**МІНІСТЕРСТВО ОСВІТИ І НАУКИ**

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМ.ІГОРЯ СІКОРСЬКОГО»**

**НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ**

Проєктна робота

«Стандартизована ітераційна генерація фігур, кривих і фракталів та стандартизоване подання даної генерації у точковому, растровому та векторному вигляді»

Виконали:  
Студенти 2 курсу  
Групи ФІ-21:  
Голуб Михайло,  
Кияшко Дарина,  
Климентьєв Максим

Перевірив:  
Хайдуров В. В.

Київ 2024

ЗМІСТ

[1. ЗАВДАННЯ 3](#_Toc20271)

[2. АКТУАЛЬНІСТЬ 4](#_Toc9965)

[3. ХІД РОБОТИ 5](#_Toc16905)

[3.1. РОЗРОБКА УНІФІКОВАНОЇ СИСТЕМИ СТВОРЕННЯ ФІГУР 5](#_Toc24352)

[3.2. РЕАЛІЗАЦІЯ КЛАСІВ-ЛЕКАЛ ДЛЯ ФРАКТАЛІВ, ЩО ВИВЧАЛИСЬ ПРОТЯГОМ СЕМЕСТРУ 9](#_Toc31815)

[3.2.1. РЕАЛІЗАЦІЯ КЛАСУ-ЛЕКАЛА ДЛЯ ФРАКТАЛІВ L-СИСТЕМ 9](#_Toc18559)

[3.2.2. РЕАЛІЗАЦІЯ КЛАСУ-ЛЕКАЛА ДЛЯ АФІННИХ ФРАКТАЛІВ 11](#_Toc13822)

[3.2.3. РЕАЛІЗАЦІЯ КЛАСУ-ЛЕКАЛА ДЛЯ ДВОВИМІРНОЇ МНОЖИНИ КАНТОРА 14](#_Toc32636)

[3.3. КОМПЛЕКСНІ ФРАКТАЛИ 16](#_Toc5059)

[3.4. ВИПАДКОВІ ФРАКТАЛИ 17](#_Toc20632)

[3.4.1. БРОУНІВСЬКЕ ДЕРЕВО 17](#_Toc2382)

[3.4.2. БРОУНІВСЬКИЙ РУХ 21](#_Toc15615)

[3.4.3. БАГАТОКУТНИЙ ФРАКТАЛ 23](#_Toc17952)

[3.5. НЕФРАКТАЛЬНІ ФІГУРИ ТА КРИВІ 24](#_Toc17677)

[3.5.1. ПРАВИЛЬНІ БАГАТОКУТНИКИ 24](#_Toc9348)

[3.5.2. ПОЛІНОМІАЛЬНІ ФУНКЦІЇ. 26](#_Toc12250)

[3.5.3. КАРДІОЇДИ 27](#_Toc5467)

[3.5.4. СПІРАЛЬ АРХІМЕДА 28](#_Toc60)

[3.5.5. ФІГУРИ ЛІССАЖУ 29](#_Toc6042)

[3.6. ВИКОРИСТАННЯ СТВОРЕНИХ КЛАСІВ-ЛЕКАЛ 30](#_Toc2797)

[4. ПРИКЛАДИ РОБОТИ СТВОРЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 35](#_Toc97)

[5. ВИСНОВКИ 58](#_Toc32427)

# ЗАВДАННЯ

Розробити уніфіковану систему створення фігур, таких як фрактали, криві та правильні багатокутники, за заданими параметрами. Розробити програмне забезпечення для побудови та виведення на екран фігур.

Презентувати роботу, сформувати звіт з проєктної роботи.

# АКТУАЛЬНІСТЬ

У сучасному світі важливу роль відіграє графічне програмне забезпечення (далі ПЗ), але через значущі відмінності в реалізації та часткову відсутність стандартизації такого ПЗ його використання стає дедалі складнішим.

Існування десятків різних графічних «двигунів» з різними інтерфейсами, методами та їх параметрів призводить до значної спеціалізації програмістів: програміст що вивчив графічний «двигун» А навряд зможе легко використати графічний «двигун» Б

Щоб вирішити таку проблему можна створювати стандартизовані точки контакту частин ПЗ: створити обов’язковий набір змінних що передаються від однієї частини ПЗ до іншого, а усі додаткові змінні винести окремо.

Для демонстрації цієї проблеми у зменшеному масштабі обрано генерацію фігур як першу частину ПЗ та їх вивід на екран як другу. Перша частина програмного забезпечення буде представлена більш-менш уніфікованими генераторами точок, а друга частина буде представлена ПЗ що буде виводити точки на екран. Обов’язковий набір змінних – масив точок та номер ітерації генерації фігури, додаткові змінні – аргументи генерації та візуалізації фігур.

# ХІД РОБОТИ

### РОЗРОБКА УНІФІКОВАНОЇ СИСТЕМИ СТВОРЕННЯ ФІГУР

Було вирішено перед початком розробки програмного забезпечення чітко визначити структуру класів, їх методи та змінні.

Було вирішено розділити код на дві частини: генеративну та візуальну. Генеративна частина буде створювати масив точок з отриманих параметрів, а візуальна буде відображати масив на екрані

Створено клас-лекало Figure в підкаталозі Figures:

class Figure:  
 def \_\_init\_\_(self, \*args):  
 pass  
  
 def check\_args(self, \*args):  
 pass  
  
 def generate\_points(self, iteration):  
 pass

Визначено призначення та аргументи методів:

* \_\_init\_\_(self, \*args) приймає усі константи потрібні для побудови фігури;
* check\_args(self, \*args) перевіряє отримані константи під час ініціалізації класу;
* generate\_points(self, iteration) повертає масив точок фігури згідно вказаної ітерації: для фракталів iteration застосовуються за прямим призначенням, у інших фігурах iteration теж змінює результат, але іншим чином.

Створено клас-лекало FigureBuilder в кореневому каталозі для швидкої ініціалізації та використання Figure:

class FigureBuilder:  
 def build(self, figure: Figure, \*args):  
 return figure.generate\_points(\*args)

Створено клас Window в кореневому каталозі для відображення отриманого від Figure.generate\_points масиву точок:

import matplotlib.animation as animation  
  
  
class Window:  
 def draw(self, input\_value, \*\*kwargs):  
 multiplayer = kwargs.get("multiplayer", 1) \* kwargs.get("multi", 1)  
 interval = kwargs.get("interval", 30)  
 markersize = kwargs.get("markersize", 0.6)  
 figsize = kwargs.get("figsize", (5, 5))  
 fps = kwargs.get("fps", 15)  
  
 cmap = kwargs.get("cmap", 'gray') *# 'inferno'* animation\_need = kwargs.get("animation\_need", False)  
 animation\_save = kwargs.get("animation\_save", False)  
  
 is\_edge = kwargs.get("is\_edge", False)  
 is\_fixed\_size = kwargs.get("is\_fixed\_size", False)  
  
 has\_axes = kwargs.get("has\_axes", True)  
 has\_background = kwargs.get("has\_background", True)  
  
 if animation\_save:  
 if not os.path.isdir("images"):  
 os.mkdir("images")  
 add\_to\_name = len(os.listdir("./images"))  
 filename = kwargs.get('filename', 'figure' + str(add\_to\_name))  
  
 if is\_edge:  
 linestyle = '-'  
 else:  
 linestyle = ''  
  
 try:  
 x, y = input\_value  
 is\_matrix = False  
 except ValueError:  
 is\_matrix = True  
  
 if is\_matrix:  
 if not isinstance(input\_value, list):  
 plt.imshow(input\_value, cmap=cmap)  
 elif not animation\_need:  
 plt.imshow(input\_value[-1], cmap=cmap)  
 else:  
 fig, ax = plt.subplots(figsize=figsize, constrained\_layout=(not has\_background))  
  
 ax.imshow(input\_value[0], cmap=cmap)  
 if not has\_axes:  
 ax.axis('off')  
  
 def update(frame):  
 ax.clear()  
 ax.imshow(input\_value[frame], cmap=cmap)  
 if not has\_axes:  
 ax.axis('off')  
  
 ani = animation.FuncAnimation(fig=fig, func=update, frames=len(input\_value), interval=interval)  
 else:  
 if is\_fixed\_size:  
 x\_limit\_left = min(x) - abs(min(x)) / 2  
 x\_limit\_right = max(x) + abs(max(x)) / 2  
 y\_limit\_bottom = min(y) - abs(min(y)) / 2  
 y\_limit\_top = max(y) + abs(max(y)) / 2  
 if not animation\_need:  
 plt.plot(x, y, marker='o', linestyle=linestyle, markersize=markersize)  
 else:  
 fig, ax = plt.subplots(figsize=figsize, constrained\_layout=(not has\_background))  
  
 if is\_fixed\_size:  
 ax.set\_xlim(x\_limit\_left, x\_limit\_right)  
 ax.set\_ylim(y\_limit\_bottom, y\_limit\_top)  
 if not has\_axes:  
 ax.axis('off')  
 ax.plot(x, y, marker='o', linestyle=linestyle, markersize=markersize)  
  
 def update(frame):  
 frame = frame \* multiplayer  
 ax.clear()  
 if is\_fixed\_size:  
 ax.set\_xlim(x\_limit\_left, x\_limit\_right)  
 ax.set\_ylim(y\_limit\_bottom, y\_limit\_top)  
 if not has\_axes:  
 ax.axis('off')  
 ax.plot(x[:frame], y[:frame], marker='o', linestyle=linestyle, markersize=markersize)  
  
 ani = animation.FuncAnimation(fig=fig, func=update, frames=len(x)//multiplayer+1, interval=interval)  
  
 if animation\_need and animation\_save:  
 ani.save("./images/" + filename + '.gif', writer=animation.PillowWriter(fps=fps))  
  
 plt.show()

Цей клас містить єдиний метод draw, що приймає масив точок в input\_value та інші опціональні аргументи, такі як:

* multiplayer – кількість точок, що виводяться кожного кадру анімації;
* interval – затримка між кадрами анімації побудови, що виводиться на екран;
* markersize – розмір крапок, які зображають точки;
* figsize – розмір вікна в умовних одиницях;
* fps – частота кадрів анімації побудови, що зберігається у файл .gif;
* cmap – палітра кольорів;
* animation\_need – вказує на те, чи потрібна анімація;
* animation\_save – вказує на те, чи потрібно зберігати анімацію;
* is\_edge – вказує на те, чи потрібно малювати ребра;
* is\_fixed\_size – вказує на те, чи потрібно зафіксувати розмір вікна під час анімації;
* has\_axes – вказує на те, чи потрібно відображати вісі;
* has\_background – вказує на те, чи потрібно відображати фон довкола побудованої фігури.

