Visualization**:
   * Ensure the world settings in NetLogo match the coordinate system used in the setup (e.g., max-pxcor: 100, max-pycor: 86, etc.).

**Explanation of the NetLogo Code**

* **Setup Procedure**:
  + Initializes the triangle vertices.
  + Sets a random starting point inside the triangle.
* **Go Procedure**:
  + Selects a random vertex.
  + Moves the current point halfway towards the selected vertex.
  + Creates a turtle at the new point's location.
  + Ticks to iterate through the procedure.

By following these steps, you can set up the Chaos Game in NetLogo, allowing easy manipulation of the model and visualization of the Sierpinski Triangle formation.

Below is a Python script that uses the **turtle** module to execute the Chaos Game. The script involves creating a window where a turtle will move to plot points according to the rules you specified:

Define the corners of the triangle.

Start from a random point within the triangle.

Iteratively choose one of the triangle's corners at random, move the turtle halfway towards that corner, and plot a point.

Here's the code:

python code

import turtle

import random

# Set up the turtle screen

window = turtle.Screen()

window.bgcolor("white")

# Create a turtle for drawing

chaos\_turtle = turtle.Turtle()

chaos\_turtle.speed(0)  # Set turtle speed to fastest

chaos\_turtle.penup()  # Don't draw lines

# Define the vertices of the triangle

A = (-200, -100)

B = (0, 200)

C = (200, -100)

# Draw the triangle

def draw\_triangle():

    chaos\_turtle.goto(A)

    chaos\_turtle.pendown()

    chaos\_turtle.goto(B)

    chaos\_turtle.goto(C)

    chaos\_turtle.goto(A)

    chaos\_turtle.penup()

draw\_triangle()

# Start with a random point inside the triangle

current\_point = (random.uniform(-100, 100), random.uniform(-50, 50))

# Function to move halfway towards a chosen vertex

def move\_halfway(point, vertex):

    return ((point[0] + vertex[0]) / 2, (point[1] + vertex[1]) / 2)

# Number of iterations

iterations = 10000

# Plot the points

for \_ in range(iterations):

    vertex = random.choice([A, B, C])

    current\_point = move\_halfway(current\_point, vertex)

    chaos\_turtle.goto(current\_point)

    chaos\_turtle.dot(2, "black")  # Draw a small dot

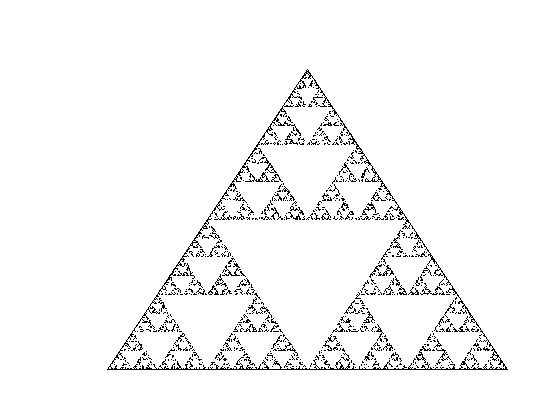
# Hide the turtle and finish

chaos\_turtle.hideturtle()

window.mainloop()  # Keeps the window open

This script will draw the initial triangle and then start plotting points inside it. You'll see how the points begin to form the Sierpinski Triangle pattern. The **chaos\_turtle.dot(2, "black")** function draws a small black dot at each new point, gradually revealing the fractal structure.

Make sure to have the Python **turtle** module installed and running in a local Python environment, as graphical output like this won't work in some online Python interpreters due to their lack of support for graphical output.



the Chaos Game can be adapted to explore fractal patterns with various other shapes beyond the standard triangle used for the Sierpinski Triangle. By modifying the number of vertices and the rules for movement towards these vertices, you can create a wide range of deterministic fractal patterns. Here are a few variations to consider:

**Square (Tetragon):**

Use four vertices, placed to form a square.

The rule might be to randomly choose any vertex and move halfway (or another fraction) towards it.

This can create patterns that are more grid-like but still possess fractal properties depending on the constraints (like skipping vertices).

**Pentagon and Hexagon:**

Using a pentagon or hexagon increases complexity.

Similar rules apply: select a vertex at random and move a fraction of the distance towards it.

Adjusting the fraction of the distance (e.g., one-third instead of half) can lead to different patterns and potentially more intricate fractal designs.

**Polygon with More Sides:**

As the number of sides increases, the potential complexity of the resulting fractal also increases.

You can experiment with polygons of 7, 8, 9, or more sides to see how the complexity of the fractal pattern evolves.

**Irregular Shapes:**

Not all vertices have to be evenly spaced or lie on a regular polygon.

By positioning vertices irregularly, you can create unique and unexpected fractal patterns.

**3D Fractals:**

Extending the Chaos Game into three dimensions, using polyhedra such as tetrahedrons, cubes, or other 3D shapes.

The principle is the same, but now you choose vertices in three-dimensional space and plot points accordingly.

**Multiple Sets of Rules:**

Apply different rules for different vertices. For example, if a certain vertex is chosen, you might move 1/3 of the way towards it, and for another vertex, you might move 2/3 of the way.

Introducing probabilistic rules where some vertices are more likely to be chosen than others can also affect the resulting pattern.

**Combining Shapes:**

You can also combine different shapes, applying the Chaos Game rules within a composite layout of multiple overlapping or adjoining shapes (like triangles and squares together).

To explore these variations, the algorithm would need adjustments mainly in terms of defining the vertices and possibly changing the fraction of the distance moved towards the selected vertex. Each variation can result in uniquely beautiful and complex patterns, showcasing the versatility and depth of the Chaos Game in exploring mathematical and natural phenomena through simple computational models.

Extending the Chaos Game to three dimensions opens up fascinating possibilities for modelling and visualizing fractal patterns in 3D space. Using polyhedra like tetrahedrons, cubes, or more complex shapes like octahedrons and icosahedrons allows for the exploration of fractal geometry in a richer, more spatially complex environment. Here are some specific ideas and types of fractals that can be modelled with 3D variations of the Chaos Game:

**Tetrahedron Fractal:**

Use the four vertices of a tetrahedron.

By choosing a vertex randomly and moving a fraction (commonly half) towards it from a current position, a fractal resembling a 3D Sierpinski gasket can be created.

Creating a tetrahedron fractal using the Chaos Game in Python using the **turtle** module presents a conceptual challenge since **turtle** is inherently a 2D graphics library. However, we can simulate a 3D appearance by strategically placing the vertices of a tetrahedron in a 2D projection. We'll do this by arranging the vertices so they visually represent a three-dimensional tetrahedron on a 2D plane.

**The Tetrahedron Fractal**

A tetrahedron is a polyhedron with four triangular faces, six straight edges, and four vertex corners. To simulate its 3D structure in 2D:

**Vertices Placement**: Place three vertices in a triangular layout for the base and add the fourth vertex above the centre of this triangle to represent the apex of the tetrahedron.

**Chaos Game Rules**:

Start with a random point inside the triangle formed by the first three vertices.

Randomly choose one of the four vertices.

Move halfway towards the chosen vertex.

Plot the point.

This will result in a pattern known as a 3D Sierpinski gasket after a sufficient number of iterations, where each face of the tetrahedron acts like a 2D Sierpinski triangle.

**Implementation in Python with Turtle**

Here's a Python script that approximates this idea using **turtle**. We will project the tetrahedron onto a 2D plane and apply the Chaos Game:

Python code

import turtle

import random

# Set up the turtle screen

window = turtle.Screen()

window.bgcolor("white")

# Create a turtle for drawing

chaos\_turtle = turtle.Turtle()

chaos\_turtle.speed(0)  # Set turtle speed to fastest

chaos\_turtle.penup()  # Don't draw lines

# Define the vertices of the tetrahedron in 2D projection

A = (-200, -100)

B = (0, 200)

C = (200, -100)

D = (0, 50)  # This is the apex, projected above the centre of the base triangle

# Draw the tetrahedron edges

def draw\_tetrahedron():

    chaos\_turtle.goto(A)

    chaos\_turtle.pendown()

    chaos\_turtle.goto(B)

    chaos\_turtle.goto(C)

    chaos\_turtle.goto(A)

    chaos\_turtle.goto(D)

    chaos\_turtle.goto(B)

    chaos\_turtle.penup()

    chaos\_turtle.goto(C)

    chaos\_turtle.pendown()

    chaos\_turtle.goto(D)

    chaos\_turtle.penup()

draw\_tetrahedron()

# Start with a random point inside the base triangle

current\_point = (random.uniform(-100, 100), random.uniform(-50, 50))

# Function to move halfway towards a chosen vertex

def move\_halfway(point, vertex):

    return ((point[0] + vertex[0]) / 2, (point[1] + vertex[1]) / 2)

# Number of iterations

iterations = 10000

# Plot the points

for \_ in range(iterations):

    vertex = random.choice([A, B, C, D])

    current\_point = move\_halfway(current\_point, vertex)

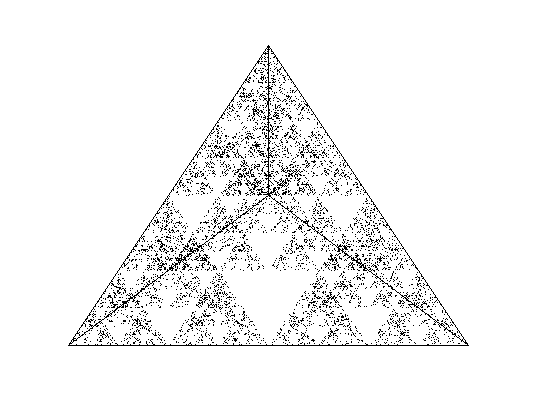
    chaos\_turtle.goto(current\_point)

    chaos\_turtle.dot(2, "black")  # Draw a small dot

# Hide the turtle and finish

chaos\_turtle.hideturtle()

window.mainloop()  # Keeps the window open

****

**Explanation**

**Vertices A, B, C** form the base of the tetrahedron, while **D** is the apex.

The initial random point and subsequent points are calculated to be inside the base triangle, but as the game progresses, points will start filling up in all four faces, each representing a 2D Sierpinski triangle.

Each face of the tetrahedron will recursively display the characteristic self-similar pattern of a Sierpinski gasket.

This script provides a simplified and projected 2D view of a tetrahedron fractal, demonstrating how complex 3D patterns can be approximated in a 2D space using simple iterative algorithms like the Chaos Game.

**Cube Fractal:**

A cube has eight vertices, allowing for more complex rules and results.

Similar to the tetrahedron, but because of the higher number of vertices, the pattern may fill more space and offer different visual textures depending on how vertices are selected and the fraction of movement.

Creating a fractal using the vertices of a cube in a simulated 3D environment using Python's **turtle** graphics requires projecting the cube into 2D space while retaining the notion of its 3D structure. Unlike the tetrahedron, a cube has more vertices and faces, offering richer potential for complex fractal patterns.

**The Cube Fractal**

A cube has eight vertices and six faces. In a typical Chaos Game on a cube, we'll follow these steps:

**Vertices Placement**: Position the cube's vertices in a 2D layout that suggests its 3D form. This might involve creating two squares, one for the front face and one offset for the rear face of the cube, connected at their corresponding corners.

**Chaos Game Rules**:

Start from a random point within the projected bounds of the cube.

Randomly select one of the eight vertices.

Move a predetermined fraction (typically half) towards the selected vertex.

Repeat and plot the point.

This will generate a cube-based fractal, potentially filling in a 3D-like pattern on a 2D plane, depending on how the vertices are chosen and visualized.

**Implementation in Python with Turtle**

Here's a conceptual Python script using **turtle** for simulating the Cube Fractal:

Python code

import turtle

import random

# Set up the turtle screen

window = turtle.Screen()

window.bgcolor("white")

# Create a turtle for drawing

chaos\_turtle = turtle.Turtle()

chaos\_turtle.speed(0)  # Set turtle speed to fastest

chaos\_turtle.penup()  # Don't draw lines

# Define the vertices of the cube in a 2D projection

A = (-100, -100)  # Front lower left

B = (100, -100)   # Front lower right

C = (100, 100)    # Front upper right

D = (-100, 100)   # Front upper left

E = (-50, -50)    # Back lower left

F = (150, -50)    # Back lower right

G = (150, 150)    # Back upper right

H = (-50, 150)    # Back upper left

vertices = [A, B, C, D, E, F, G, H]

# Draw the cube edges

def draw\_cube():

    # Front face

    chaos\_turtle.goto(A)

    chaos\_turtle.pendown()

    chaos\_turtle.goto(B)

    chaos\_turtle.goto(C)

    chaos\_turtle.goto(D)

    chaos\_turtle.goto(A)

    chaos\_turtle.penup()

    # Back face

    chaos\_turtle.goto(E)

    chaos\_turtle.pendown()

    chaos\_turtle.goto(F)

    chaos\_turtle.goto(G)

    chaos\_turtle.goto(H)

    chaos\_turtle.goto(E)

    chaos\_turtle.penup()

    # Connecting edges

    for front, back in zip([A, B, C, D], [E, F, G, H]):

        chaos\_turtle.goto(front)

        chaos\_turtle.pendown()

        chaos\_turtle.goto(back)

        chaos\_turtle.penup()

draw\_cube()

# Start with a random point near the centre of the cube

current\_point = (0, 0)

# Function to move halfway towards a chosen vertex

def move\_halfway(point, vertex):

    return ((point[0] + vertex[0]) / 2, (point[1] + vertex[1]) / 2)

# Number of iterations

iterations = 10000

# Plot the points

for \_ in range(iterations):

    vertex = random.choice(vertices)

    current\_point = move\_halfway(current\_point, vertex)

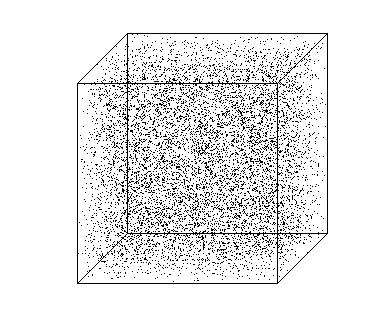
    chaos\_turtle.goto(current\_point)

    chaos\_turtle.dot(2, "black")  # Draw a small dot

# Hide the turtle and finish

chaos\_turtle.hideturtle()

window.mainloop()  # Keeps the window open



Explanation and Considerations

Vertices Representation: The vertices are arranged to give a 3D perspective effect, with A-D representing the front face and E-H the back face of the cube.

Fractal Generation: As the points are plotted, the fractal pattern begins to fill the space defined by the cube's projected vertices. Over many iterations, a complex texture emerges, resembling a 3D fractal depicted in 2D.

Visual Texture: Depending on how the vertices are selected (e.g., certain rules or biases towards choosing front vs. back vertices), different patterns and densities will develop across the cube's faces.

This simulation provides a unique way to visualize fractal patterns in a quasi-3D format using basic 2D graphics tools, demonstrating the flexibility and potential for creative mathematical exploration even within constraints.

**Icosahedron and Dodecahedron Fractals:**

These polyhedra, with 20 and 12 vertices respectively, allow for highly intricate fractal patterns.

The more vertices involved, the richer the emergent complexity, which can create highly detailed fractal clouds or shells.

Simulating the Chaos Game with an icosahedron or a dodecahedron to create fractals involves significantly more complexity due to the increased number of vertices and the intricate geometrical arrangements. These polyhedra are higher-order shapes with 20 and 12 vertices respectively, and their fractal patterns can be extremely intricate, resembling natural phenomena like microscopic radiolarians or complex viral structures.

**Icosahedron and Dodecahedron Fractal Overview**

**Icosahedron**:

Consists of 20 vertices, 30 edges, and 20 equilateral triangle faces.

An icosahedron fractal can be visualized by projecting its vertices into 2D while maintaining a representation of its 3D structure, but this loses some spatial relations.

**Dodecahedron**:

Features 12 vertices, 30 edges, and 12 pentagonal faces.

Similar to the icosahedron, a 2D projection can be used for visualization, although it's quite challenging to represent accurately without proper 3D software.

Let's simplify and fully reproduce a feasible implementation for a conceptual 2D approximation of an icosahedron using Python's **turtle** module. Given the complexity and number of vertices (20) in an icosahedron, it's important to understand that a full 3D visualization isn't possible with **turtle**. Instead, we can represent a subset of vertices to simulate the process and give a feel for the Chaos Game applied to a more complex shape.

Below, I will provide Python code that uses a simplified projection of an icosahedron's vertices in 2D, assuming arbitrary placements that would form a visually distinct pattern. This example won't accurately project the true 3D structure but will help understand the Chaos Game mechanics on a complex polyhedral base.

**Simplified Python Code for an "Icosahedron" Chaos Game**

python code

import turtle

import random

def setup\_window():

    # Set up the turtle screen

    window = turtle.Screen()

    window.bgcolor("white")

    return window

def setup\_turtle():

    # Create a turtle for drawing

    chaos\_turtle = turtle.Turtle()

    chaos\_turtle.speed(0)  # Fastest speed

    chaos\_turtle.penup()

    return chaos\_turtle

def define\_vertices():

    # Simplified "icosahedron" vertices in 2D

    vertices = [

        (-200, 0), (-160, -100), (-120, 0), (-80, -100), (-40, 0), (0, -100),

        (40, 0), (80, -100), (120, 0), (160, -100), (200, 0), (160, 100),

        (120, 200), (80, 100), (40, 200), (0, 100), (-40, 200), (-80, 100),

        (-120, 200), (-160, 100)

    ]

    return vertices

def draw\_vertices(chaos\_turtle, vertices):

    # Draw dots for vertices

    chaos\_turtle.color('red')

    for vertex in vertices:

        chaos\_turtle.goto(vertex)

        chaos\_turtle.dot(10)

def chaos\_game(chaos\_turtle, vertices, iterations=10000):

    # Start from a random point

    current\_point = (0, 0)

    chaos\_turtle.goto(current\_point)

    chaos\_turtle.dot(5, 'blue')

    # Function to move halfway towards a chosen vertex

    def move\_halfway(point, vertex):

        return ((point[0] + vertex[0]) / 2, (point[1] + vertex[1]) / 2)

    # Perform the Chaos Game

    for \_ in range(iterations):

        vertex = random.choice(vertices)

        current\_point = move\_halfway(current\_point, vertex)

        chaos\_turtle.goto(current\_point)

        chaos\_turtle.dot(3, 'black')  # Draw a small dot to represent the point

def main():

    window = setup\_window()

    chaos\_turtle = setup\_turtle()

    vertices = define\_vertices()

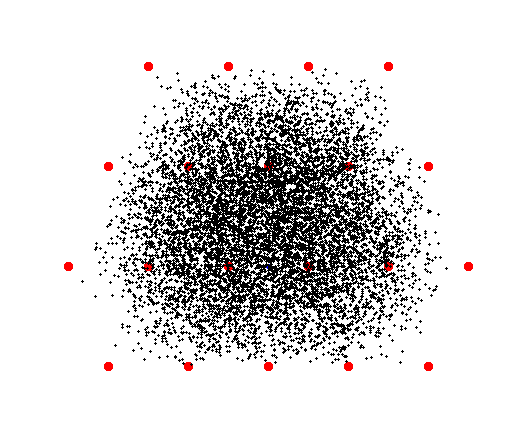
    draw\_vertices(chaos\_turtle, vertices)

    chaos\_game(chaos\_turtle, vertices)

    window.mainloop()  # Keeps the window open

if \_\_name\_\_ == '\_\_main\_\_':

    main()



Explanation

Window and Turtle Setup: The setup\_window and setup\_turtle functions prepare the turtle graphics environment.

