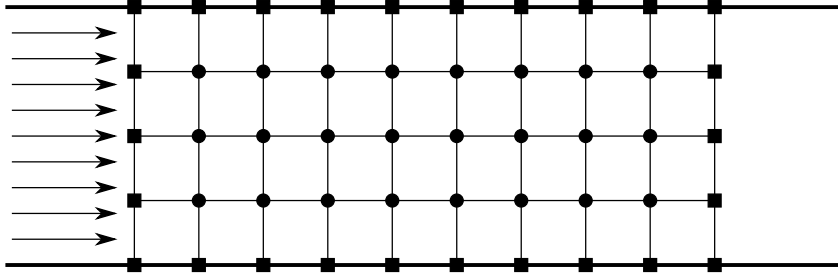


Incompressible Flow

田浦

1 やりたいこと

2次元ハーゲン・ポアズイユ流を以下の領域でシミュレートしたい



2 方程式

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u - \nabla p + c\Delta u \quad (1)$$

$$\nabla \cdot u = 0 \quad (2)$$

3 解法

非圧縮の方程式の普通の解き方.

右辺の p を求めるために式 (1) の両辺の $\nabla \cdot$ をとる.

$$\frac{\partial \nabla \cdot u}{\partial t} + \nabla \cdot (-(u \cdot \nabla)u - \nabla p + c\Delta u)h \quad (3)$$

連続の式 (2) より $\nabla \cdot u = 0$ なので

$$0 = \nabla \cdot (-(u \cdot \nabla)u - \nabla p + c\Delta u) \quad (4)$$

よって

$$-\nabla \cdot (u \cdot \nabla)u - \Delta p + c\nabla \cdot \Delta u = 0 \quad (5)$$

さらに

$$\nabla \cdot \Delta u = \Delta(\nabla \cdot u) = 0 \quad (6)$$

なので,

$$\Delta p = -\nabla \cdot (u \cdot \nabla)u \quad (7)$$

よって, 1 ステップの計算は,

1. 式 (8) を用いて p を求める.

$$\Delta p = -\nabla \cdot (u \cdot \nabla) u \quad (8)$$

2. 求まった p を用いて, $u(t+h)$ を求める.

$$u(t+h) \approx u(t) + (-(u \cdot \nabla)u - \nabla p + c\Delta u)h \quad (9)$$

4 計算の方法 + よくわからないこと

一番単純に各格子点に p, u を割り当てて (スタaggered 格子とかは必要だとわかるまでとりあえずやらないで), 差分法で解こうと思っている.

その際に境界条件の与え方がよくわからない.

1. 圧力の式 (8) を解くときの境界条件は?

どこぞには, ノイマン条件を課すみたいなのが書いてある.

$$\frac{\partial p}{\partial n} = 0 \quad (10)$$

2. u に関する境界条件は?

$$\text{上下の壁沿いについては } u = 0 \quad (11)$$

でよいのだろうか (多分).

では左右の空いているところはしたら良い? (ここが一番良くわからない)

とりあえずよくわからないながら,

$$\text{左 (流入して来る所) については } u = (1, 0), \quad (12)$$

$$\text{右 (流出して行く所) については } \frac{\partial u}{\partial n} = 0 \quad (13)$$

という条件を課している. 後者の n は x 軸方向のベクトル. 最終的に得たい答えはポアズイユ流の, $u_x = (y - 1/2)^2$ みたいな答えなので, 明らかに違うのだが, と言って, まさかこれを境界条件というわけにも行かず, どうしたものかという感じ.

ちなみに差分法で計算するときの表面的な問題として, 格子点の数を $m \times n$ とすると, 式 (8) の右辺を作る時に微分を差分で計算する関係で, 右辺が計算できるのは, 内部の点 $((m-2) \times (n-2))$ だけになる. 従って, 連立一次方程式を用いて求まる p も内部の点だけとなる (ノイマン条件を課しているので, 外周の点は内部の点と同じとして計算する. つまり, $p[0, j] = p[1, j]$, $p[m-1, j] = p[m-2, j]$, $p[i, 0] = p[i, 1]$, $p[i, n-1] = p[i, n-2]$).

