Санкт-Петербургский Национальный Исследовательский Университет Информационных Технологий, Механики и Оптики

Факультет инфокоммуникационных технологий

# Проект по теме: «Фракталы, а также их построение в Python»

Выполнила:

Драчева Е.А.

Проверил:

Фамилия И.О.

Санкт-Петербург, 2023

# Введение

Целью данной лабораторной работы является изучение фракталов, а также способов их построений. В результате данной работы будут посмотрены некоторые виды фракталов, а также изучена их природа.

# Основная часть работы

Что же такое фрактал? Фрактал — это геометрическая фигура, демонстрирующая одни и те же свойства независимо от степени увеличения (либо же самоподобная фигура).

Если же говорить об обрисовке фракталов в Python, то сама концепция покажется сложной, т. к. истинная природа определяется рекурсией. Говоря о графиках и их вычерчивании, обычно считается, что они образованы пикселями или векторами, но количество пикселей или векторов всегда ограничено, а фракталы по определению бесконечно рекурсивны. Поэтому при обрисовке фракталов в какой-то момент необходимо остановиться.

В этом же проекте хотелось бы остановиться на определенных видах фракталов. Один из самых популярных – снежинка Коха. На рисунке 1 изображена прямая Коха, это основа, которая помогает получить снежинку Коха (о ней чуть далее).

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 1 – прямая Коха

Далее рассмотрим треугольник Серпинского (см. рисунок 2).

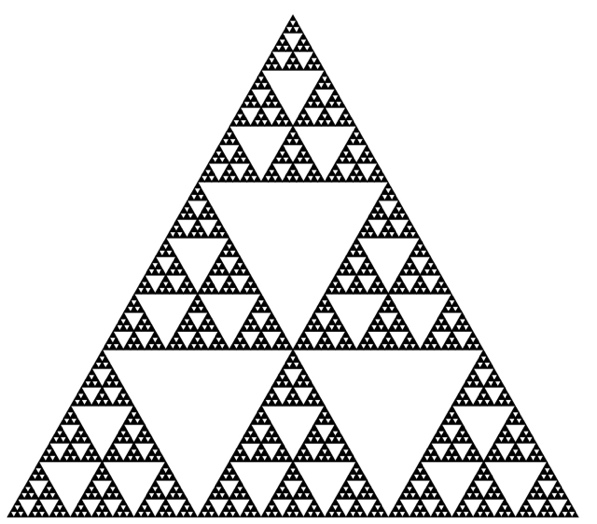


Рисунок 2 – треугольник Серпинского

И последний – фрактал Барнсли, или папоротник (см. рисунок 3)

Изображение выглядит как текст, лист

Автоматически созданное описание

Рисунок 3 – фрактал Барнсли

**Снежинка Коха**

Для начала хотелось бы реализовать алгоритм, который будет строить снежинку Коха (т. к. это один из самых популярных фракталов). В принципе идея алгоритма изображена на рисунке 1. Внесу небольшое пояснение. На 1 шаге – обычная прямая. На 2 шаге эта пряма разбивается на 3 равных части, а посередине образуется как бы равносторонний треугольник, каждая сторона которого равна 1/3 длине прямой на 1 шаге. На 3 шаге с каждой прямой происходит то же самое. И так далее… Теперь рассмотрим саму снежнику по аналогии с вышесказанным (Рисунок 4).