Vertices Definition: define\_vertices provide a manually set list of 20 points meant to represent an "icosahedron" in a simplified 2D projection.

Drawing Vertices: The draw\_vertices function marks each vertex with a red dot for visual reference.

Chaos Game Simulation: The chaos\_game function iteratively selects vertices, moves halfway towards them, and plots points, simulating the fractal formation.

This code provides an interactive and visual exploration of the Chaos Game on a complex structure in a simplified form. To get a true 3D representation, more advanced graphical programming or 3D modelling software would be required.

To model and visualize complex 3D fractals like those based on an icosahedron or dodecahedron in Python, several tools and libraries can be utilized. These libraries enable the creation, manipulation, and rendering of 3D graphics and are well-suited for handling the computation and visualization of fractal geometry in three dimensions. Here are some of the most effective Python tools and libraries for this purpose:

Matplotlib with mplot3d:

Description: Matplotlib is primarily a 2D plotting library, but it includes a toolkit, mplot3d, which adds simple 3D plotting capabilities. While not as powerful or flexible as some purely 3D visualization tools, it's good for basic 3D plotting tasks.

Use Case: Simple 3D scatter plots and wireframes can be created to visualize points and connectivity in 3D space, suitable for demonstrating fractal structures with less visual complexity.

Mayavi:

Description: Mayavi is a powerful 3D scientific data visualization library that leverages the VTK (Visualization Toolkit) engine. It provides a high-level interface for 3D visualization and is capable of producing high-quality 3D visuals.

Use Case: Ideal for complex 3D rendering of fractals, allowing for interactive exploration of data, and capable of handling large datasets with sophisticated rendering techniques like volume rendering and mesh plotting.

Plotly:

Description: Plotly is a versatile library that can produce interactive plots. It supports 3D plotting functionalities and can be used both in Python scripts and Jupyter notebooks for interactive visualizations.

Use Case: Suitable for creating interactive 3D scatter plots and surface plots, allowing users to zoom, pan, and rotate visualizations to explore fractal structures interactively.

Pygame:

Description: Pygame is a set of Python modules designed for writing video games, but it can also be used for other multimedia applications. It includes graphics and sound libraries designed to work with 2D visuals but can be hacked for simple pseudo-3D effects.

Use Case: While not suitable for true 3D modelling, Pygame can be used to simulate 3D by manipulating 2D graphics in ways that give a depth perception effect.

Blender Python API:

Description: Blender is a professional-level open-source 3D modelling and rendering tool that includes a comprehensive API for Python scripting. This API allows for extensive control over the Blender environment for creating, modifying, and rendering 3D models programmatically.

Use Case: Perfect for detailed 3D modelling and fractal generation, with the ability to script the generation of geometries, materials, and lighting to produce photorealistic renderings of complex fractals.

VPython (Visual Python):

Description: VPython is designed for easy creation of navigable 3D visualizations. It simplifies the process of creating 3D graphics using Python, making it accessible to those new to programming or 3D graphics.

Use Case: Excellent for educational purposes and for those who are beginning to explore 3D programming. It's intuitive for creating and manipulating 3D objects and animations directly in a notebook environment.

Suggested Approach:

For creating and visualizing complex 3D fractals like those based on polyhedra:

Mayavi or the Blender Python API would be the most suitable due to their powerful 3D capabilities and flexibility in handling intricate geometric transformations and renderings.

For interactive web-based presentations, Plotly offers a user-friendly and accessible platform.

These tools provide a range of functionalities from basic to advanced 3D visualization needs, suitable for different levels of project complexity and user expertise in programming and 3D modelling.

While Matplotlib's **mplot3d** toolkit is not as advanced as some dedicated 3D visualization tools, it's sufficient for creating basic 3D fractals and is an excellent choice for integrating 3D visualizations into Python scripts due to its compatibility and ease of use. Here’s how you can create a simple 3D fractal representation using Matplotlib. We'll simulate a Sierpinski tetrahedron fractal, a straightforward extension of the 2D Chaos Game to 3D.

**Step-by-Step Guide to Creating a 3D Fractal with Matplotlib**

**Set Up Your Environment**: Install Matplotlib if it's not already available.

Bash code

**pip install matplotlib**

**Define the Tetrahedron Vertices**: These will be the four points in 3D space that define a tetrahedron.

**Implement the Chaos Game Logic**: Start from a random point within the tetrahedron, select one of the vertices at random, move halfway towards it, and plot the point.

**Plotting with Matplotlib**: Use the **mplot3d** toolkit to create the plot.

**Python Code Implementation**

Here's how you can implement this in Python using Matplotlib’s **mplot3d** toolkit:

Python code

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

import random

# Define the vertices of the tetrahedron

vertices = [(1, 1, 1), (-1, -1, 1), (-1, 1, -1), (1, -1, -1)]

# Choose a random initial point within the tetrahedron (simplifying assumption near origin)

current\_point = (0.5, 0.5, 0.5)

# Number of iterations

iterations = 10000

# Function to move halfway towards a chosen vertex

def move\_halfway(point, vertex):

    return ((point[0] + vertex[0]) / 2, (point[1] + vertex[1]) / 2, (point[2] + vertex[2]) / 2)

# Store the points to plot

points = [current\_point]

for \_ in range(iterations):

    vertex = random.choice(vertices)

    current\_point = move\_halfway(current\_point, vertex)

    points.append(current\_point)

# Extracting the x, y, z coordinates

x, y, z = zip(\*points)

# Create a 3D plot

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

ax.scatter(x, y, z, c='blue', s=1)  # Smaller dots

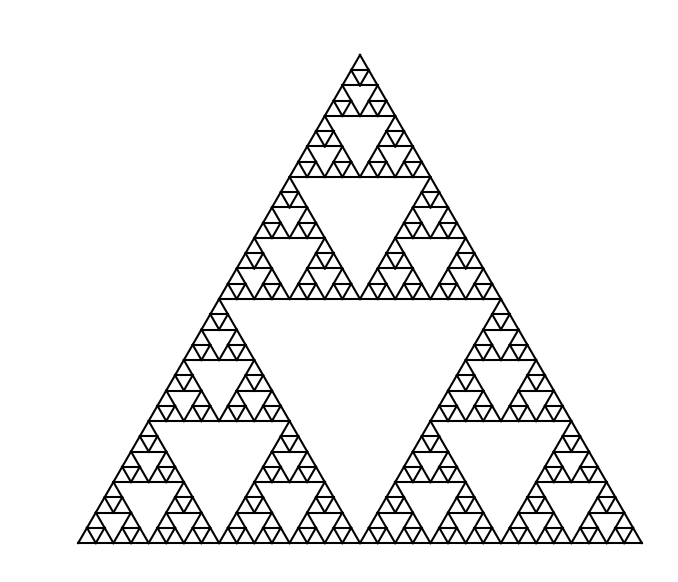
ax.set\_xlabel('X Coordinate')

ax.set\_ylabel('Y Coordinate')

ax.set\_zlabel('Z Coordinate')

plt.title('3D Chaos Game Fractal: Sierpinski Tetrahedron')

plt.show()

****

**Explanation**

**Vertices Setup**: The vertices of the tetrahedron are defined in 3D space.

**Chaos Game Logic**: The function **moves halfway** computes the midpoint between the current point and a randomly chosen vertex.

**3D Plotting**: The points are plotted in a 3D scatter plot where each point represents a step in the Chaos Game. Over many iterations, a fractal pattern emerges that resembles a 3D Sierpinski tetrahedron.

This code provides a basic but effective demonstration of generating a 3D fractal using Python and Matplotlib. You can adjust the number of iterations, the initial point, and even the vertex positions to explore different aspects of fractal geometry.

**Chaos Game Rules for These Polyhedra**

For both shapes, the basic Chaos Game rules apply:

Start from a random point inside or near the projected shape.

Randomly select one of the vertices.

Move a set fraction (usually 1/2 or 1/3) towards the selected vertex.

Plot the point and repeat.

**Challenges and Considerations**

**3D Visualization Limitation**: The **turtle** graphics library in Python is inherently 2D, which limits the ability to fully appreciate the 3D nature of the patterns without a true 3D rendering environment.

**Complexity in Vertex Representation**: Accurately projecting an icosahedron or dodecahedron onto a 2D plane while retaining their complex symmetries is challenging.

**Different Movement Rules:**

Instead of moving halfway towards the chosen vertex, varying the fraction or applying different fractions for different vertices can drastically alter the fractal's appearance.

Introducing probabilistic biases towards certain vertices can also generate unique patterns, where some parts of the shape become denser or more intricate than others.

**3D Fractals with Hollow Structures:**

By adjusting the rules to skip over certain vertices or to move in non-linear paths (like skipping one vertex in the sequence), hollow or shell-like structures can emerge, which are visually and structurally different from denser fractals.

**Animation and Dynamic Visualization:**

Dynamically visualizing the growth of these fractals in a 3D space can offer deeper insights into their structure and development over time.

Animation can also help in understanding how fractals fill up space and how their complexity evolves.

**Physical and Natural Phenomena Simulation:**

These models can simulate the growth patterns of certain types of corals, sponges, or other natural fractal-like structures.

They can also provide insights into diffusion-limited aggregation processes which resemble how particles cluster in nature.

**Mathematical and Theoretical Research:**

Studying the properties of 3D fractals can contribute to fields such as topology, computational geometry, and mathematical theory regarding dimensions and scaling laws.

The exploration of 3D fractals with the Chaos Game in computational models requires 3D graphics capabilities, such as those provided by software libraries like OpenGL, Unity 3D, or programming environments that support 3D plotting and visualization. The development and analysis of these patterns can be highly resource-intensive due to the calculations and graphical rendering involved but offer a profound way to explore the intersection of mathematics, art, and natural phenomena.

The Sierpinski Triangle, one of the most recognized fractals, showcases an intriguing intersection of simplicity in construction and complexity in outcome. The triangle emerges through iterative processes, whether through deterministic methods or stochastic simulations such as the Chaos Game.

**Understanding the Sierpinski Triangle:**

**Deterministic Method**:

Start with an equilateral triangle.

Remove the inverted triangle from the centre, dividing the remaining area into three smaller equilateral triangles.

Repeat the process for each of the smaller triangle’s ad infinitum.

Each iteration removes smaller triangles from the centre of the remaining triangles, leading to the fractal structure.

**Chaos Game Method**:

Define three vertices of a triangle.

Begin from a random point within or near the triangle.

Randomly select one of the vertices.

Move halfway (or another fixed ratio) towards the selected vertex and mark a point.

Repeat the previous two steps thousands of times.

Instead of removing parts of the triangle, the points plotted gradually fill up the areas not removed in the deterministic method, converging on the same fractal pattern.

**Visualization of the Sierpinski Triangle**

Let's use Matplotlib in Python to visualize the Sierpinski Triangle using the deterministic method, illustrating the process step by step. This visualization helps demonstrate the construction and emerging pattern of the Sierpinski Triangle without relying on the randomness of the Chaos Game.

Python code

import matplotlib.pyplot as plt

import numpy as np

def plot\_triangle(vertices, color='k'):

    """Plot a triangle from given vertices."""

    for i in range(3):

        start = vertices[i]

        end = vertices[(i + 1) % 3]

        plt.plot([start[0], end[0]], [start[1], end[1]], color)

def sierpinski(vertices, depth):

    """Recursively remove the centre triangles."""

    if depth == 0:

        plot\_triangle(vertices)

    else:

        # Calculate the midpoints of each side

        mid\_01 = ((vertices[0][0] + vertices[1][0]) / 2, (vertices[0][1] + vertices[1][1]) / 2)

        mid\_12 = ((vertices[1][0] + vertices[2][0]) / 2, (vertices[1][1] + vertices[2][1]) / 2)

        mid\_20 = ((vertices[2][0] + vertices[0][0]) / 2, (vertices[2][1] + vertices[0][1]) / 2)

        # Sub-triangles

        sierpinski([vertices[0], mid\_01, mid\_20], depth - 1)

        sierpinski([vertices[1], mid\_12, mid\_01], depth - 1)

        sierpinski([vertices[2], mid\_20, mid\_12], depth - 1)

# Initial vertices of the main triangle

vertices = np.array([[0, 0], [1, 0], [0.5, np.sin(np.pi/3)]])

# Create the plot

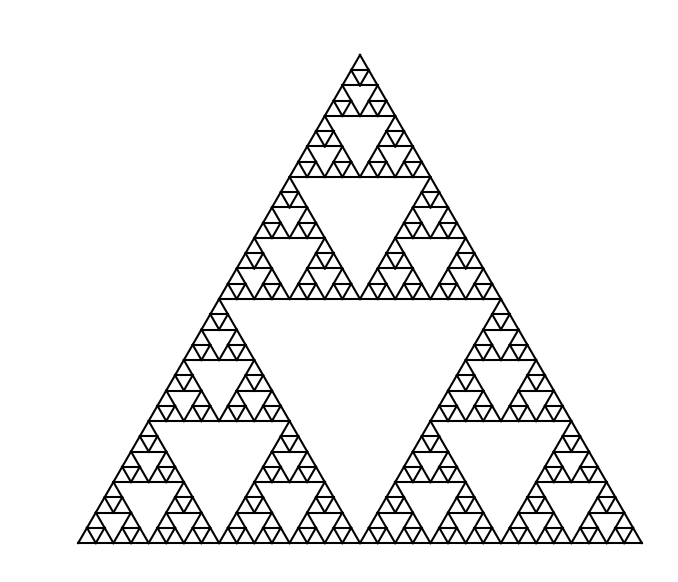
plt.figure(figsize=(8, 8))

sierpinski(vertices, depth=5)

plt.gca().set\_aspect('equal')

plt.axis('off')  # Hide the axes

plt.show()

****

**Explanation of the Code**

**Plotting Function**: **plot\_triangle** draws a triangle based on provided vertices.

**Recursive Function**: **sierpinski** is a recursive function that plots triangles down to the specified recursion depth (**depth**). It calculates midpoints of each triangle side to define the vertices of the smaller triangles.

**Visual Output**: Each recursion removes the centre triangle visually by not drawing it and recursively divides each of the remaining smaller triangles.

This deterministic approach provides a clear visualization of the Sierpinski Triangle, emphasizing its self-similar, fractal nature. Each iteration increases the resolution of the fractal, revealing its intricate pattern.

Consider where a point within the triangle ends up when it moves halfway to a corner. Whichever corner is picked, it will always end up in the triangle next to that corner somewhere, never in the middle. explain in detail

The phenomenon you're referring to is central to understanding the dynamics of the Chaos Game when applied to generate the Sierpinski Triangle. It illustrates a foundational property of fractals generated by iterative geometric processes—specifically, how iterative midpoint displacement within a geometric shape like a triangle result in a fractal pattern.

**Geometric Setup**

In the Sierpinski Triangle setup of the Chaos Game:

You begin with a large equilateral triangle.

You define its three vertices, which we'll label as A, B, and C.

**Moving Halfway to a Corner**

When a random point within the triangle moves "halfway" toward one of the corners, the following occurs:

**Choice of Vertex**: A vertex (corner) of the triangle is selected at random. Let's say vertex A is chosen.

**Movement**: The point moves halfway towards vertex A. This means if you draw a line segment from the current position of the point to vertex A, the point moves to the midpoint of that line segment.

**Geometric Consequence**

Moving to this midpoint has specific geometric implications:

**Location**: The new position of the point is always closer to the chosen vertex (A in our example) than any other part of the triangle.

**Inclusion in Sub-triangle**: By moving halfway towards a corner, the point effectively positions itself within one of the three smaller triangles that would result if lines were drawn from each vertex to the centroid (geometric centre) of the original triangle. Each of these smaller triangles shares a vertex with the original triangle and has one vertex at the centroid and one midpoint of a side of the original triangle.

**Why It Never Ends Up in the Middle**

**Fractal Division**: The process of repeatedly moving halfway towards a randomly chosen vertex divides the triangle into smaller triangles recursively. Each move places the point within a smaller triangle adjacent to the vertex chosen, never in the central void that forms as the fractal pattern develops.

**Centroid Exclusion**: As the process repeats, certain areas of the triangle—specifically, the middle region formed by connecting the midpoints of the sides of the original triangle—never contain any points. This region is the first to be 'removed' in a deterministic, middle-out removal process to form the Sierpinski Triangle and is never revisited in the stochastic midpoint displacement process.

**Visual and Mathematical Implications**

This recursive process of moving halfway towards a corner effectively maps any point within the initial triangle into smaller and smaller sub-triangles adjacent to the vertices. The result is a fractal pattern where the density of points increases near the vertices and decreases (to zero) in the areas corresponding to the middle of the original and subsequent triangles.

**Summary**

The Chaos Game, by leveraging simple rules of geometric displacement, illustrates a complex dynamic system's behaviour. The points progressively populate the perimeter of the recurring triangles, illustrating how simple deterministic rules can lead to complex patterns. This emergent behaviour is characteristic of fractal geometry and highlights a fundamental aspect of chaos theory—sensitivity to initial conditions leading to seemingly complex and ordered structures.

Now consider every sequence of two corners. If B is chosen and then A, the dot has to appear in the lower left section of triangle A.

the observation about the behaviour of the Chaos Game when considering sequences of corners is a critical aspect in understanding how the game's rules generate specific placements of dots within the Sierpinski Triangle. Let's explore the geometric implications of selecting two specific corners in sequence, such as B followed by A, and how this sequence influences the position of the dot within the fractal structure.

**Understanding Sequences in the Chaos Game**

When playing the Chaos Game to generate the Sierpinski Triangle, each move involves selecting a corner at random and then moving halfway toward it. This step is simple on its own, but when sequences of moves are considered, the complexity and fractal nature of the pattern begin to emerge.

**Example Sequence: Corner B followed by Corner A**

**First Move - Choosing Corner B**:

Start from any point within the initial triangle.

Move halfway toward corner B.

This action places the point somewhere within the smaller triangle that has corner B as one of its vertices (the base right triangle in the standard orientation).

**Second Move - Choosing Corner A**:

From the new position (somewhere in the smaller triangle near B), move halfway toward corner A.

This specific sequence (moving toward B first, then A) ensures that the new position will be within the section of the triangle that lies closer to both B and A, specifically the lower left section of the smaller triangle that shares its vertex with A in the original triangle.

**Geometric Consequences of the Sequence**

**Positioning within Sub-triangles**:

The initial move puts the point in the vicinity of B.

The subsequent move toward A shifts the point diagonally across the interior of the Sierpinski Triangle, aiming for a region that is structurally aligned between A and the initial influence of B.

The dot ends up in one of the smaller triangles formed by the recursive division of the triangle that includes A and is closer to the base of the original triangle. This is typically a lower section if A is at the top in a typical upright triangle representation.

**Fractal Recurrence and Position Refinement**

**Recursive Division**:

Each choice of corners refines the possible location of the dot within increasingly smaller triangles.

The sequence of choices directs the dot into specific sub-regions of these triangles, effectively determining which part of the fractal structure the dot will populate.

**Mapping to Sierpinski Sections**:

In the standard Sierpinski Triangle, every iteration removes the middle triangle and leaves three sub-triangles. By repeatedly moving towards specific corners according to the rules of the Chaos Game, the dot continues to find its place within smaller triangles adjacent to these corners.

