計算機実習 問題 12.12 2 次元自己回避型ランダムウォーク

早稲田大学先進理工学部物理学科 B4 藤本將太郎

2014/06/14

1　シミュレーションの目的

溶媒中の高分子を特徴づける基本的な幾何学的量は両端間の距離の 2 乗平均 $< R^2(N) >$ である。ここで N は単量体の数である。高分子鎖の希薄溶液では、$< R^2(N) >$ の漸近的依存性は

$$< \Delta R^2(N) > \sim N^{2\nu} \quad (N \gg 1) \tag{1}$$

で与えられ、指数 ν は 3 次元の場合 $\nu \simeq 0.592$ である。また、以下で議論する高分子のモデルでは、2 次元の場合は正確に $\nu = 3/4$ であることが知られている。式 (1) の比例定数は単量体の構造や溶媒による。それに対して、指数 ν はこれらの詳細にはよらない。

ここで溶媒中の鎖状高分子の全体的な特徴を取り込んだランダムウォークのモデルについて考える。高分子鎖を格子上に置き (図 12.3b 参照)、単量体どうしの相互作用を無視すると、この簡単なランダムウォークのモデルは、次元や格子の対称性にはよらず、$\nu = 1/2$ を与える。しかしながら、この ν の値は実験とは一致しないので、高分子の重要な物理的性質を見落としていることがわかる。そこで、高分子の重要な特徴である、2 個の単量体が空間の同じ位置を占有することができない (排除体積条件) という点を考慮に入れた、より現実的なモデルを考えよう。自己回避型ランダムウォーク (SAW) は、曲がる高分子鎖についてこの拘束条件を取り込んだ、よく知られたモデルの 1 つである。原点から出発し、どの格子点も 2 度以上は訪れないという拘束条件に従う N 歩のランダムウォーク全体の集合を考える。問題 12.12 では 2 次元自己回避型ランダムウォークのモンテカルロ・シミュレーションを扱う。

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　2 次元自己回避型ランダムウォークのシミュレーション実行ファイル

プログラムの内容について、簡単な解説をつけておく。
このプログラムは主に 1 つのクラスと 4 つの関数から構成され、このクラス SAW が今回のシミュレーションで特に重要な部分である。他の関数、例えば 115 行目の plot_graph は、グラフを 1 つのウィンドウに複数並べて配置するための汎用的な関数であり、この関数内の変数は matplotlib.pyplot 内の変数以外には依存していない。また、151 行目の if 節以降の関数 ex_a,ex_b,ex_b_fit はそれぞれ問題 a、b で用いる関数である

(ex_b_fit は ν の計算に用いる)。

クラス SAW は初期化関数の他に self_avoiding_rw_d2、self_avoiding_rw_Rosenbluth という 2 つの関数を含む。これらの関数はそれぞれ問題 a、b で考える、異なったアルゴリズムの 2 次元自己回避型ランダムウォークをシミュレートするものである。2 つの関数で共通のものとして配列 lattice がある。これは 2 次元の配列であり、ブール値を保持して、True のときそこには単量体が存在すると見なす。行方向を x、列方向を y とする。

次に、時間発展ごとにどのような処理を行うかについて説明を加えると、self_avoiding_rw_d2 では 1 つ前の座標を (_x,_y) で保持し、現在の位置との差をとってベクトルにし、これを vec とする。次の遷移の方向は vec の向きから見て、前・右・左の 3 種類であり、乱数を用いてこの中から等確率で選ぶようにする。もしすでにその位置に粒子があるとき (49 行)、新しいループを開始する。そうでなければその位置に単量体を置いてこの時の両端間距離 $R_m^2(n)$(m:m 番目の高分子鎖、n:n 個目の単量体) を記録する。active_walkers によって、N ステップまで高分子鎖を作ることができた数をカウントし、これによって N 個の高分子鎖をつくることができる割合 $f(N)$ を求める。また、$R^2(N)$ の平均値を求めるとき、その N まで高分子鎖を作ることができたものだけで平均をとるためにも、active_walkers を用いている。

self_avoiding_rw_Rosenbluth では、リスト path の中に可能な (つまり空白サイトの) 座標を記録し、その数によって場合分けを行う。問題 b に示されているように、これが 0 ならループを終了、1 ならその点に移動して $W[n] = (1/3)W[n-1]$ とし、2 なら等確率で移動して $W[n] = (2/3)W[n-1]$ とする。可能なパスが 3 つあるとき同様に等確率で移動し、$W[n] = W[n-1]$ とする。1 ステップごとに R^2 を計算し、保存する。全ての walker についての計算が終了したら、

