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

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.

2. 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.

3. ХІД РОБОТИ

3.1. 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.

3.2.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.

3.2.1. 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.

3.2.2. 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.

3.2.3. 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.

3.3.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:

```
import numpy as np
class MorphingFractal:
    """
    Generate points for complex fractals
    """

    def __init__(self, fractal_type: str, parameter: complex or
int, threshold: float = 2.0, width: int = 1000, height: int =
1000):
        """
        Initialize the fractal with the given parameters,  but
before checks if parameters are correct

        # Parameters:
        fractal_type: str - the type of fractal ('Julia',
'Mandelbrot', 'Multibrot', "BurningShip", "Tricorn", "SinJulia",
"HyperbolicTangent").
        parameter: complex or int - the parameter for the fractal
(complex for Julia, int for Multibrot exponent).
        max_iterations: int - the maximum number of iterations.
        threshold: float - the threshold for fractal calculation.
        width: int - the width of the generated fractal image.
        height: int - the height of the generated fractal image.
        """
        self.check_args(fractal_type, parameter, threshold, width,
height)
        self.fractal_type = fractal_type
        self.c = parameter if fractal_type in ["Julia",
"SinJulia","HyperbolicTangent"] else None
        self.exponent = parameter if fractal_type == "Multibrot"
else 2
        self.threshold = threshold
        self.width = width
        self.height = height

    def check_args(self, *args):
        """
        Checks if parameters are correct

        if not - raises ValueError with appropriate message
        """

        fractal_type, parameter, threshold, width, height = args

        valid_fractal_types = ["Julia", "Mandelbrot", "Multibrot",
"BurningShip", "Tricorn", "SinJulia", "HyperbolicTangent"]
        if fractal_type not in valid_fractal_types:
            raise ValueError(f"Invalid fractal type
'{fractal_type}'. Valid types are: {valid_fractal_types}")
        if not isinstance(fractal_type, str):
            raise TypeError("fractal_type must be a string")
```

```
        if not isinstance(parameter, (complex, int)):
            raise TypeError("parameter must be a complex number or
an integer")

        if not isinstance(threshold, float):
            raise TypeError("threshold must be a float")

        if not isinstance(width, int) or width <= 0:
            raise ValueError(f"width must be a positive integer")

        if not isinstance(height, int) or height <= 0:
            raise ValueError(f"height must be a positive integer")

    def generate_points(self, iteration):
        """
        Generate the points of the fractal

        # Returns:
        A numpy array representing the iterations for each point
in the fractal
        """
        x_min, x_max = -2, 2
        y_min, y_max = -2, 2

        x = np.linspace(x_min, x_max, self.width)
        y = np.linspace(y_min, y_max, self.height)

        X, Y = np.meshgrid(x, y)
        C = X + 1j * Y

        if self.fractal_type in ["Julia", "SinJulia",
"HyperbolicTangent"]:
            Z = C
            c = self.c
        else:
            Z = np.zeros(C.shape, dtype=complex)
            c = C

        result = np.zeros(Z.shape, dtype=int)
        mask = np.ones(Z.shape, dtype=bool)

        results = []

        for _ in range(iteration):
            if self.fractal_type == "Julia":
                Z[mask] = Z[mask] ** 2 + c
            elif self.fractal_type == "Mandelbrot":
                Z[mask] = Z[mask] ** 2 + C[mask]
            elif self.fractal_type == "Multibrot":
                Z[mask] = Z[mask] ** self.exponent + C[mask]
            elif self.fractal_type == "BurningShip":
                Z[mask] = (np.abs(Z[mask].real) + 1j *
```

```
np.abs(Z[mask].imag)) ** 2 + C[mask]
            elif self.fractal_type == "Tricorn":
                Z[mask] = np.conj(Z[mask] ** 2) + C[mask]
            elif self.fractal_type == "SinJulia":
                Z[mask] = np.sin(Z[mask] ** 2) + c
            elif self.fractal_type == "HyperbolicTangent":
                Z[mask] = np.tanh(Z[mask] ** 2) + c

            mask = np.abs(Z) < self.threshold
            result[mask] += 1
            results.append(result.copy())

        return results
```

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.

3.4.ВИПАДКОВІ ФРАКТАЛИ

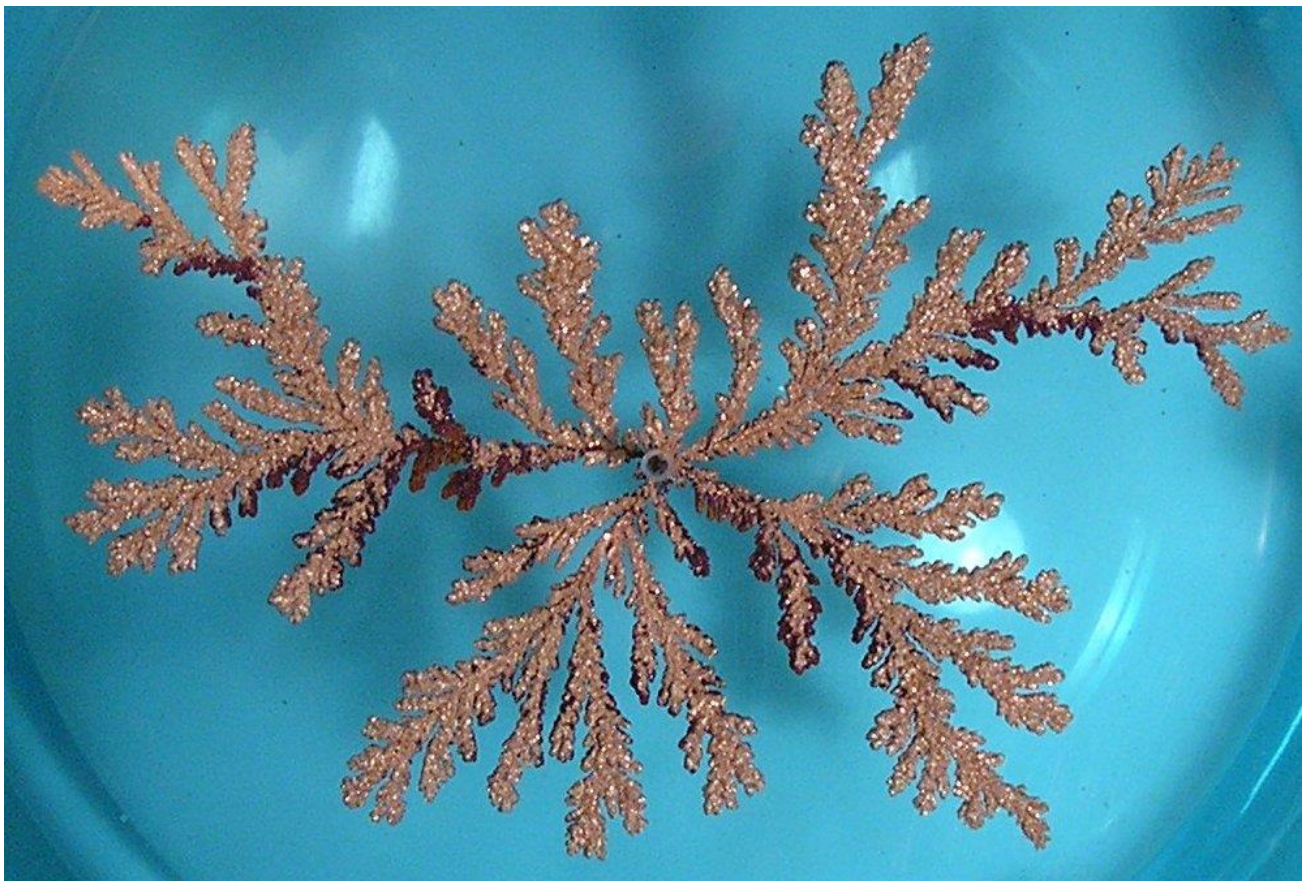
Випадковий (стохастичний) фрактал – фрактал що задається випадковим (стохастичним або псевдовипадковим) процесом. Кожна ітерація фракталу наближає фрактал до точної передачі стохастичного процесу, при чому наблизитись до абсолютно точної передачі не можливо.

