計算機実習 問題 14.1 パーコレーション・クラスターのフラクタル次元

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 7 日

1　シミュレーションの目的

フラクタル図形の面積は、一般にユークリッド次元での物体として面積を求めることはできない。このことを特徴づけるための量として、フラクタル次元が定義されている。この概念の説明のために普通のユークリッド幾何学における次元について、簡単な考えのいくつかを復習することにする。質量が M で半径が R の円や球状の物体を考えてみる。物体の半径を R から $2R$ と 2 倍にすると、もし物体が円形であれば質量は 2^2 に、球状であれば 2^3 に増大する。質量と長さの間のこのような関係は

$$M(R) \sim R^D \quad (\text{質量次元}) \tag{1}$$

と表すことができる。ここで D は物体の次元である。式 (1) は、形を保ったまま物体の差し渡しの大きさを b 倍にするとその物体の質量が b^D に増大することを意味している。このような質量と長さの間のスケーリングの関係は、次元についての我々の直感的理解と密接な関係がある。物体の次元 D と、その物体が置かれているユークリッド次元 d とが等しければ、質量密度 $\rho = M/R^d$ は

$$\rho(R) \propto M/R^d \sim R^0 \tag{2}$$

とスケールされる。質量と長さの関係が式 (1) を満たし、かつ $D = d$ であるような物体はコンパクトであると言われる。

式 (1) を使ってフラクタル次元が定義できる。もしある対象について式 (1) が成り立ち、D の値が空間次元 d と異なる値をとるとき、その対象はフラクタルであると呼ぶ。もしある物体に対して式 (1) が満たされ $D < d$ であるとき、その密度はすべての R に対して同じではなく、

$$\rho(R) \propto M/R^d \sim R^{D-d} \tag{3}$$

のようにスケールされる。$D < d$ なので、フラクタルな物体の密度は大きなスケールで見るほど小さくなる。密度のこのようなスケール依存性はフラクタルな対象がどの程度入り組んでいるか、あるいはひも状であるかについての定量的な尺度である。つまり、フラクタルな対象の特徴の 1 つは、あらゆる大きさのでこぼこな穴が存在すことである。

フラクタルな対象の持つもうひとつの重要な特徴は、長さのスケールのある範囲にわたって、それらが同じに見えることである。この自己相似性またはスケール不変性は、フラクタルな対象の一部を取り出し、それ

を全ての方向について同じ倍率で拡大しても、その拡大された図形はもとのものと区別できないことを意味する。

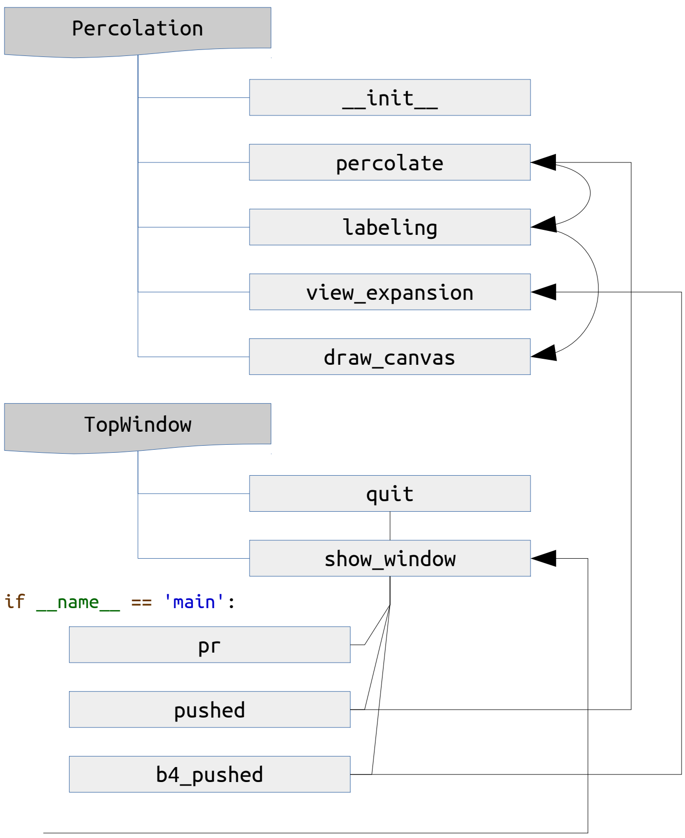
問題 14.1 では簡単なモンテカルロ法を用いて、パーコレーション・クラスターのフラクタル次元の計算を行う。M と R 間のベキ乗則の関係に対する説得力のある証拠を得る場合や、フラクタル次元を高い精度で求める場合には数十組のデータが必要であることを思い出そう。したがって、それらの問題での限られたシミュレーションに基づいた結論は注意深く解釈されなければならない。