$$< R^2(N) >= \frac{\sum_i W_i(N) R_i^2(N)}{\sum_i W_i(N)} \tag{2}$$

の式を用いて $< R^2(N) >$ を計算する。

```
1   #! /usr/bin/env python
2   # -*- coding:utf-8 -*-
3   #
4   # written by Shotaro Fujimoto, June 2014.
5
6   import numpy as np
7
8
9   class SAW:
10
11      """ Class for the simulation of SAW (Self-Avoiding Random Walk).
12
13      functions:
14      self_avoiding_rw_d2(N, x0=0, y0=0)
15      self_avoiding_rw_Rosenbluth(N, x0=0, y0=0)
16      """
17
```

```
18      def __init__(self, walker=1000, x0=0, y0=0):
19          self.walker = walker
20          self.x0 = x0
21          self.y0 = y0
22
23      def self_avoiding_rw_d2(self, N, x0=0, y0=0):
24
25          walker = self.walker
26          f = np.array([[1, 0], [0, 1]])
27          r = np.array([[0, 1], [-1, 0]])
28          l = np.array([[0, -1], [1, 0]])
29          R_2 = np.zeros([walker, N], 'i')
30
31          for m in range(walker):
32              x, y = x0 + N + 1, y0 + N + 1
33              lattice = np.zeros([2 * N + 3, 2 * N + 3], dtype=bool)
34              lattice[x][y] = True
35              lattice[x][y + 1] = True
36              _x, _y = x, y
37              x, y = x, y + 1
38              R_2[m][0] = 1
39              for n in range(1, N):
40
41                  vec = x - _x, y - _y
42                  _x, _y = x, y
43
44                  p = np.random.rand() * 3
45                  if p < 1:     # direction = 'forward'
46                      x, y = np.dot(f, vec) + (x, y)
47                  elif p < 2:   # direction = 'right'
48                      x, y = np.dot(r, vec) + (x, y)
49                  else:         # direction = 'left'
50                      x, y = np.dot(l, vec) + (x, y)
51
52                  if lattice[x][y]:
53                      break
54                  else:
55                      lattice[x][y] = True
56                      R_2[m][n] = (x - x0 - N - 1) ** 2 + (y - y0 - N - 1) ** 2
57
```

```
58          active_walkers = np.zeros(N)
59          for m in range(N):
60              i = 0
61              for k in range(walker):
62                  if R_2[k][m]:
63                      i += 1
64              active_walkers[m] = i
65
66          def f(N):
67              return active_walkers[N - 1] / float(walker)
68
69          def ave_R_2():
70              return np.sum(R_2, axis=0, dtype=np.float32) / active_walkers
71
72          self.f_N = [f(n) for n in range(1, N + 1)]
73          self.ave_R_2 = ave_R_2()
74
75      def self_avoiding_rw_Rosenbluth(self, N, x0=0, y0=0):
76
77          walker = self.walker
78          W = np.zeros([walker, N], 'f')
79          R_2 = np.zeros([walker, N], 'i')
80
81          for m in range(walker):
82              x, y = x0 + N + 1, y0 + N + 1
83              lattice = np.zeros([2 * N + 3, 2 * N + 3], dtype=bool)
84              lattice[x][y] = True
85              lattice[x][y + 1] = True
86              x, y = x, y + 1
87              W[m][0] = 1
88              R_2[m][0] = 1
89              for n in range(1, N):
90                  path = []
91                  if not lattice[x - 1][y]:
92                      path.append((x - 1, y))
93                  if not lattice[x + 1][y]:
94                      path.append((x + 1, y))
95                  if not lattice[x][y - 1]:
96                      path.append((x, y - 1))
97                  if not lattice[x][y + 1]:
```

```
98                      path.append((x, y + 1))
99
100                 if len(path) == 0:
101                     break
102                 if len(path) == 1:
103                     x = path[0][0]
104                     y = path[0][1]
105                     W[m][n] = W[m][n - 1] / 3.
106                 if len(path) == 2:
107                     p = np.random.rand() * 2
108                     idx = int(p)
109                     x = path[idx][0]
110                     y = path[idx][1]
111                     W[m][n] = (W[m][n - 1] * 2.) / 3.