Серед випадкових фракталів обрано:

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

3.4.1. 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.

3.4.2. 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.

3.4.3. 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.

3.5.НЕФРАКТАЛЬНІ ФІГУРИ ТА КРИВІ

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

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

3.5.1. 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.

3.5.2. ПОЛІНОМІАЛЬНІ ФУНКЦІЇ.

Поліноміальна функція – функція виду $f(x) = a_0 + a_1x + a_2x^2 + \dots$

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.

3.5.3. 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.

3.5.4. 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 O 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.

3.5.5. 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.

3.6.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:

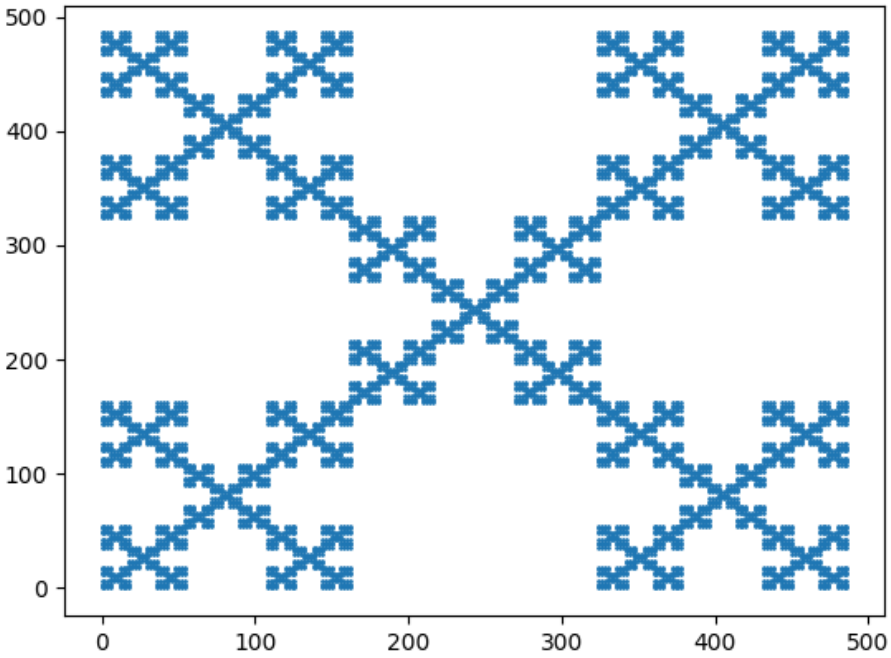
```
if __name__ == "__main__":
    app = App()

    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,
```

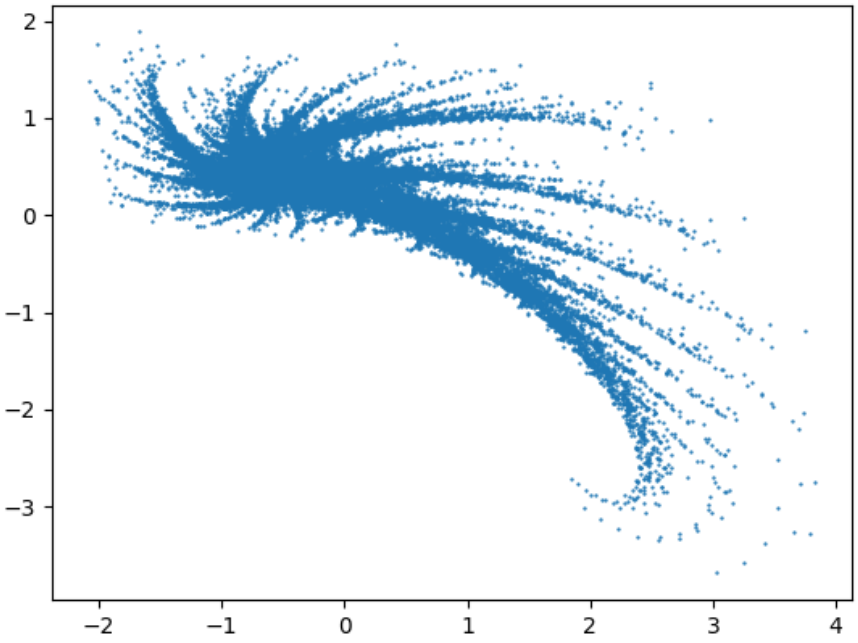
```
animation_save=False, multi=1, fps=10, is_edge=False,
is_fixed_size=True, has_background=False, has_axes=False)
    app.create_figure("DefaultPolynomialFunction", -10.0, 10.0,
[0, 0, 0, 1],
                      it=1000, animation_need=True,
animation_save=False, multi=200, is_fixed_size=True,
has_background=False, 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)
```

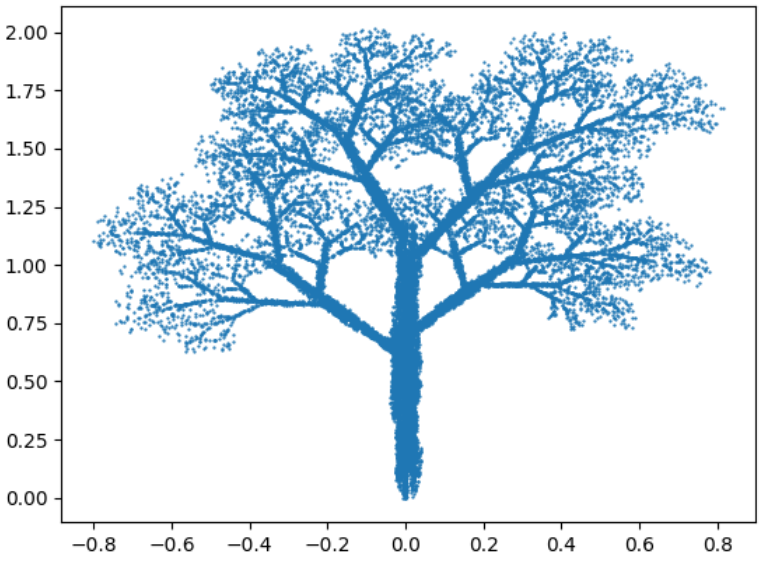
4. ПРИКЛАДИ РОБОТИ СТВОРЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