Створено клас-лекало Director в кореневому каталозі для швидкої побудови і відображення побудованих об’єктів, використовуючи класи Window та FigureBuilder:

class FigureDirector:  
 def build(self, figure: Figure, \*\*kwargs):  
 if "it" in kwargs:  
 Window().draw(FigureBuilder().build(figure, kwargs['it']), \*\*kwargs)  
 elif "iterations" in kwargs:  
 Window().draw(FigureBuilder().build(figure, kwargs['iterations']), \*\*kwargs)  
 else:  
 Window().draw(FigureBuilder().build(figure), \*\*kwargs)

Цей клас під час ініціалізації отримує клас-лекало фігури, яку потрібно створити та усі необхідні аргументи для її створення.

Створено клас App для швидкої ініціалізації екземплярів класу Director:

class App:  
 def \_\_init\_\_(self):  
 self.figures = {  
 *# “name” : Figure.Figure*   
 }  
  
 def create\_figure(self, name, \*args, \*\*kwargs):  
 if name in self.figures:  
 FigureDirector().build(self.figures[name](\*args), \*\*kwargs)  
 else:  
 raise ValueError("Wrong figure name")

Цей клас міститиме назви усіх класів-лекал фігур та посилання на них у змінній self.figures. Через метод create\_figure можна швидко створити та відобразити потрібну фігуру з потрібними параметрами.

### РЕАЛІЗАЦІЯ КЛАСІВ-ЛЕКАЛ ДЛЯ ФРАКТАЛІВ, ЩО ВИВЧАЛИСЬ ПРОТЯГОМ СЕМЕСТРУ

Було обрано фрактали, що вивчались протягом семестру, як найпростіші для реалізації, тож було вирішено їх реалізувати найпершими. До фракталів, що вивчались протягом семестру, входять:

* L-системи;
* Афінні;
* Двовимірна (матрична) множина Кантора.

### РЕАЛІЗАЦІЯ КЛАСУ-ЛЕКАЛА ДЛЯ ФРАКТАЛІВ L-СИСТЕМ

Фрактал L-системи це фрактал, що використовує текстовий рядок для ітерацій: символ F робить крок вперед на визначену довжину, +/- повертають напрямок кроку на визначений кут, усі інші символи використовуються для ітерування. Набір правил складається з пар (символ, рядок), під час ітерування кожне правило замінює кожен символ з пари на рядок.

На базі класа Figure було реалізовано клас-лекало для фракталів L-систем:

import numpy as np  
class LsystemFractal:  
 *"""  
 Lsystem implementation of fractals (see compgraph Lab3)  
 Only radians  
 """* def \_\_init\_\_(self, axiom: str, rules: dict, fi: float, dfi: float, \*args):  
 *"""  
 Initiates Lsystem fractal with given parameters but before checks if parameters are correct  
  
 # Parameters:  
 axiom: string (starting L-axiom)  
 rules: dict (rules for how to change each letter (not specific symbol) in iteration)  
 max\_iterations: int (how many iterations)  
 fi: float (starting angular) (now only radians)  
 dfi: float (angular velocity) (now only radians)  
 """* if args != ():  
 raise ValueError(f"Wrong number of arguments, {args} excess")  
 self.list\_to\_check = [str, dict, float, float] *# Change if count of arguments changes* self.check\_args(axiom, rules, fi, dfi)  
  
 self.axiom = axiom  
 self.rules = rules  
 self.fi = fi  
 self.dfi = dfi  
  
 def check\_args(self, \*args):  
 *"""  
 Checks if parameters are correct  
  
 if not - raises ValueError with appropriate message  
 """* for argument\_index in range(len(args)):  
 if type(args[argument\_index]) is not self.list\_to\_check[argument\_index]:  
 raise ValueError(f"Wrong argument {args[argument\_index]}, which is {type(args[argument\_index])} type, expected {self.list\_to\_check[argument\_index]} type")  
  
 def generate\_points(self, iteration):  
 *"""  
 Generates specified iteration of Lsystem fractal  
  
 # Returns:  
 (N+1 shape, N+1 shape) arrays of x and y coordinates  
 """* result = self.axiom  
 for iteration in range(iteration):  
 new\_axiom = ''  
 for word\_place in range(len(result)):  
 if result[word\_place] in self.rules.keys():  
 new\_axiom += self.rules[result[word\_place]]  
 else:  
 new\_axiom += result[word\_place]  
 result = new\_axiom  
  
 N = len(result)  
 L = 2  
 x = np.zeros(N+1)  
 y = np.zeros(N+1)  
 for i in range(N):  
 x[i+1] = x[i]  
 y[i+1] = y[i]  
 if result[i] == 'F':  
 x[i+1] += L\*np.cos(self.fi)  
 y[i+1] += L\*np.sin(self.fi)  
 elif result[i] == '+':  
 self.fi += self.dfi  
 elif result[i] == '-':  
 self.fi -= self.dfi  
 return x, y

Метод \_\_init\_\_ приймає початковий рядок axiom, словник правил rules, початковий напрямок кроку fi та дельту зміни напрямку кроку dfi. Після цього виконується перевірка на відсутність зайвих параметрів та на правильність типів параметрів. Після успішного виконання перевірки, отримані параметри записуються як змінні класу. У разі неуспішного виконання перевірки, виконання коду зупиняється та викликається відповідна помилка.

Метод generate\_points приймає номер ітерації iteration яку потрібно згенерувати. Метод ітерує рядок iteration разів, після чого обчислює розміщення точок отриманого фракталу на координатній площині та повертає його.

### РЕАЛІЗАЦІЯ КЛАСУ-ЛЕКАЛА ДЛЯ АФІННИХ ФРАКТАЛІВ

Афінний фрактал (система ітерованих функцій) це фрактал який кожної ітерації створює наступну точку шляхом афінного перетворення останньої створеної точки.

На базі класу Figure створено клас-лекало для афінних фракталів:

import numpy as np  
class AffineFractal:  
 *"""  
 Build fractals using affine transformation (see compgraph Lab4)  
 """* def \_\_init\_\_(self, list\_of\_lists\_of\_parameter: list, skip\_first\_n\_points: int=10\*\*2, standart\_type:bool =True, \*args):  
 *"""  
 Initiates affine fractal with given parameters but before checks if parameters are correct  
  
 # Parameters:  
 list\_of\_lists\_of\_parameter: list (list with lists in it (For a,b,c,d,e,f and, if needed, p))  
 skip\_first\_n\_points: int (skip first n points of the output)  
 standart\_type: bool (if true -- Decart, if false -- Polar coordinates)  
 """* if args != ():  
 raise ValueError(f"Wrong number of arguments, {args} excess")  
 self.list\_to\_check = [list, int, bool]  
 self.check\_args(list\_of\_lists\_of\_parameter, skip\_first\_n\_points, standart\_type)  
  
 self.skip\_first\_n\_points = skip\_first\_n\_points  
 self.standart\_type = standart\_type  
 if self.standart\_type:  
 if len(list\_of\_lists\_of\_parameter) == 7:  
 self.a, self.b, self.c, self.d, self.e, self.f, self.p = list\_of\_lists\_of\_parameter  
 else:  
 self.a, self.b, self.c, self.d, self.e, self.f = list\_of\_lists\_of\_parameter  
 self.p = [1/len(self.a)] \* len(self.a)  
 else:  
 if len(list\_of\_lists\_of\_parameter) == 7:  
 self.r, self.s, self.t, self.fi, self.e, self.f, self.p = list\_of\_lists\_of\_parameter  
 else:  
 self.r, self.s, self.t, self.fi, self.e, self.f = list\_of\_lists\_of\_parameter  
 self.p = [1/self.r.\_\_len\_\_()] \* len(self.r)  
  
 def check\_args(self, \*args):  
 *"""  
 Checks if parameters are correct  
  
 if not - raises ValueError with appropriate message  
 """* for argument\_index in range(len(args)):  
 if type(args[argument\_index]) is not self.list\_to\_check[argument\_index]:  
 raise ValueError(f"Wrong argument {args[argument\_index]}, which is {type(args[argument\_index])} type, expected {self.list\_to\_check[argument\_index]} type")  
  
 previous\_parameter = None  
 for parameter in args[0]:  
 if type(parameter) is not list: *# checks if what it received is list* raise ValueError(f"Wrong parameter {parameter}, which is {type(parameter)} type, expected list type")  
  
 for parameter\_index in range(len(parameter)): *# checks if what it received is list where each element is int or float* if type(parameter[parameter\_index]) is not int and type(parameter[parameter\_index]) is not float:  
 raise ValueError(f"Wrong parameter {parameter[parameter\_index]}, which is {type(parameter[parameter\_index])} type, expected int or float type")  
  
 if previous\_parameter is not None: *# checks len of each parameter* if len(previous\_parameter) != len(parameter):  
 raise ValueError(f"Wrong length of parameter {parameter} in row {args[0].index(parameter)}, expected {len(previous\_parameter)}")  
 if len(args[0]) != 6 and len(args[0]) != 7:  
 raise ValueError(f"Wrong size of list {args[0]} whose len is: {len(args[0])}, expected 6 or 7")  
 previous\_parameter = parameter  
  
 def generate\_points(self, iteration):  
 *"""  
 Generates dot of affine fractal on each iteration  
 """* result = np.array(  
 [[0.0, 0.0]]\*iteration  
 )  
 size\_of\_variation = len(self.p)  
 if self.standart\_type:  
 for i in range(iteration-1): *# how to handle 'iteration = 0'? It should return starting configuration* variant = np.random.choice(size\_of\_variation, 1, p=self.p)  
 variant = variant[0]  
 xk = self.a[variant]\*result[i, 0] + self.b[variant]\*result[i, 1] + self.e[variant]  
 yk = self.c[variant]\*result[i, 0] + self.d[variant]\*result[i, 1] + self.f[variant]  
 result[i+1] = [xk, yk]  
 *# i += 1* else:  
 for i in range(iteration-1):  
 variant = np.random.choice(size\_of\_variation, 1, p=self.p)  
 variant = variant[0]  
 xk = self.r[variant]\*np.cos(self.t[variant])\*result[i, 0] - self.s[variant]\*np.sin(self.fi[variant])\*result[i, 1] + self.e[variant]  
 yk = self.r[variant]\*np.sin(self.t[variant])\*result[i, 0] + self.s[variant]\*np.cos(self.fi[variant])\*result[i, 1] + self.f[variant]  
 result[i+1] = [xk, yk]  
 *# i += 1* return result[self.skip\_first\_n\_points:, 0], result[self.skip\_first\_n\_points:, 1]

Метод \_\_init\_\_ приймає список параметрів lists\_of\_parametr, кількість точок з початку які не потрібно виводити skip\_first\_n\_points, standart\_type визначає чи є афінні перетворення у декартових координатах чи у полярних. Після отримання параметрів виконується перевірка на відсутність зайвих параметрів та правильність введених аргументів. Після успішного виконання перевірки, отримані аргументи записуються як змінні класу. У разі неуспішного виконання перевірки, виконання коду зупиняється та викликається відповідна помилка.

