計算機実習 問題 14.2 フラクタル次元のくりこみ群による計算

早稲田大学先進理工学部物理学科 B4 藤本將太郎

2014 年 7 月 9 日

1　シミュレーションの目的

パーコレーション・クラスターを粗視化していくと，$p > p_c$ では粗視化後の占有確率 p' は元の占有確率 p に比べて大きくなるように見える．一方で $p < p_c$ の時には $p' < p$ となるように見える．$p = p_c$ のときには，すべての長さのスケールが存在し，系を観測するためにどの長さのスケールを使うかは問題でなくなり、したがって $p' = p_c$ となる．

今回のシミュレーションでは，もとの格子がセルの端から端を連結しているかどうかを表す 1 個の格子点で置き換える．また，くりこみ後の各セルは他のすべてのセルとは無関係に，そのセルが占有される確率 p' を持つという簡単な近似を行うことにする．このようなくりこみの操作によってクラスターを特徴づける量がどのように変化するかということを観測することによって，有限のサイズのシミュレーションにおいても，臨界指数を計算することができるようになる．

2　作成したプログラム

本シミュレーションで作成したプログラムを以下に示す．

2.1　くりこみを用いてフラクタル次元を求めるプログラム

今回シミュレーションに用いたプログラムは，問題 14.1 に用いたプログラムの大部分を流用している．ただし，プログラムの実行のためのダイアログは使用しておらず，くりこみ後の格子の描画を行うなどの目的で，関数 draw_canvas の引数の扱いを若干変更してある．また，パーコレートしなかった場合には，再帰的に関数を実行し直すようにし，パーコレートしていない格子が得られないようにしてある (62〜64 行目)．

今回新たに定義したのは関数 renormalization で，その名の通り，くりこみを行う関数である．事前にパーコレーション・クラスターが得られている状態で呼び出し，格子のサイズがくりこみのスケール因子 b で割り切れる時のみ実行される (割り切れない場合にはエラーが送出される)．くりこみのルールは，上端と下端が連結している時に格子は占有されるとし，上端と下端が連結しないときにはくりこみ後の格子点は占有されないことにする．この操作によって得られたくりこみ後の格子と，もとのパーコレーション・クラスターの 2 つに関して，その占有された格子点の数を数え，それを 2 乗した値を返す．これをループで回して 10 回の平均の値を求め，その値からフラクタル次元 D を求めることができる．

```python
#! /usr/bin/env python
# -*- coding:utf-8 -*-
#
# written by Shotaro Fujimoto, July 2014.

from Tkinter import *
import numpy as np
import matplotlib.pyplot as plt

class Percolation:

    def __init__(self, L=16, p=0.5927):
        self.L = L # lattice size
        self.p = p
        self.sub = None
        self.lattice = np.zeros([self.L, self.L], dtype=bool)

    def percolate(self):
        if self.sub is None or not self.sub.winfo_exists():
            lattice = self.lattice
            rn = np.random.random([self.L, self.L])
            lattice[rn < p] = True
            lattice[rn >= p] = False
            self.lattice = lattice

    def labeling(self):
        label = np.zeros([self.L+2, self.L+2], dtype=int)
        n = 1
        r = range(1, self.L+1)
        for i in r:
            for j in r:
                if self.lattice[i-1][j-1]:
                    nn = []
                    if label[i-1][j] > 0: nn.append(label[i-1][j])
                    if label[i][j-1] > 0: nn.append(label[i][j-1])
                    if len(nn) > 0:
                        label[i][j] = min(nn)
                    else:
                        label[i][j] = n
                        n += 1
```

```python
        tag = range(1, n+1)

        for i in reversed(r):
            for j in reversed(r):
                if label[i][j] > 0:
                    nn = []
                    if label[i+1][j] > 0: nn.append(label[i+1][j])
                    if label[i][j+1] > 0: nn.append(label[i][j+1])
                    nn.append(label[i][j])
                    min_tag = min(nn)
                    nn = set([x for x in nn if x != min_tag])
                    for t in nn:
                        tag[t-1] = min_tag
                        label[label == t] = tag[t-1]

        self.lattice = label[1:-1, 1:-1]
        left = set(self.lattice[0])
        right = set(self.lattice[self.L-1])
        top = set([self.lattice[t][0] for t in range(self.L)])
        bottom = set([self.lattice[t][self.L-1] for t in range(self.L)])
        self.ptag = (left.intersection(right)|top.intersection(bottom))-set([0])
        if len(self.ptag) == 0:
            self.percolate()
            self.labeling()
        return self.lattice, self.ptag

    def renormalization(self, b=2):
        if self.L % b != 0:
            raise ValueError("lattice cannot be divided by scale factor b")

        lattice = np.zeros([self.L, self.L])
        lattice[self.lattice == list(self.ptag)[0]] = 1
        rlattice = np.zeros([self.L/b, self.L/b])
        for i in range(self.L/2):
            ic = 2*i
            for j in range(self.L/2):
                jc = 2*j
                if lattice[ic, jc]*lattice[ic, jc+1] == 1 or \
                    lattice[ic+1, jc]*lattice[ic+1, jc+1] == 1:
                    rlattice[i, j] = 1
```

```

        self.rlattice = rlattice*list(self.ptag)[0]

        M = np.sum(lattice)
        rM = np.sum(rlattice)
        M_2 = M*M
        rM_2 = rM*rM
        return M_2, rM_2

    def draw_canvas(self, rect, L):
        default_size = 640 # default size of canvas
        r = int(default_size/(2*L))
        fig_size = 2*r*L
        margin = 10
        sub = Tk()

        sub.title('figure  '+'(p=%s)'% str(self.p))
        self.canvas = Canvas(sub, width=fig_size+2*margin,
                             height=fig_size+2*margin)
        self.canvas.create_rectangle(margin, margin,
                                    fig_size+margin,fig_size+margin,
                                    outline='black', fill='white')
        self.canvas.pack()
        c = self.canvas.create_rectangle
        colors = ['blue', 'green', 'red', 'purple']
        colordict = dict(zip(list(self.ptag),
                             colors * (int(len(self.ptag)/len(colors)) + 1)
                            )
                        )

        nonzero_rect = np.nonzero(rect)
        for m, n in zip(nonzero_rect[0], nonzero_rect[1]):
            if rect[m][n] in self.ptag:
                c(2*m*r+margin, 2*n*r+margin,
                  2*(m+1)*r+margin, 2*(n+1)*r+margin,
                  outline='', fill=colordict[rect[m][n]])
            else:
                c(2*m*r+margin, 2*n*r+margin,
                  2*(m+1)*r+margin, 2*(n+1)*r+margin,
                  outline='', fill='black')
```

```

        sub.mainloop()

if __name__ == ’__main__’:
    L = 16
    p = 0.5927
    per = Percolation(L, p)
    b = 2
    trial = 10

    M_2, rM_2 = [], []
    for t in range(trial):
        per.percolate()
        per.labeling()
        m_2, rm_2 = per.renormalization(b)
        M_2.append(m_2)
        rM_2.append(rm_2)

#    per.draw_canvas(per.lattice, L)
#    per.draw_canvas(per.rlattice, L/b)
    ave_M_2 = np.average(M_2)
    ave_rM_2 = np.average(rM_2)
    D = np.log(ave_M_2/ave_rM_2)/(2*np.log(b))

    print ’D = %f’ % D