そうやって求まった $(m \times n \text{ 点の}) p$ を用いて式 (9) の右辺を計算するときも, やはり微分を差分で作る関係で, 右辺が計算できるのは内部の点だけとなる. そのため外周の点に対しては $u(t+h)$ の値が (少なくとも素直には) 求まらない. 壁沿いについては物理的に直感にあう境界条件 (式??) があるわけだが, 左右の空いているところはさてどうするのが良いのだろうか.

コードを添付します. matplotlib が必要です. python3 で走らせてくれれば勝手にアニメーションが始まるはずです.

やってみると最初のうち中なか変化がないけど, よく見ると徐々に色が変わってきます. そしてそのうちに破綻した感じにぐちゃぐちゃになります.

何が悪いのか.

- 境界条件が悪い
- dt の値が大きすぎる
- オイラー法が荒すぎる
- 空間微分が荒すぎる
- ポアソン方程式の行列の係数のところでバグを入れている
- ...

など色々あるとおもうのですが. あまりこれ以上一人で悶々と悩みたくないで, 頭の整理を兼ねてこんな文章を書いてみたというわけです.

5 付録: コード

```
1  #!/usr/bin/python3
2  # -*- coding: utf-8 -*-
3  import matplotlib.pyplot as plt
4  import matplotlib.animation as animation
5  import numpy as np
6  import scipy.sparse
7  import scipy.sparse.linalg
8  import time
9  import pdb
10
11 # 次元で 2 ハーゲン・ポアズイユ流をシミュレート
12 #
13 #          ux = 0
14 # +-----+
15 # ---->
16 # ---->
17 # ---->
18 # ---->
19 # +-----+
20 #          ux = 0
21
22 def partial_by_x(f):
23     """
24     f : scalar field (m x n array)
25     returns :  $\partial f / \partial x$  ((m - 1) x n array)
26     """
27     m,n = f.shape
28     p = f[1:,:] - f[:-1,:]
29     assert(p.shape == (m - 1, n)), (p.shape, (m - 1, n))
30     return p
31
32 def partial_by_y(f):
33     """
34     f : scalar field (m x n array)
35     returns :  $\partial f / \partial y$  (m x (n - 1) array)
```

```

36     """
37     m,n = f.shape
38     p = f[:,1:] - f[:, :-1]
39     assert(p.shape == (m, n - 1)), (p.shape, (m, n - 1))
40     return p
41
42 def grad_f(f):
43     """
44     f : scalar field (m x n array)
45     returns :  $\partial (f \partial / x, \partial f \partial / y)$  (2 x (m - 1) x (n - 1) array)
46     """
47     m,n = f.shape
48     gx = partial_by_x(f)
49     gy = partial_by_y(f)
50     r = np.vstack([ gx[:,1:], gy[1:,:] ]).reshape((2, m - 1, n - 1))
51     return r
52
53 def u_dot_grad_f(u, f):
54     """
55     u : vector field (2 x m x n)
56     f : scalar field (m x n)
57     compute  $(\cdot \nabla u) f = (u_0 \partial \partial / x + u_1 \partial \partial / y) f$ 
58     """
59     d,m,n = u.shape
60     assert(d == 2), d
61     assert(f.shape == (m, n)), (f.shape, (m, n))
62     g = u[0,1:,1:] * partial_by_x(f[:,1:]) + u[1,1:,1:] * partial_by_y(f[1:,:])
63     assert(g.shape == (m - 1, n - 1)), (g.shape, (m - 1, n - 1))
64     return g
65
66 # v : vector field
67 # compute  $(\cdot \nabla u) v$ 
68 def u_dot_grad_v(u, v):
69     """
70     u : vector field (2 x m x n)
71     v : vector field (2 x m x n)
72     compute  $(\cdot \nabla u) v = ((u_0 \partial \partial / x + u_1 \partial \partial / y) v_0, (u_0 \partial \partial / x + u_1 \partial \partial / y) v_1)$ 
73     """
74     d,m,n = v.shape
75     assert(d == 2), d
76     assert(u.shape == (d,m,n)), (u.shape, (d,m,n))
77     gx = u_dot_grad_f(u, v[0,:,:])
78     gy = u_dot_grad_f(u, v[1,:,:])
79     assert(gx.shape == (m - 1, n - 1)), (gx.shape, (m - 1, n - 1))
80     assert(gy.shape == (m - 1, n - 1)), (gy.shape, (m - 1, n - 1))
81     r = np.vstack([ gx, gy ]).reshape((2, m - 1, n - 1))
82     return r
83
84 def div(v):
85     """
86     v : vector field (2 x m x n)
87     compute  $\nabla \cdot v = \partial v_0 \partial / x + \partial v_1 \partial / y$ 
88     """
89     d,m,n = v.shape
90     g = partial_by_x(v[0,:,:]) + partial_by_y(v[1,1,:])
91     assert(g.shape == (m - 1, n - 1)), (g.shape, (m - 1, n - 1))
92     return g
93
94 def laplace_f(f):
95     """
96     f : scalar field (m x n)
97     returns :  $\Delta f = \partial^2 f \partial^2 / x^2 + \partial^2 f \partial^2 / y^2$  ((m - 1) x (n - 1))
98     """
99     m,n = f.shape
100     l = f[:-2,1:-1] + f[2:,1:-1] + f[1:-1,-2:] + f[1:-1,2:] - 4 * f[1:-1,1:-1]
101     assert(l.shape == (m - 2, n - 2)), (l.shape, (m - 2, n - 2))
102     return l
103