Метод generate\_points приймає номер ітерації iteration яку потрібно згенерувати. Метод створює масив довжиною iteration з початковою точкою (0, 0). Після цього, виконується ітерація iteration разів. Метод повертає частину масиву від позиції skip\_first\_n\_points до кінця.

### РЕАЛІЗАЦІЯ КЛАСУ-ЛЕКАЛА ДЛЯ ДВОВИМІРНОЇ МНОЖИНИ КАНТОРА

Множина Кантора це фрактал у просторі [0,1]N. Кожна ітерація замінює кожну заповнену частину простору на певний набір заповнених шматків простору. При N = 1 простір є відрізком, при N = 2 – квадратом, при N = 3 – кубом.

У двовимірному просторі множина Кантора кожної ітерації замінює кожен заповнений квадрат на певну структуру з квадратів меншого розміру.

На базі класу Figure створено клас-лекало для двовимірної множини Кантора:

import numpy as np  
class MatrixFractal:  
 *"""  
 Matrix implementation of fractals (see compgraph MKR)  
 """* def \_\_init\_\_(self, coefs:np.ndarray, \*args):  
 *"""  
 Initiates Matrix fractal with given coefs but before checks if parameters are correct  
  
 # Parameters:  
 coefs: np.ndarray (coefs of matrix which will be used in iterations)  
 """* if args != ():  
 raise ValueError(f"Wrong number of arguments, {args} excess")  
 self.list\_to\_check = [np.ndarray]  
 self.check\_args(coefs)  
  
 self.coefs = coefs  
  
 def check\_args(self, \*args):  
 *"""  
 Checks if parameters are correct  
  
 if not - raises ValueError with appropriate message  
 """* for argument\_index in range(len(args)):  
 if type(args[argument\_index]) is not self.list\_to\_check[argument\_index]:  
 raise ValueError(f"Wrong argument {args[argument\_index]}, which is {type(args[argument\_index])} type, expected {self.list\_to\_check[argument\_index]} type")  
  
  
 def generate\_points(self, iterations=3):  
 *"""  
 Generates more and more big matrix fractal on each iteration  
  
 # Updates:  
 Matrix each iteration  
  
 # Returns:  
 (row ^ N, col ^ N) matrix that should be displayed as image (plt.imshow)  
 """* def redo\_array(array, out\_array=None, \*args):  
 *"""  
 Makes array from array of arrays using recursion  
  
 array like [  
 [  
 [1, 1, 1],  
 [1, 1, 1],  
 [1, 1, 1]  
 ],  
  
 [  
 [1, 1, 1],  
 [1, 0, 1],  
 [1, 1, 1]  
 ]  
 ]  
 turns into array like [  
 [1, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1],  
 [1, 1, 1, 1, 1, 1]  
 ]  
 """* if len(array.shape) == 2:  
 if out\_array is None:  
 out\_array = array  
 return out\_array  
 else:  
 return np.concatenate((out\_array, array), axis=1)  
  
 for i in range(array.shape[0]):  
 if len(args) < 1 and out\_array is None:  
 out\_array = redo\_array(array[i], out\_array, \*args, i)  
 elif len(args) == 1:  
 out\_array = redo\_array(array[i], out\_array, \*args, i)  
 else:  
 out\_array = np.concatenate((out\_array, redo\_array(array[i], None, \*args, i)), axis=0)  
 return out\_array  
  
 matrix = np.ones((1, 1))  
 result = [matrix]  
 for it in range(iterations):  
 matrix = np.array([[coef \* matrix for coef in row] for row in self.coefs])  
 if matrix.shape[:-2] == (1, 1):  
 matrix = matrix.reshape(matrix.shape[:-2])  
 matrix = redo\_array(matrix)  
 result.append(1-matrix)  
 return result

Метод \_\_init\_\_ приймає коефіцієнти матриці. Після цього виконується перевірка на відсутність зайвих параметрів. Після успішного виконання перевірки, отриманий параметр записуються як змінна класу. У разі неуспішного виконання перевірки, виконання коду зупиняється та викликається відповідна помилка.

Метод generate\_points приймає номер ітерації iteration яку потрібно згенерувати. Метод ітерує матрицю iteration разів, після чого обчислює розміщення точок отриманого фракталу на координатній площині та повертає його.

### КОМПЛЕКСНІ ФРАКТАЛИ

Комплексний фрактал - фрактал, який будується ітеративним застосуванням математичних правил до комплексних чисел

Було побудовано такі комплексні фрактали:

* Julia Set
* Mandelbrot Set
* Multibrot Set
* Sinusoidal Julia Set
* Hyperbolic Tangent
* Burning Ship
* Tricorn

На базі класу Figure створено клас-лекало для комплексних фракталів:

\* КОД З ФАЙЛУ MorphingFractal.py \*

Метод \_\_init\_\_ приймає такі аргументи: 1) назва фракталу: 'Julia', 'Mandelbrot', 'Multibrot', 'BurningShip', 'Tricorn', 'SinJulia', 'HyperbolicTangent' 2) parameter (complex,int) - грає ключову роль у визначенні форми і структури фракталу 3) max\_iterations - визначає максимальну кількість кроків, які виконуються для визначення, чи належить точка на комплексній площині до фрактала 4) treshold- граничне значення, що використовується для визначення, чи належить точка до фрактала 5) width, height - ширина і висота згенерованого зображення. Після цього виконується перевірка на відсутність зайвих параметрів та на правильність типів параметрів. У разі неуспішного виконання перевірки, виконання коду зупиняється та викликається відповідна помилка.

Метод generate\_points приймає один аргумент iteration, який визначає кількість ітерацій для кожної точки. Відповідає за генерування точок, що представляють зображення фрактала.

### ВИПАДКОВІ ФРАКТАЛИ

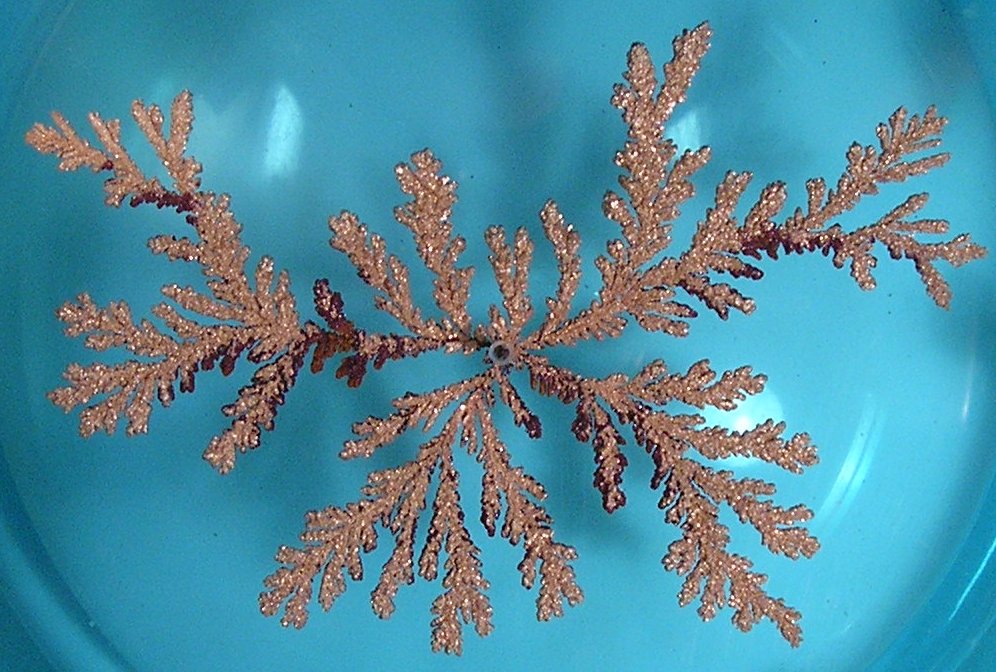
Випадковий (стохастичний) фрактал – фрактал що задається випадковим (стохастичним або псевдовипадковим) процесом. Кожна ітерація фракталу наближає фрактал до точної передачі стохастичного процесу, при чому наблизитись до абсолютно точної передачі не можливо.

Серед випадкових фракталів обрано:

* Броунівське дерево;
* Броунівський рух;
* Багатокутний фрактал (не має справжньої загальноприйнятої назви).

### БРОУНІВСЬКЕ ДЕРЕВО

Броунівське дерево — дерево, створене під впливом фізичного процесу, відомого як агрегація обмежена дифузією.



Мал. 3.1. Дерево мідних кристалів у мідному купоросі.

На базі класу Figure створено клас-лекало для броунівського дерева:

import numpy as np  
import numba  
class BrownianTree:  
 *"""  
 Uses brownian motion to generate brownian tree fractal. When it near drawn dot it draws self position and stopps.  
 """* def \_\_init\_\_(self, print\_need=False):  
 *"""  
 it exists only for printing  
 """* self.print\_need = print\_need  
  
 def generate\_points(self, iteration):  
 *"""  
 Creates circle (matrix that has 1 in center of circle, 2 after circle and 0 in other places) and then release brownian walker (random walker) in it and when it near drawn dot it draws self position and stopps.  
  