The specific sequence of corner choices thus progressively maps the dot into one of the corners of these smaller triangles, which are geometrically consistent across scales due to the fractal’s self-similarity.

**Conclusion**

Each sequence of moves in the Chaos Game doesn't just randomly distribute points within the triangle; it strategically confines them to smaller and smaller sections, adhering to the fractal's recursive pattern. This method illustrates how deterministic rules (like moving halfway toward a corner) applied in a stochastic process (random corner selection) can produce intricately ordered and beautiful patterns found in fractal geometry. This behaviour—where sequences like B followed by A lead to specific regional confinement—highlights the underlying order in what might initially seem like a chaotic system.

Each sequence of k characters defines a triangle at the kth level of magnification.

the statement highlights an essential aspect of the Sierpinski Triangle's fractal nature when generated through the Chaos Game or similar iterative methods. Each sequence of selections—each choice of corners in this context—progressively defines increasingly smaller triangles within the overall structure. This can be understood through the concept of "levels of magnification" in the fractal, where each additional character (representing a corner of the triangle) in a sequence further refines the location within the fractal.

**Exploring Levels of Magnification**

In the context of the Sierpinski Triangle:

**Initial Triangle (Level 0)**:

This is the entire triangle, with no subdivisions.

**First Division (Level 1)**:

After the first division (removing the central upside-down triangle), three smaller triangles remain, each corresponding to a corner of the original triangle. These smaller triangles represent the first level of magnification. If you choose a vertex A, B, or C, the resulting smaller triangle adjacent to that vertex is defined.

**Subsequent Divisions (Higher Levels)**:

Each subsequent choice of a vertex refines the previous choice by selecting a smaller triangle within the chosen triangle from the previous step.

For example, a sequence of two vertices, say "AB," means moving first toward vertex A, then within the triangle section near A, moving next toward vertex B. This sequence places the point within a specific sub-triangle of the triangle adjacent to A at the second level of magnification.

**Illustration with a Sequence**

**Sequence**: Consider a sequence of vertices "ABC".

**First Choice (A)**: Move toward vertex A. This defines the first-level triangle (a small triangle near vertex A).

**Second Choice (B)**: Within the triangle chosen by the first move, the second move toward vertex B defines a smaller triangle within the first triangle. This is the second level of magnification.

**Third Choice (C)**: Continuing within the context of the triangle defined by the previous choices, a move toward C defines an even smaller triangle within the second one. This is the third level of magnification.

**Fractal Self-Similarity and Recursive Definition**

The self-similar nature of the Sierpinski Triangle means that each triangle at each level is a smaller version of the whole. Each sequence of choices (k characters) effectively zooms in on the fractal, defining a specific location within the structure. The more characters in a sequence, the greater the level of magnification, and the smaller and more precise the defined triangle.

**Mathematical and Computational Implications**

In computational models or mathematical descriptions, this characteristic allows for detailed exploration and analysis of fractal geometry. Each step in such a sequence can be modelled or computed to investigate properties like boundary definitions, area ratios, or limit behaviours at infinite magnifications.

In conclusion, sequences defining magnification levels in fractals like the Sierpinski Triangle offers a profound insight into the structural organisation of fractals and the underlying mathematical principles driving their formation. Each sequence not only determines a unique point within the fractal but also maps out a selection path that progressively reveals finer details of the fractal architecture.

Using the Chaos Game with a square and altering the jumping distance (the fraction of the distance moved towards a selected vertex) demonstrates how simple rule modifications can result in different complex patterns. For a square, as you mentioned, if we stick to moving halfway (0.5 of the distance) towards each randomly chosen corner, the result fills the square uniformly, not creating a fractal pattern. However, increasing the jump distance can lead to more complex, patterned distributions, often resembling a 'tartan' grid pattern.

To explore this behaviour with a square, let's create a Python script using the turtle module. We'll visualise the outcomes for two scenarios: moving halfway to the corner and moving with a more significant jump value to illustrate the difference.

**Python Script: Chaos Game with a Square**

Here's a Python script that uses the turtle graphics library to simulate the Chaos Game on a square with different jump values.

Python code

import turtle

import random

def setup\_window():

    """ Set up the turtle screen. """

    window = turtle.Screen()

    window.bgcolor("white")

    return window

def setup\_turtle():

    """ Create a turtle for drawing. """

    chaos\_turtle = turtle.Turtle()

    chaos\_turtle.speed(0)  # Fastest speed

    chaos\_turtle.penup()

    return chaos\_turtle

def draw\_square(chaos\_turtle, side\_length):

    """ Draw a square for the Chaos Game. """

    start\_pos = [-side\_length / 2, -side\_length / 2]

    for \_ in range(4):

        chaos\_turtle.goto(start\_pos[0], start\_pos[1])

        chaos\_turtle.pendown()

        start\_pos[0] += side\_length

        chaos\_turtle.goto(start\_pos[0], start\_pos[1])

        start\_pos[0], start\_pos[1] = start\_pos[1], start\_pos[0]

        chaos\_turtle.penup()

    chaos\_turtle.home()

def chaos\_game(chaos\_turtle, vertices, jump, iterations=5000):

    """ Perform the Chaos Game on a square. """

    current\_point = [0, 0]

    for \_ in range(iterations):

        vertex = random.choice(vertices)

        # Move towards the vertex by a fraction defined by 'jump'

        current\_point[0] += (vertex[0] - current\_point[0]) \* jump

        current\_point[1] += (vertex[1] - current\_point[1]) \* jump

        chaos\_turtle.goto(current\_point[0], current\_point[1])

        chaos\_turtle.dot(3)  # Draw a small dot

def main():

    window = setup\_window()

    chaos\_turtle = setup\_turtle()

    side\_length = 300  # Define the size of the square

    vertices = [

        (-side\_length / 2, -side\_length / 2),

        (side\_length / 2, -side\_length / 2),

        (side\_length / 2, side\_length / 2),

        (-side\_length / 2, side\_length / 2)

    ]

    draw\_square(chaos\_turtle, side\_length)

    chaos\_turtle.color('red')

    # Chaos game with jump = 0.5

    chaos\_game(chaos\_turtle, vertices, jump=0.5)

    # Reset turtle and try a larger jump

    chaos\_turtle.clear()

    draw\_square(chaos\_turtle, side\_length)

    chaos\_turtle.color('blue')

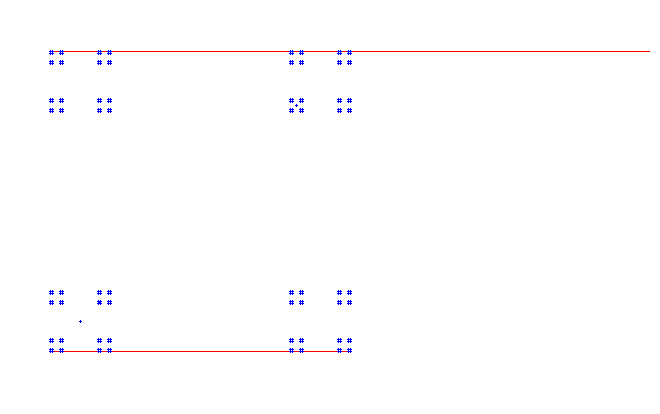
    # Chaos game with jump = 0.8

    chaos\_game(chaos\_turtle, vertices, jump=0.8)

    window.mainloop()  # Keeps the window open

if \_\_name\_\_ == '\_\_main\_\_':

    main()

****

Here's the translated code into NetLogo:

Netlogo code

**to** setup-window  
 ;; Set up the turtle screen  
 clear-all  
 set-patch-size 2  
 set-background-color white  
 reset-ticks  
**end**  
  
**to** setup-turtle  
 ;; Create a turtle for drawing  
 create-turtles 1  
 [  
 set size 0.5  
 pen-up  
 ]  
**end**  
  
**to** draw-square [side-length]  
 ;; Draw a square for the Chaos Game  
 let start-x (- side-length / 2)  
 let start-y (- side-length / 2)  
 repeat 4  
 [  
 ask turtles  
 [  
 setxy start-x start-y  
 pen-down  
 forward side-length  
 right 90  
 pen-up  
 ]  
 set start-x start-x + side-length  
 set start-y start-y + side-length  
 ]  
**end**  
  
**to** chaos-game [vertices jump iterations]  
 ;; Perform the Chaos Game on a square  
 let current-point (list 0 0)  
 repeat iterations  
 [  
 let vertex random-one-of vertices  
 let new-x (item 0 vertex - item 0 current-point) \* jump + item 0 current-point  
 let new-y (item 1 vertex - item 1 current-point) \* jump + item 1 current-point  
 ask turtles  
 [  
 setxy new-x new-y  
 pen-down  
 dot 3  
 pen-up  
 ]  
 set current-point (list new-x new-y)  
 ]  
**end**  
  
**to** setup  
 setup-window  
 setup-turtle  
 let side-length 300 ;; Define the size of the square  
 let vertices [(list (- side-length / 2) (- side-length / 2))  
 (list (side-length / 2) (- side-length / 2))  
 (list (side-length / 2) (side-length / 2))  
 (list (- side-length / 2) (side-length / 2))]  
  
 draw-square side-length  
 ask turtles [set color red]  
  
 ;; Chaos game with jump = 0.5  
 chaos-game vertices 0.5 5000  
  
 ;; Reset turtle and try a larger jump  
 clear-all  
 setup-window  
 setup-turtle  
 draw-square side-length  
 ask turtles [set color blue]  
  
 ;; Chaos game with jump = 0.8  
 chaos-game vertices 0.8 5000  
**end**

This code translates the Python Chaos Game into NetLogo. It sets up the window, creates the turtles for drawing, defines the square, and performs the Chaos Game with different jump values. You can run this code in NetLogo to observe the Chaos Game visualisation.

**Script Details and Explanation**

**Setup Functions**: Initialize the drawing environment and define the square with given vertices.

**Chaos Game Implementation**: Using the Chaos Game, the script moves the turtle from an initial point towards a randomly chosen vertex by a specified jump fraction.

**There are two scenarios**: First, with a **jump = 0.5**, which fills the square uniformly, and then with a **jump = 0.8**, which leads to a more patterned or 'tartan' appearance, as the points more frequently cross the square, leading to distinct lines and crossings.

This script provides a direct way to visualize how different jumping fractions impact the patterns produced in the Chaos Game. It demonstrates the versatility and depth of such simple algorithmic rules in creating complex patterns from straightforward geometric bases.

The use of a pentagon in the Chaos Game, especially when implementing a jump factor derived from the reciprocal of the golden ratio, offers a fascinating approach to generating unique fractal patterns. The golden ratio (𝜙*ϕ*), approximately 1.618034, has profound connections to aesthetics and natural patterns, and its reciprocal provides an intriguing balance between too much and too little displacement.

When applying the Chaos Game to a pentagon with a jump equal to 1𝜙*ϕ*1​ or approximately 0.618034, you tend to see patterns that beautifully align with the pentagon's symmetry. These patterns often reveal deeper, fractal-like structures within the confines of the pentagon's boundary, including patterns reminiscent of the Cesaro fractal within the shape's interior.

**How This Happens:**

**Golden Ratio**: The reciprocal of the golden ratio provides a displacement that is neither too small to cause overlapping of points too frequently nor too large to send points outside the pentagon often. This "optimal" displacement tends to produce intricate and aesthetically pleasing patterns.

**Pentagon Geometry**: The angles and distances in a regular pentagon allow for a dynamic that enhances the visual complexity when points jump by 1𝜙*ϕ*1​ towards each vertex. The pentagon's vertices are positioned such that each jump retains an echo of the initial geometric order, enhancing the fractal-like behaviour inside the shape.

**Cesaro Fractal Resemblance**: Inside the boundary formed by the overall shape, the jumping rule tends to generate a pattern that has similarities to the Cesaro fractal. The Cesaro fractal itself is formed by repeatedly subdividing line segments and altering them by a certain angle, creating a jagged, recursive boundary. In the Chaos Game within a pentagon with jumps of 1𝜙*ϕ*1​, similar jagged, self-repeating structures can appear, especially noticeable along the lines defining the interior of the boundary.

**Implementing in Python with Turtle**

Here is a Python script using the **turtle** module to implement the Chaos Game with a pentagon and a jump of *ϕ*1​:

Python code

import turtle

import random

import math

def setup\_window():

    """ Set up the turtle screen. """

    window = turtle.Screen()

    window.bgcolor("white")

    window.tracer(0)

    return window

def setup\_turtle():

    """ Create a turtle for drawing. """

    chaos\_turtle = turtle.Turtle()

    chaos\_turtle.speed(0)  # Fastest speed

    chaos\_turtle.penup()

    return chaos\_turtle

def draw\_pentagon(chaos\_turtle, side\_length):

    """ Draw a pentagon for the Chaos Game. """

    start\_angle = 54  # To point the pentagon top upwards

    chaos\_turtle.right(start\_angle)

    for \_ in range(5):

        chaos\_turtle.pendown()

        chaos\_turtle.forward(side\_length)

        chaos\_turtle.right(72)

    chaos\_turtle.penup()

    chaos\_turtle.home()

def chaos\_game(chaos\_turtle, vertices, jump, iterations=10000):

    """ Perform the Chaos Game on a pentagon. """

    current\_point = [0, 0]  # Start in the center for simplicity

    for \_ in range(iterations):

        vertex = random.choice(vertices)

        current\_point[0] += (vertex[0] - current\_point[0]) \* jump

        current\_point[1] += (vertex[1] - current\_point[1]) \* jump

        chaos\_turtle.goto(current\_point[0], current\_point[1])

        chaos\_turtle.dot(3)  # Draw a small dot

def main():

    window = setup\_window()

    chaos\_turtle = setup\_turtle()

    side\_length = 200

    angle = 360 / 5

    vertices = []

    for i in range(5):

        angle\_rad = math.radians(angle \* i - 90)

        vertices.append((math.cos(angle\_rad) \* side\_length, math.sin(angle\_rad) \* side\_length))

    draw\_pentagon(chaos\_turtle, side\_length)

    chaos\_turtle.color('blue')

    # Chaos game with jump = 1 / phi (golden ratio)

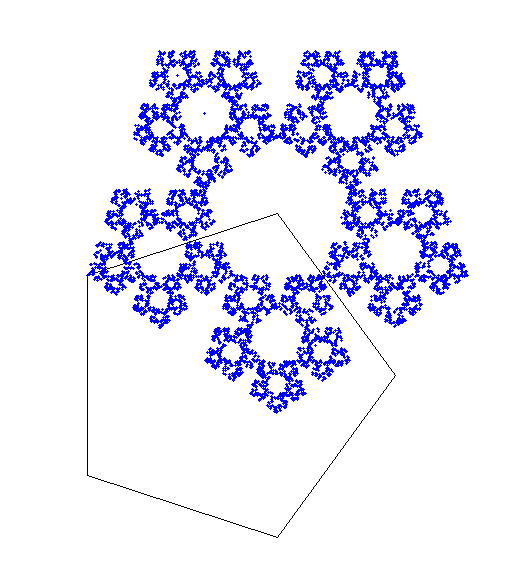
    jump = 1 / ((1 + math.sqrt(5)) / 2)

    chaos\_game(chaos\_turtle, vertices, jump=jump)

    window.mainloop()  # Keeps the window open

if \_\_name\_\_ == '\_\_main\_\_':

    main()



Here's the translation of the Python code into NetLogo:

Netlogo code

**to** setup-window  
 ;; Set up the turtle screen  
 clear-all  
 set-patch-size 2  
 set-background-color white  
 reset-ticks  
**end**  
  
**to** setup-turtle  
 ;; Create a turtle for drawing  
 create-turtles 1  
 [  
 set size 0.5  
 pen-up  
 ]  
**end**  
  
**to** draw-pentagon [side-length]  
 ;; Draw a pentagon for the Chaos Game  
 let start-angle 54 ;; To point the pentagon top upwards  
 right start-angle  
 repeat 5  
 [  
 ask turtles  
 [  
 pen-down  
 forward side-length  
 right 72  
 pen-up  
 ]  
 ]  
 set heading 0  
 setxy 0 0  
**end**  
  
**to** chaos-game [vertices jump iterations]  
 ;; Perform the Chaos Game on a pentagon  
 let current-point (list 0 0) ;; Start in the center for simplicity  
 repeat iterations  
 [  
 let vertex random-one-of vertices  
 let new-x (item 0 vertex - item 0 current-point) \* jump + item 0 current-point  
 let new-y (item 1 vertex - item 1 current-point) \* jump + item 1 current-point  
 ask turtles  
 [  
 setxy new-x new-y  
 pen-down  
 dot 3  
 pen-up  
 ]  
 set current-point (list new-x new-y)  
 ]  
**end**  
  
**to** setup  
 setup-window  
 setup-turtle  
 let side-length 200  
 let angle 360 / 5  
 let vertices []  
 repeat 5  
 [  
 let angle-rad (radians (angle \* ? - 90))  
 let new-x (cos angle-rad) \* side-length  
 let new-y (sin angle-rad) \* side-length  
 set vertices lput (list new-x new-y) vertices  
 ]  
  
 draw-pentagon side-length  
 ask turtles [set color blue]  
  
 ;; Chaos game with jump = 1 / phi (golden ratio)  
 let jump (1 / ((1 + sqrt 5) / 2))  
 chaos-game vertices jump 10000  
**end**

This code translates the Python Chaos Game into NetLogo. It sets up the window, creates the turtles for drawing, defines the pentagon, and performs the Chaos Game with the specified vertices and jump value. You can run this code in NetLogo to observe the Chaos Game visualisation.

**Script Explanation:**

**Setup**: The **turtle** is initialised along with the screen.

**Pentagon Drawing**: The Pentagon is drawn based on a given side length.

**Chaos Game**: Use the vertices of the pentagon and move by a factor of 1𝜙*ϕ*1​ towards a randomly chosen vertex.

This script visualises the unique fractal pattern you can expect from such a setup, emphasising the intriguing impact of mathematical constants like the golden ratio on fractal geometry.

The Chaos Game can be extended to other polygons, such as hexagons, where specific jump values like 2332​ can produce distinctive fractal boundaries that resemble the Koch snowflake curve. This observation underlines the role of jump distances in defining the limits and internal patterns of fractals generated by iterative point placement in regular polygons.

**How Jump Values Influence Fractal Patterns in a Hexagon:**

**Jump of 2/3​**: This jump distance is crucial for the fractal pattern observed with a hexagon. Unlike the more commonly used 1/2​ which fills the shape more uniformly, 2/3​ encourages point convergence towards the edges and vertices, leading to intricate, repeated boundary formations.

**Koch Curve Resemblance**: The Koch curve, known for its "triangular" protrusions creating a snowflake-like appearance, can manifest in the boundary behaviour of hexagons when using 2/3​ jump distances. This is due to the greater displacement toward each vertex, which, given the hexagonal symmetry, creates repetitive geometric protrusions resembling the Koch fractal.

