计算物理第10次作业

吴远清 2018300001031

第一题

题目分析

对于n=50的情况下, enumeration由于内存占用极高(在n=19时,使用c++编写的程序内存占用就已超过64g).因此只能通过simulation方法来进行模拟.

对于SAW, $n=1$时, 不存在任何限制, 因此总的可能路径为$T_n=4$. 对于$n>1$的情况下, 下一步均有三种可能(不能掉头走), 因此总的可能路径为:

$$T_n = 4 \times 3^{n-1} \tag{1}$$

实际的满足SAW的路径为C_n,则对应的概率为:

$$P_n = \frac{C_n}{T_n} \tag{2}$$

实际上,我们通过simulation方法测得的是P_n.

测定P_n后,我们有:

$$\frac{P_{n+1}}{P_n} \sim \frac{\mu}{3}(1+\frac{\gamma-1}{n}) \tag{3}$$

根据P_n和n的关系,我们即可求出μ和γ的值.

代码实现

首先引入相关包.

其中,`scipy`用于曲线拟合,计算出μ和γ

```
import numpy as np
from random import random
from tqdm import tqdm
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
```

之后设定拟合函数, 形式与(3)式相同.参数为μ和γ

```
def f(x,mu,gamma):
    return (mu/3)*(1+(gamma-1)/x)
```

之后实现SAW类, 实现对SAW过程的模拟, 计算出P_n

```
class SAW():
    def __init__(self, steps = 50, n = 10**6):
        self.n = n
        self.steps = steps
    def simulate(self):
```

```
        nSurvive = 0
        for i in tqdm(range(self.n)):
            if self.oneWay():
                nSurvive += 1
        return (nSurvive/self.n)
    def oneWay(self):
        direction = [(1,0),(-1,0),(0,1),(0,-1)]
        path = [[0,0]]
        lastDir = None
        for step in range(self.steps):
            dir = int(random()*4)
            if lastDir == None:
                path.append([path[-1][0] + direction[dir][0], path[-1][1] +
direction[dir][1]])
                lastDir = dir
            else:
                while lastDir+dir == 1 or lastDir+dir == 5:
                    dir = int(random()*4)
                path.append([path[-1][0] + direction[dir][0], path[-1][1] +
direction[dir][1]])
                lastDir = dir
            for i in path[:-1]:
                if i == path[-1]:
                    return False
        return True
```

计算$n = 1 \sim n = 50$的P_n

```
pn = []
for i in range(1,51):
    s = SAW(steps = i)
    pn.append(s.simulate())
    print(pn[-1])
```

计算出模拟出的C_n,与enumeration的结果做对照.

```
with open('data.txt', 'r') as fp:
    data = np.array(json.load(fp))
tot = 4
for i in data:
    print(tot * i)
    tot = tot * 3
```

最后, 拟合$P_n \sim n$的关系, 计算出μ和γ.

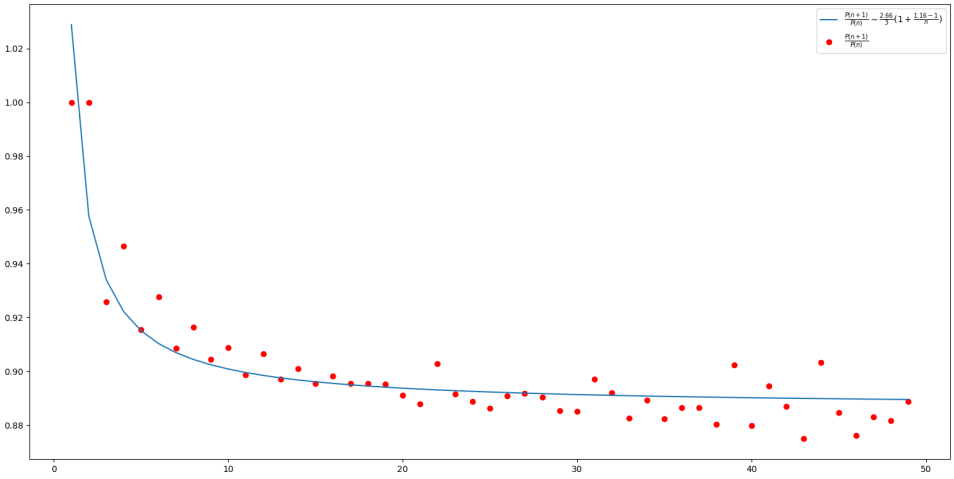
```
x = np.array(range(1,50))
y = data[1:]/data[0:-1]
popt, pcov = curve_fit(f, x, y)
print("mu = {}, gamma = {}".format(popt[0], popt[1]))
plt.scatter(x,y,label = '$\\frac{P(n+1)}{P(n)}$', color = 'r')
plt.plot(x, f(range(1,50), popt[0], popt[1]), label = "$\\frac{P(n+1)}
{P(n)}\\sim\\frac{%.2f}{3}(1+\\frac{%.2f - 1}{n})$"%(popt[0], popt[1]))
plt.legend()
plt.show()
```