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 パーコレーション・クラスターのフラクタル次元を求めるプログラム

以下に、プログラム全体の概略図を示す。



このプログラムにおいて、クラス Percolate がシミュレーションに関して本質的な部分である。クラス TopWindow はサイトの占有確率 p を入力したり、シミュレーションを実行したり、得られた結果を図にしたりするためのボタンなどを、画面に配置するものである。

クラス Percolate では、関数として percolate, labeling, view_expansion, draw_canvas が定義されている。

関数 percolate は、与えられた確率で各サイトに粒子を配置することを行っている。

関数 labeling ではクラスターのラベル付けを行っている。教科書 [1]13 章に書かれているラベル付けのアルゴリズムを参考にしており、まず格子を表す行列で行番号、列番号の若いものから順番に走査する。一つ前の行と列を調べ、そこに粒子が存在する場合はそれらのうち小さい数を自分のラベルとし、一方そこに粒子が存在しない場合には、全体の通し番号 n を、新たに自身のラベルとする。ここまでが 30〜43 行目の内容である。この操作だけでは、隣接するクラスターが異なるラベルを持つということが起きて、適切なラベル付けはできないので、次に、これらのサイトに与えた数字を、隣接するクラスタが存在するとき、小さい方の数にラベルを揃えることを考える。ここでは新たにラベルの数字に対応する tag という配列を用意し、この配列の中の数を変えることによって、ラベルにおける数と、そのラベルが変更されるべき数を表すことができる。実際のアルゴリズムとしては、始めと逆向きに走査し、一つ行／列の数字が大きい 2 点に粒子が存在する場合、自分自身を含めた 3 点での最小値を求め、最小でない点におけるラベルを持つサイトをすべて、その最小値で置き換える。この操作を繰り返し行うことにより、クラスタのラベル付けを効率的よく適切に行うことができる。こうして得られた self.lattice(59 行目) の、上下左右の行と列をみて、上と下、左と右で、どちらにも存在するようなラベルを探す。そのようなラベルを持つクラスターは、パーコレートしていると見なし、そのラベルは self.ptag に保存される (60〜64 行目)。

関数 view_expansion は、pecolate, labeling が実行された後に呼び出される必要があり、この関数によって、視野拡大法を用いたサイト・パーコレーションのフラクタル次元を求めることができる。具体的なアルゴリズムとしては、まず,self.ptag の 1 個目のラベルをもつサイトのラベルを 1 として、その他のサイトは全て 0 とする。次に、視野の大きさ b を $3, 5, \cdots, 2k+1, \cdots$ というように変えていき、そのサイズの正方形を収めることのできる領域で、そのうちゼロでない領域に対して、そのそれぞれの点を原点とした 1 辺が b の正方形の中に、占有されたサイトがいくつあるか ($M(b)$) を数え上げる。各 b に対して、全ての可能な原点の取り方で得られた $M(b)$ を平均することで、より正確な値が得られる (67〜78 行目)。b について得られた $M(b)$ を、scipy パッケージの最小 2 乗法モジュールを用いてベキ乗則に近似し、もとのデータとあわせて両対数グラフにして画面に表示する (83〜110 行目)。