**Implementing the Chaos Game with a Hexagon Using Python and Turtle**

Here's how you can implement this with a Python script using the **turtle** module to simulate the Chaos Game on a hexagon with a jump of 2/3​, aiming to observe the Koch curve-like boundaries:

Python code

import turtle

import random

import math

def setup\_window():

    """ Set up the turtle screen. """

    window = turtle.Screen()

    window.bgcolor("white")

    return window

def setup\_turtle():

    """ Create a turtle for drawing. """

    chaos\_turtle = turtle.Turtle()

    chaos\_turtle.speed(0)  # Fastest speed

    chaos\_turtle.penup()

    return chaos\_turtle

def draw\_hexagon(chaos\_turtle, side\_length):

    """ Draw a hexagon for the Chaos Game. """

    start\_angle = 90  # To point the hexagon flat side up

    chaos\_turtle.right(start\_angle)

    for \_ in range(6):

        chaos\_turtle.pendown()

        chaos\_turtle.forward(side\_length)

        chaos\_turtle.right(60)

    chaos\_turtle.penup()

    chaos\_turtle.home()

def chaos\_game(chaos\_turtle, vertices, jump, iterations=10000):

    """ Perform the Chaos Game on a hexagon. """

    current\_point = [0, 0]  # Start in the center for simplicity

    for \_ in range(iterations):

        vertex = random.choice(vertices)

        current\_point[0] += (vertex[0] - current\_point[0]) \* jump

        current\_point[1] += (vertex[1] - current\_point[1]) \* jump

        chaos\_turtle.goto(current\_point[0], current\_point[1])

        chaos\_turtle.dot(3)  # Draw a small dot

def main():

    window = setup\_window()

    chaos\_turtle = setup\_turtle()

    side\_length = 200

    angle = 360 / 6

    vertices = []

    for i in range(6):

        angle\_rad = math.radians(angle \* i - 90)

        vertices.append((math.cos(angle\_rad) \* side\_length, math.sin(angle\_rad) \* side\_length))

    draw\_hexagon(chaos\_turtle, side\_length)

    chaos\_turtle.color('blue')

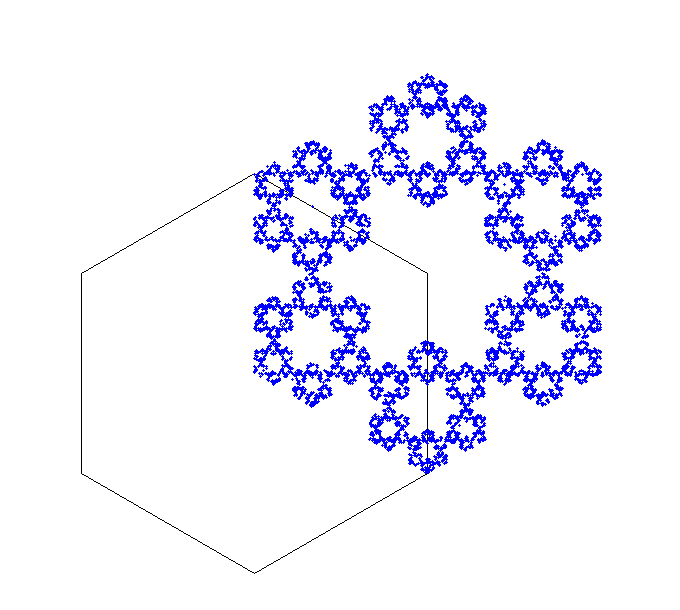
    # Chaos game with jump = 2/3

    chaos\_game(chaos\_turtle, vertices, jump=2/3)

    window.mainloop()  # Keeps the window open

if \_\_name\_\_ == '\_\_main\_\_':

    main()



the translation of the Python code into NetLogo:

NetLogo code

**to** setup-window  
 ;; Set up the turtle screen  
 clear-all  
 set-patch-size 2  
 set-background-color white  
 reset-ticks  
**end**  
  
**to** setup-turtle  
 ;; Create a turtle for drawing  
 create-turtles 1  
 [  
 set size 0.5  
 pen-up  
 ]  
**end**  
  
**to** draw-hexagon [side-length]  
 ;; Draw a hexagon for the Chaos Game  
 let start-angle 90 ;; To point the hexagon flat side up  
 right start-angle  
 repeat 6  
 [  
 ask turtles  
 [  
 pen-down  
 forward side-length  
 right 60  
 pen-up  
 ]  
 ]  
 set heading 0  
 setxy 0 0  
**end**  
  
**to** chaos-game [vertices jump iterations]  
 ;; Perform the Chaos Game on a hexagon  
 let current-point (list 0 0) ;; Start in the center for simplicity  
 repeat iterations  
 [  
 let vertex random-one-of vertices  
 let new-x (item 0 vertex - item 0 current-point) \* jump + item 0 current-point  
 let new-y (item 1 vertex - item 1 current-point) \* jump + item 1 current-point  
 ask turtles  
 [  
 setxy new-x new-y  
 pen-down  
 dot 3  
 pen-up  
 ]  
 set current-point (list new-x new-y)  
 ]  
**end**  
  
**to** setup  
 setup-window  
 setup-turtle  
 let side-length 200  
 let angle 360 / 6  
 let vertices []  
 repeat 6  
 [  
 let angle-rad (radians (angle \* ? - 90))  
 let new-x (cos angle-rad) \* side-length  
 let new-y (sin angle-rad) \* side-length  
 set vertices lput (list new-x new-y) vertices  
 ]  
  
 draw-hexagon side-length  
 ask turtles [set color blue]  
  
 ;; Chaos game with jump = 2/3  
 chaos-game vertices (2 / 3) 10000  
**end**

This code translates the Python Chaos Game with a hexagon into NetLogo. It sets up the window, creates the turtles for drawing, defines the hexagon, and performs the Chaos Game with the specified vertices and jump value. You can run this code in NetLogo to observe the Chaos Game visualisation.

**Script Explanation:**

**Setup and Drawing**: The script initialises the drawing environment, sets up the turtle, and draws a hexagon.

**Chaos Game Execution**: Using the hexagon's vertices, it moves by a factor of 2/3​ towards a randomly chosen vertex. This larger jump compared to 1/2​ encourages a more defined exploration of the space towards and along the hexagon's boundaries, potentially leading to patterns that visually resemble parts of the Koch curve, especially along the edges.

**Conclusion:**

This experiment with a hexagon and a jump of 2/3​ illustrates how specific parameters in a seemingly simple algorithm can generate complex and aesthetically significant fractal geometries. This approach highlights the intersection of geometry, randomness, and iterative algorithms in creating fascinating patterns that resonate with well-known fractal structures like the Koch curve.

In the Chaos Game, the patterns that emerge can be significantly influenced by the jump factor (the fraction of the distance moved towards a chosen vertex) and by restricting or altering the rules for choosing vertices. By limiting which vertices can be selected under certain conditions or by setting rules that dictate the sequence of vertex selection, you can create a diverse array of intricate and unique fractal patterns.

**Basic Principles of Restricting Vertex Choices**

**Restricting Random Selection**:

**Rule-based Selection**: Implement rules that restrict vertex choices based on previous choices. For example, you might rule out selecting the same vertex twice in succession or create a rule where the next vertex must be two positions away from the current one in a polygon.

**Conditional Logic**: Apply conditional logic to influence the choice of the next vertex based on factors like the current position within the shape, the last few vertices selected, or even more complex state conditions.

**Geometric and Dynamic Constraints**:

**Adjacent Vertex Skipping**: In a regular polygon, a straightforward rule could be to skip the next vertex clockwise or counterclockwise. This changes the usual dynamics and can lead to different fractal patterns, as seen in the "no consecutive repeat" rule commonly used in triangular Sierpinski setups.

**Vertex Path Memory**: By maintaining a memory of the last few vertices visited and setting rules that avoid repeating these vertices, new patterns can emerge. This approach can lead to fractals with more 'gaps' or complex tiling patterns.

**Examples of Restricting Choices**

**Sierpinski Triangle Modified**:

**Rule**: Do not return to the vertex just used.

**Effect**: This rule creates a path that tends to move across the triangle more dramatically, filling in areas that would otherwise be left blank in the standard Sierpinski process.

**Square Fractal Patterns**:

**Rule**: Alternate between non-adjacent vertices only.

**Effect**: This creates a crossing pattern, filling the square with a more complex network of lines, potentially leading to intricate lattice-like structures.

**Pentagon with Controlled Sequences**:

**Rule**: Choose vertices in a repeating pattern (e.g., skipping one vertex each time).

**Effect**: This can result in spiralling or star-shaped patterns within the pentagon, emphasising certain symmetries inherent in the five-sided shape.

**Implementing in Code**

To implement these ideas in a programming environment like Python using the turtle module, you would modify the Chaos Game vertex selection part to incorporate these rules. Here’s an example of how you might implement a rule that skips one vertex clockwise in a pentagon:

python code

def chaos\_game(chaos\_turtle, vertices, jump, iterations=10000):

    current\_point = [0, 0]  # Start in the center for simplicity

    last\_vertex\_index = None  # Track the last vertex used

    for \_ in range(iterations):

        if last\_vertex\_index is None:

            vertex\_index = random.randint(0, len(vertices) - 1)

        else:

            # Skip one vertex clockwise

            vertex\_index = (last\_vertex\_index + 2) % len(vertices)

        vertex = vertices[vertex\_index]

        current\_point[0] += (vertex[0] - current\_point[0]) \* jump

        current\_point[1] += (vertex[1] - current\_point[1]) \* jump

        chaos\_turtle.goto(current\_point[0], current\_point[1])

        chaos\_turtle.dot(3)  # Draw a small dot

        last\_vertex\_index = vertex\_index  # Update the last vertex used

**Conclusion**

By restricting vertex choices in the Chaos Game, you can manipulate the fractal outcomes in various ways, exploring mathematical creativity and discovering new patterns. Each set of rules can yield drastically different visuals, demonstrating the rich potential for variation even within the constraints of simple geometric shapes.

The example uses a square and a jump ratio of 0.5 with the added rule that the same vertex cannot be used twice.

The modification you're proposing to the Chaos Game, where the same vertex cannot be selected consecutively, adds an interesting dynamic when playing the game on a square with a jump ratio of 0.5. This rule ensures that the points do not merely converge towards the center but also explore other areas of the square more thoroughly, potentially leading to a more uniform distribution of points across the entire area of the square.

**Python Script for the Chaos Game on a Square with Modified Rule**

Here's how you can implement this modification using Python and the **turtle** module. This script will simulate the Chaos Game on a square, using a jump ratio of 0.5, with the rule that the same vertex cannot be chosen consecutively:

Python code

import turtle

import random

def setup\_window():

    """ Set up the turtle screen. """

    window = turtle.Screen()

    window.bgcolor("white")

    window.tracer(0)

    return window

def setup\_turtle():

    """ Create a turtle for drawing. """

    chaos\_turtle = turtle.Turtle()

    chaos\_turtle.speed(0)  # Fastest speed

    chaos\_turtle.penup()

    return chaos\_turtle

def draw\_square(chaos\_turtle, side\_length):

    """ Draw a square for the Chaos Game. """

    start\_pos = [-side\_length / 2, -side\_length / 2]

    chaos\_turtle.goto(start\_pos[0], start\_pos[1])

    for \_ in range(4):

        chaos\_turtle.pendown()

        chaos\_turtle.forward(side\_length)

        chaos\_turtle.left(90)

    chaos\_turtle.penup()

def chaos\_game(chaos\_turtle, vertices, jump, iterations=10000):

    """ Perform the Chaos Game on a square with no repeat vertex rule. """

    current\_point = [0, 0]  # Start in the center for simplicity

    last\_vertex = None  # Store the last vertex to avoid repeating it

    for \_ in range(iterations):

        vertex = random.choice([v for v in vertices if v != last\_vertex])

        current\_point[0] += (vertex[0] - current\_point[0]) \* jump

        current\_point[1] += (vertex[1] - current\_point[1]) \* jump

        chaos\_turtle.goto(current\_point[0], current\_point[1])

        chaos\_turtle.dot(3, 'blue')  # Draw a small dot

        last\_vertex = vertex  # Update the last vertex

def main():

    window = setup\_window()

    chaos\_turtle = setup\_turtle()

    side\_length = 200

    vertices = [

        (-side\_length / 2, -side\_length / 2),

        (side\_length / 2, -side\_length / 2),

        (side\_length / 2, side\_length / 2),

        (-side\_length / 2, side\_length / 2)

    ]

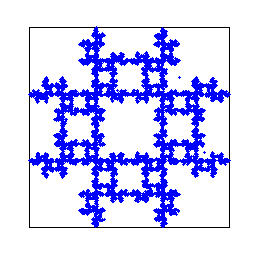
    draw\_square(chaos\_turtle, side\_length)

    chaos\_game(chaos\_turtle, vertices, jump=0.5)

    window.mainloop()  # Keeps the window open

if \_\_name\_\_ == '\_\_main\_\_':

    main()

****

Here's the translation of the Python code into NetLogo:

NetLogo code

**to** setup-window  
 ;; Set up the turtle screen  
 clear-all  
 set-patch-size 2  
 set-background-color white  
 reset-ticks  
**end**  
  
**to** setup-turtle  
 ;; Create a turtle for drawing  
 create-turtles 1  
 [  
 set size 0.5  
 pen-up  
 ]  
**end**  
  
**to** draw-square [side-length]  
 ;; Draw a square for the Chaos Game  
 let start-pos list (- side-length / 2) (- side-length / 2)  
 ask turtles  
 [  
 setxy item 0 start-pos item 1 start-pos  
 repeat 4  
 [  
 pen-down  
 forward side-length  
 left 90  
 ]  
 pen-up  
 ]  
**end**  
  
**to** chaos-game [vertices jump iterations]  
 ;; Perform the Chaos Game on a square with no repeat vertex rule  
 let current-point (list 0 0) ;; Start in the center for simplicity  
 let last-vertex nobody ;; Store the last vertex to avoid repeating it  
 repeat iterations  
 [  
 let possible-vertices []  
 foreach vertices  
 [  
 ifelse-value (self != last-vertex)  
 [  
 set possible-vertices lput self possible-vertices  
 ]  
 []  
 ]  
 let vertex random-one-of possible-vertices  
 let new-x (item 0 vertex - item 0 current-point) \* jump + item 0 current-point  
 let new-y (item 1 vertex - item 1 current-point) \* jump + item 1 current-point  
 ask turtles  
 [  
 setxy new-x new-y  
 pen-down  
 dot 3 blue ;; Draw a small dot  
 pen-up  
 ]  
 set last-vertex vertex ;; Update the last vertex  
 set current-point (list new-x new-y)  
 ]  
**end**  
  
**to** setup  
 setup-window  
 setup-turtle  
 let side-length 200  
 let vertices [(list (- side-length / 2) (- side-length / 2)) ((side-length / 2) (- side-length / 2)) ((side-length / 2) (side-length / 2)) ((- side-length / 2) (side-length / 2))]  
  
 draw-square side-length  
 ask turtles [set color blue]  
  
 ;; Chaos game with jump = 0.5  
 chaos-game vertices 0.5 10000  
**end**

This NetLogo code translates the Python Chaos Game with a square and a no-repeat vertex rule. It sets up the window, creates the turtle for drawing, defines the square, and performs the Chaos Game with the specified vertices and jump value. You can run this code in NetLogo to observe the Chaos Game visualization.

**Explanation of the Script**

**Setup**: The turtle environment is initialized, and a square is drawn where the Chaos Game will be played.

**Chaos Game Function**: In the **chaos\_game** function, the points are plotted by moving halfway (0.5) towards a randomly chosen vertex, with the condition that the same vertex cannot be chosen consecutively. This is enforced by filtering out the last selected vertex from the list of possible choices.

**Vertex Selection**: A list comprehension **[v for v in vertices if v != last\_vertex]** ensures that the vertex chosen in the current iteration is not the same as the one selected in the previous iteration.

**Impact of the No Consecutive Vertex Rule**

This rule prevents the process from converging quickly to a static equilibrium near the centre of the square, which might happen if the same vertex could be chosen repeatedly. Instead, the points explore the space more dynamically, creating a more even distribution across the square area, which could lead to discovering more varied and interesting patterns within the confines of the geometric structure.

The Chaos Game provides a rich playground for exploring the intersection of geometric shapes, randomness, and emergent patterns. By introducing further restrictions on vertex selection, one can manipulate the emerging patterns, programming the randomness to yield more complex or specific designs. Here’s a deeper look into how additional rules might affect the outcomes, especially when using shapes like squares, pentagons, hexagons, or more complex polygons.

1. Neighbouring Vertices Exclusion

**Rule**: Prohibit the selection of any vertex adjacent to the most recently chosen vertex.

**Effect**: This rule forces the point to "jump" across the shape, ensuring a wider coverage and potentially creating spiral or zigzag patterns. In polygons like hexagons or octagons, this can generate intricate web-like structures as the point continually skips over adjacent vertices.

**Example**: In a hexagon, if the point is currently at vertex A, the next vertex cannot be B or F (if vertices are labelled clockwise).

2. Opposite Vertices Preference

**Rule**: Prefer or exclusively choose the vertex directly opposite the current vertex in symmetrical polygons.

**Effect**: This strategy can create starkly contrasting lines that intersect the polygon, often leading to symmetrical patterns that reflect across the shape’s centre. This is particularly interesting in shapes like squares and hexagons, where opposite vertices clearly define lines of symmetry.

**Example**: In a square, the next vertex must be C from vertex A.

3. Area-based Vertex Selection

**Rule**: Choose vertices based on a specific sub-area within the shape, such as aiming for vertices that will keep the point within a designated subset of the shape or avoiding vertices that would move the point into another area.

**Effect**: This can be used to fill specific areas more densely than others or to create patterns that only populate certain parts of the shape, such as making a pattern that resembles a specific symbol or letter by avoiding or favouring particular trajectories.

**Example**: In a complex polygon, choose vertices that maintain the point within a sub-area shaped like a star or heart.

4. Conditional Vertex Choice Based on External Factors

**Rule**: Use external factors like the current iteration number, the time of day, or input from an external device to influence vertex choice.

**Effect**: This can randomize the pattern further or synchronize the pattern creation with external events, adding an additional layer of complexity or interaction with the environment.

**Example**: Use the system clock's seconds value mod the number of vertices to choose the next vertex, creating a pattern that changes based on the time of day.