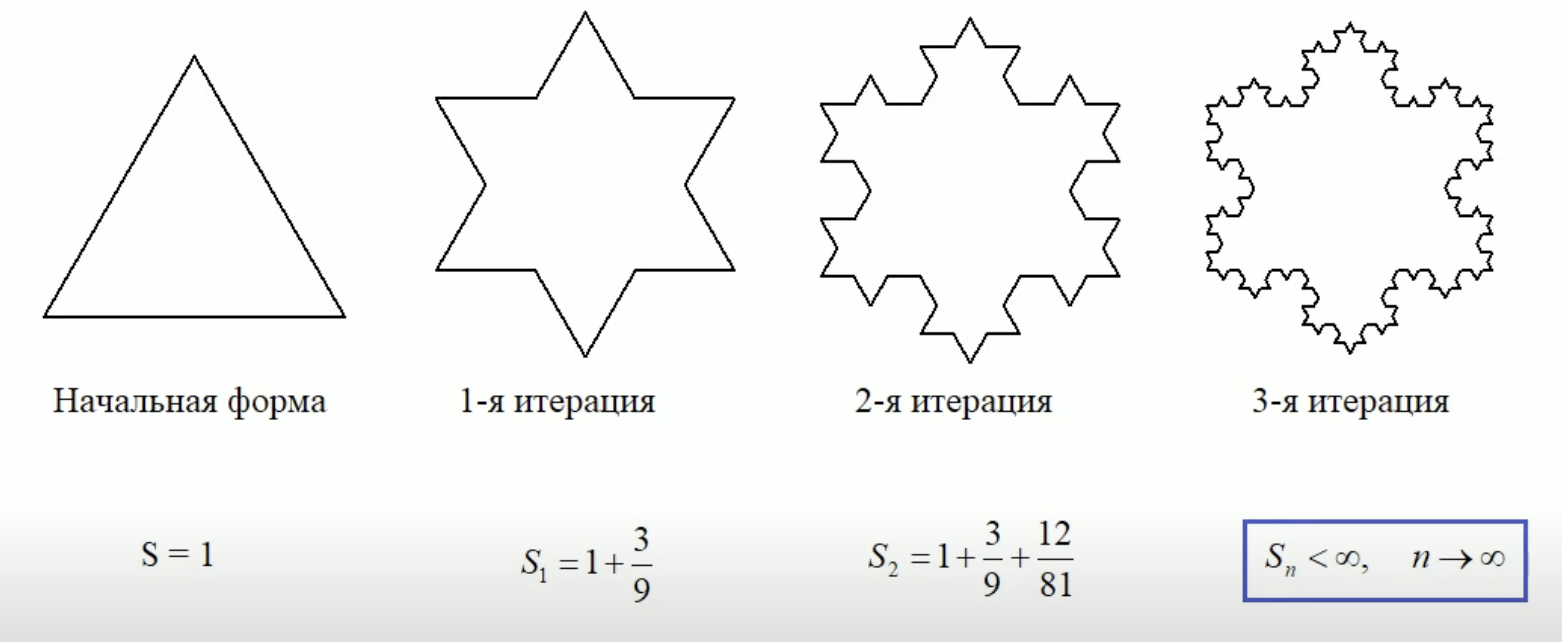


Рисунок 4 – этапы построения снежинки Коха

Начнем же программировать. Т. к. фрактал – непрерывный (то есть не состоит из точек, только прямые), то было принято решение использовать библиотеку turtle, которая легко и просто предоставляет возможность реализовать графику. Для начала напишем функцию, которая будет рисовать сегмент прямой Коха (см. рисунок 5). Далее будем использовать эту функцию.