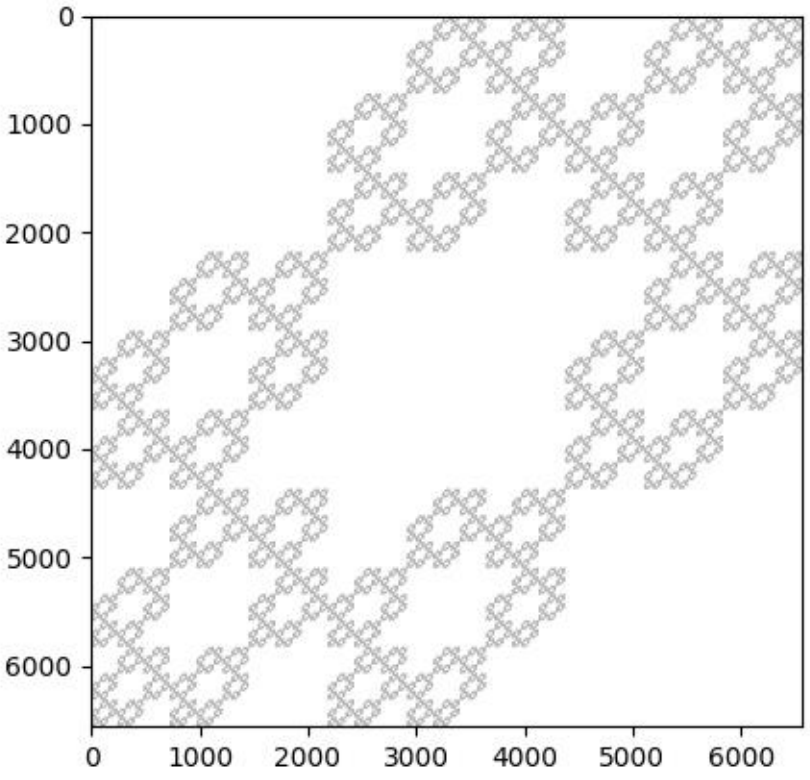
Мал. 4.1. L-системний фрактал, схожий на вишивку.



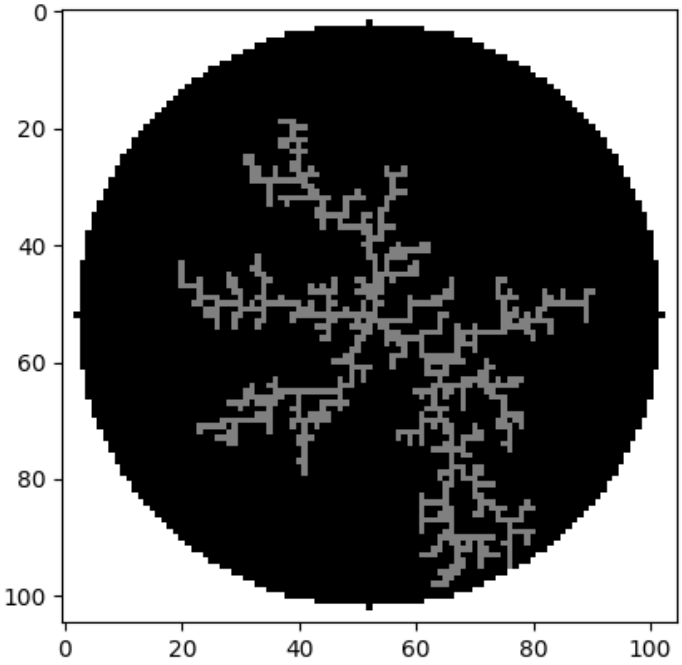
Мал. 4.2. Афінний фрактал, схожий на комету.



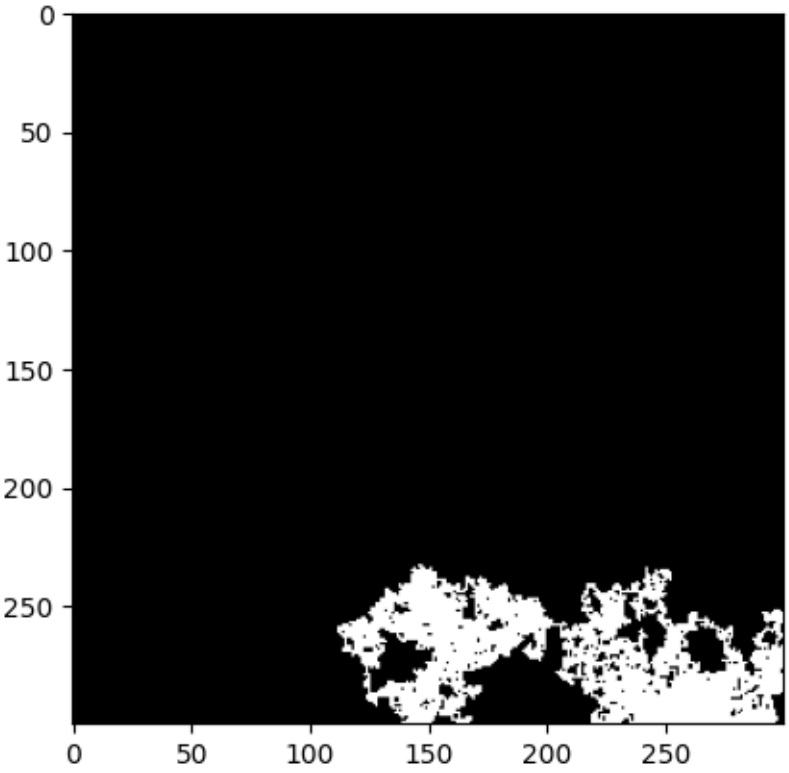
Мал. 4.3. Афінний фрактал, другий варіант фрактального дерева.