结果分析

将计算得到的P_n代入(2)式, 算得C_n, 并与enumeration得到的C_n做比较:

n	Enumeration	Simulation
1	4	4
2	12	12
3	36	36
4	100	99.99
5	284	283.95
6	780	779.80
7	2172	2170.45
8	5916	5915.52
9	16268	16263.38
10	44100	44129.36
11	120292	120323.91
12	324932	3234356.16
13	881500	881998.62
14	2374444	2373411.25
15	6416596	6414535.39
16	17245332	17233209.49
17	46466676	46442418.54
18	124658732	123771920067.43

可见,两种方法得到的结果非常接近. 当n增大时, 误差增大, 这是由于, 我们始终保持模拟次数不变, 因此当步长增加时(可能性增多), 精度会下降



拟合得到的曲线如上图,计算出来的结果为:

$$\mu = 2.66$$

$$\gamma = 1.16$$

第二题

问题分析

对于Koch分形, 我们可以这样考虑.

Koch分形的过程可以看作是一个算符A作用于一个线段$l \in \mathbb{R}^2 \times \mathbb{R}^2$上,得到若干个新的线段$l_1, l_2, \cdots, l_m$,不同的规则下的Koch分形实际上是算符$A$的不同.

对于n阶的Koch分形K_n, 我们有:

$$\{l_i | l_i \in K_n\} = \{Al_i' | l_i' \in K_{n-1}\} \tag{1}$$

由上式,我们通过递归来产生Koch分形.

代码实现

引入相关包

```
import matplotlib.pyplot as plt
from math import tan
from tqdm import tqdm
```

定义算符A(产生分形的规则), 此处为正方形

```
def squareUnit(p1, p2):
    newPoint = []
    if p1[1] == p2[1]:
        dx = p2[0] - p1[0]
        newPoint.append([p1[0] + dx/3, p1[1]])
```