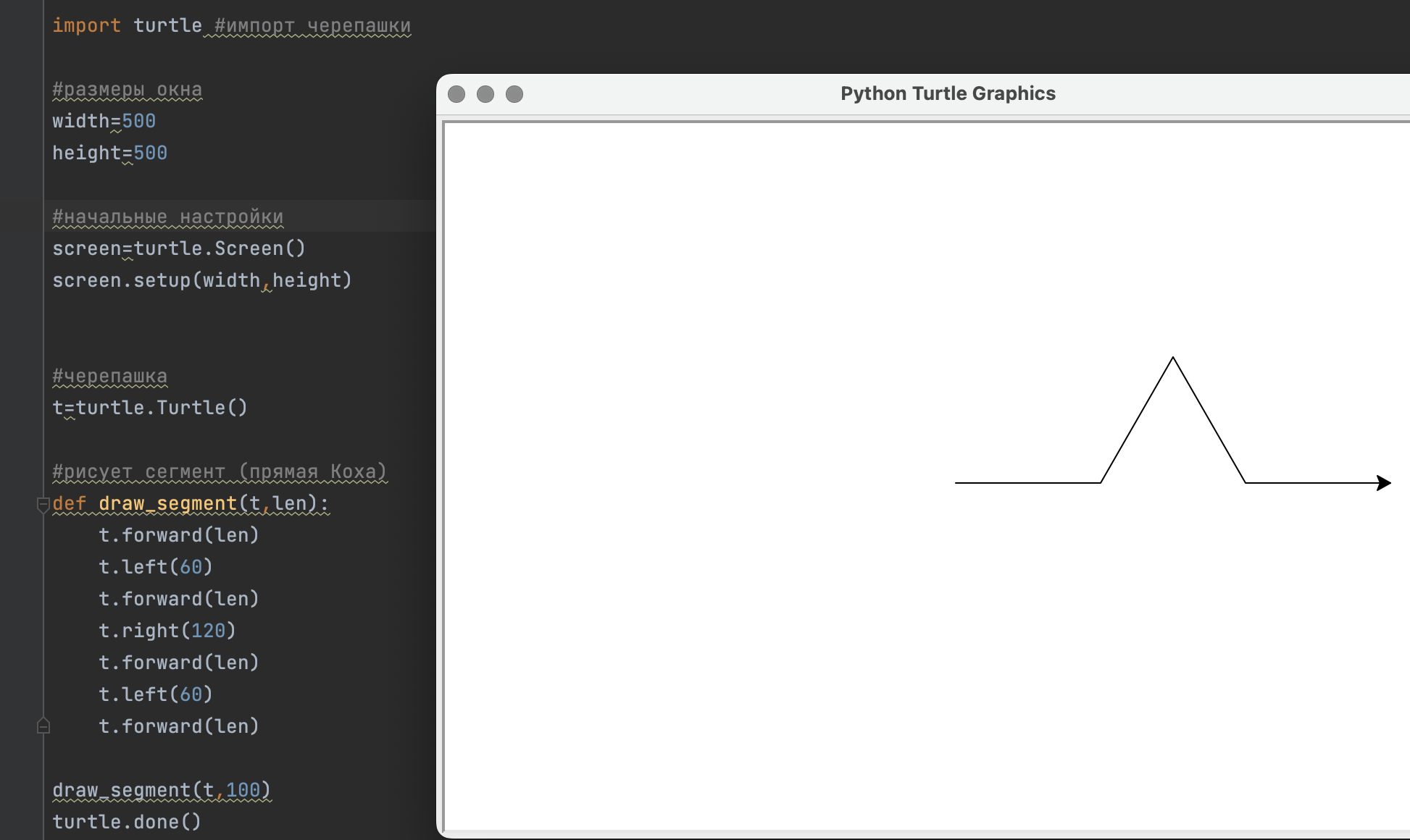


Рисунок 5 – сегмент прямой Коха

Теперь поработаем с рекурсией и сделаем так, чтобы получилось что-то похожее на реальный фрактал прямой Коха (рисунок 6).

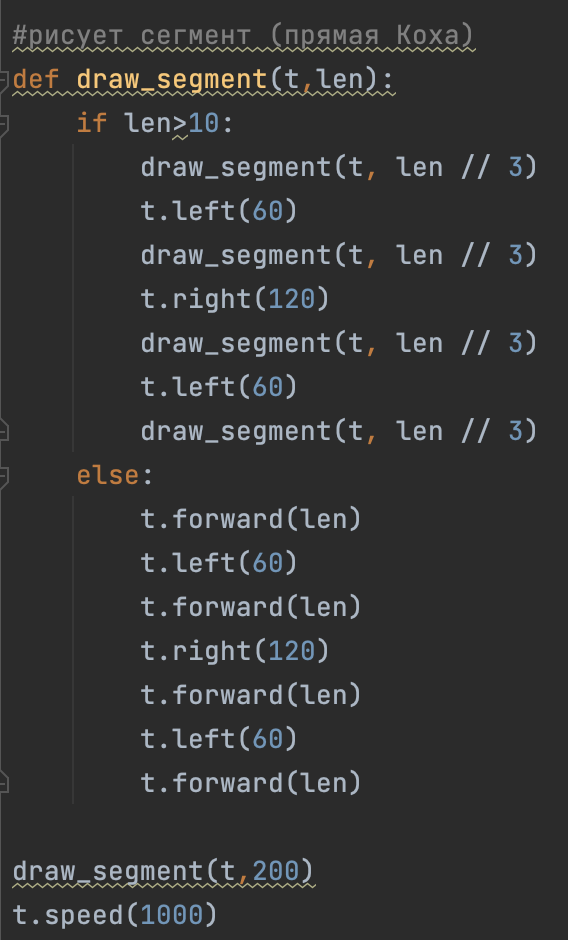


Рисунок 6 – видоизмененная часть кода

Итог - получаем прямую Коха (рисунок 7).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 7 – прямая Коха

Тогда останется немного изменить код – сделать так, чтобы черепашка поворачивалась, а затем снова чертила (как в треугольнике), а также в самой функции поменять 10 на 5, чтобы быстрее получить ответ и поставить большое число на скорость (Рисунок 8).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 8 – дополненный код

Вот такая снежинка получилась в итоге (Рисунок 9).