112                 if len(path) == 3:
113                     p = np.random.rand() * 3
114                     idx = int(p)
115                     x = path[idx][0]
116                     y = path[idx][1]
117                     W[m][n] = W[m][n - 1]
118
119                 lattice[x][y] = True
120                 R_2[m][n] = (x - x0 - N - 1) ** 2 + (y - y0 - N - 1) ** 2
121
122         self.ave_R_2 = np.sum(W * R_2, axis=0, dtype=np.float32) \
123                         / np.sum(W, axis=0, dtype=np.float32)
124
125
126 def plot_graph(x_data, y_data, x_labels, y_labels,
127                xscale='linear', yscale='linear', aspect='auto'):
128     """ Plot the graph about y_data for each x_data.
129     """
130     import matplotlib.pyplot as plt
131
132     d = len(y_data)
133     if not len(x_data) == len(y_data) == len(x_labels) == len(y_labels):
134         raise ValueError("Arguments must have the same dimension.")
135     if d == 0:
136         raise ValueError("At least one data for plot.")
137     if d > 9:
```

```
138             raise ValueError("""So much data for plot in one figure.
139                                 Please divide two or more data sets.""")
140
141         fig = plt.figure(figsize=(9, 8))
142         subplot_positioning = [
143             '11', '21', '22', '22', '32', '32', '33', '33', '33']
144         axes = []
145         for n in range(d):
146             lmn = int(subplot_positioning[d - 1] + str(n + 1))
147             axes.append(fig.add_subplot(lmn))
148
149         for i, ax in enumerate(axes):
150             ymin, ymax = min(y_data[i]), max(y_data[i])
151             ax.set_aspect(aspect)
152             ax.set_xscale(xscale)
153             ax.set_yscale(yscale)
154             ax.set_xlabel(x_labels[i], fontsize=16)
155             ax.set_ylabel(y_labels[i], fontsize=16)
156             ax.set_ymargin(0.05)
157             ax.plot(x_data[i], y_data[i])
158
159         fig.subplots_adjust(wspace=0.2, hspace=0.5)
160         fig.tight_layout()
161         plt.show()
162
163     if __name__ == '__main__':
164
165         def ex_a(N):
166             rw = SAW()
167             rw.self_avoiding_rw_d2(N)
168
169             print 'f(%d) = ' % N + str(rw.f_N[N - 1])
170             print '<R^{2}(%d)> = ' % N + str(rw.ave_R_2[N - 1])
171
172             x_labels = [r'$N$'] * 2
173             y_labels = [r'$f(N)$', r'$<R^{2}(N)>$']
174             plot_graph([range(1, N + 1)] * 2, [rw.f_N, rw.ave_R_2],
175                        x_labels, y_labels)
176
177         def ex_b(N):
```

```
178         rw = SAW()
179         rw.self_avoiding_rw_Rosenbluth(N)
180 
181         for n in [4, 8, 16, 32]:
182             print '<R^{2}(%d)> = ' % n + str(rw.ave_R_2[n - 1])
183 
184         x_labels = [r'$N$']
185         y_labels = [r'$<R^{2}(N)>$']
186         plot_graph([range(1, N + 1)], [rw.ave_R_2], x_labels, y_labels,
187                    xscale='log', yscale='log', aspect='equal')
188 
189     def ex_b_fit(N):
190         import scipy.optimize as optimize
191         from math import sqrt
192 
193         trial = 100
194         nu = np.zeros(trial)
195         for i in range(trial):
196             rw = SAW()
197             rw.self_avoiding_rw_Rosenbluth(N)
198 
199             parameter0 = [1.0, 0.75]  # C, nu
200 
201             def fit_func(parameter0, n, r_2):
202                 C = parameter0[0]
203                 nu = parameter0[1]
204                 residual = r_2 - C * (n ** (2 * nu))
205                 return residual
206 
207             result = optimize.leastsq(fit_func, parameter0,
208                                       args=(np.array(range(1, N + 1)), rw.ave_R_2))
209 
210             nu[i] = result[0][1]
211 
212         print 'nu =', np.average(nu), u'pm', np.std(nu) / sqrt(rw.walker)
213 
214 #    ex_a(N=25)
215 #    ex_b(N=32)
216     ex_b_fit(N=32)
```