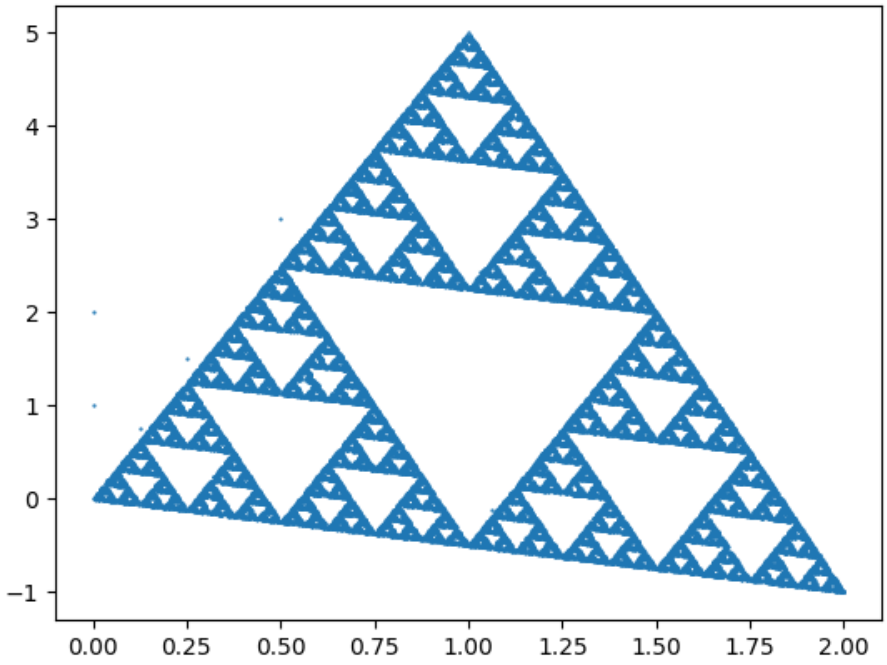
Мал. 4.4. Двовимірна множини кантора, килим Серпинського



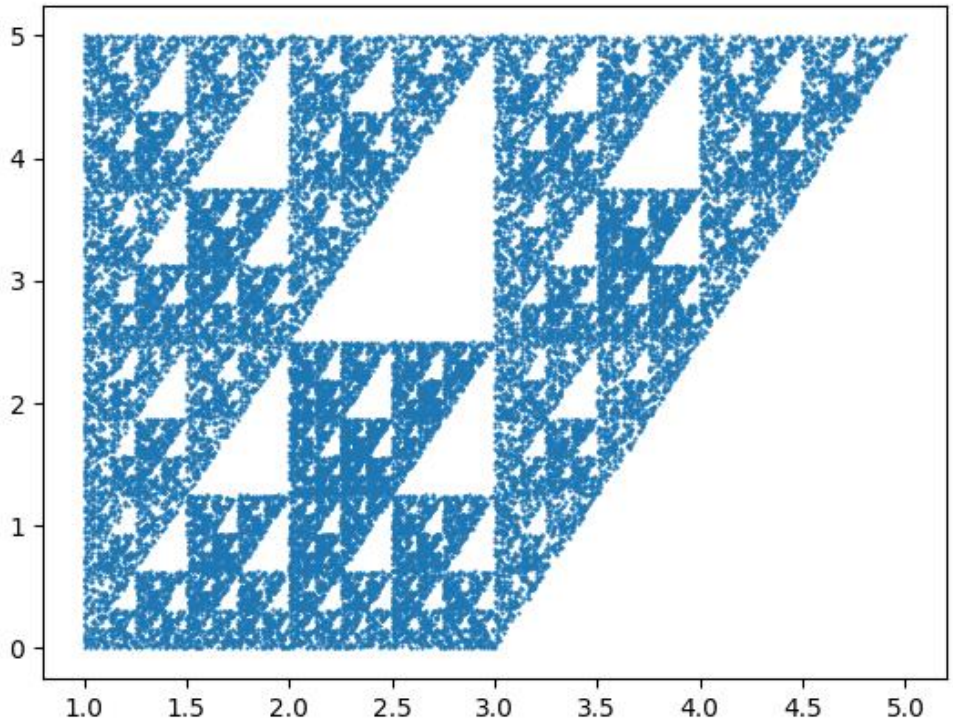
Мал. 4.5. Випадковий фрактал, Броунівське дерево.



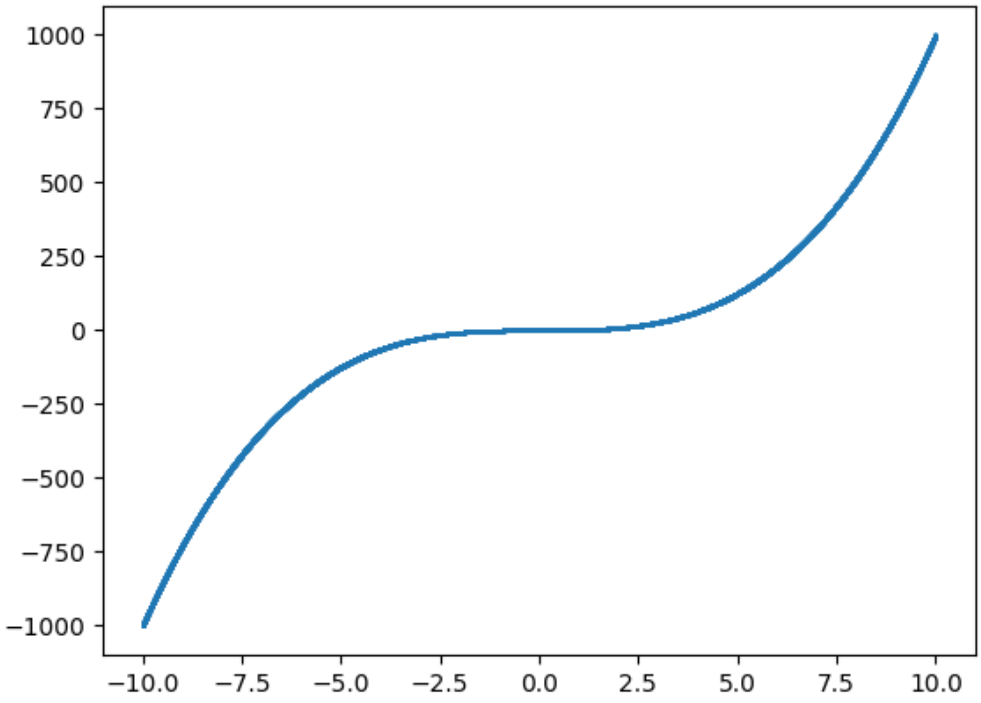
Мал. 4.6. Випадковий фрактал, Броунівський рух.