 # Returns:  
 (iteration\*2+5, iteration\*2+5) matrix that should be displayed as matrix (plt.matshow)  
 """* @numba.njit()  
 def create\_circle(matrix, x\_limit, y\_limit, radius, squareSize):  
 *"""  
 # Returns:  
 (iteration\*2+5, iteration\*2+5) matrix that has 1 in center of circle, 2 after circle and 0 in other places  
 """* for row in range(squareSize):  
 for col in range(squareSize):  
  
 if row == x\_limit and col == y\_limit:  
 matrix[row, col] = 1  
  
 elif np.sqrt((x\_limit-row)\*\*2 + (y\_limit-col)\*\*2) > radius:  
 matrix[row, col] = 2  
 return matrix  
  
 @numba.njit()  
 def checkAround(x, y, squareSize, matrix):  
 *"""  
 Checks if there is friend or exit from circle around point (x, y), if not, chooses random way to go  
  
 # Returns:  
 x - x coordinate,  
 y - y coordinate,  
 friend\_found - if there is dot nearby,  
 edge\_near - if there is edge nearny,  
 exit\_from\_circle - if it exited from circle  
 """* friend\_found = False  
 exit\_from\_circle = False  
 edge\_near = False  
  
 if (x + 1) > squareSize - 1 or (x - 1) < 1 or (y + 1) > squareSize - 1 or (y - 1) < 1:  
 edge\_near = True  
  
 if not edge\_near:  
 neighbor\_down = matrix[x + 1, y]  
 if neighbor\_down == 1:  
 friend\_found = True  
 if neighbor\_down == 2:  
 exit\_from\_circle = True  
  
 neighbor\_up = matrix[x - 1, y]  
 if neighbor\_up == 1:  
 friend\_found = True  
 if neighbor\_up == 2:  
 exit\_from\_circle = True  
  
 neighbor\_right = matrix[x, y+1]  
 if neighbor\_right == 1:  
 friend\_found = True  
 if neighbor\_right == 2:  
 exit\_from\_circle = True  
  
 neighbor\_left = matrix[x, y-1]  
 if neighbor\_left == 1:  
 friend\_found = True  
 if neighbor\_left == 2:  
 exit\_from\_circle = True  
  
 if not friend\_found and not edge\_near:  
 variant = np.random.choice(np.array([0, 1, 2, 3]), 1)  
 x, y = [(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)][variant[0]]  
  
 return x, y, friend\_found, edge\_near, exit\_from\_circle  
  
 radius = iteration  
 x\_limit = radius + 2  
 y\_limit = radius + 2  
 squareSize = radius\*2+5  
  
 matrix = np.zeros((squareSize, squareSize))  
  
 matrix = create\_circle(matrix, x\_limit, y\_limit, radius, squareSize)  
  
 rwalkers\_count = 0  
 rwalkers\_count\_stopped = 0  
  
 is\_completed = False  
  
 matrixs\_for\_animation = []  
  
 anim\_matrix\_range = np.arange(0, 40000, radius//3)  
  
 while not is\_completed:  
 rwalkers\_count += 1  
 np.random.seed()  
  
 theta = 2 \* np.pi \* np.random.random()  
  
 x = int(radius \* np.cos(theta)) + x\_limit  
 y = int(radius \* np.sin(theta)) + y\_limit  
  
 friend\_found = False  
 edge\_near = False  
  
 while not friend\_found and not edge\_near:  
 x\_new, y\_new, friend\_found, edge\_near, exit\_from\_circle = checkAround(x, y, squareSize, matrix)  
  
 if friend\_found:  
 matrix[x, y] = 1  
 rwalkers\_count\_stopped += 1  
 if rwalkers\_count\_stopped in anim\_matrix\_range:  
 if self.print\_need:  
 print("Random dots used on the field:", rwalkers\_count, "from which", rwalkers\_count\_stopped, "was drawn")  
 matrixs\_for\_animation.append(matrix.copy())  
  
 else:  
 x, y = x\_new, y\_new  
  
 if friend\_found and exit\_from\_circle:  
 if self.print\_need:  
 print("Dots drawn in the field:", rwalkers\_count\_stopped)  
 is\_completed = True  
  
 matrixs\_for\_animation.append(matrix.copy())  
 return matrixs\_for\_animation

Метод \_\_init\_\_ приймає аргумент, що позначає чи потрібно виводити в консоль прогрес виконання розрахунків.

Метод generate\_points приймає радіус iteration який потрібно згенерувати. Метод ітерує матрицю стільки разів, скільки потрібно для торкання границі, після чого повертає результат.

### БРОУНІВСЬКИЙ РУХ

Броунівський рух це рух частки у середовищі, причому частка значно більше за частинки середовища. Фрактал броунівського руху – шлях такої (таких) частинок.

На базі класу Figure створено клас-лекало для броунівського руху:

import numpy as np  
class BrownianMotion:  
 def \_\_init\_\_(self, size: int, \*args):  
 if args != ():  
 raise ValueError(f"Wrong number of arguments, {args} excess")  
 self.list\_to\_check = [int]  
 self.check\_args(size)  
  
 self.size = size  
  
 def check\_args(self, \*args):  
 *"""  
 Checks if parameters are correct  
  
 if not - raises ValueError with appropriate message  
 """* for argument\_index in range(len(args)):  
 if type(args[argument\_index]) is not self.list\_to\_check[argument\_index]:  
 raise ValueError(f"Wrong argument {args[argument\_index]}, which is {type(args[argument\_index])} type, expected {self.list\_to\_check[argument\_index]} type")  
  
 def generate\_points(self, iteration):  
 *"""  
 Generates dot of brownian motion on each iteration  
  
 # Returns:  
 (size, size) matrix that should be displayed as image (plt.imshow)  
 """* def checkAround(x, y, size\_limit):  
 *"""  
 Checks if there is friend or exit from circle around point (x, y), if not, chooses random way to go  
  
 # Returns:  
 x - x coordinate,  
 y - y coordinate,  
 """* coefs = [0, 1, 2, 3]  
 if x - 1 < 0:  
 coefs.remove(0)  
 if x + 1 > size\_limit - 1:  
 coefs.remove(1)  
 if y - 1 < 0:  
 coefs.remove(2)  
 if y + 1 > size\_limit - 1:  
 coefs.remove(3)  
 array = np.array([0, 1, 2, 3])  
 variant = np.random.choice(array[coefs], 1)  
 x, y = [(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)][variant[0]]  
  
 return x, y  
  
 matrix = np.zeros((self.size, self.size))  
 matrix\_for\_animation = []  
  
 np.random.seed()  
 x = np.random.randint(0, self.size)  
 y = np.random.randint(0, self.size)  
  
 for i in range(iteration):  
 matrix[x, y] = 1  
 if i in np.arange(0, iteration, self.size//3):  
 matrix\_for\_animation.append(matrix.copy())  
 x, y = checkAround(x, y, self.size)  
  
 matrix\_for\_animation.append(matrix.copy())  
 return matrix\_for\_animation

Метод \_\_init\_\_ приймає та зберігає розмір матриці відображення.

Метод generate\_points приймає номер ітерації iteration яку потрібно згенерувати. Метод ітерує матрицю iteration разів, після чого повертає результат.

### БАГАТОКУТНИЙ ФРАКТАЛ

Багатокутний фрактал, для ініціалізації якого береться довільний багатокутник (точки його вершин) та довільна точка в середині багатокутника. Під час ітерування створюється нова точка посередині між останньою точкою та випадковою іншою створеною точкою.

На базі класу Figure створено клас-лекало для багатокутного фракталу:

import numpy as np  
class NangularFractal:  
 *"""  
 Fractal that on start has n-dots, starts with random dot and goes on half of distance to randomly chosen dot given  
 """* def \_\_init\_\_(self, dots\_array: np.ndarray, \*args):  
 *"""  
 Initiates Nangular fractal with given parameters but before checks if parameters are correct  
  
 # Parameters:  
 dots\_array: np.ndarray (starting n-dots of [x, y])  
 """* if args != ():  
 raise ValueError(f"Wrong number of arguments, {args} excess")  
 self.list\_to\_check = [np.ndarray]  
 self.check\_args(dots\_array)  
  
 self.dots\_array = dots\_array  
 self.result = dots\_array.copy()  
  
 def check\_args(self, \*args):  
 *"""  
 Checks if parameters are correct  
  
 if not - raises ValueError with appropriate message  
 """* for argument\_index in range(len(args)):  
 if type(args[argument\_index]) is not self.list\_to\_check[argument\_index]:  
 raise ValueError(f"Wrong argument {args[argument\_index]}, which is {type(args[argument\_index])} type, expected {self.list\_to\_check[argument\_index]} type")  
  
 if args[0].shape[1] != 2:  
 raise ValueError(f"Wrong shape of array {args[0]}, which is {args[0].shape} shape, expected (n, 2)")  
  
 def generate\_points(self, iteration):  
 *"""  
 Generates `iteration` number of dots  
 # Returns:  
 (iteration + 4, 2) arrays of x and y coordinates  
 """* x = np.random.randint(min(self.dots\_array[:, 0]), max(self.dots\_array[:, 0]))  
 y = np.random.randint(min(self.dots\_array[:, 1]), max(self.dots\_array[:, 1]))  
  
 self.result = np.concatenate((self.result, np.array([[x, y]])))  
  
 for i in range(iteration):  
 random\_dot = self.dots\_array[np.random.choice(self.dots\_array.shape[0], 1)][0]  
 x = (x+random\_dot[0]) / 2  
 y = (y+random\_dot[1]) / 2  
 self.result = np.concatenate((self.result, np.array([[x, y]])))  
  
 return self.result[:, 0], self.result[:, 1]

Метод \_\_init\_\_ приймає початкові вершини багатокутника, перевіряє правильність параметрів та зберігає їх у змінні класу.

Метод generate\_points приймає номер ітерації iteration та виконує вищеописаний випадковий рух iteration разів.

### НЕФРАКТАЛЬНІ ФІГУРИ ТА КРИВІ

У цьому розділі описано створення нефрактальних фігур та кривих, таких як:

* Правильні багатокутники;
* Поліноміальні функції;
* Кардіоїди;
* Спіраль Архімеда;
* Фігури Ліссажу.

### ПРАВИЛЬНІ БАГАТОКУТНИКИ

Для генерації правильного багатокутника достатньо обрати рівновіддалені точки на колі. На базі класу Figure створено клас-лекало для правильного багатокутника:

import numpy as np  
class RegularPolygon:  
 *"""  
 Creates regular polygon with constant radius and variable number of verticies  
 """* def \_\_init\_\_(self, radius: float, fi: float, overdot: bool = True, \*args):  
 *"""  
 Initiates RegularPolygon with given radius but before checks if parameters are correct  
  
 # Parameters:  
 radius: float (Distance from center to verticies)  
 fi: float (Aangle offset)  
 overdot: bool (Add extra dot at the end same as the first dot )  
 """* if args != ():  
 raise ValueError(f"Wrong number of arguments, {args} excess")  
 self.list\_to\_check = [float, float, bool] *# Change if count of arguments changes* self.check\_args(radius, fi, overdot)  
  
 self.radius = radius  
 self.fi = fi  
 self.overdot = overdot  
  
 def check\_args(self, \*args):  
 *"""  
 Checks if parameters are correct  
  
 if not - raises ValueError with appropriate message  
 """* for argument\_index in range(len(args)):  
 if type(args[argument\_index]) is not self.list\_to\_check[argument\_index]:  
 raise ValueError(  
 f"Wrong argument {args[argument\_index]}, which is {type(args[argument\_index])} type, expected {self.list\_to\_check[argument\_index]} type")  
 if args[0] <= 0:  
 raise ValueError("Radius must be positive")  
 return 1  
  
 def generate\_points(self, iteration: int):  
 *"""  
 Returns regular polygon with 'iteration' vertices  
 """* if self.overdot:  
 x = np.zeros(iteration + 1)  
 y = np.zeros(iteration + 1)  
 else:  
 x = np.zeros(iteration)  
 y = np.zeros(iteration)  
 angle = self.fi % np.pi  
 d\_angle = np.pi \* 2 / iteration  
 for i in range(iteration):  
 x[i], y[i] = self.radius \* np.cos(angle), self.radius \* np.sin(angle)  
 angle += d\_angle  
 if self.overdot:  
 x[-1], y[-1] = x[0], y[0]  
 return x, y

Метод \_\_init\_\_ приймає радіус кола radius, зміщення початкової точки fi та чи потрібна додаткова початкова в кінці масиву точок overdot. Після цього виконується перевірка на відсутність зайвих параметрів та на правильність типів параметрів. Після успішного виконання перевірки, отримані параметри записуються як змінні класу. У разі неуспішного виконання перевірки, виконання коду зупиняється та викликається відповідна помилка.