```

```

104 def laplace_v(v):
105     """
106     v : vector field (2 x m x n)
107     returns :  $\Delta v = \partial^2 v_0 / \partial x^2 + \partial^2 v_0 / \partial y^2,$ 
108                $\partial^2 v_1 / \partial x^2 + \partial^2 v_1 / \partial y^2$ 
109               (2 x (m - 1) x (n - 1))
110     """
111     d,m,n = v.shape
112     assert(d == 2), d
113     lx = laplace_f(v[0,:,:])
114     ly = laplace_f(v[1,:,:])
115     assert(lx.shape == (m - 2, n - 2)), (lx.shape, (m - 2, n - 2))
116     assert(ly.shape == (m - 2, n - 2)), (ly.shape, (m - 2, n - 2))
117     l = np.vstack([ lx, ly ]).reshape((d, m - 2, n - 2))
118     return l
119
120 def make_laplace_matrix(m, n):
121     """
122     make matrix A for (m x n) region ((m * n) x (m * n) matrix).
123     assume neumann boundary condition
124     """
125     D = {}
126     for i in range(m):
127         for j in range(n):
128             D[i,j,i,j] = -4
129             if i + 1 < m:
130                 D[i,j,i+1,j] = 1
131             else:
132                 D[i,j,i,j] += 1
133             if i > 0:
134                 D[i,j,i-1,j] = 1
135             else:
136                 D[i,j,i,j] += 1
137             if j + 1 < n:
138                 D[i,j,i,j+1] = 1
139             else:
140                 D[i,j,i,j] += 1
141             if j > 0:
142                 D[i,j,i,j-1] = 1
143             else:
144                 D[i,j,i,j] += 1
145     I = {}
146     for i in range(m):
147         for j in range(n):
148             I[i,j] = len(I)
149     N = m * n
150     A = scipy.sparse.dok_matrix((N, N))
151     for (i,j,ii,jj),v in D.items():
152         A[I[i,j],I[ii,jj]] = v
153     return scipy.sparse.csr_matrix(A)
154
155 def solve_poisson(p, f):
156     """
157     f : scalar field (m x n)
158     solve  $\Delta p = f$ 
159     """
160     m,n = f.shape
161     N = m * n
162     A = make_poisson_matrix(m, n)
163     p[1:-1,1:-1] = scipy.sparse.linalg.spsolve(A, f.reshape(N)).reshape((m, n))
164     p[0,1:-1] = p[1,1:-1]
165     p[-1,1:-1] = p[-2,1:-1]
166     p[1:-1,0] = p[1:-1,1]
167     p[1:-1,-1] = p[1:-1,-2]
168     p[0,0] = p[1,1]
169     p[-1,-1] = p[-2,-2]
170     p[0,-1] = p[1,-2]
171     p[-1,0] = p[-2,1]

