Python で微分方程式の数値解法

水谷正大

2017 年 10 月 25 日

1　微分方程式の数値解法

$\boldsymbol{x}(t) = (x_1(t), \dots, x_n(t))$ の初期条件 $\boldsymbol{x}(t_0) = (x_1(t_0), \dots, x_n(t_0))$ のもとで空間 M 上の微分方程式系

$$\begin{aligned}
\frac{dx_1}{dt} &= v_1(\boldsymbol{x}, t), \quad x_1(t_0) = x_{10} \\
&\vdots \\
\frac{dx_n}{dt} &= v_n(\boldsymbol{x}, t), \quad x_n(t_0) = x_{n0}
\end{aligned}$$

をベクトル表記して、次のように表す。

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{v}(\boldsymbol{x}, t), \qquad \boldsymbol{x}(t_0) = \boldsymbol{x}_0$$

この微分方程式は各点 $\boldsymbol{x}$ および時間 t 毎に接線ベクトル $\boldsymbol{v} \in \mathbb{R}^n$ を与えているとみることができ、$\boldsymbol{v} : M \to \mathbb{R}^n$ を**ベクトル場** (vector filed) ということがある。

時間 Δt の経過による $\boldsymbol{x}$ の変化を

$$\frac{d\boldsymbol{x}}{dt} = \lim_{\Delta t \to 0} \frac{\boldsymbol{x}(t + \Delta t) - \boldsymbol{x}(t)}{\Delta t}$$

と考えていることから、有限ではあるが十分小さい時間経過 Δt 後に対しては

$$\frac{\boldsymbol{x}(t + \Delta t) - \boldsymbol{x}(t)}{\Delta t} \approx \boldsymbol{f}(\boldsymbol{x}, t)$$

が期待できる。そこで微小であるが有限の時間刻み Δt ごとに、t をパラメータとする**解曲線** $\boldsymbol{x}(t)$ 上に近いと考えられる離散的な値を逐次的に求める微分方程式の数値解法がさまざまに考案されてきた。

時刻 $t = t_0$ における初期条件 $\boldsymbol{x}(t_0)$ から逐次的に求めて得られる点列を $\{\boldsymbol{x}_n\}(n = 0, 1, 2, \dots)$ と表記する ($\boldsymbol{x}_0 = \boldsymbol{x}(t_0)$)。以下では、簡単のために時間刻みを Δt を一定とする。

ここでは $\mathbb{R}^3$ 内の Lorentz 方程式

$$\begin{aligned}
\frac{dx}{dt} &= -\sigma x + \sigma y \\
\frac{dy}{dt} &= rx - y + xz \\
\frac{dz}{dt} &= -bz + xy
\end{aligned}$$

を取り上げよう。ここでは Lorenz による数値計算 (1963) にならってパラメータ $\sigma = 10$, $r = 24.74$, $b = 8/3$ を固定しておく。

1.1　Euler 法

微分方程式の数値解法として、**Euler 法**とは、点列 $\boldsymbol{x}_0, \boldsymbol{x}_1, \dots, \boldsymbol{x}_n, \dots$ を次のようにして逐次的に求める方法である。

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \Delta t \boldsymbol{f}(\boldsymbol{x}_n, t_n), \quad t_n = t_0 + n\Delta t.$$

次の Python 関数 `euler_orbit(x0, T, dt, f)` は、Euler 法を使って解軌道の数値リストを求めている。vectorfield はベクトル場 $\boldsymbol{v}$ を定める m 個の関数名のリスト $[v_1, \ldots, v_m]$ である。