Метод generate\_points приймає кількість вершин iteration яку потрібно згенерувати. Метод створює необхідну кількість вершин, та повертає масив координат вершин.

### ПОЛІНОМІАЛЬНІ ФУНКЦІЇ.

Поліноміальна функція – функція виду f(x) = a0 + a1x + a2x2 + …

На базі класу Figure створено клас-лекало для поліноміальної функції:

import numpy as np  
class DefaultPolynomialFunction:  
 *"""  
 Creates regular polygon with constant radius and variable number of verticies  
 """* def \_\_init\_\_(self, start: float, stop: float, polynom: list, \*args):  
 *"""  
 Initiates DefaultPolynomialFunction with range and polynom but before checks if parameters are correct  
  
 # Parameters:  
 start: float (Fitst x)  
 stop: float (Last x)  
 polynom: list ([a, b, c,...] Multipliers for x\*\*i where i is position in list)  
 """* if args != ():  
 raise ValueError(f"Wrong number of arguments, {args} excess")  
 self.list\_to\_check = [float, float, list] *# Change if count of arguments changes* self.check\_args(start, stop, polynom)  
  
 self.start = start  
 self.stop = stop  
 self.polynom = polynom  
  
 def check\_args(self, \*args):  
 *"""  
 Checks if parameters are correct  
  
 if not - raises ValueError with appropriate message  
 """* for argument\_index in range(len(args)):  
 if type(args[argument\_index]) is not self.list\_to\_check[argument\_index]:  
 raise ValueError(  
 f"Wrong argument {args[argument\_index]}, which is {type(args[argument\_index])} type, expected {self.list\_to\_check[argument\_index]} type")  
  
 def generate\_points(self, iteration):  
 *"""  
 Returns graph with '1/iteration' precision  
 """* x = np.array([x / iteration + self.start for x in range(int((self.stop - self.start) \* iteration))])  
 y = np.zeros(len(x))  
 for i in range(len(self.polynom)):  
 y += (x \*\* i) \* self.polynom[i]  
 return x, y

Метод \_\_init\_\_ приймає початок start та кінець stop інтервалу на якому буде обчислюватись функція та список коефіцієнтів полінома polynom. Після цього виконується перевірка на відсутність зайвих параметрів та на правильність типів параметрів. Після успішного виконання перевірки, отримані параметри записуються як змінні класу. У разі неуспішного виконання перевірки, виконання коду зупиняється та викликається відповідна помилка.

Метод generate\_points приймає кількість точок на одиницю iteration яку потрібно згенерувати. Метод створює необхідну кількість точок, обчислює значення ординат, та повертає масив з них.

### КАРДІОЇДИ

На базі класу Figure створено клас-лекало для кардіоїди:

import numpy as np  
class CardioidCurve:  
 *"""  
 Generate points for a Cardioid curve  
 """* def \_\_init\_\_(self, a: float):  
 *"""  
 Initialize the Cardioid curve with the given parameter, but before checks if parameters are correct  
  
 # Parameters:  
 a: float - the coefficient defining the size of the cardioid  
 """* self.check\_args(a)  
 self.a = a  
  
 def check\_args(self, \*args):  
 *"""  
 Checks if parameters are correct  
  
 if not - raises ValueError with appropriate message  
 """* expected\_types = [float]  
 if len(args) != 1:  
 raise ValueError(f"Expected 1 argument, but got {len(args)}")  
 if not isinstance(args[0], expected\_types[0]):  
 raise ValueError(f"Expected {expected\_types[0]} but got {type(args[0])} for argument {args[0]}")  
  
 def generate\_points(self, iteration):  
 *"""  
 Generate the points of the Cardioid curve.  
  
 # Returns:  
 A tuple of numpy arrays (x, y) representing the coordinates of the curve.  
 """* t = np.linspace(0, 2 \* np.pi, iteration)  
 x = self.a \* (1 - np.cos(t)) \* np.cos(t)  
 y = self.a \* (1 - np.cos(t)) \* np.sin(t)  
 return x, y

Метод \_\_init\_\_ приймає параметр a. Після цього виконується перевірка на відсутність зайвих параметрів та на правильність типів параметрів. Після успішного виконання перевірки, отримані параметри записуються як змінні класу. У разі неуспішного виконання перевірки, виконання коду зупиняється та викликається відповідна помилка.

Метод generate\_points приймає кількість точок iteration яку потрібно згенерувати. Метод створює необхідну кількість точок та повертає масив з них.

### СПІРАЛЬ АРХІМЕДА

Спіраль Архімеда — крива, яку описує точка M під час її рівномірного руху зі швидкістю v уздовж прямої, що рівномірно обертається у площині навколо однієї зі своїх точок О із кутовою швидкістю ω. Спіраль названо ім'ям Архімеда, який вивчав її властивості.

На базі класу Figure створено клас-лекало для спіралі Архімеда:

import numpy as np  
class ArchimedeanSpiral:  
 def \_\_init\_\_(self, a: float, b: float):  
 self.a = a  
 self.b = b  
  
 def check\_args(self, \*args):  
 expected\_types = [float, float]  
 if len(args) != len(expected\_types):  
 raise ValueError(f"Expected {len(expected\_types)} arguments, but got {len(args)}")  
 for arg, expected\_type in zip(args, expected\_types):  
 if not isinstance(arg, expected\_type):  
 raise ValueError(f"Expected {expected\_type} but got {type(arg)} for argument {arg}")  
  
 def generate\_points(self, iteration):  
 t = np.linspace(0, 10 \* np.pi, iteration)  
 r = self.a + self.b \* t  
 x = r \* np.cos(t)  
 y = r \* np.sin(t)  
 return x, y

Метод \_\_init\_\_ приймає початковий радіус спіралі a та швидкість збільшення радіусу b. Після цього виконується перевірка на відсутність зайвих параметрів та на правильність типів параметрів. Після успішного виконання перевірки, отримані параметри записуються як змінні класу. У разі неуспішного виконання перевірки, виконання коду зупиняється та викликається відповідна помилка.

Метод generate\_points приймає кількість точок iteration яку потрібно згенерувати. Метод створює необхідну кількість точок та повертає масив з них.

### ФІГУРИ ЛІССАЖУ

Фігури Ліссажу — замкнуті траєкторії, які прокреслюються точкою, що здійснює одночасно два гармонійних коливання у двох взаємно перпендикулярних напрямках. Вперше вивчені французьким науковцем Ж. Ліссажу.

На базі класу Figure створено клас-лекало для фігур Ліссажу:

import numpy as np  
class LissajousCurve:  
 *"""  
 Generate points for a Lissajous curve   
 """* def \_\_init\_\_(self, a: float, b: float, delta: float):  
 *"""  
 Initialize the Lissajous curve with given parameters, but before checks if parameters are correct  
  
 # Parameters:  
 a: float - the coefficient for the x component  
 b: float - the coefficient for the y component  
 delta: float - the phase shift for the x component  
 """* self.check\_args(a, b, delta)  
 self.a = a  
 self.b = b  
 self.delta = delta  
  
 def check\_args(self, \*args):  
 *"""  
 Checks if parameters are correct  
  
 if not - raises ValueError with appropriate message  
 """* a, b, delta = args  
 if not all(isinstance(arg, (int, float)) for arg in args):  
 raise ValueError("Arguments a, b, and delta must be numeric")  
   
 def generate\_points(self, iteration):  
 *"""  
 Generate the points of the Lissajous curve.  
  
 # Returns:  
 A tuple of numpy arrays (x, y) representing the coordinates of the curve  
 """* t = np.linspace(0, 2 \* np.pi, iteration)  
 x = self.a \* np.sin(self.a \* t + self.delta)  
 y = self.b \* np.sin(self.b \* t)  
 return x, y

Метод \_\_init\_\_ приймає коєфіцієнти частоти вздовж осі абцис a та вздовж осі ординат b. Після цього виконується перевірка на відсутність зайвих параметрів та на правильність типів параметрів. Після успішного виконання перевірки, отримані параметри записуються як змінні класу. У разі неуспішного виконання перевірки, виконання коду зупиняється та викликається відповідна помилка.

Метод generate\_points приймає кількість точок iteration яку потрібно згенерувати. Метод створює необхідну кількість точок та повертає масив з них.