最後に draw_canvas についてだが、これは Tk の Canvas メソッドを用いて、占有サイトを黒の正方形で表すものである。また、パーコレーション・クラスターは、青または緑、赤、紫の正方形で描画される。効率化のための工夫として、canvas.create_rectangle を毎回直接呼び出すのではなく、ローカル変数 c に置き換えておき、それを呼び出すことによって、無駄な遅延が出ないようになっている。また、デフォルトサイズを決めておき、そのサイズから、描画するキャンバスの大きさと正方形の大きさを決定している。

```
1   #! /usr/bin/env python
2   # -*- coding:utf-8 -*-
3   #
4   # written by Shotaro Fujimoto, June 2014.
5
6   from Tkinter import *
7   import numpy as np
8   import sys
9   import matplotlib.pyplot as plt
10  import scipy.optimize as optimize
11
```

```
class Percolation:

    def __init__(self):
        self.sub = None
        self.L = 61 # lattice size
        self.p = 0.5
        self.lattice = np.zeros([self.L, self.L], dtype=bool)

    def percolate(self, p=0.5):
        self.p = p
        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
        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])

    def view_expansion(self):
        lattice = np.zeros([self.L, self.L])
        lattice[self.lattice == list(self.ptag)[0]] = 1
        M_b = []
        s = np.sum
        ave = np.average
        append = M_b.append
        for k in range(1, int(self.L)/2):
            nonzero = np.nonzero(lattice[k:-k,k:-k])
            tmp = np.array([0])
            for i, j in zip(nonzero[0]+k, nonzero[1]+k):
                tmp = np.append(tmp, s(lattice[i-k:i+k+1, j-k:j+k+1]))
            append(ave(tmp))

        b = np.array([2.*k+1 for k in range(1, int(self.L)/2)])
        M_b = np.array(M_b)

        def fit_func(parameter0, b, M_b):
            log = np.log
            c1 = parameter0[0]
            c2 = parameter0[1]
            residual = log(M_b) - c1 - c2*log(b)
            return residual

        parameter0 = [0.1, 2.0]
        result = optimize.leastsq(fit_func, parameter0, args=(b[:-1], M_b[:-1]))
```

```
        c1 = result[0][0]
        D = result[0][1]

        def fitted(b, c1, D):
            return np.exp(c1)*(b**D)

        fig = plt.figure("Fractal Dimesion")
        ax = fig.add_subplot(111)
        ax.plot(b, M_b, '-o', label="p = %f" % self.p)
        ax.plot(b, fitted(b, c1, D), label="fit func: D = %f" % D)
        ax.set_xlabel(r'$b$', fontsize=16)
        ax.set_ylabel(r'$M(b)$', fontsize=16)
        ax.set_xscale('log')
        ax.set_yscale('log')
        ax.set_ymargin(0.05)
        fig.tight_layout()
        plt.legend(loc='best')
        print "D = %f" % D
        plt.show()

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

        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
        rect = self.lattice
        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')


class TopWindow:

    def quit(self):
        self.root.destroy()
        sys.exit()

    def show_window(self, pr, pushed, b4_pushed):
        self.root = Tk()
        self.root.title('Percolation')
        f = Frame(self.root)
        self.label = Label(f, text='p =')
        self.label.pack(side='left')
        self.entry = Entry(f, width=20)
        self.entry.pack(side='left')
        self.entry.delete(0, END)
        self.entry.insert(0, 0.5927)
        self.entry.focus_set()

        b1 = Button(f, text='run', command=pushed)
        b1.pack(side='left', expand=YES, fill='x')

        b4 = Button(f, text='count', command=b4_pushed)
        b4.pack(side='left', expand=YES, fill='x')

        b2 = Button(f, text='write canvas to sample.eps', command=pr)
```

```
            b2.pack(side='left', expand=YES, fill='x')

            b3 = Button(f, text='quit', command=self.quit)
            b3.pack(side='right', expand=YES, fill='x')

            f.pack(fill='x')

            self.root.mainloop()

    if __name__ == '__main__':
        top = TopWindow()
        per = Percolation()
        count = 1

        def pr():
            global count
            p = float(top.entry.get())
            d = per.canvas.postscript(file="figure_%d(p=%s).eps" % (count, str(p)))
            print "saved the figure to a eps file"
            count += 1

        def pushed():
            p = float(top.entry.get())
            per.percolate(p)
            per.labeling()
            per.draw_canvas()

        def b4_pushed():
            if not per.ptag:
                print "Can't count. (There is no percolation cluster.)"
            else:
                per.view_expansion()

        top.show_window(pr, pushed, b4_pushed)

```

3　実習課題

a. $L = 61$ の正方格子上で $p = 0.5927$ におけるサイト・パーコレーションのパターンを作成せよ。端から端まで連結したクラスターを得る前に多数の配置を生成しなければならない理由を考えよ。端から端まで連結したクラスターが入り組んだパターンを持つことを感覚的に理解するために、1 つの配置を出力して、端から端まで連結したクラスター内の点に印をつけよ。その中に、一方が占有され、一方が占有されていない最隣接格子点の組は多く存在するか。

$p = 0.5927$ におけるサイト・パーコレーションのパターンを作成し、これを図 1 に示す。パーコレートしたクラスタを青色で示し、その他のクラスタは黒色で描画してある。パーコレートしたクラスターを得るために多数の配置を生成しなければならないのは、$p = 0.5927$ という値が、パーコレーション閾値 p_c に近い値であることによる [2]。すなわち、これより小さい p では、ほとんどの配置でパーコレーションは起こらない。パーコレーションクラスター内の点で、一方が占有され、一方が占有されていない再隣接格子点の組は多く存在しており、空洞が多く、境界が入り組んだ構造を持っていることが分かる。