Implementing Advanced Rules in Python

Let's consider a scenario using the neighbour vertices exclusion rule in a pentagon. Here's how you might modify the Chaos Game to incorporate this rule using Python and the **turtle** module:

Python code

import turtle

import random

import math

def setup\_turtle():

    window = turtle.Screen()

    window.bgcolor("white")

    t = turtle.Turtle()

    t.speed(0)

    t.penup()

    return t

def draw\_polygon(t, sides, length):

    angle = 360 / sides

    start\_angle = 90 if sides % 2 == 0 else 180 / sides

    t.right(start\_angle)

    for \_ in range(sides):

        t.pendown()

        t.forward(length)

        t.right(angle)

    t.penup()

    t.home()

def chaos\_game(t, vertices, jump, restrictions, iterations=10000):

    current\_point = [0, 0]

    last\_vertex = None

    for \_ in range(iterations):

        viable\_vertices = [v for i, v in enumerate(vertices) if not restrictions(last\_vertex, i)]

        vertex = random.choice(viable\_vertices)

        current\_point[0] += (vertex[0] - current\_point[0]) \* jump

        current\_point[1] += (vertex[1] - current\_point[1]) \* jump

        t.goto(current\_point[0], current\_point[1])

        t.dot(3, 'blue')

        last\_vertex = vertices.index(vertex)

def neighbor\_restriction(last\_index, current\_index, sides=5):

    if last\_index is None:

        return False

    return abs(last\_index - current\_index) == 1 or abs(last\_index - current\_index) == sides - 1

def main():

    t = setup\_turtle()

    sides = 5

    length = 200

    vertices = [(length \* math.cos(2 \* math.pi \* i / sides), length \* math.sin(2 \* math.pi \* i / sides)) for i in range(sides)]

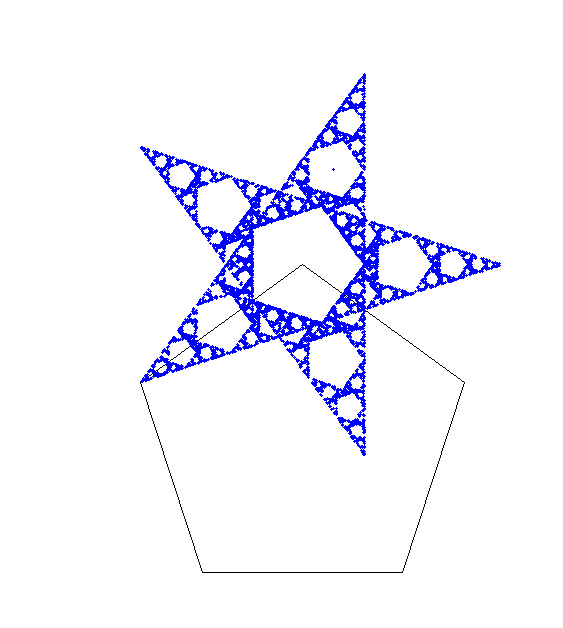
    draw\_polygon(t, sides, length)

    chaos\_game(t, vertices, 0.5, lambda last, current: neighbor\_restriction(last, current, sides))

    turtle.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

    main()



Here's the translation of the Python code into NetLogo:

NetLogo code

**to** setup-turtle  
 ;; Set up the turtle screen  
 clear-all  
 set-patch-size 2  
 set-background-color white  
  
 ;; Create a turtle for drawing  
 create-turtles 1  
 [  
 set size 1  
 pen-up  
 ]  
**end**  
  
**to** draw-polygon [sides length]  
 ;; Draw a polygon for the Chaos Game  
 let angle 360 / sides  
 let start-angle ifelse-value (sides mod 2 = 0) [90] [180 / sides]  
 ask turtles  
 [  
 right start-angle  
 repeat sides  
 [  
 pendown  
 forward length  
 right angle  
 ]  
 pen-up  
 home  
 ]  
**end**  
  
**to** chaos-game [vertices jump restrictions iterations]  
 ;; Perform the Chaos Game on a polygon with restrictions  
 let current-point (list 0 0)  
 let last-vertex nobody  
 repeat iterations  
 [  
 let viable-vertices []  
 foreach vertices  
 [  
 let i position self vertices  
 ifelse-value (not restrictions last-vertex i)  
 [  
 set viable-vertices lput self viable-vertices  
 ]  
 []  
 ]  
 let vertex random-one-of viable-vertices  
 let new-x (item 0 vertex - item 0 current-point) \* jump + item 0 current-point  
 let new-y (item 1 vertex - item 1 current-point) \* jump + item 1 current-point  
 ask turtles  
 [  
 setxy new-x new-y  
 pendown  
 dot 3 blue ;; Draw a small dot  
 pen-up  
 ]  
 set last-vertex vertex ;; Update the last vertex  
 set current-point (list new-x new-y)  
 ]  
**end**  
  
**to** neighbor-restriction [last-index current-index sides]  
 ;; Apply neighbor restriction to vertices  
 if last-index = nobody [report **false**]  
 report abs(last-index - current-index) = 1 or abs(last-index - current-index) = sides - 1  
**end**  
  
**to** main  
 setup-turtle  
 let sides 5  
 let length 200  
 let vertices []  
 repeat sides  
 [  
 let i position self vertices  
 let angle 360 / sides \* i  
 set vertices lput (list (length \* cos (angle / 180 \* pi)) (length \* sin (angle / 180 \* pi))) vertices  
 ]  
 draw-polygon sides length  
 ask turtles [set color blue]  
  
 ;; Chaos game with jump = 0.5 and neighbor restriction  
 chaos-game vertices 0.5 neighbor-restriction sides  
  
 reset-ticks  
**end**

This NetLogo code translates the Python Chaos Game with a polygon and neighbor restriction. It sets up the window, creates the turtle for drawing, defines the polygon, and performs the Chaos Game with the specified vertices, jump value, and neighbour restriction. You can run this code in NetLogo to observe the Chaos Game visualization.

This script modifies the Chaos Game to exclude neighbouring vertices in a pentagon, potentially leading to unique fractal patterns that reflect the constraints of skipping over adjacent vertices. Such restrictions can vastly change the resulting dynamics and aesthetics of the patterns formed, demonstrating the versatility and depth of the Chaos Game in exploring mathematical and artistic concepts through programming.

The chaos game is a method of computing an Iterated Function System (IFS). An IFS consists of one of more contractive functions (that is, functions which may reduce but not increase the distance between any two points) and select from them randomly at each iteration. The output of each function becomes the input to the next.

Indeed, the Chaos Game is intrinsically linked to the concept of Iterated Function Systems (IFS), which are used extensively in fractal geometry to produce intricate and fascinating patterns that are self-similar across different scales. Understanding how the Chaos Game relates to IFSs reveals more profound insights into the mathematical theory and practical applications of fractals.

**Basics of Iterated Function Systems (IFS)**

An IFS is typically defined by the following components:

**Contractive Functions**: These transformations reduce the distance between points in a space. In mathematical terms, a function 𝑓 is contractive on a space if there exists a constant 𝑐 between 0 and 1 such that for all points 𝑥 and 𝑦, the distance 𝑑(𝑓(𝑥),𝑓(𝑦))≤𝑐×𝑑(𝑥,𝑦). The constant 𝑐*c* is known as the contraction factor.

**Random Selection**: The functions are chosen randomly in each iteration, and the output of one function becomes the input to the next. This randomness is critical in exploring the space and ensures that the fractal is filled out evenly over many iterations.

**Fixed Point (Attractor)**: Due to the contractive nature of the functions, the IFS tends to converge toward a fixed point or a set of points that are invariant under the functions of the IFS. This set of points often forms a fractal.

**The Chaos Game as an IFS**

In the Chaos Game:

Each move towards a vertex can be considered as applying a contractive affine transformation. For a triangle, these transformations might be:

𝑇𝐴(𝑥,𝑦)=(0.5×𝑥+0.5×𝑥𝐴,0.5×𝑦+0.5×𝑦𝐴) for vertex A

𝑇𝐵(𝑥,𝑦)=(0.5×𝑥+0.5×𝑥𝐵,0.5×𝑦+0.5×𝑦𝐵) for vertex B

𝑇𝐶(𝑥,𝑦)=(0.5×𝑥+0.5×𝑥𝐶,0.5×𝑦+0.5×𝑦𝐶) for vertex C Here, (𝑥𝐴,𝑦𝐴)), (𝑥𝐵,𝑦𝐵), and (𝑥𝐶,𝑦𝐶) are the coordinates of the vertices A, B, and C respectively.

Each of these functions is contractive with a factor of 0.5, ensuring that distances are halved with each application. The transformations are randomly selected, adhering to the principles of an IFS.

**Applications and Visualization**

**Fractal Geometry**: The Sierpinski Triangle is a classic example generated by an IFS using a triangle as the base shape. Other fractals, like the Barnsley Fern or various fractal trees, can also be generated using more complex IFSs with more transformations.

**Data Compression**: IFSs can be used in image compression, where the image is represented as a fractal, allowing for high compression levels.

**Modelling Natural Phenomena**: Many natural shapes, such as clouds, mountain ranges, and coastlines, can be modelled using fractals generated by IFSs, reflecting their self-similar structure.

The Chaos Game provides a compelling visual and intuitive approach to understanding IFSs. It emphasizes how simple iterative rules, when applied under constraints of contraction and randomness, can lead to complex, organized structures emerging from seemingly chaotic processes. This connection between simple mathematical rules and complex natural patterns is one of the most fascinating aspects of fractal geometry.

Michael Barnsley formulated the Collage Theorem, a foundational concept in the field of fractals. It offers a compelling mathematical framework, demonstrating how Iterated Function Systems (IFS) can create any given shape or, more precisely, any compact set in a metric space. This theorem is not just of theoretical interest but also has practical applications in image compression and synthesis.

**Understanding the Collage Theorem**

**Basic Principle**: The Collage Theorem states that for any non-empty compact set 𝐸*E* in a metric space (such as an image or shape), and for any desired degree of accuracy 𝜖>0*ϵ*>0, there exists an IFS whose attractor 𝐴*A* (the fractal resulting from repeatedly applying the IFS) is within 𝜖*ϵ* of 𝐸*E* in terms of the Hausdorff distance (a way to measure the distance between subsets of a space).

**How It Works**:

**Selection of Functions**: The theorem suggests that you can "cover" or "approximate" any shape by carefully choosing and applying a set of contractive transformations. Each part of the shape 𝐸*E* is mapped onto itself by one of the transformations in the IFS.

**Covering the Shape**: You construct the IFS so that the union of the images of the shape under the transformations is as close as possible to the original shape. Essentially, the original shape is "collaged" from smaller copies of itself.

**Application to IFS**

If you want to recreate a particular shape using an IFS:

**Choose a Set of Transformations**: Identify transformations that, when applied, produce smaller parts that can be pieced together to resemble the entire shape.

**Apply the Collage Theorem**: Adjust these transformations so that when they are applied repeatedly (as in an IFS), the limit set of the process—the attractor—is as close as possible to the desired shape.

**Implications and Uses**

**Image Compression**: The Barnsley Fern is a famous example where a particular set of affine transformations reproduces a complex figure resembling a fern. This principle is utilized in fractal image compression, where images are stored not as pixel arrays but as parameters defining transformations in an IFS.

**Pattern Generation**: The theorem is used in graphic design and digital art to create intricate patterns and textures that are both efficient to compute and aesthetically appealing.

**Practical Example: Creating a Fractal Leaf**

Consider you want to design a fractal leaf using the Collage Theorem. You would proceed by:

**Identifying Key Features**: Recognize main parts of the leaf such as the stem, tip, and body.

**Defining Transformations**: Develop transformations that reduce the whole leaf into these smaller components. For instance, one transformation might scale down the whole leaf to fit just the tip, another for the body, and a third for the stem.

**Iterative Application**: By applying these transformations iteratively, you construct an IFS whose attractor closely resembles the leaf.

**Conclusion**

The Collage Theorem not only enriches our understanding of fractals but also bridges the gap between abstract mathematical theory and practical application, showing how complex structures can be systematically reconstructed from simple, repeated patterns. This approach underscores the deep interconnectedness between geometry, art, and computation in contemporary mathematical sciences.

the Collage Theorem, formulated by Michael Barnsley, has significant practical implications, particularly in the field of image compression. This theorem forms the basis of fractal image compression, a method that leverages the properties of fractals to store image data efficiently. Here’s a detailed exploration of how the Collage Theorem is applied in fractal image compression and the benefits and challenges it presents.

**Fractal Image Compression: Overview**

**Concept**: Fractal image compression is based on the idea that in many natural images (such as those of landscapes, plants, and various physical structures), parts of the image resemble other parts. The Collage Theorem supports the concept that any image can be represented as a fixed point of an iterated function system (IFS), where the IFS consists of contractive transformations. Each transformation encodes a specific part of the image, and when these transformations are applied repeatedly, they reconstruct the original image.

**Application of the Collage Theorem**

**Image Segmentation**: The image to be compressed is divided into ranges (blocks of the image that need to be compressed) and domains (larger blocks that can be transformed to approximate the ranges).

**Searching for Similarities**: For each range block, the algorithm searches for a domain block that can be transformed to closely match the range. This transformation involves affine transformations including scaling, rotation, and contrast adjustments, aligning with the contractive nature stipulated by the Collage Theorem.

**Storing Transformations**: Instead of storing pixel values directly, the compression algorithm stores the parameters of the transformations (affine coefficients, offsets, contrast settings, etc.). Because these parameters take up less space than the original pixels, the image can be compressed effectively.

**Decompression**: To reconstruct the image, the stored transformations are applied iteratively starting from any initial image (typically a blank image), and due to the contractive nature of the transformations guaranteed by the Collage Theorem, the process converges to the attractor of the IFS, which closely resembles the original image.

**Benefits of Fractal Image Compression**

**High Compression Ratios**: Fractal compression can achieve higher ratios than traditional methods like JPEG, especially at higher levels of compression.

**Resolution Independence**: One of the unique features of fractal compression is that it's resolution-independent. You can scale the decompressed image to higher resolutions without the typical loss in quality seen in pixel-based images.

**Efficiency in Decoding**: The decompression process is relatively fast and can be implemented efficiently on hardware, making it suitable for streaming applications where decompression speed is critical.

**Challenges and Limitations**

**Computational Intensity in Compression**: Finding the best domain blocks and the optimal transformations for each range block is computationally intensive and time-consuming, making the compression process slower than other methods.

**Artifacts**: Like other compression methods, fractal compression can introduce artifacts, especially at high compression ratios. These are typically a result of the approximation methods used in identifying domain blocks and transformations.

**Limited Applications**: Due to its computational demands and the nature of its efficiencies (better at higher compressions), fractal compression has found limited application compared to more straightforward methods like JPEG or PNG for general use.

**Conclusion**

The Collage Theorem provides a robust theoretical foundation for fractal image compression, offering an elegant mathematical framework for understanding and applying fractals in practical scenarios. While the method presents certain challenges, particularly in terms of compression time, its unique advantages make it a fascinating area of study and an ongoing field of research in image processing.

Barnsley’s fern is an IFS of four functions on an (x,y) coordinate.

1. xn+1 = 0 yn+1 = 0.16yn
2. xn+1 = 0.85xn + 0.04yn  yn+1 = -0.04xn + 0.85yn + 1.6
3. xn+1 = 0.2xn - 0.26yn  yn+1 = 0.23xn + 0.22yn + 1.6
4. xn+1 = -0.15xn + 0.28yn  yn+1 = 0.26xn + 0.24n + 0.44

We choose functions with the following probabilities:

1% 2. 85% 3 and 4: 7%

The details you've provided on Barnsley's Fern offer a fascinating glimpse into the application of Iterated Function Systems (IFS) to generate complex fractal patterns. Barnsley's Fern uses a set of four affine transformations applied to coordinates in the (𝑥,𝑦) plane, each with specific probabilities that dictate their selection frequency during the iteration process. This fractal is a classic example in the study of IFS because it demonstrates the power of relatively simple mathematical rules to create intricate natural patterns and underscores such systems' probabilistic nature.

**Analysis of Barnsley's Fern IFS**

Each of the four transformations plays a distinct role in forming the overall shape of the fern:

**Stem of the Fern**:

The first transformation:

𝑥𝑛+1=0,𝑦𝑛+1=0.16𝑦𝑛​

This transformation scales 𝑦 by 0.16 and does not change 𝑥, resulting in a vertical contraction. It is applied only 1% of the time, generating the stem's vertical structure.

**Successively Larger Leaflets**:

The second transformation:

𝑥𝑛+1=0.85𝑥𝑛+0.04𝑦𝑛,𝑦𝑛+1=−0.04𝑥𝑛+0.85𝑦𝑛+1.6

Applied 85% of the time, this transformation primarily builds the bulk of the leaf. It scales, rotates slightly, and translates the points, contributing to the larger, repetitive pattern that makes up the main body of the fern.

**Left Leaflet**:

The third transformation:

𝑥𝑛+1=0.2𝑥𝑛−0.26𝑦𝑛,𝑦𝑛+1=0.23𝑥𝑛+0.22𝑦𝑛+1.6

With a 7% probability, this transformation helps form the smaller leaflets on the left side, adding asymmetry and complexity to the fern.

**Right Leaflet**:

The fourth transformation:

𝑥𝑛+1=−0.15𝑥𝑛+0.28𝑦𝑛,𝑦𝑛+1=0.26𝑥𝑛+0.24𝑦𝑛+0.44

Also, at 7% probability, this mirrors the role of the third transformation, but for the right side, it balances the overall appearance of the fern.

**Visual Representation and Theoretical Implications**

These transformations collectively ensure that each point generated from one iteration feeds into the next, creating a self-similar structure characteristic of fractals. The different scaling, rotation, and translation parameters in each transformation influence the fern's appearance, while the probabilities control how densely different parts of the fern are filled in.

Adjusting the parameters and probabilities can generate different types of fractal plants, demonstrating the adaptability and power of fractal geometry in modelling natural phenomena. The process vividly illustrates how deterministic algorithms (governed by fixed rules and probabilities) can generate structures of great complexity and beauty, mimicking the randomness found in nature.

This system provides a method for creating visually appealing patterns and a tool for exploring mathematical concepts in symmetry, scaling, and recursion, making it a valuable educational and artistic resource.

Here's a Python example using the **turtle** module to generate Barnsley's Fern based on the provided IFS and probabilities:

Python code

import turtle

import random

# Define the transformations with their probabilities

transformations = [

    (lambda x, y: (0, 0.16 \* y), 0.01),  # Stem

    (lambda x, y: (0.85 \* x + 0.04 \* y, -0.04 \* x + 0.85 \* y + 1.6), 0.85),  # Larger leaflets

    (lambda x, y: (0.2 \* x - 0.26 \* y, 0.23 \* x + 0.22 \* y + 1.6), 0.07),  # Left leaflet

    (lambda x, y: (-0.15 \* x + 0.28 \* y, 0.26 \* x + 0.24 \* y + 0.44), 0.07)  # Right leaflet

]

# Initialize the turtle

t = turtle.Turtle()

t.speed(0)

t.penup()

t.setpos(0, -300)

t.pendown()

# Set the number of iterations between screen updates

iterations\_between\_updates = 500

# Iterate and draw the fern

for i in range(10000):

    # Choose a random transformation based on probabilities

    rand\_num = random.random()

    probability\_sum = 0

    chosen\_transform = None

    for transform, probability in transformations:

        probability\_sum += probability

        if rand\_num < probability\_sum:

            chosen\_transform = transform

            break

    # Apply the chosen transformation

    new\_x, new\_y = chosen\_transform(t.xcor(), t.ycor())

    t.goto(new\_x, new\_y)

    # Check if it's time to update the screen

    if (i + 1) % iterations\_between\_updates == 0:

        turtle.update()

# Hide the turtle and display the final drawing

t.hideturtle()

turtle.done()

This code uses the **turtle** module to visually represent Barnsley's Fern. Each transformation function is randomly selected based on its associated probability, and the turtle moves accordingly to draw the fern. Adjust the number of iterations (**range(10000)**) to change the density of the fern.