3 実習課題

a. 正方格子上の自己回避型ランダムウォークを考える。任意の格子点を選んで、そこを原点とし、第 1 歩を”上向き”にとる。他の 3 方向の第 1 歩から生成される自己回避型ランダムウォークは、単に全体を回転させるだけなので、それらを別に考える必要はない。原点に戻ることができないという拘束条件から、第 2 歩では 3 方向が可能である。乱数を生成して 3 方向からその 1 つを選び、引き続く各ステップも同様にして生成する。しかし通常はその歩行 (ランダムウォーク) は無限には続かない。注意すべき点は、偏りのない結果を得るために、1 歩あるいはそれ以上の歩数の後に自己交差を生じてしまう場合でも、同様に乱数 (たとえば 1,2,3) を生成しなければならないことである。次のステップで自己交差が起こるときには、統計性を正しく保つためにその歩行は停止しなければならない。そして新しい歩行を再び原点から始める。この簡単なアルゴリズムを実現するプログラムを書いて、N 個の単量体からなる高分子鎖を生成することができた回数の割合 $f(N)$ を記録せよ。すでに訪れた格子点を記録するために、格子点を 2 次元配列で表すと都合がよい。$f(N)$ の定性的な N 依存性はどうなるか。適当と考えられる N の最大値はどれほどか。それらの N の値について両端間の距離の 2 乗平均の値 $< R^2(N) >$ を求めよ。

上に示したプログラムを用いて、N 個の単量体からなる高分子鎖を生成することができた回数の割合 $f(N)$ と、両端間の距離の 2 乗平均の値 $< R^2(N) >$ を求め、横軸 N として図 1 に示した。この図から分かるように、$f(N)$ は $N > 3$ で、N の増加に対して指数関数的に減少していることが分かる。そこで、適当と考えられる N の最大値は、$f(N)$ の大きさがほぼ 0 となる $N = 50$ 程度であるとしてよいだろう。次に、$< R^2(N) >$ の値についてグラフを観察すると、この曲線を直線に近似することはできず、また、これまで得られたような 2 乗に比例するようなグラフよりは”なだらか”である。したがって、ν は $1/2 < \nu < 1$ を満たす値であると分かる。

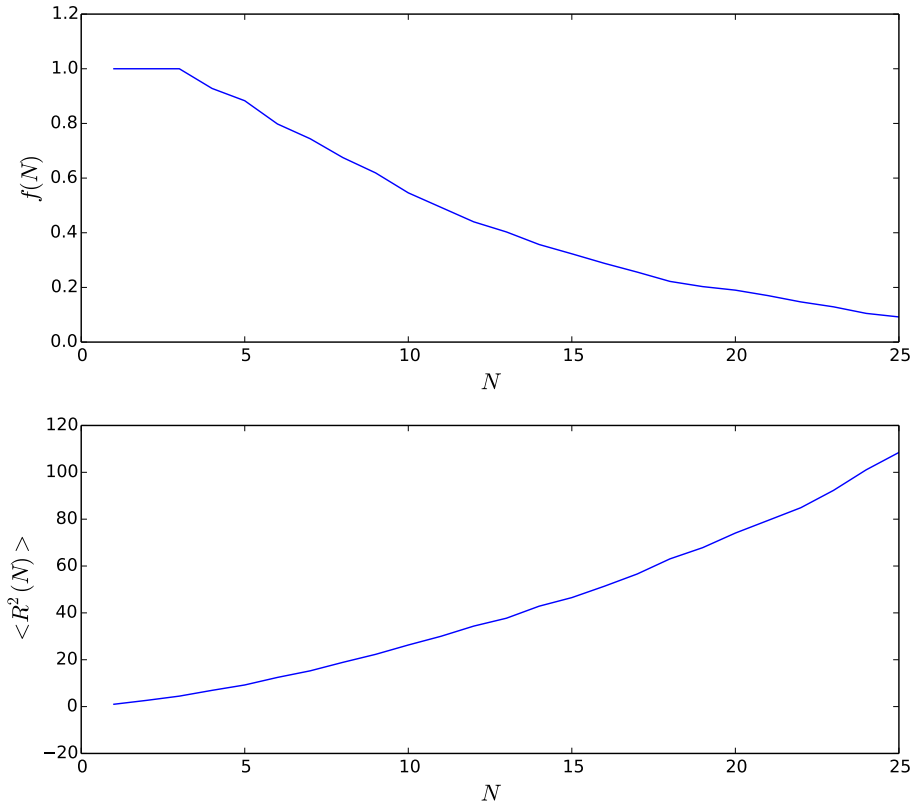

図 1　N 個の単量体からなる高分子鎖を生成することができた回数の割合 $f(N)$ と、両端間の距離の 2 乗平均の値 $< R^2(N) >$

b. 設問 a で用いた単純な標本抽出法の欠点は、長い鎖では全く効率がよくないこと、つまり成功する確率の割合が指数関数的に減少することである。この非効率を解決するために、いくつかの”強化法”が考案されている。最初に、ローゼンブルスとローゼンブルス (Rosenbluth and Rosenbluth) が提案した、N 歩からなる各歩行に重み関数 $W(N)$ を関連づけた比較的簡単な方法について議論しよう。第 1 歩を北向きに取ることは常に可能なので $W(1) = 1$ である。N 歩のすべての可能な配置を同等に数えるために、$N > 1$ での $W(N)$ は次の可能性にしたがって決定される。