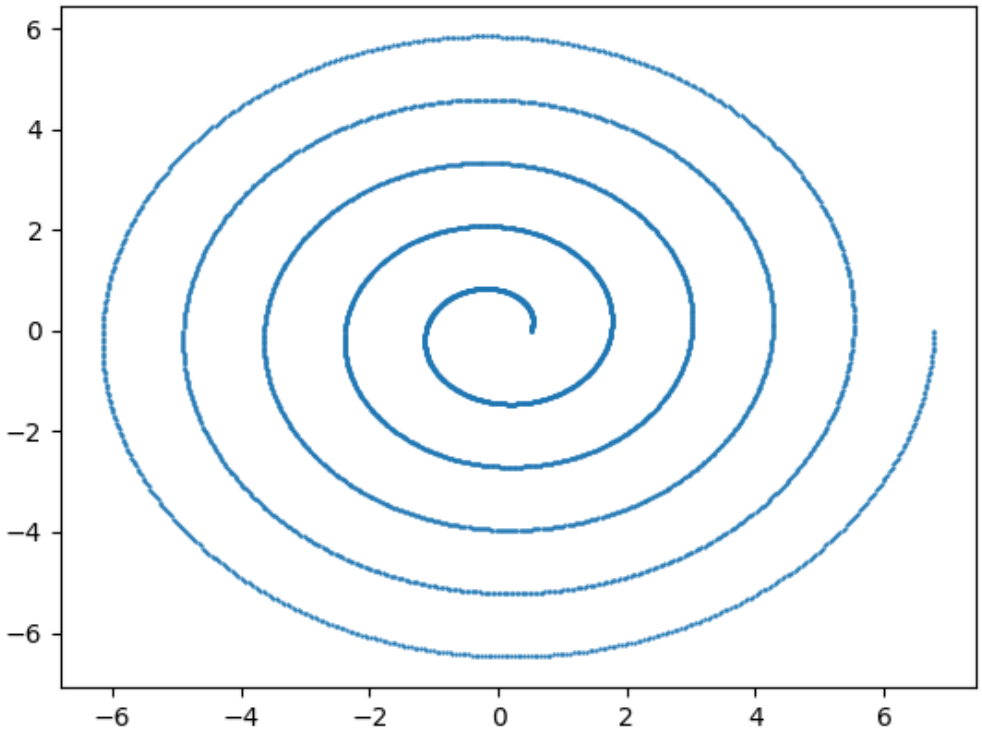
Мал. 4.7. Випадковий фрактал, трикутник Серпінського.



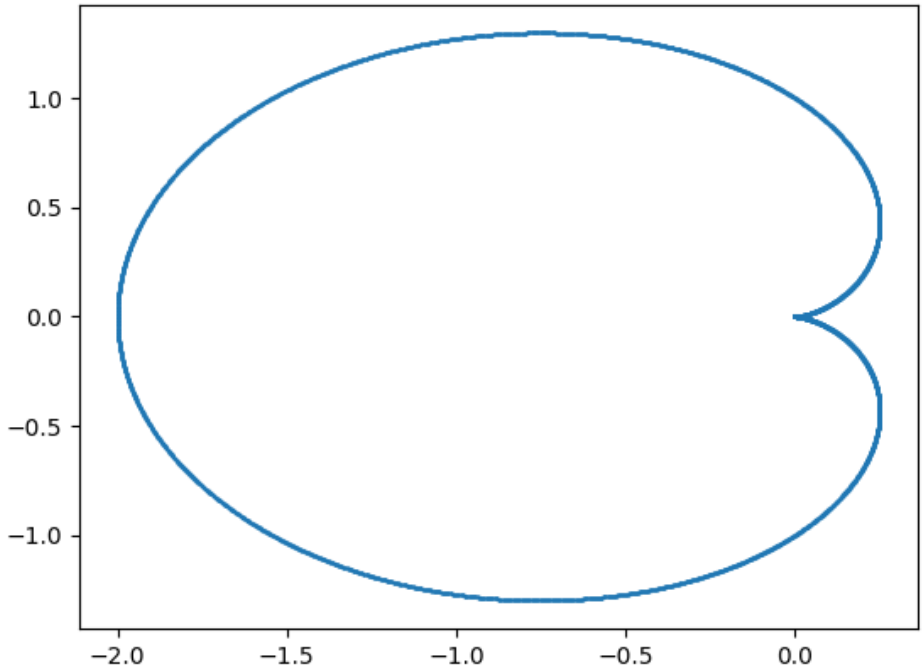
Мал. 4.8. Випадковий фрактал, трапеція, яка виявилась фракталом з трьох трикутників.



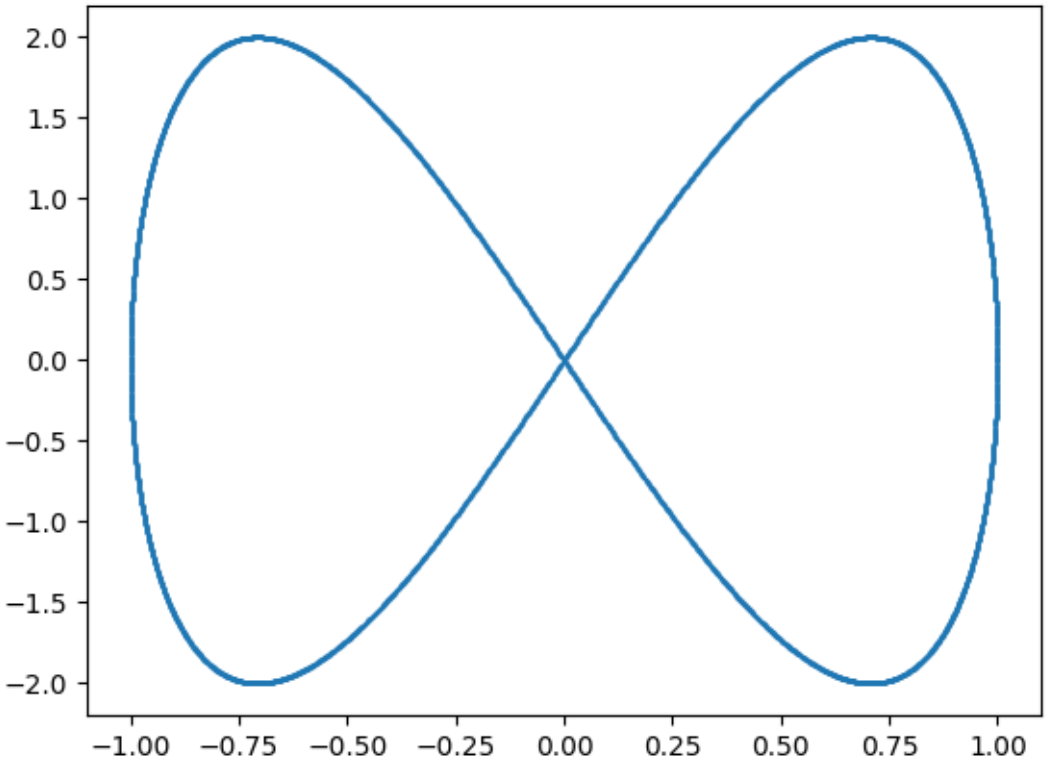
Мал. 4.9. Поліноміальна функція, x^3 на проміжку [-10, 10].