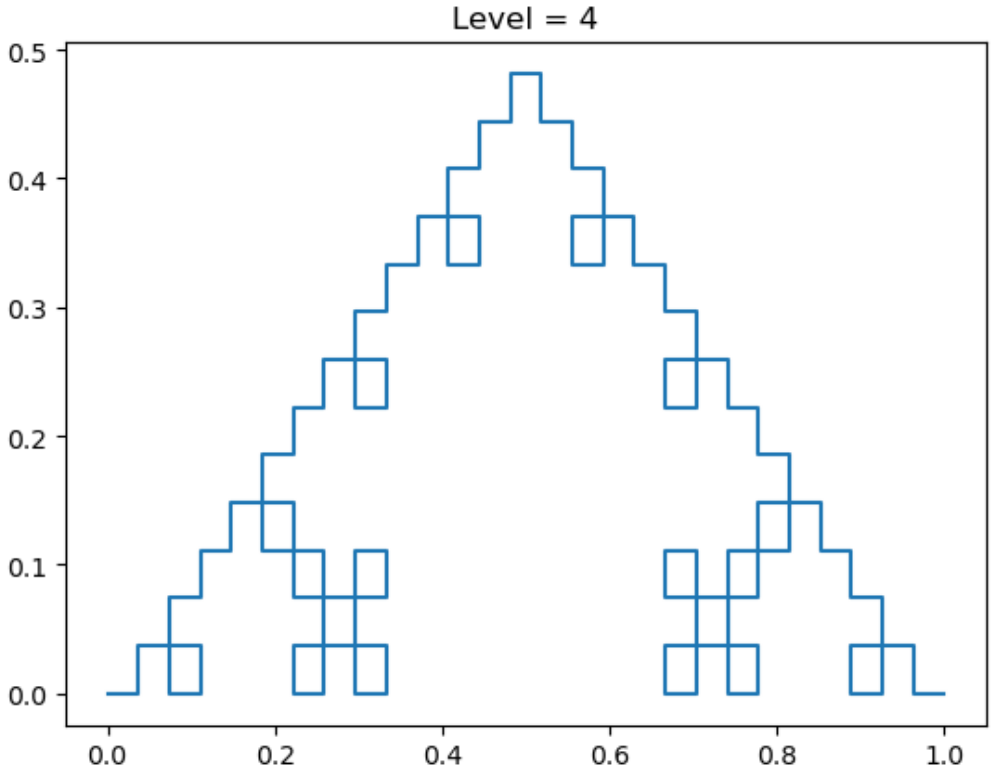
```
        newPoint.append([p1[0] + dx/3, p1[1]+dx/3])
        newPoint.append([p1[0] + 2*dx/3, p1[1] + dx/3])
        newPoint.append([p1[0] + 2*dx/3, p1[1]])
    else:
        dy = p2[1] - p1[1]
        newPoint.append([p1[0], p1[1] + dy/3])
        newPoint.append([p1[0]-dy/3, p1[1] + dy/3])
        newPoint.append([p1[0]-dy/3, p1[1] + 2*dy/3])
        newPoint.append([p1[0], p1[1] + 2*dy/3])
    return newPoint
```

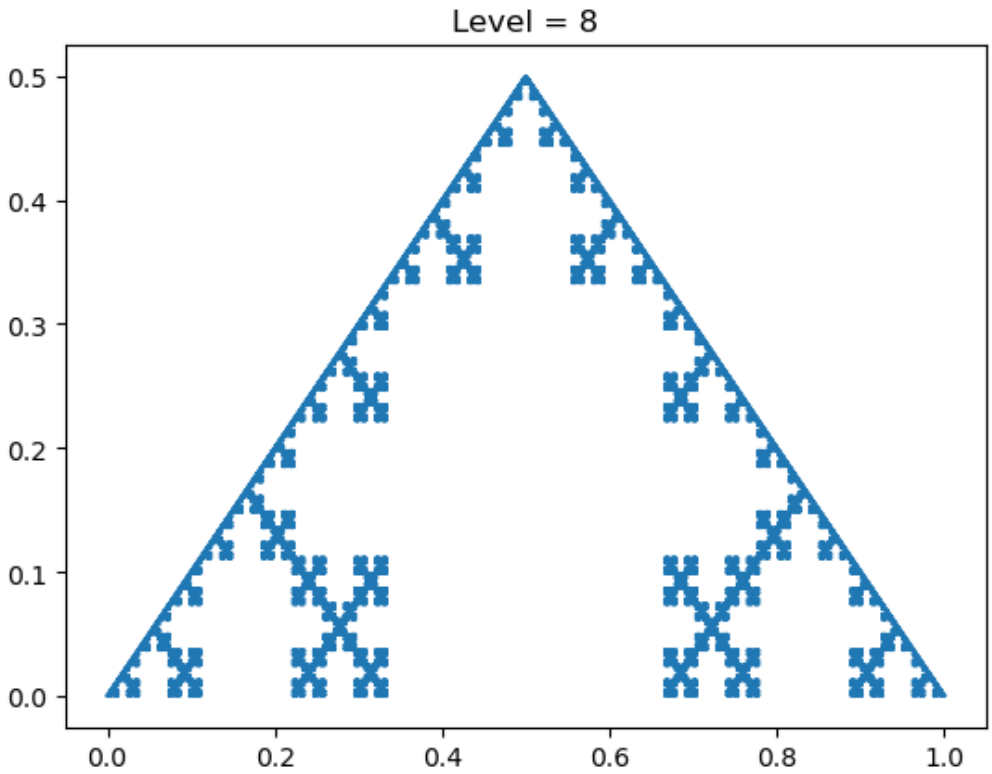
定义`fractalCurve`类,实现递归计算和绘制分形.