```

```

172
173 def step(u, p, dt):
174     """
175     calc 1 step.
176     u : velocity field (vector field) 2 x m x n
177     p : pressure field (scalar field) m x n
178     """
179     d,m,n = u.shape
180     assert(d == 2), d
181     assert(p.shape == (m,n)), (p.shape, (m, n))
182     c = 1 # 1/Re (Re : Reynolds constant)
183     # calc righthand side of the poission equation
184     #  $(\cdot \nabla u)u$ 
185     vgu = u.dot_grad_v(u, u)
186     assert(vgu.shape == (d, m - 1, n - 1)), (vgu.shape, (d, m - 1, n - 1))
187     #  $-\nabla \cdot (\cdot \nabla u)u$ 
188     div_vgu = -div(vgu)
189     assert(div_vgu.shape == (m - 2, n - 2)), (div_vgu.shape, (m - 2, n - 2))
190     # solve  $\Delta p = -\nabla \cdot (\cdot \nabla u)u$ 
191     solve_poisson(p, div_vgu)
192     #  $\nabla p$ 
193     gp = grad_f(p)
194     assert(gp.shape == (d, m - 1, n - 1)), (gp.shape, (d, m - 1, n - 1))
195     #  $-(\cdot \nabla u)u - \nabla p + c \Delta u$ 
196     dudt = -vgu[:,1:,1:] - gp[:,1:,1:] + c * laplace_v(u)
197     assert(dudt.shape == (d, m - 2, n - 2)), (dudt.shape, (d, m - 2, n - 2))
198     u[:,1:-1,1:-1] += dudt * dt
199     # impose boundary condition on the right open boundary
200     u[:, -1, 1:-1] += u[:, -2, 1:-1]
201
202 #
203 # 以下の simulate1 ~ simulate3 は
204 # アニメーションのやり方を試行錯誤しているもの
205 # simulate3 が一番シンプル
206 #
207
208 def simulate1():
209     d = 2 # dimension (2)
210     m = 200 # x-axis 200
211     n = 50 # y-axis 50
212     dt = 0.001
213     u = np.zeros((d, m, n))
214     u[0,0,:] = 1.0 # left open boundary
215     p = np.zeros((m, n))
216     n_steps = 200
217     n_imgs = 10
218     imgs = []
219     fig = plt.figure()
220     for i in range(n_steps):
221         print("step %d" % i)
222         while len(imgs) / n_imgs < (i + 1) / n_steps:
223             imgs.append([visualize_f(u[0])])
224             print(" add img -> %d" % len(imgs))
225         step(u, p, dt)
226     ani = animation.ArtistAnimation(fig, imgs, interval=500)
227     plt.show()
228     return ani
229
230 def shrink(z):
231     m,n = z.shape
232     return z[:,m-1,:n-1].reshape((m - 1) * (n - 1))
233
234 def visualize_f(f):
235     """
236     f : scalar field
237     """
238     m,n = f.shape
239     x = np.linspace(0, m - 1, m)

```

```

240     y = np.linspace(0, n - 1, n)
241     x,y = np.meshgrid(x, y, indexing='ij')
242     return plt.pcolor(x, y, f)
243
244 def simulate2():
245     d = 2                        # dimension (2)
246     m = 400                    # x-axis 200
247     n = 100                    # y-axis 50
248     dt = 0.01
249     u = np.zeros((d, m, n))
250     u[0,0,:] = 1.0             # boundary
251     p = np.zeros((m, n))
252     n_steps = 100
253     # fig = plt.figure()
254     for i in range(n_steps):
255         print("step %d" % i)
256         if i == 0:
257             field = visualize_f(u[0])
258             plt.legend()
259         else:
260             field.set_array(shrink(u[0]))
261         yield [field]
262         step(u, p, dt)
263
264 def do_animation(iterator, **kwargs):
265     def anime_fun(*args):
266         try:
267             return next(iterator)
268         except StopIteration:
269             return []
270     ani = animation.FuncAnimation(plt.gcf(), anime_fun, **kwargs)
271     plt.show()
272     return ani
273
274 def simulate3():
275     d = 2                        # dimension (2)
276     m = 100                    # x-axis 200
277     n = 50                     # y-axis 50
278     dt = 0.01
279     u = np.zeros((d, m, n))
280     u[0,0,:] = 1.0             # boundary
281     p = np.zeros((m, n))
282     n_steps = 100
283     # fig = plt.figure()
284     for i in range(n_steps):
285         print("step %d" % i)
286         if i == 0:
287             field = plt.imshow(u[0])
288             fig = plt.gcf()
289             #plt.clim() # clamp the color limits
290             plt.colorbar()
291         else:
292             #field.set_array(shrink(u[0]))
293             field.set_data(u[0])
294         step(u, p, dt)
295         plt.pause(0.5)
296
297 def main():
298     # return simulate1()
299     # return do_animation(simulate2(), interval=100)
300     return simulate3()
301
302 ani = main()

```