時刻 t_0 における m 次元（2 行目の `width` に次元の値が格納される）の初期点リスト $x0 = [x_{01}, \ldots, x_{0m}]$ から出発して、時刻 t_0 から $t_0 + T$ までを指定した時間刻み dt から逐次的に定まる $N = \lfloor T/dt \rfloor$ 個からなる m 次元ベクトル値のリスト $[\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N]$ を返す（3 行目で $\boldsymbol{x}_0$ を $\boldsymbol{x}$ とおいて、while 文で更新した点を `orbit.append(list(x))` によって数値軌道リスト `orbit` に次々と追加している）。

```
1  def euler_orbit(x0, T, dt, vectorfield):
2      width = len(x0)
3      x = x0
4      t = 0
5      orbit = []
6      orbit.append(list(x))
7      while t <= T:
8          vx = list(map(lambda v: v(t, *x), vectorfield))
9          for i in range(width):
10             x[i] += dt * vx[i]
11         orbit.append(list(x))
12         t += dt
13     return(orbit)
```

8 行目で Python 的技法 `vx = list(map(lambda v:  v(t, *x), vectorfield))` を使っている。`*x` は可変長引数を表す。`lambda v:v(t,*x)` は**無名関数**（ラムダ関数）として、`t, *x` をセットして関数 `v(t, *x)` を定義している。

ただし、この関数 `v` の実体は `map` を使って、ベクトル場 $\boldsymbol{v}(t, \boldsymbol{x})$ を定めている関数名のリスト `vectorfild` から 1 つずつ得ていることがポイントである。`map` はリスト `vectorfild` から各関数要素（オブジェクト）を取り出して無名関数定義に渡す。無名関数 `lambda v:  v(t, *x)` は既に値が格納された引数 `t` と可変引数 `*x` を使って関数値を計算する。その結果、`vx` には与えられた点でのベクトル場値 `vx` がリストとして格納される。

`vx` はリストであるため、値 $\boldsymbol{x} + dt \cdot \boldsymbol{vx}$ を計算して改めて $\boldsymbol{x}$ とするために 9,10 行目のように、for 文を使ってリスト要素毎の代入計算 `x[i] += dt * vx[i]` をしている。

1.2　可変長引数

Python の記号であるアスタリスク (`*`) は通常、乗算（掛け算）として利用される（文字列についても `'apple' * 4` は `'appleappleappleapple'` と掛け算の意味として使う）。しかし、関数定義の際の引数の並びに、アスタリスク (*) を 1 つ付けて**可変長引数**（variable arguments）の意味を持たせるように使うことができる。アスタリスク (*) を 2 つ付けて**キーワード可変長引数**とすることもできる。

関数定義において、引数にアスタリスク (*) を 1 つ付ける任意の個数の引数が、タプル型オブジェクト（括弧 (と) に挟まれたカンマ区切りの並び）の要素に格納される。

次の例を見てみよう。関数 `func` は形式的には `x, y` および `arg` の 3 つの引数を取るように見えるが、`'a', 'b', 'c', 'd', 'e', 'f'` の 6 つの引数を与えると、`x` に `'a'` が、`y` に `'b'` が、そして `arg` には残りの `'c', 'd', 'e', 'f'` がセットされたことが print で確認できる。つまり、引数 `x, y` 以降に渡した任長さの引数並びが `args` にタプル型オブジェクトの要素として格納されたことが分かる。

```
>> def func(x, y, *args):
>>     print a, b, args
>> print func('a', 'b', 'c', 'd', 'e', 'f')
('a', 'b', ('c', 'd', 'e', 'f'))
>> print func('a', 'b', 'c', 'd')
('a', 'b', ('c', 'd'))
>> print func('a', 'b', 'c')
```

```
('a', 'b', ('c',))
>> print func('a', 'b')
('a', 'b', ())
```

1.3 Euler 法で Lorenz 系を解いてみる

Lorenz 方程式（論文内の式 (25), (26), (27)）を Euler 法で解いてみよう。次のスクリプト `diff_euler.py` は、ベクトル場 `vector = [v1, v2, v3]` で定まる Lorenz の微分方程式を初期点 `x0 = [0.1, 0.1, 0.1]`、時間区間 1 、刻み幅 0.01 で（Euler 法を逐次的に適用して）100 個の要素からなる軌道リスト `orbit` を求めている。