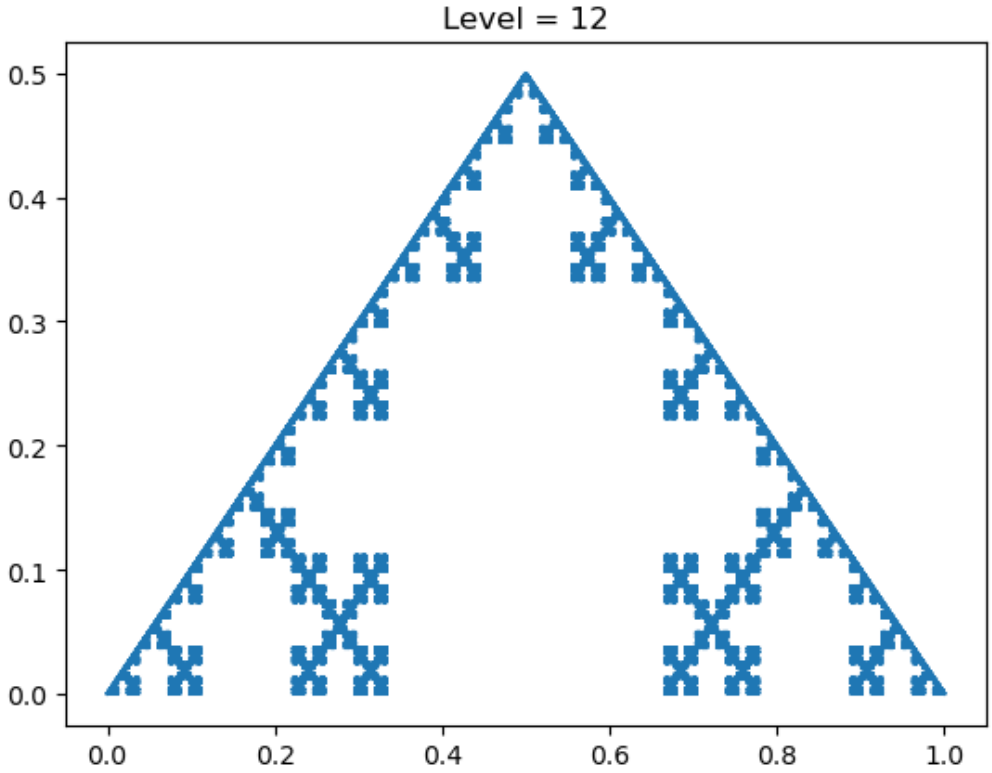
```
class fractalCurve():
    def __init__(self, level = 10, generator = squareUnit):
        self.level = level
        self.pattern = []
        self.generator = generator
    def generate(self, level):
        print(level)
        if level == 1:
            self.pattern = [[0,1], [0,0]]
        else:
            self.generate(level - 1)
            plt.plot(self.pattern[0], self.pattern[1])
            plt.title("Level = {}".format(level - 1))
            plt.show()
            newPattern = [[],[]]
            for i in tqdm(range(len(self.pattern[0]) - 1)):
                #print([self.pattern[0][i], self.pattern[1][i]], [self.pattern[0]
[i+1], self.pattern[1][i+1]])
                newPoint = self.generator([self.pattern[0][i], self.pattern[1]
[i]], [self.pattern[0][i+1], self.pattern[1][i+1]])
                newPattern[0].append(self.pattern[0][i])
                newPattern[1].append(self.pattern[1][i])
                for j in newPoint:
                    newPattern[0].append(j[0])
                    newPattern[1].append(j[1])
            newPattern[0].append(self.pattern[0][-1])
            newPattern[1].append(self.pattern[1][-1])
            self.pattern = newPattern[:]
    def plot(self):
        self.generate(self.level)
        plt.plot(self.pattern[0], self.pattern[1])
        plt.title("Level = {}".format(self.level))
        plt.show()
```

结果展示

我计算了1~12阶的分形,由于空间限制,此处仅展示三个.


Level = 4
0.5
0.4
0.3
0.2
0.1
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Level = 8
0.5
0.4
0.3
0.2
0.1
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Level = 12
0.5
0.4
0.3
0.2
0.1
0.0
0.0
0.2
0.4
0.6
0.8
1.0