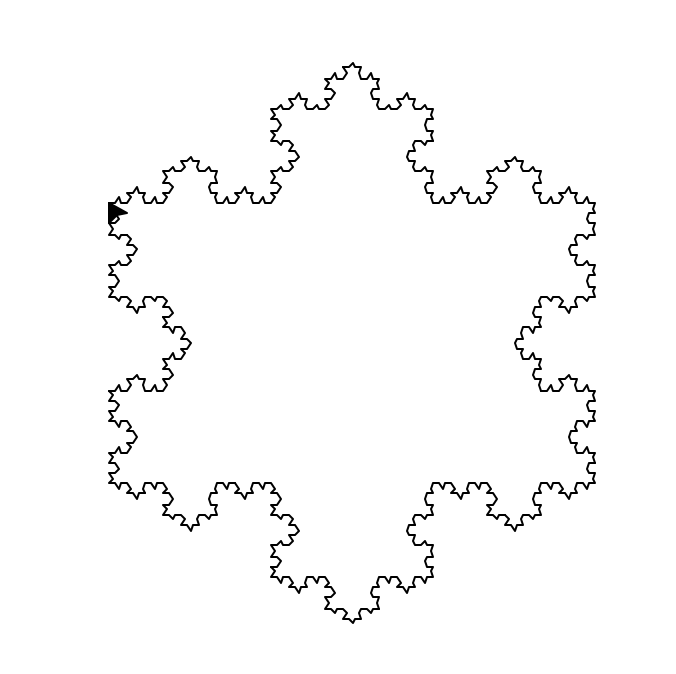


Рисунок 9 – снежинка Коха

Код:

# импорт черепашки  
import turtle  
  
#размеры окна  
width=500  
height=500  
  
#начальные настройки  
screen=turtle.Screen()  
screen.setup(width,height)  
  
  
#черепашка  
t=turtle.Turtle()  
t.speed(10000)  
  
#рисует сегмент (прямая Коха)  
def draw\_segment(t,len):  
 if len>5:  
 draw\_segment(t, len // 3)  
 t.left(60)  
 draw\_segment(t, len // 3)  
 t.right(120)  
 draw\_segment(t, len // 3)  
 t.left(60)  
 draw\_segment(t, len // 3)  
 else:  
 t.forward(len)  
 t.left(60)  
 t.forward(len)  
 t.right(120)  
 t.forward(len)  
 t.left(60)  
 t.forward(len)  
  
#сегменты как стороны треугольника  
draw\_segment(t,100)  
t.right(120)  
draw\_segment(t,100)  
t.right(120)  
draw\_segment(t,100)  
  
turtle.done()

**Треугольник Серпинского**

В основе построения треугольника Серпинского лежат уже не прямые, а точки. Поэтому библиотека с черепашкой не подходит. Также в качестве усложнения задания будем выводить не точки, а стрелки, которые будут указывать на следующую точку при построении.

Для понимания построения использовалось следующее видео - <https://www.youtube.com/watch?v=kbKtFN71Lfs&t=0s> (именно благодаря нему пишется этот проект). Если кратко объяснить, то изначально берется 3 точки (вершины треугольника – неважно какого). Берется рандомная точка. Каждой точке присваиваются значения 1 и 2, 3 и 4, 5 и 6. Рандомно бросается кубик – какое число выпало, к такой точке мы приближаемся от этой точки наполовину. И т. д. – от последней поставленной точке. Если ускорить процесс – получим треугольник Серпинского. Это называется играми хаоса. Запрограммируем этот алгоритм и проверим на практике!