```

3 実習課題

a. もとの格子と，大きさ $L' = L/b$ にくりこまれた格子の両方で，$p = p_c$ における端から端まで連結したクラスター内の占有された格子点の数の 2 乗 $< M^2 >$ および，$< M'^2 >$ をそれぞれ計算せよ．$< M^2 > \sim R^{2D}$ であり，$< M'^2 > \sim (R/b)^{2D}$ でもあるので，関係式 $b^{2D} =< M^2 > / < M'^2 >$ から D を求めることができる．長さのスケール変換のための因子を $b = 2$ と選んで，第 13.5 節で用いたのと同じブロック法を用いよ．$L = 16$，$p = 0.5927$ の場合に定性的な結果を得るには，10 個の端から端まで連結したクラスターについて平均を取れば十分である．

スケール変換のための因子を $b = 2$ とおいて，もとの格子のサイズ $L = 16$ としたときに，もとの格子と，くりこまれた格子の両方について，パーコレーション・クラスター内の占有サイトの数の 2 乗を計算し，その値から関係式

$$b^{2D} = \frac{< M^2 >}{< M'^2 >} \tag{1}$$

を用いてフラクタル次元 D を求めた．実際に 10 回目の試行で得られたもとのパーコレーション・クラスターと，くりこまれた後の格子点を，図 1，図 2 に示す．このとき，1 回のプログラム (すなわち 10 個のサンプル) で得られたフラクタル次元 D は $D \approx 1.933$ であった．また，試行回数を 50 回にし，10 回のプログラムの実行によって得られた平均値は $D \approx 1.925$ となり，その偏差 σ は $\sigma \approx 0.01193$ となった．

図 1　$p = 0.5927$ において得られたサイト・パーコレーションの例

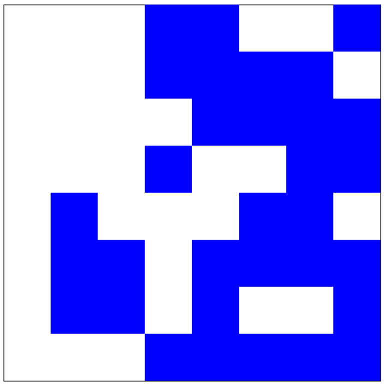

図 2　$b = 2$ によってくりこまれたパーコレーションクラスター

4　まとめ

くりこみの考え方を用いた簡単な計算によって，パーコレーション・クラスターのおおよそのフラクタル次元を求めることができた．

参考文献

[1] ハーベイ・ゴールド, ジャン・トボチニク, 石川正勝・宮島佐介訳 『計算物理学入門』, ピアソン・エデュケーション, 2000.