1. 可能な 3 通りのステップすべてで自己交差の拘束条件が破られる場合。重みを $W(N) = 0$ として歩行は終了し、原点から出発する新しい歩行が生成される。
2. 3 通りのステップが可能な場合、$W(N) = W(N-1)$ とする。
3. $m(1 \leq m < 3)$ 個のステップだけが可能な場合。この場合には $W(N) = (m/3)W(N-1)$ とし、乱数を生成して可能な m 通りのステップの中から 1 つを選ぶ。

$< R^2(N) >$ の正しい値は、i 番目の試行で得られた $R^2(N)$ の値 $R_i^2(N)$ にその歩行の重み $W_i(N)$ をつけて得られる。したがって

$$< R^2(N) >= \frac{\sum_i W_i(N) R_i^2(N)}{\sum_i W_i(N)} \tag{3}$$

と書くことができる。ここで和はすべての試行についてとられる。ローゼンブルスの方法をモンテ

カルロ法のプログラムに取り入れて、$N = 4, 8, 16, 32$ について $< R^2(N) >$ を計算せよ。そして、$< R^2(N) >$ の N に対する両対数プロットから指数 ν を求めよ。その ν の値はランダムウォークの値 $\nu = 1/2$ と区別することができるか。

ローゼンブルスの方法を取り入れて、$N = 4, 8, 16, 32$ について $< R^2(N) >$ を計算し、その結果を表 1 に示した。また、$< R^2(N) >$ の N に対する両対数プロットを図 2 に示した。このグラフから読み取れることとして、$< R^2(N) >$ は N のベキ乗に比例しており、そしてグラフの傾きは 1 よりも大きいので、指数 ν を求めると、その値は単純なランダムウォークのときの値 $\nu = 1/2$ よりも大きくなることが予想される。実際に 100 回の試行から ν を求めると、$\nu = 0.732232182648 \pm 0.000938990585164$ となった。これは単純なランダムウォークの値 $\nu = 1/2$ と区別できる。

表 1　ローゼンブルスの方法で求めた $< R^2(N) >$

N	$< R^2(N) >$
4	6.38406
8	17.0735
16	50.2673
32	143.723

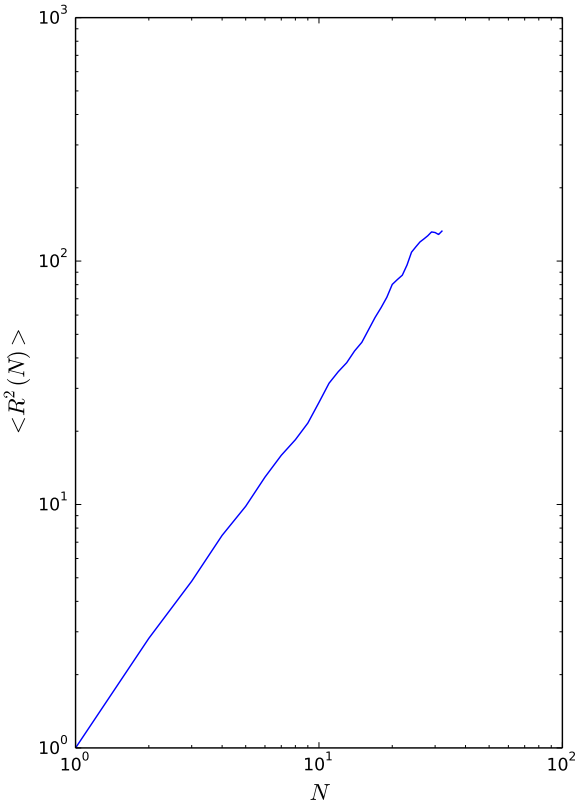

図 2　$< R^2(N) >$ の N に対する両対数プロット

4　まとめ

自己回避型ランダムウォークのモンテカルロ･シミュレーションを行い､両端間の距離の 2 乗平均 $< R^2(n) >$ の値が単純なランダムウォークのときの値 $1/2$ と区別できることを確かめた。

5　参考文献

- 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.