Для начала решим задачу проще – не будем вводить стрелки, а просто построим обычный треугольник по алгоритму, описанному выше. Для реализации будем использовать библиотеку random и matplotlib (для генерации рандомных чисел и построения). Оси зададим до 500. А также сами зададим координаты треугольника, чтобы было нагляднее и треугольник был большим. Далее ставим точки, а также генерируем ту самую рандомную точку на поле. Затем фиксируем начальную точку и ту, в которую перейдем. При этом рандомно выбираем индекс вершины треугольника.

Код:

# импорт библиотек  
import matplotlib.pyplot as plt  
import random  
  
# настройки  
plt.axis([0,500,0,500]) # оси  
  
# количество повторений  
n=5000  
  
# задаем вершины треугольника и рисуем их на графике  
trinagle=[[0,100],[490,490],[460,130]]  
  
for point\_tr in trinagle:  
 plt.plot(point\_tr[0],point\_tr[1],marker='.')  
  
# выбираем рандомную точку на графике  
x=random.randint(0,500)  
y=random.randint(0,500)  
plt.plot(x,y,marker='.')  
  
count\_point=0 # кол-о нарисованных точек  
point=(x,y) # координаты текущей точки  
next\_point=(0,0) # координаты следующей точки  
while count\_point<=n: # цикл, пока все не нарисуем  
 flag=random.randint(0,2) # выбираем рандомно индекс точки треугольника  
 next\_point=((point[0]+trinagle[flag][0])/2,(point[1]+trinagle[flag][1])/2)  
   
 plt.plot(point[0], point[1], marker='.')  
 point=next\_point # текущая точка становится равна предыдущей  
 count\_point+=1 # кол-о точек поставленных увеличилось  
  
plt.show()

Посмотрим на результат работы программы для различных значений n (рисунки 10–12).

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 10 – результат работы программы для n=100

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 11 – результат работы программы для n=500

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 12 – результат работы программы для n=2000

На рисунке 13 уже отчетливо выражен треугольник Серпинского! К чему и должны были прийти.

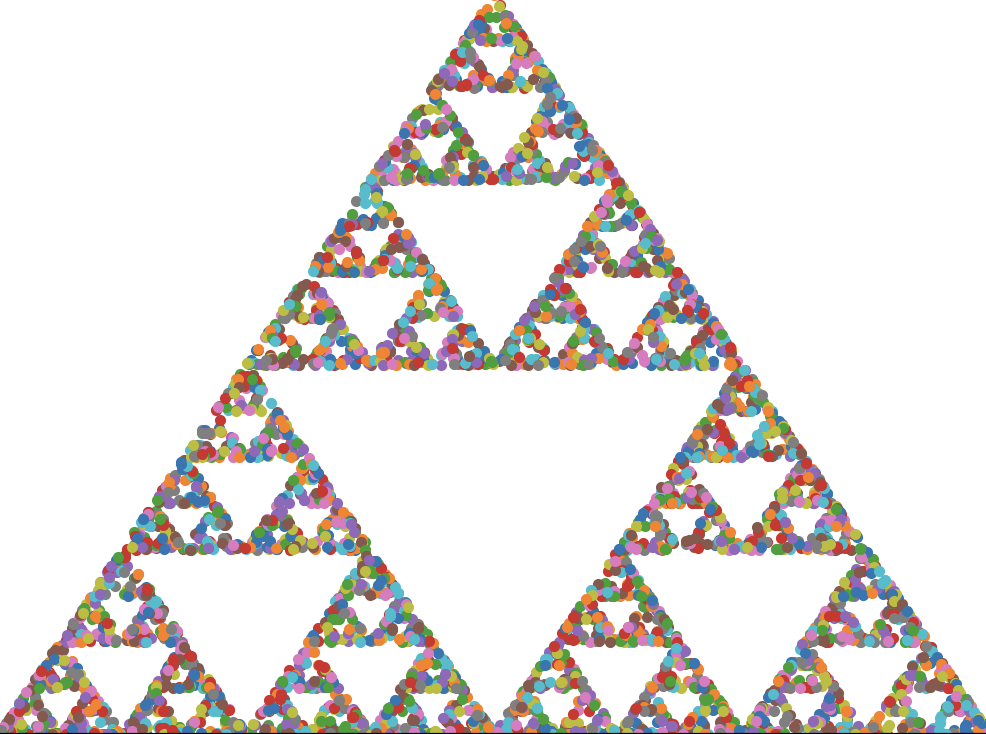


Рисунок 13 – треугольник Серпинского

Теперь усложним задачу и вместо точек будем рисовать маленькие стрелки, которые будут указывать на направление следующей точки. Достаточно написать функцию, которая бы определяла направление стрелки в зависимости от положения точек, а также делала длину стрелки небольшой.