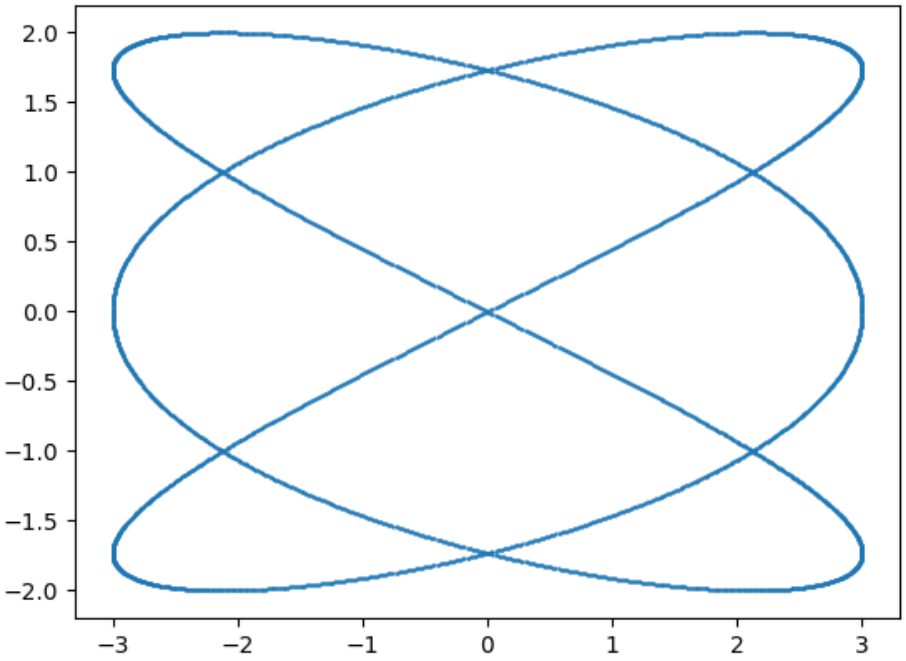
Мал. 4.10. Спіраль архімеда.



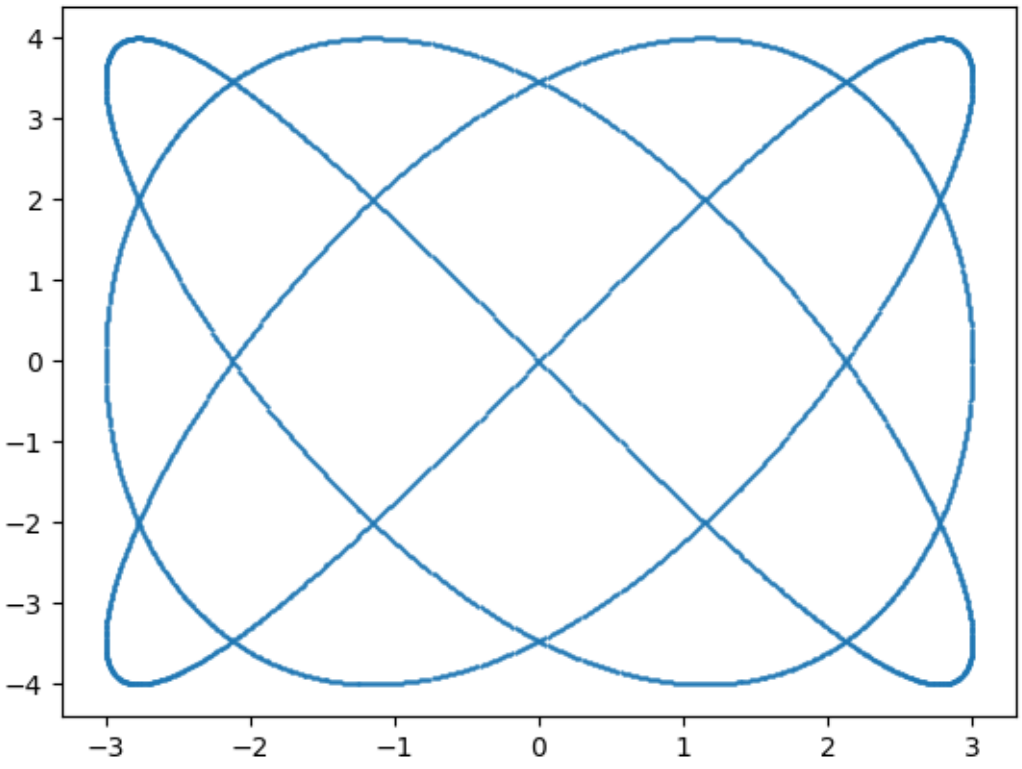
Мал. 4.11. Кардіоїда.



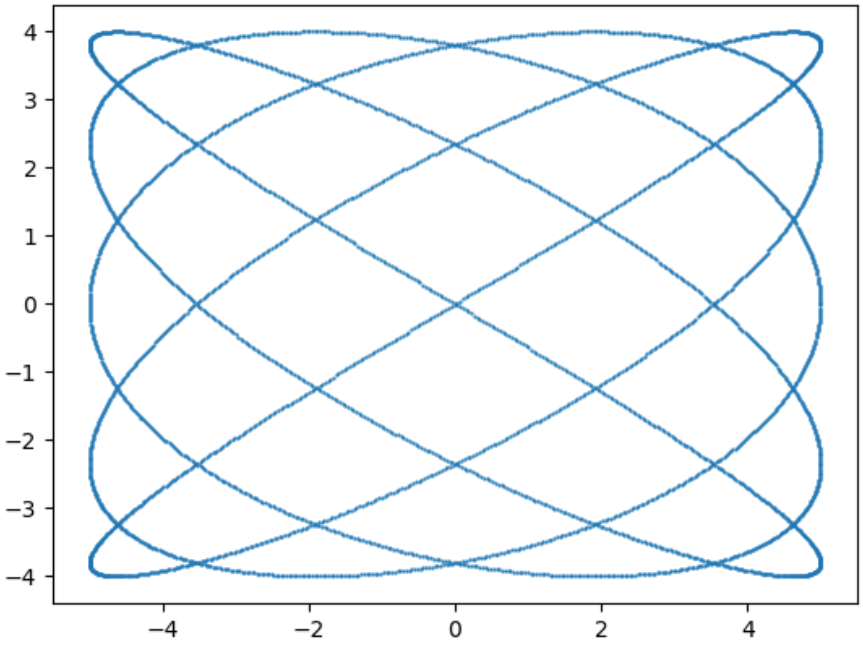
Мал. 4.12. Фігура ліссажу, a = 1, b = 2, delta = pi/2.



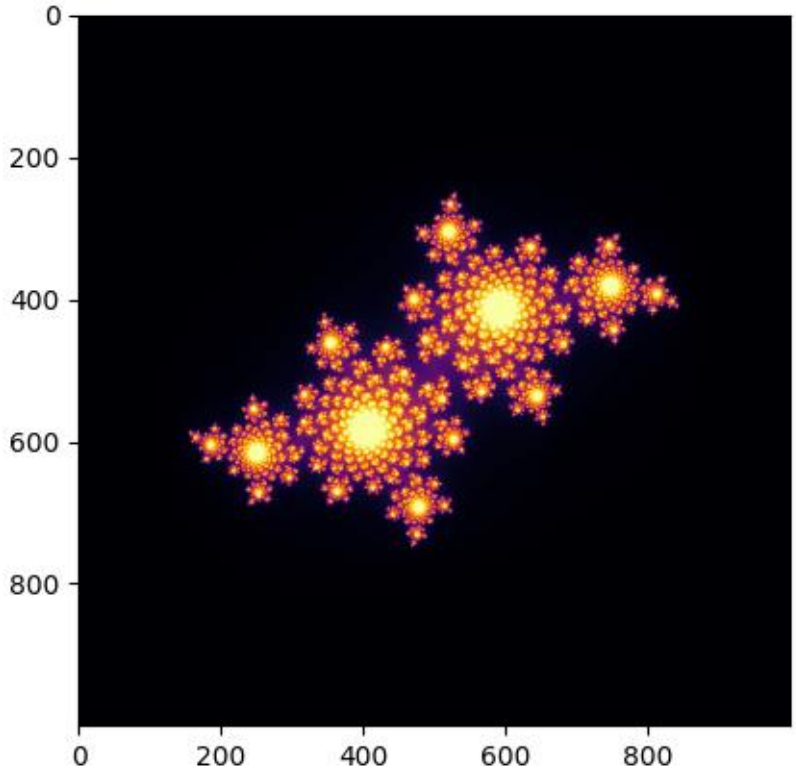
Мал. 4.13. Фігура ліссажу, a = 3, b = 2, delta = pi/2.