diff_euler.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  # vector fields v1, v2, v3 of Lorenz eq as functions of t, x, y, z.
5  p, r, b = 10.0, 28.0, 8.0/3.0
6  def v1(t, x, y, z):
7          return(-p * x + p * y)
8  def v2(t, x, y, z):
9          return(-x * z + r * x - y)
10 def v3(t, x, y, z):
11         return(x * y - b * z)
12
13 def euler_orbit(x0, T, dt, vectorfield):
14     width = len(x0)
15     x = x0
16     t = 0
17     orbit = []
18     while t <= T:
19         orbit.append(list(x))
20         vecx = list(map(lambda v: v(t, *x), vectorfield))
21         for i in range(width):
22             x[i] += dt * vecx[i]
23         #print x,
24         t += dt
25     return(orbit)
26
27 vector = [v1, v2, v3]
28 x0 = [0.1, 0.1, 0.1]
29 dt = 0.01
30 T = 1.0
31
32 orbit = euler_orbit(x0, T, dt, vector)
33 print(orbit)
```

演習 1.1 `diff_euler.py` を実行してみなさい。`T= 50, dt = 0.01` としたとき、`orbit` の各要素（3 つの要素（x, y, z 成分からなる）に関して、どれか 2 つだけを取り出して（3 組 $(x, y), (y, z), (x, z)$ のいずれか）csv 形式のファイル `lorenz_euler.csv` に書き出して、各点を「つない」で折れ線でプロットしてみなさい

ここでは、グラフィックス描画モジュール matplotlib とその関連モジュールを使って、解軌道を 3 次元プロットしてみよう。描画された画像はマウスで回転して解軌道の様子が観察できる。次のプログラムの先頭の 2,3,4 行は先のスクリプト `diff_euler.py` の先頭でインポートする。先のスクリプト `diff_euler.py` の 32 行目の後に、以下の 11 行目以下を加えておく。モジュール matplotlib を使って描くために、時間刻み `dt` ごとに逐次的に求めた x, y, z-成分リスト $[x_i, y_i, z_i](i = 0, 1, 2, \dots)$ からなるリストから、x-成分、y-成分および z-成分だけが並んだリスト `xorbit`,`yorbit`,`zorbit` を用意している。22 行目でこれらのリストを matplotlib

に渡して軌道を描いている。このような形でプロットするのは numpy や matplotlib の仕様である。

diff_euler.py 続き

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D# for 3dim plot
....
....
orbit = euler_orbit(x0, T, dt, vector)

xlist = []
ylist = []
zlist = []
for x, y, z in orbit:
    xlist.append(x)
    ylist.append(y)
    zlist.append(z)

# plot in 3-din space
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot(xorbit, yorbit, zorbit)
ax.set_xlabel('x(t)')
ax.set_ylabel('y(t)')
ax.set_zlabel('z(t)')
plt.show()
```

演習 1.2　上のように修正加筆した `diff_euler.py` で、`T= 50, dt = 0.01` として、得られる軌道要素を描きなさい。

1.3.1　Numpy を使って効率化を図る

上のスクリプトの 9-15 行目のようにして matplotlib に渡すデータを整形することが二度手間のように思える。Euler 法で得られる軌道要素のリストは長さ 3 の x, y, z 成分からなるリストを要素としたリスト (長さ N とする）で、数学的には $N \times 3$（N 行 3 列）の行列である。目的とする `xorbit`, `yorbit` および `zorbit` は、この $N \times 3$ 行列の各列の要素からなるリストである。

モジュール NumPy はこうした場合に強力な手段を与える。NumPy (通常 `import numpy as np` としてインポートされる）で最も重要なクラス np.ndarray(object) を使うと多次元配列の簡単で多彩な取扱が可能になる（object の各次元ごとの要素数が等しいとする)。クラス np.ndarray のコンストラクタとして通常 np.array(リスト) によって（多次元配列を）生成する。