Here's an example of how you can implement Barnsley's Fern in NetLogo:

NetLogo code

**globals** [  
 iterations-between-updates  
]  
  
**to** setup  
 clear-all  
 reset-ticks  
 set iterations-between-updates 500  
   
 ; Initialize turtle settings  
 create-turtles 1 [  
 setxy 0 -300  
 pen-down  
 ]  
   
 ; Create buttons  
 create-buttons 1 [  
 setxy -10 10  
 set size 2  
 set color white  
 set label "Run Simulation"  
 set action "run-simulation"  
 ]  
   
 create-buttons 1 [  
 setxy -10 -10  
 set size 2  
 set color white  
 set label "Increase Iterations"  
 set action "increase-iterations"  
 ]  
**end**  
  
**to** run-simulation  
 draw-fern  
**end**  
  
**to** draw-fern  
 ; Fern drawing code  
 let transformations [  
 [(0)(0.16 \* [ycor])] with [random-float 1 < 0.01] ; Stem  
 [(0.85 \* [xcor] + 0.04 \* [ycor])(-0.04 \* [xcor] + 0.85 \* [ycor] + 1.6)] with [random-float 1 < 0.85] ; Larger leaflets  
 [(0.2 \* [xcor] - 0.26 \* [ycor])(0.23 \* [xcor] + 0.22 \* [ycor] + 1.6)] with [random-float 1 < 0.07] ; Left leaflet  
 [(-0.15 \* [xcor] + 0.28 \* [ycor])(0.26 \* [xcor] + 0.24 \* [ycor] + 0.44)] with [random-float 1 < 0.07] ; Right leaflet  
 ]  
   
 ask turtles [  
 repeat 10000 [  
 let chosen-transform random-one-of transformations  
 let new-x first chosen-transform  
 let new-y second chosen-transform  
 setxy new-x new-y  
   
 ; Check if it's time to update the display  
 if ticks mod iterations-between-updates = 0 [  
 tick  
 ]  
 ]  
 ]  
**end**  
  
**to** increase-iterations  
 set iterations-between-updates iterations-between-updates + 100  
**end**

This NetLogo code defines a procedure **draw-fern** that implements Barnsley's Fern. We represent the transformations and their associated probabilities within a list called **transformations**. The turtle then iterates through 10,000 steps, applying a randomly chosen transformation at each step.

The display is updated every **iterations-between-updates** ticks to improve performance. You can adjust the **iterations-between-updates** variable to change how often the display updates occur.

Let's delve into the details of linear transformations and the matrices associated with scaling, shearing, and rotation.

**Linear Transformations:**

A linear transformation is a function that maps vectors from one space to another while preserving certain properties, such as straight lines, planes, and the origin. In two-dimensional space, linear transformations can be represented by matrices.

**Scaling Matrix:**

A scaling matrix is a type of linear transformation that enlarges or shrinks an object. In a 2D space, a scaling matrix is represented as:

A group of black letters

Description automatically generated

Here, 𝑎 and 𝑏 represent the scaling factors in the x and y directions. Multiplying a point (𝑥,𝑦) by this matrix will result in a new point (𝑎𝑥,𝑏𝑦), where the x-coordinate is scaled by 𝑎*a* and the y-coordinate is scaled by 𝑏.

**Shearing Matrix:**

A shearing matrix is a linear transformation that distorts the shape of an object by shifting points along one axis. There are two types of shearing: horizontal (x-direction) and vertical (y-direction).

Horizontal Shearing: The shearing matrix for horizontal shearing is represented as:

A number and a line

Description automatically generated with medium confidence

Here, 𝑎 represents the shearing factor in the x-direction. When a point (𝑥,𝑦) is multiplied by this matrix, the x-coordinate is shifted by 𝑎𝑦*ay*, resulting in a sheared shape.

Vertical Shearing: The shearing matrix for vertical shearing is represented as:

A number and number in a row

Description automatically generated with medium confidence

Here, 𝑏 represents the shearing factor in the y-direction. When a point (𝑥,𝑦) is multiplied by this matrix, the y-coordinate is shifted by 𝑏𝑥, resulting in a vertically sheared shape.

**Rotation Matrix:**

A rotation matrix is a linear transformation that rotates points around the origin by a specified angle 𝜃*θ*. The rotation matrix in 2D space is given by:

A close-up of a number

Description automatically generated

Here, 𝜃 represents the angle of rotation. When a point (𝑥,𝑦) is multiplied by this matrix, it rotates counterclockwise by an angle 𝜃 around the origin.

**Summary:**

**Scaling**: Enlarges or shrinks objects along the x and y directions.

**Shearing**: Distorts the shape of objects by shifting points along the x or y-axis.

**Rotation**: Rotates points counterclockwise around the origin by a specified angle 𝜃.

These linear transformations are fundamental concepts in linear algebra and play crucial roles in computer graphics, image processing, and many other fields.

To visualize the effect of scaling, shearing, and rotation matrices, we can create a simple Python example using matplotlib to plot points before and after applying these transformations. Let's implement each transformation and visualize its effect on a set of points.

Python code

import numpy as np

import matplotlib.pyplot as plt

# Define the points to be transformed

points = np.array([[0, 0], [1, 1], [2, 2], [3, 3]])

# Define scaling matrix

def scaling\_matrix(a, b):

    return np.array([[a, 0], [0, b]])

# Define shearing matrix

def shearing\_matrix(a, b):

    return np.array([[1, a], [b, 1]])

# Define rotation matrix

def rotation\_matrix(theta):

    cos\_theta = np.cos(theta)

    sin\_theta = np.sin(theta)

    return np.array([[cos\_theta, -sin\_theta], [sin\_theta, cos\_theta]])

# Plot points before transformation

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.scatter(points[:, 0], points[:, 1], color='blue')

plt.title('Before Transformation')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

# Apply scaling transformation

scaled\_points = np.dot(points, scaling\_matrix(2, 0.5))

# Plot points after scaling transformation

plt.subplot(1, 2, 2)

plt.scatter(scaled\_points[:, 0], scaled\_points[:, 1], color='red')

plt.title('After Scaling')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.tight\_layout()

plt.show()

# Apply shearing transformation

sheared\_points = np.dot(points, shearing\_matrix(0.5, 0.5))

# Plot points after shearing transformation

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.scatter(sheared\_points[:, 0], sheared\_points[:, 1], color='green')

plt.title('After Shearing')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

# Apply rotation transformation

rotated\_points = np.dot(points, rotation\_matrix(np.pi/4))  # Rotate by 45 degrees

# Plot points after rotation transformation

plt.subplot(1, 2, 2)

plt.scatter(rotated\_points[:, 0], rotated\_points[:, 1], color='purple')

plt.title('After Rotation')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.tight\_layout()

plt.show()

A comparison of a white sheet with red dots

Description automatically generated

In this example:

We define a set of points **[0, 0], [1, 1], [2, 2], [3, 3]**.

Using the defined transformation matrices, we then apply scaling, shearing, and rotation transformations to these points.

Finally, we plot the points before and after each transformation to visualise their effects.

You can adjust the parameters of the transformations (such as scaling factors, shearing factors, and rotation angles) to observe different effects on the points.

combinations of these transformations can be achieved by multiplying their corresponding matrices together. This allows us to create more complex transformations by composing simpler ones. Let's explore how we can combine scaling, shearing, and rotation transformations in Python using matrix multiplication:

python code

import numpy as np

import matplotlib.pyplot as plt

# Define the points to be transformed

points = np.array([[0, 0], [1, 1], [2, 2], [3, 3]])

# Define scaling matrix

def scaling\_matrix(a, b):

    return np.array([[a, 0], [0, b]])

# Define shearing matrix

def shearing\_matrix(a, b):

    return np.array([[1, a], [b, 1]])

# Define rotation matrix

def rotation\_matrix(theta):

    cos\_theta = np.cos(theta)

    sin\_theta = np.sin(theta)

    return np.array([[cos\_theta, -sin\_theta], [sin\_theta, cos\_theta]])

# Define combination of transformations: scaling -> shearing -> rotation

def combined\_transformation(points, scale\_factor, shear\_factor, rotation\_angle):

    scaling = scaling\_matrix(scale\_factor, scale\_factor)

    shearing = shearing\_matrix(shear\_factor, shear\_factor)

    rotation = rotation\_matrix(rotation\_angle)

    # Combine transformations: scaling -> shearing -> rotation

    combined\_matrix = np.dot(rotation, np.dot(shearing, scaling))

    transformed\_points = np.dot(points, combined\_matrix)

    return transformed\_points

# Plot points before transformation

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.scatter(points[:, 0], points[:, 1], color='blue')

plt.title('Before Transformation')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

# Apply combined transformation: scaling -> shearing -> rotation

transformed\_points = combined\_transformation(points, 2, 0.5, np.pi/4)

# Plot points after combined transformation

plt.subplot(1, 2, 2)

plt.scatter(transformed\_points[:, 0], transformed\_points[:, 1], color='red')

plt.title('After Combined Transformation')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.tight\_layout()

plt.show()

A screenshot of a graph

Description automatically generated

In this example:

We define a combination of transformations: scaling -> shearing -> rotation.

We define scaling, shearing, and rotation matrices with specific parameters.

We then multiply these matrices together to form a combined transformation matrix.

Finally, we apply this combined transformation to the set of points and plot them before and after the transformation to visualise its effect.

By composing multiple transformations in this way, we can create more complex and exciting transformations that achieve specific effects on our data.

An affine transformation is a type of geometric transformation that combines linear transformations (such as scaling, rotation, and shearing) with an additional translation. In other words, an affine transformation preserves points, straight lines, and planes, but it allows for shifting, rotating, scaling, and shearing objects in space.

**Components of an Affine Transformation:**

**Linear Transformation**: The linear part of an affine transformation consists of scaling, rotation, and shearing. These operations are applied to the coordinates of the points being transformed.

**Translation**: The translation component involves shifting the points in space by a fixed amount along each axis. Unlike linear transformations, translations move all points in the same direction by the same distance without changing their relative positions.

**Mathematical Representation:**

An affine transformation in 2D space is typically represented by a matrix multiplication followed by a vector addition. Given a point (𝑥,𝑦) to be transformed, the transformation can be expressed as:

A group of black letters

Description automatically generated

A black text on a white background

Description automatically generated**Application in Computer Graphics and Image Processing:**

Affine transformations are widely used in computer graphics, image processing, and computer vision. They are fundamental for geometric modelling, image registration, image warping, and object tracking.

**Connection to Iterated Function Systems (IFS):**

In fractals and chaos theory, an Iterated Function System (IFS) is a collection of affine transformations, each associated with a probability. These transformations are repeatedly applied to points in space, leading to the generation of complex fractal patterns. Each transformation in an IFS consists of a linear transformation (represented by a matrix) and a translation (represented by a vector).

**Summary:**

An affine transformation combines linear transformations with a translation.

It preserves points, straight lines, and planes while allowing for shifting, rotating, scaling, and shearing.

Affine transformations are widely used in computer graphics, image processing, and computer vision.

In fractals, affine transformations are the building blocks of Iterated Function Systems (IFS), which generate complex patterns through repeated transformations.

Let's create a Python example that demonstrates an affine transformation using a simple graphical interface. We'll use the **matplotlib** library to plot points before and after the transformation. We'll also integrate some AI-based thinking by allowing the user to interactively adjust the transformation parameters.

Python code

import numpy as np

import matplotlib.pyplot as plt

from matplotlib.widgets import Slider

class AffineTransformationDemo:

    def \_\_init\_\_(self):

        self.fig, self.ax = plt.subplots()

        plt.subplots\_adjust(left=0.25, bottom=0.35)

        self.ax.set\_aspect('equal', adjustable='box')

        self.ax.grid(True)

        self.axcolor = 'lightgoldenrodyellow'

        self.ax\_a = plt.axes([0.25, 0.1, 0.65, 0.03], facecolor=self.axcolor)

        self.ax\_b = plt.axes([0.25, 0.15, 0.65, 0.03], facecolor=self.axcolor)

        self.ax\_c = plt.axes([0.25, 0.2, 0.65, 0.03], facecolor=self.axcolor)

        self.ax\_d = plt.axes([0.25, 0.25, 0.65, 0.03], facecolor=self.axcolor)

        self.ax\_e = plt.axes([0.25, 0.3, 0.65, 0.03], facecolor=self.axcolor)

        self.ax\_f = plt.axes([0.25, 0.35, 0.65, 0.03], facecolor=self.axcolor)

        self.slider\_a = Slider(self.ax\_a, 'a', -2.0, 2.0, valinit=1)

        self.slider\_b = Slider(self.ax\_b, 'b', -2.0, 2.0, valinit=0)

        self.slider\_c = Slider(self.ax\_c, 'c', -2.0, 2.0, valinit=0)

        self.slider\_d = Slider(self.ax\_d, 'd', -2.0, 2.0, valinit=1)

        self.slider\_e = Slider(self.ax\_e, 'e', -10.0, 10.0, valinit=0)

        self.slider\_f = Slider(self.ax\_f, 'f', -10.0, 10.0, valinit=0)

        self.slider\_a.on\_changed(self.update)

        self.slider\_b.on\_changed(self.update)

        self.slider\_c.on\_changed(self.update)

        self.slider\_d.on\_changed(self.update)

        self.slider\_e.on\_changed(self.update)

        self.slider\_f.on\_changed(self.update)

        self.points = np.array([[0, 0], [1, 1], [2, 2], [3, 3]])

        self.trans\_points = self.points.copy()

        self.update(None)

    def update(self, val):

        a = self.slider\_a.val

        b = self.slider\_b.val

        c = self.slider\_c.val

        d = self.slider\_d.val

        e = self.slider\_e.val

        f = self.slider\_f.val

        transform\_matrix = np.array([[a, b], [c, d]])

        translation\_vector = np.array([e, f])

        self.trans\_points = np.dot(self.points, transform\_matrix) + translation\_vector

        self.ax.clear()

        self.ax.scatter(self.points[:, 0], self.points[:, 1], color='blue', label='Original Points')

        self.ax.scatter(self.trans\_points[:, 0], self.trans\_points[:, 1], color='red', label='Transformed Points')

        self.ax.legend()

        self.ax.set\_xlabel('X-axis')

        self.ax.set\_ylabel('Y-axis')

        self.ax.set\_title('Affine Transformation Demo')

        self.ax.grid(True)

        self.fig.canvas.draw\_idle()

if \_\_name\_\_ == "\_\_main\_\_":

    demo = AffineTransformationDemo()

    plt.show()

A graph with red and blue dots

Description automatically generated

This code creates a graphical interface using Matplotlib with sliders for adjusting the parameters of an affine transformation matrix and a translation vector. The points before and after the transformation are plotted in different colours, allowing the user to see the transformation's effect in real time. This interactive demonstration provides a visually appealing and educational understanding of affine transformations.

Barnsley's Fern is a classic example of a fractal generated by an Iterated Function System (IFS). It's named after mathematician Michael Barnsley, who introduced it in his book "Fractals Everywhere" in 1988. The fern is created through a series of mathematical transformations applied iteratively to a set of points in a coordinate system.

**Key Concepts:**

**Iterated Function System (IFS)**: An IFS is a mathematical framework used to create fractal patterns by repeatedly applying a set of affine transformations to points in space. Each transformation is associated with a probability of being chosen and the points generated from one iteration are used as input for the next iteration.

**Affine Transformations**: Affine transformations are geometric transformations that include scaling, rotation, shearing, and translation. In Barnsley's Fern, these transformations create the intricate patterns of the fern leaves.

**Algorithm:**

The algorithm for generating Barnsley's Fern involves four affine transformations, each with its associated probabilities. These transformations are applied iteratively to a set of initial points, typically within a specific range. Here's a simplified version of the algorithm:

**Define Affine Transformations**: Set up four affine transformations representing the stem and different parts of the fern leaves. Each transformation includes scaling, rotation, and translation.

**Choose Transformation with Probability**: Randomly select one of the four transformations based on their associated probabilities. This determines which part of the fern will be drawn.

**Apply Transformation**: Apply the chosen transformation to the current point to generate a new point.

**Repeat**: Repeat steps 2 and 3 for a large number of iterations to create a detailed fern pattern.

**Affine Transformations for Barnsley's Fern:**

The affine transformations used in Barnsley's Fern typically include:

**Stem**: A straight line representing the stem of the fern.

**Main Leaf**: A larger leaflet that forms the main body of the fern.

**Left Leaflet**: A smaller leaflet on the left side of the fern.

**Right Leaflet**: A smaller leaflet on the right side of the fern.

Each transformation involves scaling, rotation, and translation to create the desired shapes and patterns.

**Visualisation:**

When the algorithm is executed for many iterations, the points generated form a fractal pattern resembling a fern leaf, the fern's self-similarity is a characteristic feature of fractals, where more minor parts of the pattern resemble the overall shape.

**Conclusion:**

Barnsley's Fern demonstrates the beauty and complexity that can arise from simple mathematical rules applied iteratively. It's a visually striking example of a fractal and highlights the power of mathematical abstraction in modelling natural phenomena.

Barnsley's Fern consists of four affine transformations, each associated with a probability. Here's a table defining these transformations:

A screenshot of a table

Description automatically generated

In this table:

**Transformation**: Describes the type of transformation (Stem, Main Leaf, Left Leaflet, or Right Leaflet).

**Probability**: Represents the likelihood of selecting each transformation during the iteration process.

**Transformation Matrix**: Defines the linear part of the affine transformation, including scaling, rotation, and shearing.

**Translation Vector**: Represents the translation part of the affine transformation, which moves the points in space.

When applied iteratively with their associated probabilities, these transformations generate the intricate fern pattern characteristic of Barnsley's Fern. Each transformation contributes to different fern parts, creating a visually appealing fractal structure.

Here's a Python code example using matplotlib to visualize the effects of each of the transformations associated with Barnsley's Fern. We'll create an interactive application where you can adjust the parameters of the transformations and observe their effects in real time:

Python code

import numpy as np

import matplotlib.pyplot as plt

from matplotlib.widgets import Slider

class BarnsleysFernDemo:

    def \_\_init\_\_(self):

        self.fig, self.ax = plt.subplots()

        plt.subplots\_adjust(left=0.25, bottom=0.35)

        self.ax.set\_aspect('equal', adjustable='box')

        self.ax.grid(True)

        self.axcolor = 'lightgoldenrodyellow'

        self.ax\_p1 = plt.axes([0.25, 0.25, 0.65, 0.03], facecolor=self.axcolor)

        self.ax\_p2 = plt.axes([0.25, 0.20, 0.65, 0.03], facecolor=self.axcolor)

        self.ax\_p3 = plt.axes([0.25, 0.15, 0.65, 0.03], facecolor=self.axcolor)

        self.ax\_p4 = plt.axes([0.25, 0.10, 0.65, 0.03], facecolor=self.axcolor)

        self.slider\_p1 = Slider(self.ax\_p1, 'Probability p1', 0, 100, valinit=1)

        self.slider\_p2 = Slider(self.ax\_p2, 'Probability p2', 0, 100, valinit=85)

        self.slider\_p3 = Slider(self.ax\_p3, 'Probability p3', 0, 100, valinit=7)

        self.slider\_p4 = Slider(self.ax\_p4, 'Probability p4', 0, 100, valinit=7)

        self.slider\_p1.on\_changed(self.update)

        self.slider\_p2.on\_changed(self.update)

        self.slider\_p3.on\_changed(self.update)

        self.slider\_p4.on\_changed(self.update)

        self.update(None)

    def update(self, val):

        p1 = self.slider\_p1.val / 100

        p2 = self.slider\_p2.val / 100

        p3 = self.slider\_p3.val / 100

        p4 = self.slider\_p4.val / 100

        # Initialize points

        points = np.zeros((50000, 2))

        points[0] = [0, 0]

        # Define transformation matrices and translation vectors

        transformations = [

            (np.array([[0, 0], [0, 0.16]]), np.array([0, 0])),

            (np.array([[0.85, 0.04], [-0.04, 0.85]]), np.array([0, 1.6])),

            (np.array([[0.20, -0.26], [0.23, 0.22]]), np.array([0, 1.6])),

            (np.array([[-0.15, 0.28], [0.26, 0.24]]), np.array([0, 0.44]))

        ]

        # Iterate and apply transformations

        for i in range(1, len(points)):

            rand = np.random.random()

            if rand < p1:

                chosen\_transform = transformations[0]

            elif rand < p1 + p2:

                chosen\_transform = transformations[1]

            elif rand < p1 + p2 + p3:

                chosen\_transform = transformations[2]

            else:

                chosen\_transform = transformations[3]

            points[i] = np.dot(chosen\_transform[0], points[i-1]) + chosen\_transform[1]

        self.ax.clear()

        self.ax.scatter(points[:, 0], points[:, 1], s=1, color='green')

        self.ax.set\_xlim(-3, 3)

        self.ax.set\_ylim(0, 10)

        self.ax.set\_title('Barnsley\'s Fern')

        self.ax.set\_xlabel('X-axis')

        self.ax.set\_ylabel('Y-axis')

        self.ax.grid(True)

        self.fig.canvas.draw\_idle()

if \_\_name\_\_ == "\_\_main\_\_":

    demo = BarnsleysFernDemo()

    plt.show()

A screenshot of a graph

Description automatically generated

This code creates an interactive application using Matplotlib, where you can adjust the probabilities of each transformation associated with Barnsley's Fern. The fern pattern is updated in real-time as you modify the probabilities, allowing you to visualize the effects of each transformation and control the appearance of the fern.