### ВИКОРИСТАННЯ СТВОРЕНИХ КЛАСІВ-ЛЕКАЛ

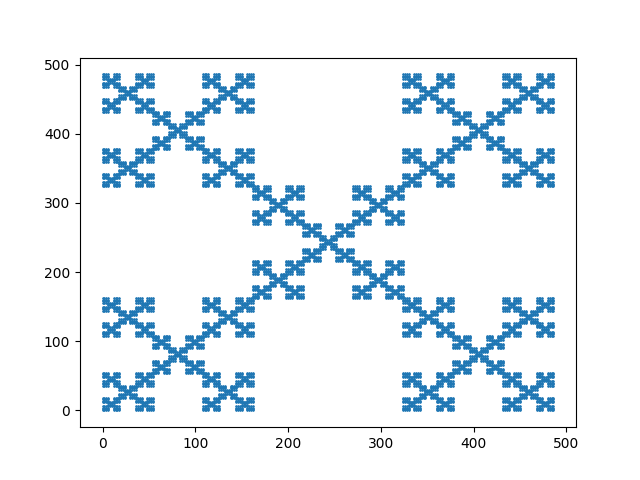
Для доступу до створених класів-лекал було доповнено клас App:

class App:  
 def \_\_init\_\_(self):  
 self.figures = {  
 *# Iteration Fractals* "Lfractal": LsystemFractal.LsystemFractal,  
 "Afractal": AffineFractal.AffineFractal,  
 "Mfractal": MatrixFractal.MatrixFractal,  
 *# Complex Fractals* "MorphingFractal": MorphingFractal.MorphingFractal,  
 *# Random Fractals* "BrownianTree": BrownianTree.BrownianTree,  
 "BrownianMotion": BrownianMotion.BrownianMotion,  
 "NangularFractal": NangularFractal.NangularFractal,  
 *# Non Fractals* "ReguralPolygon": RegularPolygon.RegularPolygon,  
 "DefaultPolynomialFunction": DefaultPolynomialFunction.DefaultPolynomialFunction,  
 "CardioidCurve": CardioidCurve.CardioidCurve,  
 "ArchimedeanSpiral": ArchimedeanSpiral.ArchimedeanSpiral,  
 "LissajousCurve": LissajousCurve.LissajousCurve,   
 }  
  
 def create\_figure(self, name, \*args, \*\*kwargs):  
 if name in self.figures:  
 FigureDirector().build(self.figures[name](\*args), \*\*kwargs)  
 else:  
 raise ValueError("Wrong fractal name")

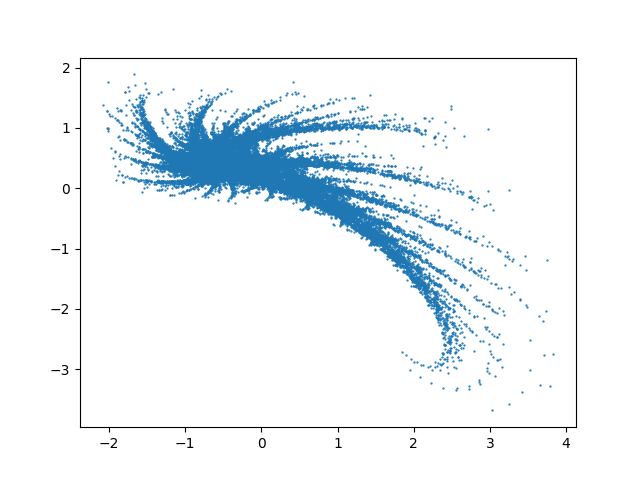
Для того щоб вивести приклади на екран, було створено наступний код:

if \_\_name\_\_ == "\_\_main\_\_":  
 app = App()  
  
 app.create\_figure("Lfractal", "F+F+F+F", {"F": "F+F-F-F+F"}, 0., np.pi/2,  
 it=5, animation\_need=True, animation\_save=False, multi=200, has\_background=False, has\_axes=False)  
 app.create\_figure("Afractal", [  
 [1.0, -0.1],  
 [0.2, -1.0],  
 [-0.3, 0.4],  
 [0.7, 0.2],  
 [0.5, -0.4],  
 [-0.2, 0.5]  
 ],  
 it=4\*10\*\*4, animation\_need=True, animation\_save=False, multi=200, has\_background=False, has\_axes=False)  
  
 app.create\_figure("Afractal", [  
 [0.0500, 0.0500, 0.6000, 0.5000, 0.5000, 0.5500],  
 [0.6000, -0.5000, 0.5000, 0.4500, 0.5500, 0.4000],  
 [0.0000, 0.0000, 0.6980, 0.3490, -0.5240, -0.6980],  
 [0.0000, 0.0000, 0.6980, 0.3492, -0.5240, -0.6980],  
 [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],  
 [0.0000, 1.0000, 0.6000, 1.1000, 1.0000, 0.7000],  
 ], 0, False,  
 it=4\*10\*\*4, animation\_need=True, animation\_save=False, multi=200, has\_background=False, has\_axes=False)  
  
 app.create\_figure("Mfractal", np.array([[0, 1, 1], [1, 0, 1], [1, 1, 0]]),  
 it=8, animation\_need=True, animation\_save=False, has\_background=False, has\_axes=False)  
  
 app.create\_figure("BrownianTree", False,  
 it=50, animation\_need=True, animation\_save=False, has\_background=False, has\_axes=False)  
 app.create\_figure("BrownianMotion", 300,  
 it=30000, animation\_need=True, animation\_save=False, has\_background=False, has\_axes=False)  
 app.create\_figure("NangularFractal", np.array([  
 [0, 0],  
 [1, 5],  
 [2, -1],  
 ]),  
 it=4\*10\*\*4, animation\_need=True, animation\_save=False, multi=200, has\_background=False, has\_axes=False)  
 app.create\_figure("NangularFractal", np.array([  
 [1, 0],  
 [3, 0],  
 [1, 5],  
 [5, 5],  
 ]),  
 it=4\*10\*\*4, animation\_need=True, animation\_save=False, multi=200, has\_background=False, has\_axes=False)  
  
 app.create\_figure("ReguralPolygon", 3., 0.5,  
 it=12, animation\_need=True, animation\_save=False, multi=1, fps=10, is\_edge=False, is\_fixed\_size=True, has\_background=False, has\_axes=False)  
 app.create\_figure("DefaultPolynomialFunction", -10.0, 10.0, [0, 0, 0, 1],  
 it=1000, animation\_need=True, animation\_save=False, multi=200, is\_fixed\_size=True, has\_background=False, has\_axes=False)  
  
 app.create\_figure("ArchimedeanSpiral", 0.5, 0.2,  
 it=4000, animation\_need=True, animation\_save=False, multi=70, is\_fixed\_size=True, has\_background=False, has\_axes=False)  
 app.create\_figure("CardioidCurve", 1.0,  
 it=4000, animation\_need=True, animation\_save=False, multi=20, is\_fixed\_size=True, has\_background=False, has\_axes=False)  
 app.create\_figure("LissajousCurve", 1.0, 2.0, np.pi/2,  
 it=4000, animation\_need=True, animation\_save=False, multi=70, is\_fixed\_size=True, has\_background=False, has\_axes=False)  
 app.create\_figure("LissajousCurve", 3.0, 2.0, np.pi/2,  
 it=4000, animation\_need=True, animation\_save=False, multi=70, is\_fixed\_size=True, has\_background=False, has\_axes=False)  
 app.create\_figure("LissajousCurve", 3.0, 4.0, np.pi/2,  
 it=4000, animation\_need=True, animation\_save=False, multi=70, is\_fixed\_size=True, has\_background=False, has\_axes=False)  
 app.create\_figure("LissajousCurve", 5.0, 4.0, np.pi/2,  
 it=4000, animation\_need=True, animation\_save=False, multi=70, is\_fixed\_size=True, has\_background=False, has\_axes=False)  
  
 app.create\_figure("MorphingFractal", "Julia", complex(-0.4, 0.6), 2.0, 1000, 1000,  
 it=100, animation\_need=True, animation\_save=False, cmap='inferno', has\_background=False, has\_axes=False)  
 app.create\_figure("MorphingFractal", "Julia", complex(0.4, 0.4), 2.0, 1000, 1000,  
 it=100, animation\_need=True, animation\_save=False, cmap='inferno', has\_background=False, has\_axes=False)  
 app.create\_figure("MorphingFractal", "Mandelbrot", 100, 2.0, 1000, 1000,  
 it=100, animation\_need=True, animation\_save=False, cmap='inferno', has\_background=False, has\_axes=False)  
 app.create\_figure("MorphingFractal", "Multibrot", 3, 2.0, 1000, 1000,  
 it=100, animation\_need=True, animation\_save=False, cmap='inferno', has\_background=False, has\_axes=False)  
 app.create\_figure("MorphingFractal", "BurningShip", 100, 2.0, 1000, 1000,  
 it=100, animation\_need=True, animation\_save=False, cmap='inferno', has\_background=False, has\_axes=False)  
 app.create\_figure("MorphingFractal", "Tricorn", 100, 2.0, 1000, 1000,  
 it=100, animation\_need=True, animation\_save=False, cmap='inferno', has\_background=False, has\_axes=False)  
 app.create\_figure("MorphingFractal", "SinJulia", complex(-0.4, 0.6), 2.0, 1000, 1000,  
 it=100, animation\_need=True, animation\_save=False, cmap='inferno', has\_background=False, has\_axes=False)  
 app.create\_figure('MorphingFractal', 'HyperbolicTangent', complex(0.1, 0.1), 2.0, 1000, 1000,  
 it=100, animation\_need=True, animation\_save=False, cmap='inferno', has\_background=False, has\_axes=False)

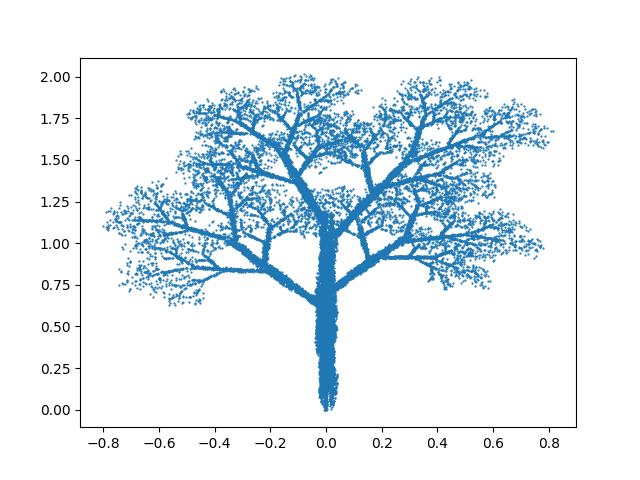
# ПРИКЛАДИ РОБОТИ СТВОРЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Накидати картинок, підписати як в 3.4.1, тобто 4.1, 4.2, 4.3… 

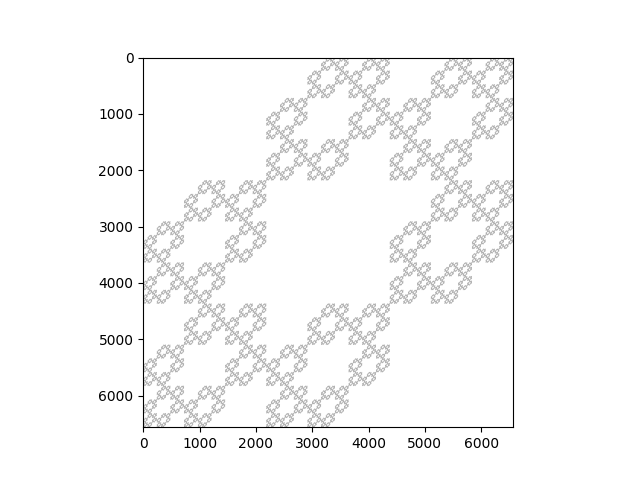
Мал. 4.1. L-системний фрактал, схожий на вишивку.