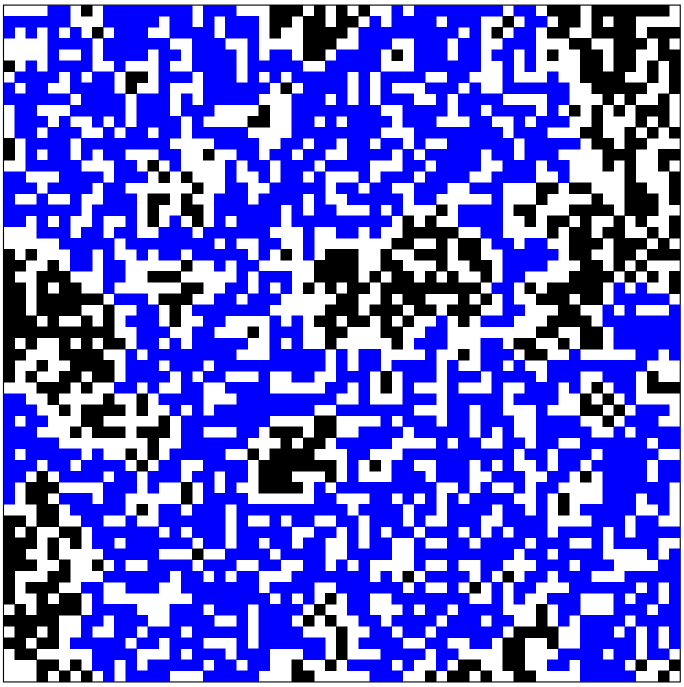


図 1　$p = 0.5927$ において得られたサイト・パーコレーションの例

b. 端から端まで連結したクラスター内の 1 つの点を選んで、その点を中心にした b^2 の正方形の内部にある点の数 $M(b)$ を数えよ。つぎに、b を大きくして、その正方形の中の点の数を数えよ。点の数の b 依存性が得られるまで、この手順を繰り返せ (視野拡大法)。この手順を無限に繰り返すことは可能か。$M(b)$ の b 依存性を用いて、定義 $M(b) \sim b^D$ により D を求めよ。クラスター内の別の点を選んで、この手順を繰り返せ。同様の結果が得られたか。多くの端から端まで連結したクラスターのそれぞれについて原点の選び方に関して $M(b)$ を平均することで、D のよりよい値を得ることができる。

パーコレーション・クラスター内の点に関して、その点を中心とした $b = 3, 5, \cdots, 2k+1, \cdots$ を 1 辺とする正方形内に含まれるパーコレーションクラスターの数を数える。この操作を、考えている系の外側に正方形がはみ出すことが無いようにしながら、b と正方形の原点の位置を変えて、全ての可能な場合について数え上げる。これは、系が有限のサイズであるために、b の大きさを無限に大きくすることはできないことによる。こうして、1 辺 b の正方形の内部に含まれるパーコレーションクラスターの点の数 $M(b)$ の平均値をそれぞれの b について求めることができる。これを横軸 b、縦軸 $M(b)$ の両対数グラフにプロットすると、図 2 のように直線で近似できることがわかり、その傾きは $D \approx 1.87$ と求められた。このような計算を 20 回行って、D の平均値を求めると $D \approx 1.84$ が得られた。また、そのときの偏差 σ は $\sigma = 0.0952$ であった。これは、大規模なコンピュータ・シミュレーションによって得られた値 $D \approx 1.89$ とも近い値となっている [2](ただし、参考文献では $p = 0.5928$ として計算しているようなので注意)。

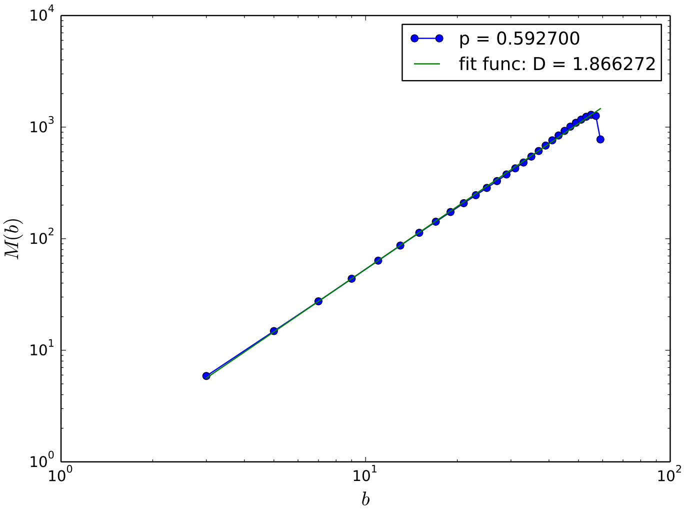


図 2　$M(b)$ と b の関係を表す両対数グラフ。フラクタル次元 D は $D \approx 1.87$ と求められた。

4　まとめ

シミュレーションによってサイト・パーコレーションのフラクタル次元を求めることができた。クラスターのラベル付けのアルゴリズムや、効率化の点で苦労したが、現時点で納得のいくプログラムを作成することができたと思う。

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.

[2] 松下 貢『フラクタルの物理 (I)』, 裳華房, 2002.