Код:

import numpy as np  
  
def arrow(point,next\_point):  
 # Задаем две точки (x1, y1) и (x2, y2)  
 x1, y1 = point  
 x2, y2 = next\_point  
   
 # Вычисляем направление вектора стрелки  
 dx, dy = x2 - x1, y2 - y1  
  
 # Нормируем вектор, чтобы его длина была 2  
 length = np.sqrt(dx\*\*2 + dy\*\*2)  
 dx = 2 \* dx / length  
 dy = 2 \* dy / length  
  
 # Рисуем стрелку с началом в точке (x1, y1) и длиной вектора (dx, dy)  
 return plt.arrow(x1, y1, dx, dy, head\_width=1, head\_length=0.4, fc='black', ec='black')

А в предыдущем коде – вместо точки вызываем данную функцию для двух точек. Также для наглядности были изменены координаты вершин треугольника и оси. Тогда протестируем программу для различных n (рисунки 14–16).

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 14 – результат работы программы для n=100

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 15 – результат работы программы для n=500

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 15 – результат работы программы для n=2000

И окончательное изображение получим для n=5000 (см. рисунок 16). На нем можно отчетливо разглядеть начальные стрелки (за счет точки, которая рандомно была, выбрала вне треугольника).

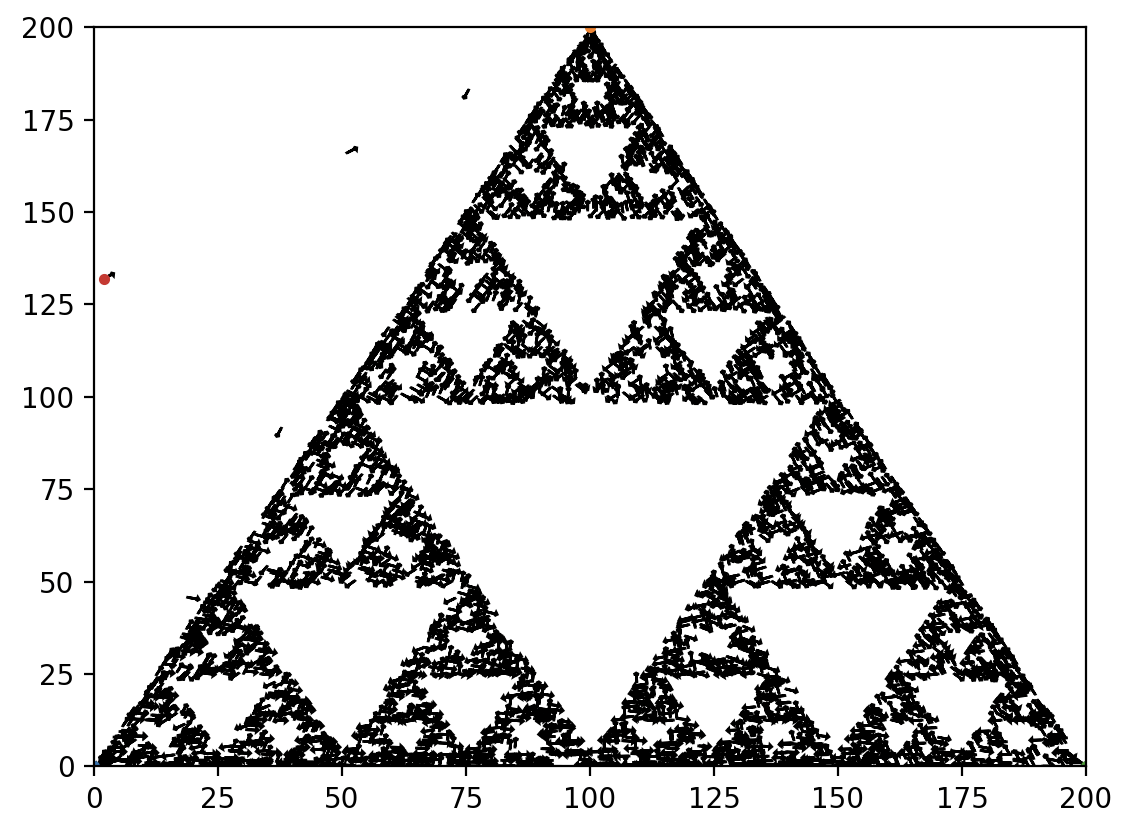


Рисунок 16 – треугольник Серпинского со стрелками, указывающими местоположение следующей точки