A Partial Iterated Function System (PIFS) is a technique used to generate fractal patterns by dividing the image space into two or more grids, each with different cell sizes. This approach offers more flexibility and control over the generation of fractals compared to traditional Iterated Function Systems (IFS).

Key Concepts:

**Iterated Function System (IFS)**: An IFS is a mathematical framework used to generate fractal patterns by iteratively applying a set of affine transformations to points in space. Each transformation is associated with a probability of being chosen.

**Partial Iterated Function System (PIFS)**: In a PIFS, the image space is partitioned into grids, typically with different cell sizes. Each grid represents a different level of detail or resolution in the fractal generation process.

How PIFS Works:

**Grid Partitioning**: The image space is divided into two or more grids containing varying-sized cells. Typically, one grid has more giant cells, while the other has smaller cells. The more giant cells may overlap with each other.

**Transformation Assignment**: Affine transformations are assigned to each cell in the grids. Each cell may have multiple transformations associated with it, with probabilities determining the likelihood of each transformation being applied.

**Iteration Process**: The fractal generation process iterates over each cell in the grids, applying the assigned transformations based on their probabilities. The transformations may include scaling, rotation, translation, and shearing.

**Combining Results**: After applying the transformations to all cells in the grids, the results are combined to create the final fractal image. This can involve blending the results from different grids or overlaying them to achieve the desired level of detail and complexity.

Advantages of PIFS:

**Control Over Detail**: PIFS allows for greater control over the level of detail and resolution in the generated fractal patterns. By partitioning the image space into grids with different cell sizes, you can vary the level of detail in other parts of the fractal.

**Flexibility**: PIFS offers flexibility in designing fractals by allowing you to experiment with different grid configurations, cell sizes, and transformation assignments. This flexibility enables the creation of a wide range of intricate and visually appealing fractal patterns.

**Efficiency**: PIFS can be computationally efficient, especially when dealing with large fractal images. Partitioning the image space into smaller grids reduces the computational complexity of applying transformations, leading to faster generation times.

Applications of PIFS:

**Computer Graphics**: PIFS is widely used in computer graphics and digital art to generate complex and visually appealing fractal patterns. It's employed in texture synthesis, terrain generation, and procedural content generation applications.

**Mathematical Exploration**: PIFS provides a framework for exploring mathematical concepts related to fractals, affine transformations, and probability distributions. It's used in educational settings to teach students about fractal geometry and computational mathematics.

Overall, Partial Iterated Function Systems offer a versatile and powerful approach to generating fractal patterns with varying levels of detail and complexity. They provide a rich playground for mathematical exploration and artistic creativity.

Let's delve deeper into how this process works and its significance in generating fractal patterns by expanding on the concept of matching affine transforms between small and large cells in a Partial Iterated Function System (PIFS).

Affine Transform Matching:

**Small Cell Identification**: In a PIFS, the image space is divided into grids with smaller and larger cells. Each small cell represents a portion of the fractal image at a finer level of detail.

**Significant Cell Identification**: Corresponding to each small cell, there is a larger cell in the grid with larger cells. These more giant cells provide a broader overview of the fractal image, capturing its general structure.

**Finding Matching Affine Transforms**: For each small cell, the goal is to find an affine transformation that closely matches the transformation applied to the corresponding large cell. This matching ensures consistency and coherence between different levels of detail in the fractal.

**Compensation for Contrast and Brightness**: After identifying potential matching affine transforms, adjustments may be made to compensate for differences in contrast and brightness between the small and large cells. This ensures seamless blending and transition between different scales of the fractal.

Significance:

**Coherence and Continuity**: Matching affine transforms between small and large cells contribute to the overall coherence and continuity of the fractal pattern. It ensures that details at different scales align correctly, creating a smooth transition between varying levels of detail.

**Preservation of Structure**: By matching affine transforms, the overall structure and features of the fractal are preserved across different scales. This helps maintain the fractal's aesthetic appeal and ensures its intricate patterns remain consistent and recognizable.

**Enhanced Visual Quality**: Matching affine transforms with compensation for contrast and brightness contributes to the visual quality of the fractal image. It helps reduce artefacts and irregularities that may arise from abrupt transitions between different scales or levels of detail.

**Artistic Expression**: Matching affine transforms also gives artists and designers greater control and flexibility in crafting visually appealing fractal patterns. It allows subtle adjustments and fine-tuning to achieve the desired artistic effect and expression.

Implementation:

Implementing affine transform matching in a PIFS involves algorithms for comparing transforms, assessing similarity, and applying adjustments for contrast and brightness. Template matching, feature matching, and histogram normalization may be employed to achieve accurate matching and compensation.

Overall, affine transform matching ensures coherence, continuity, and visual quality in fractal patterns generated using a Partial Iterated Function System. It facilitates the creation of captivating and aesthetically pleasing fractal imagery across different scales and levels of detail.

Storing the transforms and grid dimensions in a fractal generation system, such as a Partial Iterated Function System (PIFS), is a critical step in generating and rendering fractal patterns. Let's explore why storing this information is essential and how it contributes to the generation of fractals:

Importance of Storing Transforms and Grid Dimensions:

**Reproducibility**: Storing the transforms and grid dimensions allows for the reproducibility of fractal patterns. By preserving the parameters used in the generation process, it becomes possible to recreate the same fractal pattern later or share it with others for analysis or artistic purposes.

**Adjustment and Iteration**: Access to the stored transforms and grid dimensions enables users to adjust and iterate on the fractal generation process. They can modify parameters, such as transformation probabilities or grid sizes, and observe the effects on the resulting fractal patterns. This iterative approach facilitates experimentation and refinement in creating desired fractal imagery.

**Optimization**: Storing the transforms and grid dimensions provides opportunities for optimization in the fractal generation algorithm. By caching or precomputing certain aspects of the transformation process, computational efficiency can be improved, leading to faster generation times and smoother real-time rendering of fractal patterns.

**Memory Efficiency**: Rather than recalculating transforms and grid dimensions every time a fractal pattern is generated, storing this information allows for memory-efficient storage and retrieval. This is particularly important when dealing with large-scale fractals or generating fractals in real-time applications with limited computational resources.

Components of Stored Information:

**Transformations**: Each stored transformation includes parameters such as scaling factors, rotation angles, translation vectors, and shearing coefficients. These parameters define how points in the image space are transformed and contribute to the overall appearance of the fractal pattern.

**Grid Dimensions**: The grid dimensions specify the layout of the image space and the arrangement of cells within the grids. This information includes the size of each cell, the number of cells in each grid, and any overlap or boundary conditions imposed on the grids.

Implementation Considerations:

**Data Structure**: The transforms and grid dimensions can be stored using appropriate data structures such as arrays, matrices, dictionaries, or custom objects. The choice of data structure depends on factors such as memory efficiency, ease of access, and compatibility with the fractal generation algorithm.

**Serialization**: Storing transforms and grid dimensions often involves serialization, where the data is converted into a format suitable for storage or transmission. Serialization techniques such as JSON, XML, or binary encoding can be used to efficiently serialize the data while preserving its structure and integrity.

In summary, storing transforms and grid dimensions in a fractal generation system is essential for reproducibility, flexibility, optimization, and memory efficiency. Users can explore, manipulate, and share fractal patterns with greater control and insight by preserving this information.

The concept of a fixed-point attractor in fractal generation refers to a specific point or set of points in the image space towards which any starting condition will converge under the iterated application of a set of affine transformations. Let's explore this concept in more detail:

Understanding Fixed-Point Attractor:

**Definition**: A fixed-point attractor is a point in the image space that remains stationary or converges to a stable position under the repeated application of affine transformations. An attractor is called because nearby points tend to gravitate towards it over successive iterations.

**Convergence Property**: Regardless of the initial starting condition, after multiple iterations of applying a set of affine transformations, the points in the image space gradually move closer to the fixed-point attractor. Eventually, they stabilize or oscillate around this attractor, forming the characteristic pattern of the fractal.

**Stability**: The fixed-point attractor is characterized by its strength, meaning that small perturbations or variations in the initial conditions do not significantly alter its position or properties. This stability ensures the fractal pattern maintains its overall structure and coherence over successive iterations.

Role of Affine Transformations:

**Affine Transformations**: Affine transformations such as scaling, rotation, translation, and shearing are applied iteratively to points in the image space. Each transformation contributes to the movement and deformation of the points, gradually shaping the fractal pattern.

**Collective Effect**: While individual transformations may produce complex patterns, the collective effect of repeatedly applying a set of transformations leads to the emergence of the fixed-point attractor. This attractor represents a stable configuration towards which the system tends to evolve.

Visual Interpretation:

**Fractal Patterns**: The fixed-point attractor often corresponds to recognizable features or structures within the fractal pattern. For example, in Barnsley's Fern, the attractor may correspond to the central stem or main leaf structure that remains relatively unchanged after multiple iterations.

**Convergence Behavior**: Observing the convergence behaviour of points in the image space provides insights into the characteristics of the fixed-point attractor. Points that move towards and stabilize around a specific region indicate the presence of a fixed-point attractor.

Practical Implications:

**Algorithm Validation**: The existence and behaviour of a fixed-point attractor serve as a validation mechanism for fractal generation algorithms. If points consistently converge towards a stable configuration, it indicates that the algorithm accurately captures the fractal system's underlying dynamics.

**Parameter Tuning**: Understanding the properties of the fixed-point attractor can inform parameter tuning and optimization strategies in fractal generation algorithms. Adjusting transformation parameters or probabilities may influence the attractor's position, stability, and complexity.

In summary, the fixed-point attractor in fractal generation represents a stable configuration towards which points in the image space converge under the repeated application of affine transformations. It serves as a fundamental concept in understanding the dynamics and behaviour of fractal patterns.

Consequently, if we begin with random pixels and then apply all the transforms iteratively, we arrive at a close approximation to the original image.

indeed, that's a fascinating consequence of the dynamics of fractal generation systems, particularly those based on iterated function systems (IFS) or partial iterated function systems (PIFS). Let's break down why this phenomenon occurs:

1. Preservation of Structure:

Fractal generation algorithms, such as those based on IFS or PIFS, are designed to preserve the overall structure and characteristics of the original image or pattern. This means that the transformations applied iteratively tend to maintain or replicate the essential features and geometry of the original image.

2. Fixed-Point Attractor:

As discussed earlier, the repeated application of affine transformations converges towards a fixed-point attractor. In generating fractal images, this attractor represents a stable configuration resembling the original image.

3. Convergence to Original Image:

When starting with random pixels and iteratively applying the set of transforms associated with a fractal generation system, the system's dynamics naturally lead to convergence towards the fixed-point attractor. Since the attractor closely approximates the original image, the resulting pattern or image becomes a close approximation of the original.

4. Emergence of Self-Similarity:

Fractal patterns exhibit self-similarity, meaning they contain smaller-scale copies of themselves at various magnification levels. The iterative application of transforms converges towards the original image and reveals the self-similar structures inherent in fractals.

5. Iterative Refinement:

As the iterations progress, the fractal pattern becomes increasingly refined and detailed, resembling the original image more closely with each iteration. This iterative refinement process enhances the fidelity of the approximation to the original image.

6. Practical Applications:

This phenomenon has practical applications in fields such as image compression, where fractal-based algorithms can efficiently represent and reconstruct images using a relatively small set of transforms and parameters.

In summary, the convergence towards a close approximation of the original image through the iterative application of transforms in fractal generation systems highlights fractals' inherent self-similarity and structure-preserving properties. It demonstrates how complex and visually appealing patterns can emerge from simple iterative processes.

We can also reproduce the image at any resolution we like. Just start with a larger random image and apply the transforms. This makes it a valuable technique for upscaling.

Exactly. This scalability is another powerful aspect of fractal generation techniques like IFS or PIFS. Here's how the process of reproducing an image at any resolution works:

1. Larger Random Image:

To reproduce an image at a higher resolution, you start with a larger random image that serves as the initial canvas for the fractal generation process. This larger image contains more pixels and provides greater detail than the original image.

2. Iterative Application of Transforms:

As with the original image, you iteratively apply the set of affine transformations to the pixels of the larger random image. These transformations dictate how the pixels are transformed and rearranged, gradually shaping the fractal pattern.

3. Scaling Effect:

As the transformations are applied, the fractal pattern evolves and expands across the canvas of the larger random image. The scaling effect ensures that the resulting fractal maintains the same structural characteristics and overall appearance as the original image but at a higher resolution.

4. Self-Similarity Across Scales:

One of the defining properties of fractals is self-similarity, meaning that the same patterns recur at different scales within the image. By upscaling the fractal pattern generated from the larger random image, you're essentially zooming in on these self-similar structures, revealing finer details and intricacies.

5. Adjusting Resolution:

The resolution of the final reproduced image can be adjusted according to the desired level of detail and fidelity. You can control the resolution and quality of the resulting fractal image by varying parameters such as the size of the larger random image and the number of iterations.

6. Practical Applications:

This technique of reproducing images at different resolutions is beneficial for upscaling, where you need to increase the size and detail of an image without losing quality or introducing artefacts. Fractal-based upscaling algorithms can efficiently generate high-resolution photos while preserving the original content and structure.

7. Image Enhancement:

In addition to upscaling, fractal generation techniques can be used for image enhancement and restoration. By iterative refining and adding detail to the original image, fractal-based algorithms can produce visually pleasing results with improved clarity and definition.

In summary, the ability to reproduce images at any resolution using fractal generation techniques offers a versatile and powerful tool for upscaling, image enhancement, and digital content creation. It leverages the self-similar properties of fractals to produce high-quality, detailed images with flexibility and efficiency.

Below is a Python example demonstrating reproducing an image at any resolution using fractal generation techniques. We'll start with a larger random image and apply affine transformations iteratively to create a fractal pattern. This process will upscale the picture, revealing finer details while maintaining the overall appearance and structure of the original image.

Python code

import numpy as np

import matplotlib.pyplot as plt

from skimage.transform import resize

# Function to apply affine transformations iteratively

def apply\_transformations(image, transforms, iterations):

    images = []

    for i in range(1, iterations + 1):

        idx = np.random.choice(len(transforms), p=[t["probability"] for t in transforms])

        matrix = transforms[idx]["matrix"]

        translation = transforms[idx]["translation"]

        # Reshape the image vector to a 2D array before matrix multiplication

        image = np.dot(matrix, image.reshape(2, -1)) + translation.reshape(2, 1)

        images.append(image)

    return images

# Function to reproduce image at a higher resolution

def reproduce\_image(image, transforms, target\_resolution, iterations, snapshot\_iterations):

    # Upscale the random image to target resolution

    upscale\_factor = target\_resolution / image.shape[0]

    image = resize(image, (target\_resolution, target\_resolution), anti\_aliasing=True)

    # Apply affine transformations iteratively

    fractal\_images = apply\_transformations(image.flatten(), transforms, iterations)

    # Reshape the flattened images to 2D arrays

    fractal\_images = [img.reshape(target\_resolution, target\_resolution) for img in fractal\_images]

    # Select snapshots of iterations for plotting

    snapshot\_indices = np.linspace(0, iterations-1, num=snapshot\_iterations, dtype=int)

    snapshot\_images = [fractal\_images[i] for i in snapshot\_indices]

    return snapshot\_images

# Define parameters

random\_image\_size = 100

target\_resolution = 1000

iterations = 10000

snapshot\_iterations = 6

# Generate a random image

random\_image = np.random.rand(random\_image\_size, random\_image\_size)

# Define affine transformations (example transformations)

transforms = [

    {"matrix": np.array([[0.85, 0.04], [-0.04, 0.85]]), "translation": np.array([0, 1.6]), "probability": 0.85},

    {"matrix": np.array([[0.2, -0.26], [0.23, 0.22]]), "translation": np.array([0, 1.6]), "probability": 0.07},

    {"matrix": np.array([[-0.15, 0.28], [0.26, 0.24]]), "translation": np.array([0, 0.44]), "probability": 0.07}

]

# Ensure probabilities sum up to 1

total\_prob = sum(t["probability"] for t in transforms)

for t in transforms:

    t["probability"] /= total\_prob

# Reproduce image at higher resolution with snapshots of iterations

fractal\_images = reproduce\_image(random\_image, transforms, target\_resolution, iterations, snapshot\_iterations)

# Plotting snapshots of iterations

plt.figure(figsize=(18, 3))

for i in range(snapshot\_iterations):

    plt.subplot(1, snapshot\_iterations, i + 1)

    plt.imshow(fractal\_images[i], cmap='gray')

    plt.title(f"Iteration {int(iterations \* (i+1) / snapshot\_iterations)}")

    plt.axis('off')

plt.show()

This code will generate a random image of a specified size and then iteratively apply affine transformations to upscale the image and create a fractal pattern. Finally, it reproduces the fractal image at a higher resolution and visualises it using Matplotlib. You can adjust the parameters, such as the size of the random image, target resolution, and the number of iterations, to see different results.

Iterations play a crucial role in various processes, including fractal generation, optimisation algorithms, numerical methods, and iterative algorithms in general. Let's delve into the concept of iterations and their significance in different contexts:

Definition of Iterations:

**Repetition**: Iterations refer to the repeated execution of instructions or operations. Each iteration involves performing the same sequence of steps multiple times, often with slight variations or updates at each iteration.

**Progressive Update**: Through iterations, a system or algorithm progresses towards a desired state, solution, or outcome. Each iteration contributes to refining, improving, or converging towards the final result.

Significance of Iterations:

**Convergence**: In many algorithms, iterations converge towards a solution or equilibrium point. The algorithm gradually approaches the desired solution by iteratively updating variables or parameters, reducing the error or discrepancy with each iteration.

**Refinement**: Iterations enable the refinement of solutions or approximations by iteratively incorporating new information or adjusting parameters. This iterative refinement process often improves final result accuracy, precision, or quality.

**Complexity Reduction**: In complex problems or systems, iterations help break down the problem into manageable steps. By addressing one iteration at a time, the complexity of the problem is reduced, making it more tractable and more accessible to solve.

**Optimization**: Many optimization algorithms rely on iterations to iteratively update candidate solutions and search for optimal or near-optimal solutions. These algorithms explore the solution space through successive iterations, identify promising regions, and refine the search towards the best solution.