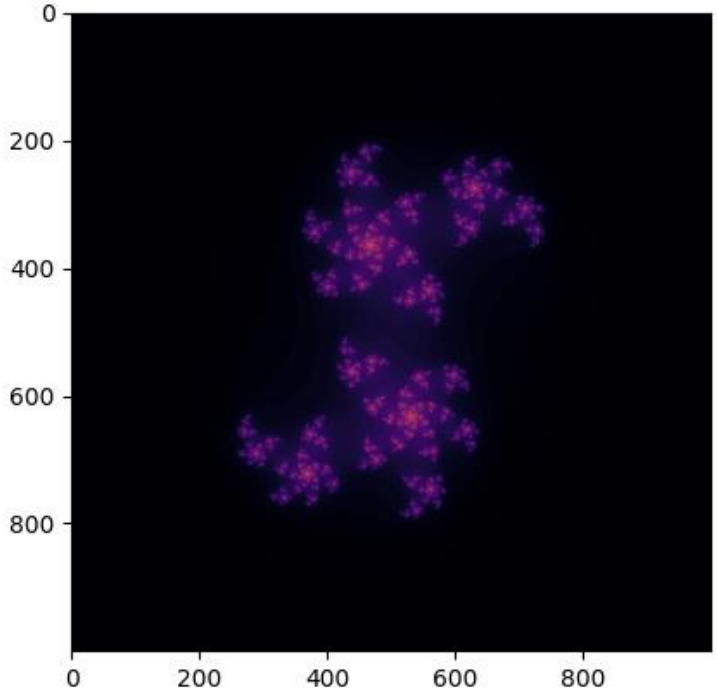
Мал. 4.14. Фігура ліссажу, a = 3, b = 4, delta = pi/2.



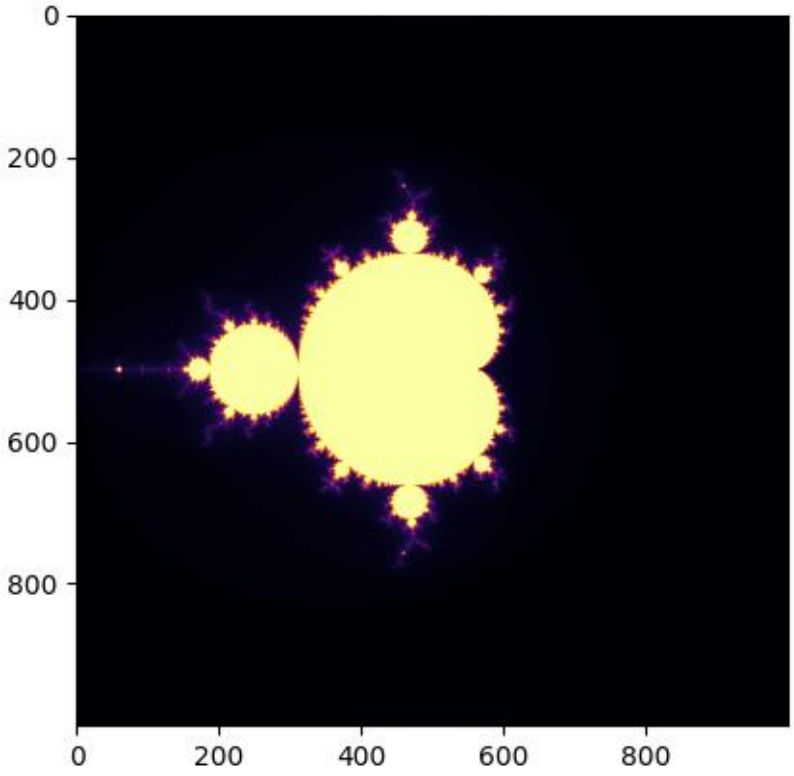
Мал. 4.15. Фігура ліссажу, a = 5, b = 4, delta = pi/2.



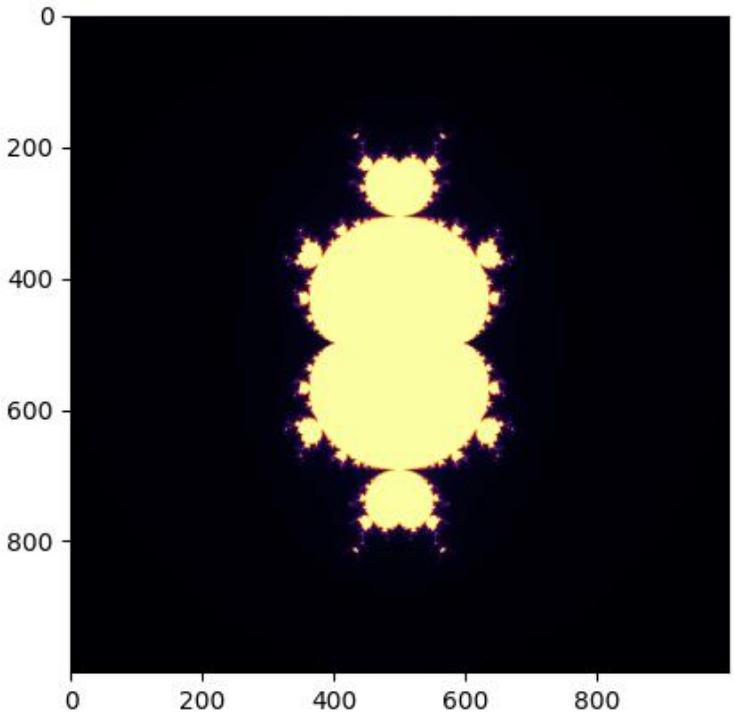
Мал. 4.16. Комплексний фрактал, множина Жуліа при C = -0.4 + i*0.6.