Также с этим кодом можно экспериментировать, меняя положение и количество точек, а также расстояния. Например – изменив количество точек на 4 и изобразив квадрат, а также изменив расстояния можно получить другой фрактал (см. рисунок 17).

Изображение выглядит как схематичный, календарь

Автоматически созданное описание

Рисунок 17 – фрактал при изменении величин

Код:

# импорт библиотек  
import matplotlib.pyplot as plt  
import random  
  
# настройки  
plt.axis([0, 500, 0, 500]) # оси  
  
# количество повторений  
n = 5000  
  
# задаем вершины и рисуем их на графике  
trinagle = [[10, 10], [10, 490], [490, 490],[490,10]]  
  
for point\_tr in trinagle:  
 plt.plot(point\_tr[0], point\_tr[1], marker='.')  
  
# выбираем точку   
x = 240  
y = 240  
plt.plot(x, y, marker='.')  
  
count\_point = 0 # кол-о нарисованных точек  
point = (x, y) # координаты текущей точки  
next\_point = (0, 0) # координаты следующей точки  
while count\_point <= n: # цикл, пока все не нарисуем  
 flag = random.randint(0, 3) # выбираем рандомно индекс точки   
 next\_point = ((point[0] + 2\*trinagle[flag][0]) /3, (point[1] + 2\*trinagle[flag][1]) /3)  
 plt.plot(point[0], point[1], marker='.')  
 point = next\_point # текущая точка становится равна предыдущей  
 count\_point += 1 # кол-о точек поставленных увеличилось  
  
plt.show()

**Фрактал Брансли (папоротник)**

Для построения этого фрактала применяются 4 афинные преобразования (см. рисунок 18).

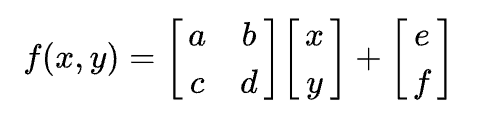


Рисунок 18 – формула для преобразований

А также понадобятся следующие таблицы с коэффициентами для вычислений (см. рисунок 19).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 19 - коэффициенты уравнений

Тогда, согласно этим уравнениям, можем перенести данные вычисления в программу.

В данном случае путем проб подобрали оптимальный размер окна, а также уменьшили размер точек в связи с масштабом.

Код:

# импорт библиотек  
import matplotlib.pyplot as plt  
import random  
  
# настройки  
plt.axis([-5,7.5,-5,15]) # оси  
  
# количество повторений  
n=5000  
  
x = 0.0  
y = 0.0  
count=0  
while count <= n:  
 # раписываем уравнения  
 r=random.random()  
 if r < 0.01:  
 x = 0.0  
 y = 0.16 \* y  
 elif r < 0.86:  
 x = 0.85 \* x + 0.04 \* y  
 y = -0.04 \* x + 0.85 \* y + 1.6  
 elif r < 0.93:  
 x = 0.2 \* x - 0.26 \* y  
 y = 0.23 \* x + 0.22 \* y + 1.6  
 else:  
 x = -0.15 \* x + 0.22 \* y  
 y = 0.26 \* x + 0.24 \* y + 0.44  
  
 count+=1  
 plt.plot(x,y,marker='.',markersize=1)  
  
plt.show()

Тогда для различных значений n получим различные рисунки (см. рисунки 20–21).

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 20 – результат работы программы для n=100

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 21 – результат работы программы для n=5000

И, наконец, вместо точек – стрелки (просто используем функцию из прошлого построения). Но из-за масштаба стрелки выражены не так ярко, как в прошлом примере (см. рисунок 22).

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 22 – результат работы программы со стрелками, указывающими положение следующей точки

# Заключение

Цель данной работы достигнута. Изучены некоторые виды фракталов, а также написаны алгоритмы их построения.