**МІНІСТЕРСТВО ОСВІТИ І НАУКИ**

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

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

**НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ**

Проєктна робота

«Стандартизована ітераційна генерація правильних багатокутників, 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](#_Toc169530280)

[2. ХІД РОБОТИ 4](#_Toc169530281)

[2.1. РОЗРОБКА УНІФІКОВАНОЇ СИСТЕМИ СТВОРЕННЯ ФІГУР 4](#_Toc169530282)

[2.2. РЕАЛІЗАЦІЯ КЛАСІВ-ЛЕКАЛ ДЛЯ ФРАКТАЛІВ, ЩО ВИВЧАЛИСЬ ПРОТЯГОМ СЕМЕСТРУ 8](#_Toc169530283)

[2.2.1. РЕАЛІЗАЦІЯ КЛАСУ-ЛЕКАЛА ДЛЯ ФРАКТАЛІВ L-СИСТЕМ 8](#_Toc169530284)

[2.2.2. РЕАЛІЗАЦІЯ КЛАСУ-ЛЕКАЛА ДЛЯ АФІННИХ ФРАКТАЛІВ 10](#_Toc169530285)

# 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.

# 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 чого 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 чого 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