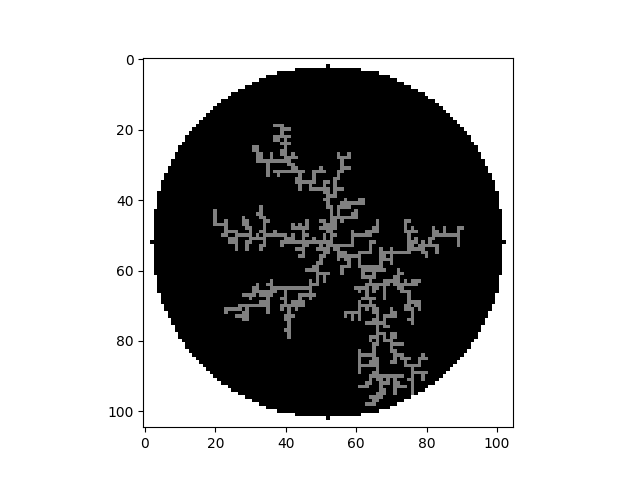
Мал. 4.2. Афінний фрактал, схожий на комету.



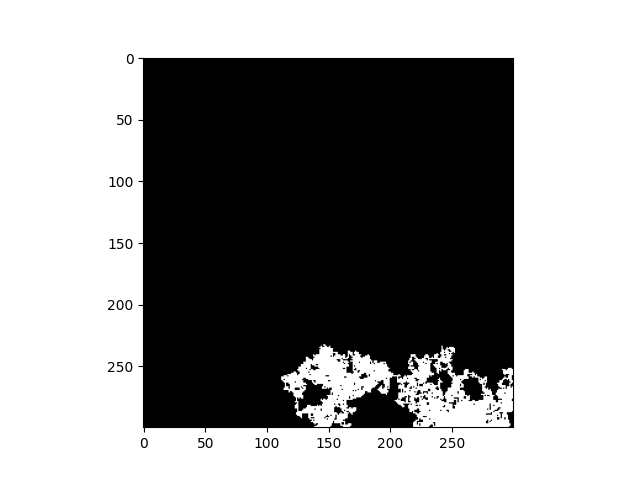
Мал. 4.3. Афінний фрактал, другий варіант фрактального дерева.



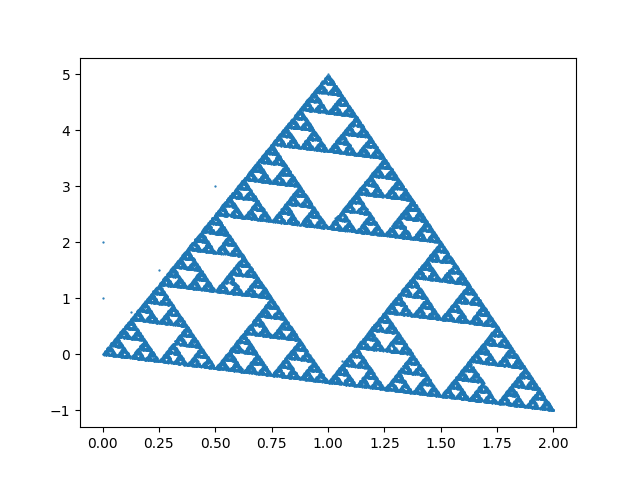
Мал. 4.4. Двовимірна множини кантора, килим Серпинського



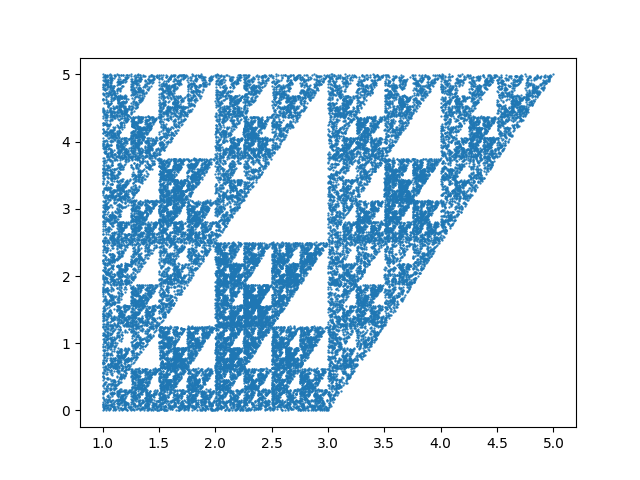
Мал. 4.5. Випадковий фрактал, Броунівське дерево.



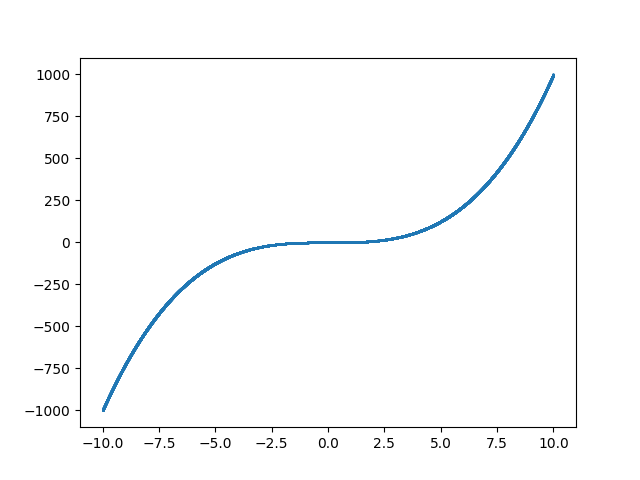
Мал. 4.6. Випадковий фрактал, Броунівський рух.



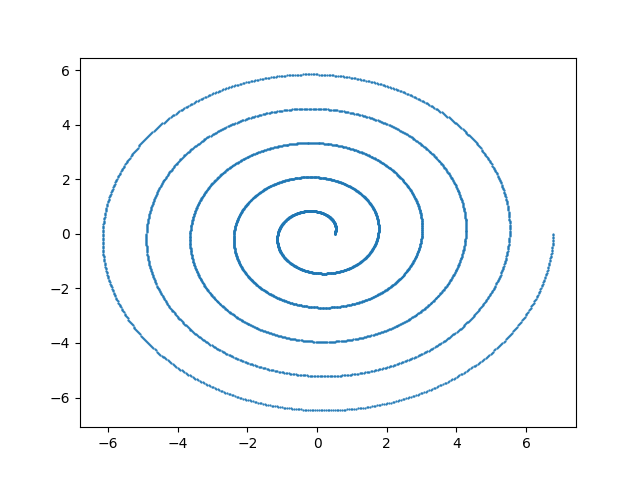
Мал. 4.7. Випадковий фрактал, трикутник Серпінського.



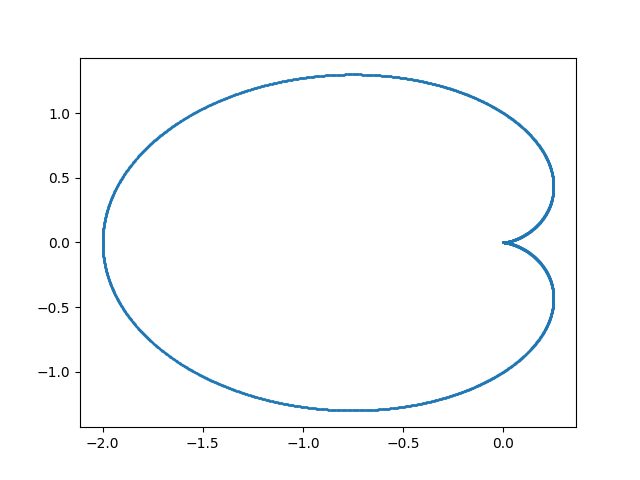
Мал. 4.8. Випадковий фрактал, трапеція, яка виявилась фракталом з трьох трикутників.



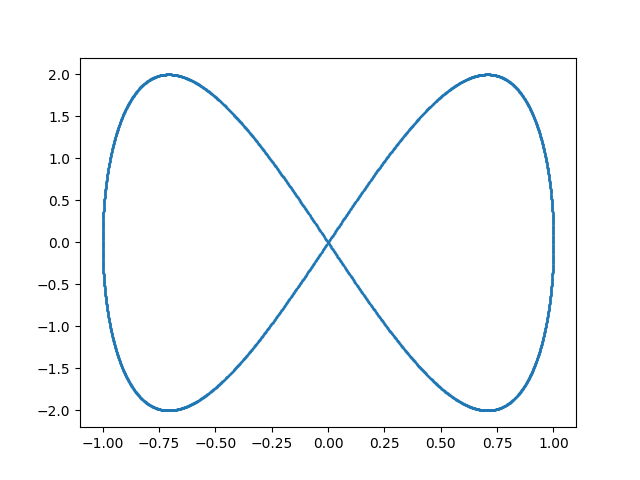
Мал. 4.9. Поліноміальна функція, x^3 на проміжку [-10, 10].



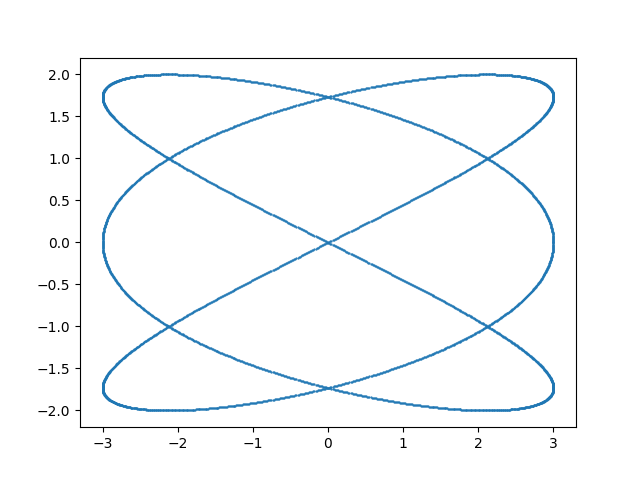
Мал. 4.10. Спіраль архімеда.



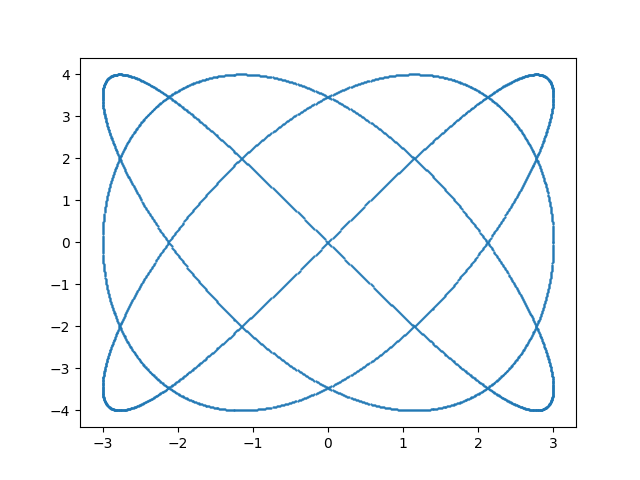
Мал. 4.11. Кардіоїда.