**Convergence Analysis**: Iterations provide insights into algorithms' behavior and performance. Convergence analysis examines how the algorithm's solution or output approaches stability or a steady state over multiple iterations, helping assess its effectiveness and reliability.

Examples of Iterative Processes:

**Fractal Generation**: In fractal generation algorithms such as IFS or PIFS, iterations repeatedly apply affine transformations to points in the image space. Each iteration contributes to the formation and refinement of the fractal pattern, leading to the emergence of intricate structures.

**Numerical Methods**: Iterative numerical methods, such as Newton's method for root finding or the Gauss-Seidel method for solving linear systems, involve updating solutions until convergence. Each iteration brings the solution closer to the proper solution of the problem.

**Machine Learning Training**: Iterative optimization algorithms, such as gradient descent in machine learning, involve adjusting model parameters to minimize the loss function. The model learns to make better predictions and generalize to unseen data through successive iterations over training data.

Iterations are fundamental to many processes and algorithms, driving convergence, refinement, and optimization towards desired outcomes. They enable the progressive improvement and convergence of solutions, making them indispensable in problem-solving and algorithmic design.

Below is a Python script that reads the content from the "KevinLeeExamples.txt" file, creates a simple UI with a dropdown menu to select different fractal ideas, and updates the graphic accordingly. Save the page [<https://larryriddle.agnesscott.org/ifskit/gallery/kevinlee/KevinLeeExamples.ifs>](https://larryriddle.agnesscott.org/ifskit/gallery/kevinlee/KevinLeeExamples.ifs) as "KevinLeeExamples.txt" in your folder.

python code

# Spiral

spiral = [

    {"matrix": [[0.935567, -0.164966], [0.164966, 0.935567]], "translation": [0.0, 0.0], "probability": 0.919682},

    {"matrix": [[0.1, 0.0], [0.0, 0.1]], "translation": [0.75, -0.75], "probability": 0.020080},

    {"matrix": [[0.1, 0.0], [0.0, 0.1]], "translation": [-0.75, 0.75], "probability": 0.020080},

    {"matrix": [[0.1, 0.0], [0.0, 0.1]], "translation": [0.75, 0.75], "probability": 0.020080},

    {"matrix": [[0.1, 0.0], [0.0, 0.1]], "translation": [-0.75, -0.75], "probability": 0.020080}

]

# Crab

crab = [

    {"matrix": [[0.5, 0.2], [0.0, 0.5]], "translation": [0.0, 0.0], "probability": 0.25},

    {"matrix": [[-0.5, 0.0], [0.2, 0.5]], "translation": [1.0, 0.0], "probability": 0.25},

    {"matrix": [[0.5, 0.2], [0.0, -0.5]], "translation": [0.0, 1.0], "probability": 0.25},

    {"matrix": [[-0.5, 0.0], [-0.2, -0.5]], "translation": [1.0, 1.0], "probability": 0.25}

]

# Eiffel Tower

eiffel\_tower = [

    {"matrix": [[0.5, 0.0], [0.0, 0.3]], "translation": [0.0, 0.0], "probability": 0.246154},

    {"matrix": [[0.5, 0.0], [0.0, 0.3]], "translation": [0.5, 0.0], "probability": 0.246154},

    {"matrix": [[0.5, 0.0], [0.0, 0.7]], "translation": [0.25, 0.3], "probability": 0.507692}

]

# Fern

fern = [

    {"matrix": [[0.0, 0.0], [0.0, 0.172033]], "translation": [0.496139, -0.09051], "probability": 0.01},

    {"matrix": [[0.075906, 0.312285], [-0.257105, 0.204233]], "translation": [0.494173, 0.132616], "probability": 0.075},

    {"matrix": [[0.82113, -0.028405], [0.029799, 0.84528]], "translation": [0.087877, 0.175709], "probability": 0.84},

    {"matrix": [[-0.023936, -0.356062], [-0.323405, 0.074403]], "translation": [0.470356, 0.259738], "probability": 0.075}

]

# Spiral 2

spiral\_2 = [

    {"matrix": [[0.143268, 0.37263], [0.0, 0.411999]], "translation": [-0.424035, 0.228385], "probability": 0.05},

    {"matrix": [[0.817528, -0.472], [0.472, 0.817528]], "translation": [-0.06, 0.051538], "probability": 0.95}

]

# Leaf

leaf = [

    {"matrix": [[0.5, 0.0], [0.0, 0.759127]], "translation": [0.0, 0.542236], "probability": 0.423635},

    {"matrix": [[0.001, 0.0], [0.0, 0.272]], "translation": [0.0, 0.0], "probability": 0.05},

    {"matrix": [[0.437567, 0.397819], [-0.086824, 0.492404]], "translation": [-0.001575, 0.428276], "probability": 0.263},

    {"matrix": [[-0.437567, -0.397819], [-0.086824, 0.492404]], "translation": [-0.003556, 0.434301], "probability": 0.263}

]

# Tree

tree = [

    {"matrix": [[0.458805, -0.225654], [0.072682, 0.60187]], "translation": [-0.001772, 0.319478], "probability": 0.31},

    {"matrix": [[0.342683, 0.37561], [-0.203368, 0.54642]], "translation": [-0.022095, 0.329764], "probability": 0.2},

    {"matrix": [[0.135817, 0.502731], [-0.313469, 0.138456]], "translation": [-0.020039, 0.217063], "probability": 0.16},

    {"matrix": [[0.253255, -0.489749], [0.307978, 0.349892]], "translation": [-0.007397, 0.198101], "probability": 0.23},

    {"matrix": [[0.066406, 0.0], [0.0, 0.479424]], "translation": [-0.014923, -0.024215], "probability": 0.1}

]

# Nautilus

nautilus = [

    {"matrix": [[0.860671, 0.401487], [-0.402177, 0.860992]], "translation": [0.108537, 0.075138], "probability": 0.93},

    {"matrix": [[0.094957, -0.000995], [0.237023, 0.002036]], "translation": [-0.746911, 0.047343], "probability": 0.02},

    {"matrix": [[0.150288, 0.0], [0.0, 0.146854]], "translation": [-0.563199, 0.032007], "probability": 0.03},

    {"matrix": [[0.324279, -0.002163], [0.005846, 0.001348]], "translation": [-0.557936, -0.139735], "probability": 0.02}

]

# Pine Tree

pine\_tree = [

    {"matrix": [[0.797881, 0.0], [0.0, 0.833333]], "translation": [0.097828, 0.168721], "probability": 0.75},

    {"matrix": [[-0.131628, -0.366787], [0.399144, -0.140643]], "translation": [0.500098, 0.180655], "probability": 0.1},

    {"matrix": [[0.099649, 0.403824], [0.408196, -0.10033]], "translation": [0.364628, 0.15777], "probability": 0.1},

    {"matrix": [[0.04918, 0.0], [0.0, 0.333333]], "translation": [0.398795, 0.009695], "probability": 0.05}

]

# Spiral 3

spiral\_3 = [

    {"matrix": [[0.25, 0.0], [0.0, 0.25]], "translation": [0.0, 0.5], "probability": 0.073459},

    {"matrix": [[0.822978, -0.475], [0.474955, 0.822724]], "translation": [0.30114, -0.173839], "probability": 0.926541}

]

# Square Snowflake

square\_snowflake = [

    {"matrix": [[-0.5, 0.0], [0.0, -0.5]], "translation": [1.0, 1.0], "probability": 0.33},

    {"matrix": [[0.5, 0.0], [0.0, -0.5]], "translation": [-1.0, 1.0], "probability": 0.33},

    {"matrix": [[-0.5, 0.0], [0.0, 0.5]], "translation": [1.0, 0.0], "probability": 0.34}

]

# Starfish

starfish = [

    {"matrix": [[0.25, 0.0], [0.0, 0.25]], "translation": [0.0, 0.5], "probability": 0.080201},

    {"matrix": [[0.727742, -0.610648], [0.610648, 0.727742]], "translation": [0.416453, -0.194195], "probability": 0.92}

]

import numpy as np

import matplotlib.pyplot as plt

import tkinter as tk

from tkinter import ttk

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

# Fractal dictionaries

fractals = {

    "Spiral": spiral,

    "Crab": crab,

    "Eiffel Tower": eiffel\_tower,

    "Fern": fern,

    "Spiral 2": spiral\_2,

    "Leaf": leaf,

    "Tree": tree,

    "Nautilus": nautilus,

    "Pine Tree": pine\_tree,

    "Spiral 3": spiral\_3,

    "Square Snowflake": square\_snowflake,

    "Starfish": starfish

}

def plot\_fractal():

    try:

        fractal\_choice = fractal\_var.get()

        fractal\_data = fractals[fractal\_choice]

        fig, ax = plt.subplots(figsize=(6, 6))  # Adjust the size of the plot

        # Plot each transformation

        for i, transform in enumerate(fractal\_data):

            matrix = np.array(transform["matrix"])

            translation = np.array(transform["translation"])

            # Apply transformation matrix

            transformed\_points = np.dot(matrix, points) + translation.reshape(2,1)

            # Plot transformed points

            ax.plot(transformed\_points[0], transformed\_points[1], 'o', markersize=1, label=f"Transformation {i+1}")  # Label each transformation

        ax.set\_aspect('equal', adjustable='box')

        ax.axis('off')

        ax.legend()  # Add legend

        canvas = FigureCanvasTkAgg(fig, master=frame)

        canvas.draw()

        canvas.get\_tk\_widget().grid(row=1, column=0, padx=10, pady=10)

        plt.show()  # Show the plot

        # Update the Tkinter window

        root.update()

        print("Plotting completed successfully.")

    except Exception as e:

        print(f"An error occurred during plotting: {e}")

# Create main window

root = tk.Tk()

root.title("Fractal Explorer")

# Create frame for UI elements

frame = tk.Frame(root)

frame.pack(padx=10, pady=10)

# Create dropdown menu

fractal\_var = tk.StringVar()

fractal\_var.set("Spiral")

fractal\_menu = ttk.Combobox(frame, textvariable=fractal\_var, values=list(fractals.keys()))

fractal\_menu.grid(row=0, column=0, padx=10, pady=10)

fractal\_menu.bind("<<ComboboxSelected>>", lambda event: plot\_fractal())

# Generate points to plot

x = np.linspace(-1, 1, 1000)

y = np.linspace(-1, 1, 1000)

points = np.array([x, y])

# Plot default fractal

plot\_fractal()

# Start tkinter main loop

root.mainloop()

To use this script, make sure you have the "KevinLeeExamples.txt" file in the same directory as the script. Run the script, and a window will appear with a dropdown menu containing the names of different fractal ideas from the file. Selecting an idea and clicking the "Draw Fractal" button will display the corresponding fractal graphic.

Creating variations of the Barnsley Fern fractal involves adjusting the parameters of the transformations. Here are a few variations you can try:

**Changing Probabilities**: Adjust the probabilities of each transformation to emphasize certain shapes or patterns.

**Scaling and Rotating**: Modify the scaling factors and rotation angles of the transformations to change the fern's overall structure.

**Translation**: Experiment with different translation values to shift the position of the fern within the plot.

**Adding New Transformations**: Introduce new transformations with different matrices and probabilities to create more complex fern shapes.

**Colour Variation: Apply colour** gradients or markers to distinguish between different fern parts.

Here's an example code snippet to create variations of the Barnsley Fern fractal:

python code

import numpy as np

import matplotlib.pyplot as plt

# Barnsley Fern parameters

fern = [

    {"matrix": [[0, 0], [0, 0.25]], "translation": [0, -0.4], "probability": 0.02},

    {"matrix": [[0.85, 0.04], [-0.04, 0.85]], "translation": [0, 1.6], "probability": 0.84},

    {"matrix": [[0.2, -0.26], [0.23, 0.22]], "translation": [0, 1.6], "probability": 0.07},

    {"matrix": [[-0.15, 0.28], [0.26, 0.24]], "translation": [0, 0.44], "probability": 0.07}

]

def barnsley\_fern(fern, iterations=10000):

    x = np.zeros(iterations)

    y = np.zeros(iterations)

    idx = np.zeros(iterations, dtype=int)

    prob\_sum = np.cumsum([transform["probability"] for transform in fern])

    for i in range(1, iterations):

        r = np.random.rand()

        idx[i] = np.searchsorted(prob\_sum, r)

        x[i], y[i] = np.dot(fern[idx[i]]["matrix"], [x[i-1], y[i-1]]) + fern[idx[i]]["translation"]

    return x, y, idx

# Variation 5: Color variation

def plot\_colored\_fern(fern, iterations=10000):

    x, y, idx = barnsley\_fern(fern, iterations)

    colors = ['green', 'darkgreen', 'yellowgreen', 'forestgreen']  # Colors for different parts of the fern

    fig, ax = plt.subplots()

    for i in range(iterations):

        ax.scatter(x[i], y[i], c=colors[idx[i]], alpha=0.5, marker='.')

    ax.set\_aspect('equal', adjustable='box')

    plt.show()

# Plot the colored fern

plot\_colored\_fern(fern)

A green leaf with white text

Description automatically generated

You can use these variations in your existing code by replacing the **fern** dictionary with one of these variations and then plotting the fractal using the **plot\_fractal()** function. Experiment with different parameter values to achieve the desired visual effects.

Variations of the fern

import numpy as np

import matplotlib.pyplot as plt

# Barnsley Fern parameters with variations

fern\_variations = [

    # Variation 1: Original Barnsley Fern

    [

        {"matrix": [[0, 0], [0, 0.16]], "translation": [0, 0], "probability": 0.01},

        {"matrix": [[0.85, 0.04], [-0.04, 0.85]], "translation": [0, 1.6], "probability": 0.85},

        {"matrix": [[0.2, -0.26], [0.23, 0.22]], "translation": [0, 1.6], "probability": 0.07},

        {"matrix": [[-0.15, 0.28], [0.26, 0.24]], "translation": [0, 0.44], "probability": 0.07}

    ],

    # Variation 2: Size variation

    [

        {"matrix": [[0, 0], [0, 0.25]], "translation": [0, -0.4], "probability": 0.01},

        {"matrix": [[0.85, 0.04], [-0.04, 0.85]], "translation": [0, 1.6], "probability": 0.85},

        {"matrix": [[0.2, -0.26], [0.23, 0.22]], "translation": [0, 1.6], "probability": 0.07},

        {"matrix": [[-0.15, 0.28], [0.26, 0.24]], "translation": [0, 0.44], "probability": 0.07}

    ],

    # Variation 3: Rotation variation

    [

        {"matrix": [[0, -0.16], [0.16, 0]], "translation": [0, 0], "probability": 0.01},

        {"matrix": [[0.85, 0.04], [-0.04, 0.85]], "translation": [0, 1.6], "probability": 0.85},

        {"matrix": [[0.2, -0.26], [0.23, 0.22]], "translation": [0, 1.6], "probability": 0.07},

        {"matrix": [[-0.15, 0.28], [0.26, 0.24]], "translation": [0, 0.44], "probability": 0.07}

    ],

    # Variation 4: Adding new transformations

    [

        {"matrix": [[0, 0], [0, 0.25]], "translation": [0, -0.4], "probability": 0.01},

        {"matrix": [[0.85, 0.04], [-0.04, 0.85]], "translation": [0, 1.6], "probability": 0.85},

        {"matrix": [[0.2, -0.26], [0.23, 0.22]], "translation": [0, 1.6], "probability": 0.07},

        {"matrix": [[-0.15, 0.28], [0.26, 0.24]], "translation": [0, 0.44], "probability": 0.07},

        {"matrix": [[0.21, -0.35], [0.15, 0.27]], "translation": [0, 0.4], "probability": 0.1}

    ],

    # Variation 5: Color variation

    [

        {"matrix": [[0, 0], [0, 0.25]], "translation": [0, -0.4], "probability": 0.01},

        {"matrix": [[0.85, 0.04], [-0.04, 0.85]], "translation": [0, 1.6], "probability": 0.85},

        {"matrix": [[0.2, -0.26], [0.23, 0.22]], "translation": [0, 1.6], "probability": 0.07},

        {"matrix": [[-0.15, 0.28], [0.26, 0.24]], "translation": [0, 0.44], "probability": 0.07}

    ]

]

def barnsley\_fern(fern, iterations=10000):

    x = np.zeros(iterations)

    y = np.zeros(iterations)

    idx = np.zeros(iterations, dtype=int)

    prob\_sum = np.cumsum([transform["probability"] for transform in fern])

    for i in range(1, iterations):

        r = np.random.rand()

        idx[i] = np.searchsorted(prob\_sum, r)

        x[i], y[i] = np.dot(fern[idx[i]]["matrix"], [x[i-1], y[i-1]]) + fern[idx[i]]["translation"]

    return x, y, idx

def plot\_barnsley\_fern\_with\_variations(variations, iterations=10000):

    variation\_labels = ["Original Fern", "Size Variation", "Rotation Variation", "New Transformations", "Color Variation"]

    for i, fern\_variation in enumerate(variations):

        x, y, \_ = barnsley\_fern(fern\_variation, iterations)

        plt.figure(figsize=(8, 6))

        plt.scatter(x, y, c='green', alpha=0.5, marker='.')

        plt.title(f"Barnsley Fern - {variation\_labels[i]}")

        plt.xlabel("X")

        plt.ylabel("Y")

        plt.gca().set\_aspect('equal', adjustable='box')

        plt.show()

# Plotting all variations of the Barnsley Fern

plot\_barnsley\_fern\_with\_variations(fern\_variations)

This Python code plots different variations of the Barnsley Fern fractal. Here's a detailed breakdown of each part:

**Importing Libraries**:

**import numpy as np**: Imports the NumPy library, which is used for numerical computing in Python.

**import matplotlib.pyplot as plt**: Imports the pyplot module from the Matplotlib library, which is used for creating visualizations in Python.

**Barnsley Fern Parameters with Variations**:

**fern\_variations**: This is a list containing different variations of the parameters for generating the Barnsley Fern fractal. Each variation is represented as a list of dictionaries, where each dictionary contains the transformation matrix, translation vector, and probability for a particular transformation.

**Barnsley Fern Function** (**barnsley\_fern**):

This function takes a set of parameters for generating the Barnsley Fern and the number of iterations as input.

It initialises arrays to store the x and y coordinates of the points generated by the fractal and an array to store the index of the transformation applied to each end.

The function iterates through the specified number of iterations, randomly selects a transformation based on its probability, applies it to the previous point, and stores the new coordinates.

Finally, it returns the x and y coordinates of the generated points and the indices of the transformations applied.

**Plot Barnsley Fern with Variations Function** (**plot\_barnsley\_fern\_with\_variations**):

This function takes a list of variations of the Barnsley Fern parameters and the number of iterations as input.

It defines labels for each variation.

Each variation generates the Barnsley Fern using the **barnsley\_fern** function, extracts the x and y coordinates, and plots the points using Matplotlib's scatter plot.

It sets the plot's title based on the variation label, sets the x and y labels, adjusts the aspect ratio of the plot, and displays the plot.

This function plots each variation of the Barnsley Fern individually.

**Plotting All Variations of the Barnsley Fern**:

This code section calls the **plot\_barnsley\_fern\_with\_variations** function, which uses the **fern\_variations** list as input to plot all variations of the Barnsley Fern.

Each variation is plotted with its corresponding label, and the resulting plot is displayed.

Overall, this code allows you to visualise different variations of the Barnsley Fern fractal using Matplotlib in Python.