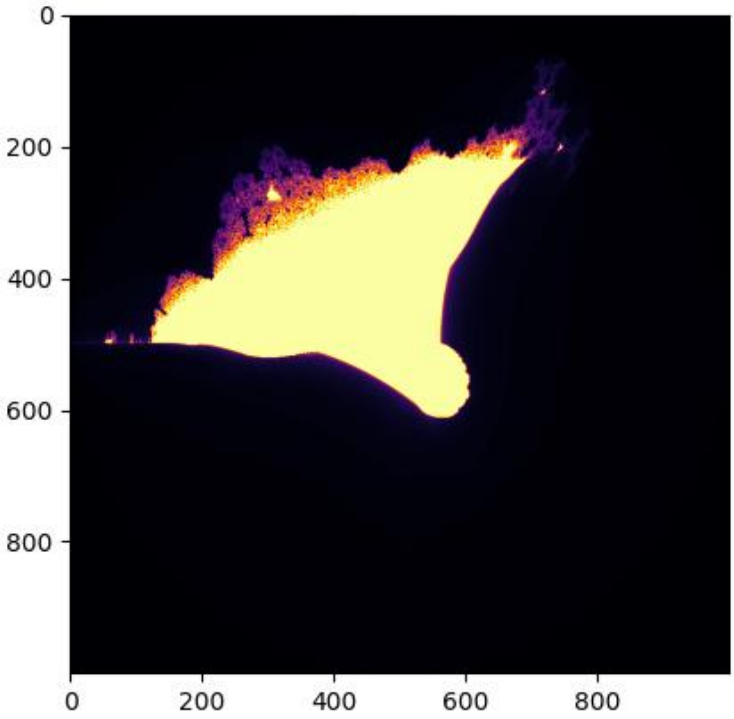
Мал. 4.17. Комплексний фрактал, множина Жуліа при C = 0.4 + i*0.4.



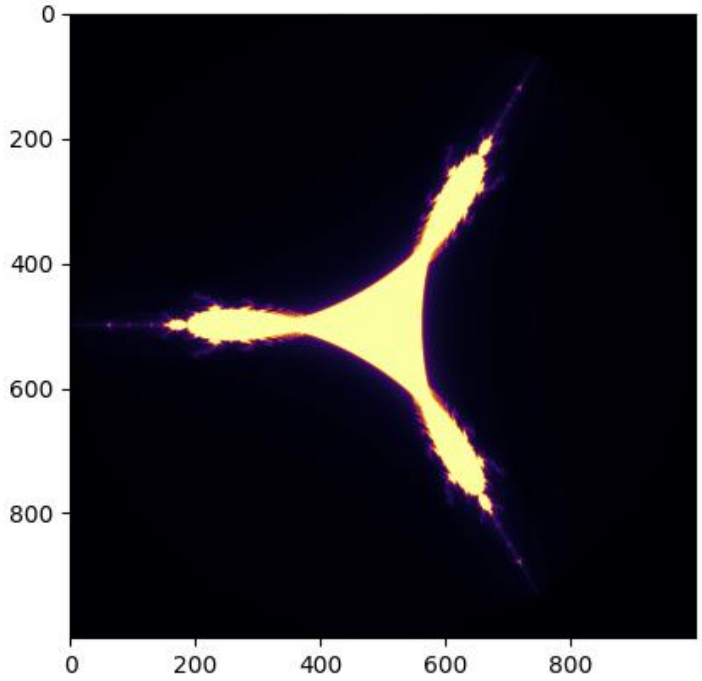
Мал. 4.18. Комплексний фрактал, Мандельброт.



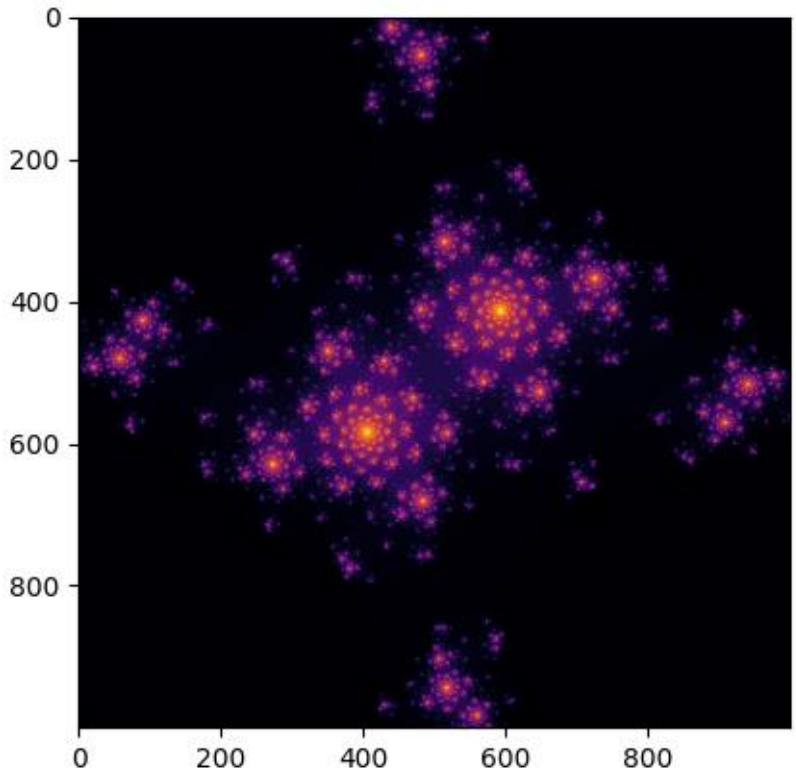
Мал. 4.19. Комплексний фрактал, Мультиброт.



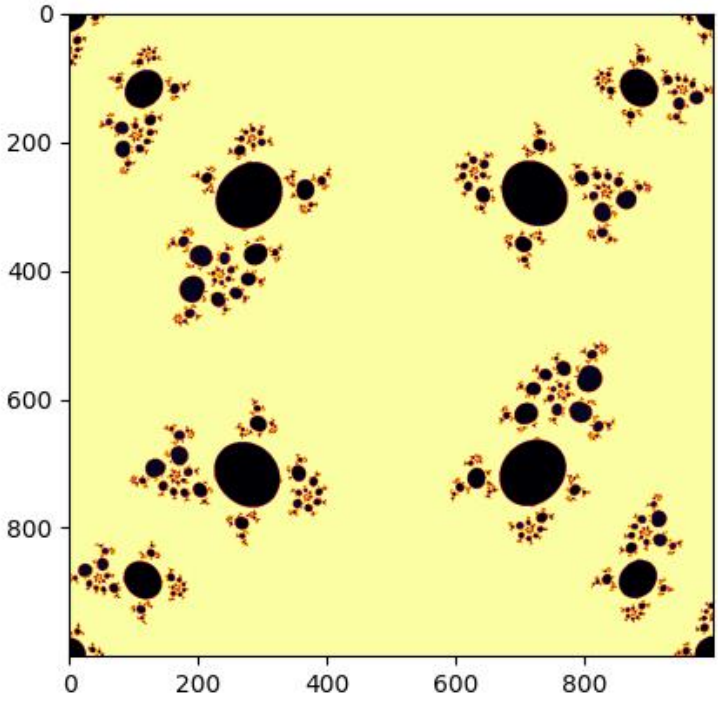
Мал. 4.20. Комплексний фрактал, Палаючий корабель.



Мал. 4.21. Комплексний фрактал, трикорн.



Мал. 4.22. Комплексний фрактал, синусоїдна множина Жуліа.



Мал. 4.23. Комплексний фрактал, гіперболічний тангенс.

5. 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.