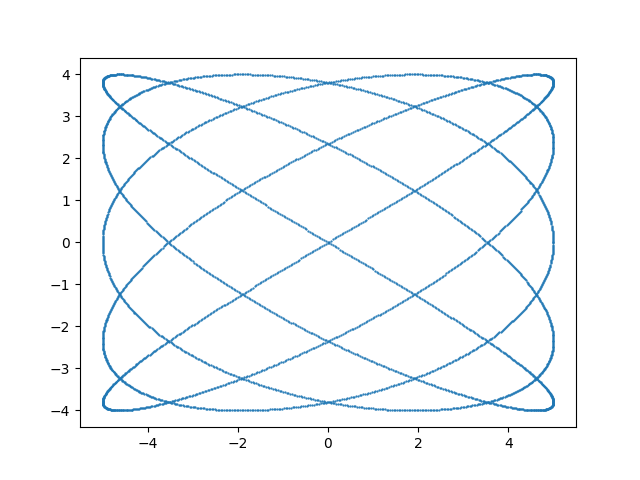
Мал. 4.12. Фігура ліссажу, a = 1, b = 2, delta = pi/2.



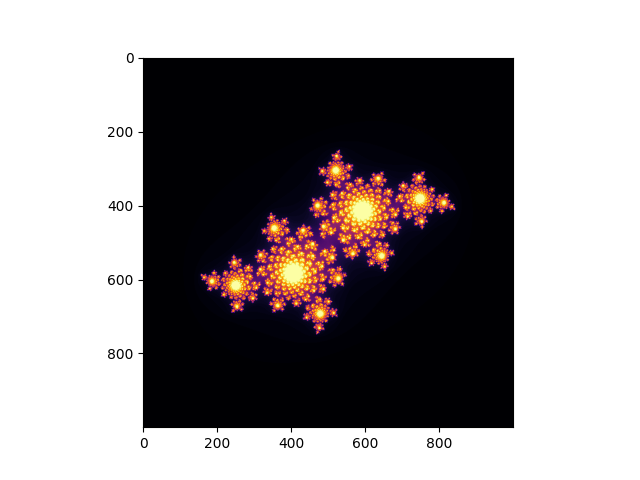
Мал. 4.13. Фігура ліссажу, a = 3, b = 2, delta = pi/2.



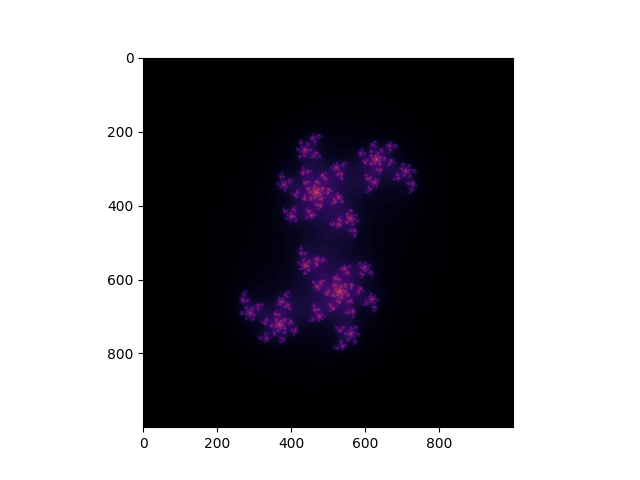
Мал. 4.14. Фігура ліссажу, a = 3, b = 4, delta = pi/2.



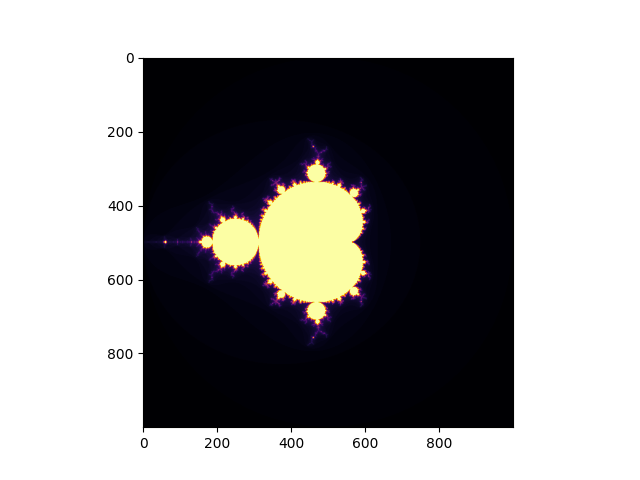
Мал. 4.15. Фігура ліссажу, a = 5, b = 4, delta = pi/2.



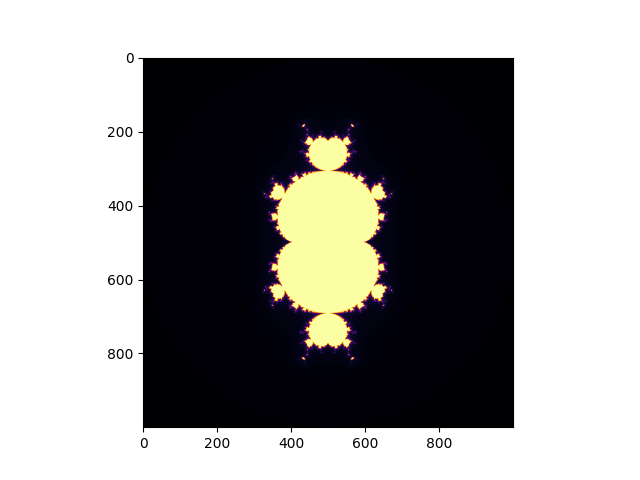
Мал. 4.16. Комплексний фрактал, множина Жуліа при С = -0.4 + i\*0.6.



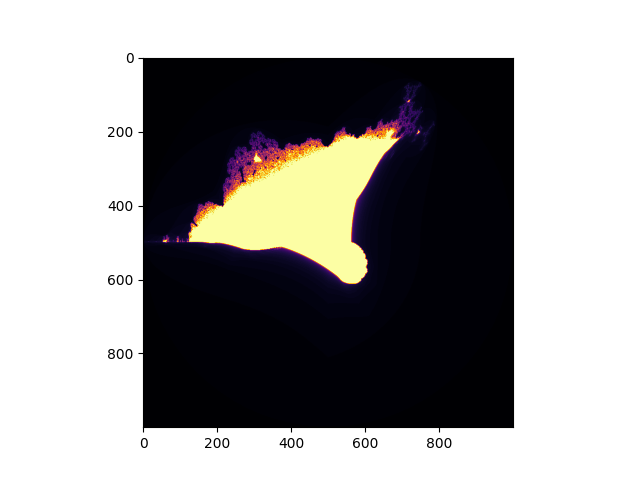
Мал. 4.17. Комплексний фрактал, множина Жуліа при С = 0.4 + i\*0.4.



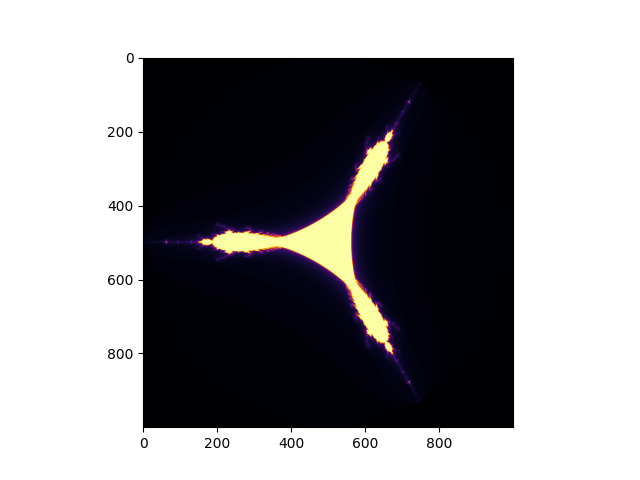
Мал. 4.18. Комплексний фрактал, Мандельброт.



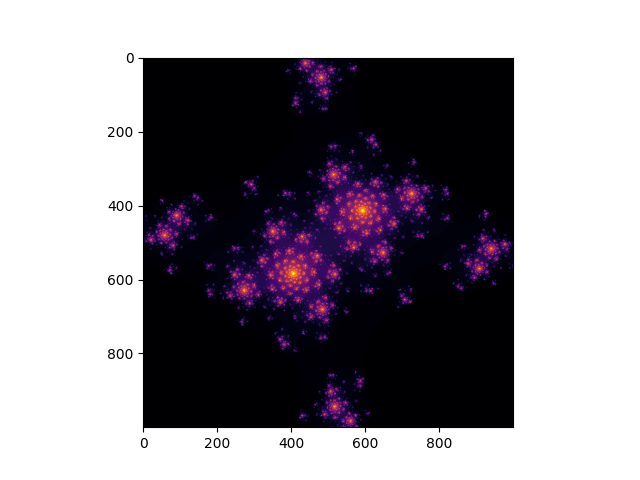
Мал. 4.19. Комплексний фрактал, Мультиброт.



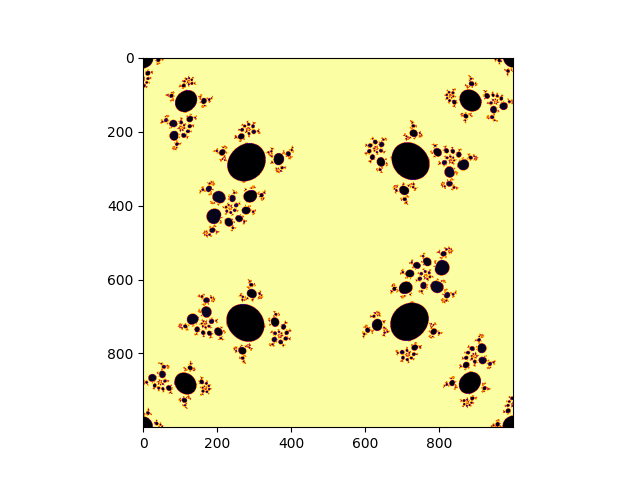
Мал. 4.20. Комплексний фрактал, Палаючий корабель.



Мал. 4.21. Комплексний фрактал, трикорн.



Мал. 4.22. Комплексний фрактал, синусоїдна множина Жуліа.



Мал. 4.23. Комплексний фрактал, гіперболічний тангенс.

# ВИСНОВКИ

* Створити уніфіковану систему генерації та виводу фігур на екран можливо. При цьому додаткові параметри лише покращують результат виводу, але не є обов’язковими.
* Всі фігури можна представити у вигляді масиву точок з певною точністю. Для багатокутників така точність є абсолютною, а для кривих та фракталів її можна нескінченно наближати до абсолютної.