次の Python Shell の例を見てみよう。多次元リスト `a` は 4×3 の配列とみなせる。Python では多次元リスト `a` のスライス（要素の切り出し）は `a[i][j]` などと行われるが、このままでは各行または各列の要素全体をスライスできない。3 行目は `a[1][2]` で行列の 2 行目 3 列目の要素を取り出す。5 行目の結果は、`a[:]` でリスト全体が取り出され、`a[:][2]` でのインデックス 2 の要素を返すが、全行にわたって 3 列目の要素全体は返さない。

```
>>> import numpy as np
>>> a = [[1,2,3], [4,5,6], [7,8,9], [10,11,12]]
>>> a[1][2]
6
>>> a[:][2]
[7, 8, 9]
>>> na = np.array(a)
>>> na[:, 2]
array([ 3,  6,  9, 12])
>>> na[1, :]
```

```
11  array([4, 5, 6])
```

7 行目で `na = np.array(a)` とリストを ndarray 化すると、8 行目で行列の 3 列目全体、10 行目で 2 行目全体が `array` として取り出されていることがわかる。ndarry 化したときの行列要素 `A` のスライスの形は `A[j, k]` であって、`A[j][k]` でないことに注意する。

NumPy を使って求めた軌道要素リスト `orbit` を 3 行目で `nporbit = np.array(orbit)` と $N \times 3$ 行列として ndarray 化する。このとき、各列要素 `xorbit, yorbit, zorbit` を別途計算することなく、16 行目のようにして matplotlib で描かせることができる。

diff_euler.py プロット法修正

```
1   ....
2   orbit = euler_orbit(x0, T, dt, vector)
3   nporbit = np.array(orbit)
4
5   #xorbit = []
6   #yorbit = []
7   #zorbit = []
8   #for x, y, z in orbit:
9   #    xorbit.append(x)
10  #    yorbit.append(y)
11  #    zorbit.append(z)
12
13  # plot in 3-din space
14  fig = plt.figure()
15  ax = fig.gca(projection='3d')
16  ax.plot(nporbit[:, 0], nporbit[:, 1], nporbit[:, 2])
17  .....
```

演習 1.3　Euler 法を使って得られた軌道要素を上のように NumPy を使って ndarray 化して `diff_euler.py` を修正し、`T= 50, dt = 0.01` として得られる軌道要素を描きなさい。

1.4　Runge-Kutta 法

Euler 法は誤差が単純に蓄積し、真の解から急速に遠ざかってしまい、実用的には微分方程式の数値解としては精度は十分ではない。微分方程式の Runge-Kutta 数値解法は時間刻み `dt` の中間点を使って大幅に誤差の発生を抑える工夫がな微分方程式の数値計算法である。

次で定義した Python 関数 `runge_kutta_orbit(x0, T, dt, vectorfield)` は Runge-Kutta 法を使って解軌道の数値リストを求めている。`vectorfield` はベクトル場 $\boldsymbol{v}$ を定める m 個の関数リスト $[v_1, \ldots, v_m]$ である。時刻 t_0 における m 次元（2 行目の `width` に次元の値が格納される）の初期点リスト $x0 = [x_{01}, \ldots, x_{0m}]$ から出発して、時刻 t_0 から t_0+T までを指定した時間刻み `dt` から逐次的に定まる $N = \lfloor T/dt \rfloor$ 個からなる m 次元ベクトル値のリスト $[\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N]$ を返す。3 行目で リスト `x0` を `x` に代入し、while 文で更新した点を `orbit.append(list(x))` によって数値軌道リスト `orbit` に次々と追加している。

Euler 法とは異なり、while 文内で時刻 $t_n = n\Delta t$ に求めた値 $\boldsymbol{x}_n$ から次の時刻 $t_{n+1} = t_n + dt$ での値 $\boldsymbol{x}_{n+1}$ を求めるために、22,23 行目で中間の刻み幅 $dt/2$ を使った増分を計算した `k1, k2, k3, k4` を使っている（計算量は 4 倍になる）

```
1  def runge_kutta(x0, T, dt, vectorfield):
2      width = len(x0)
3      x = x0
4      t = 0.0
5      orbit = []
6      while t <= T:
7          orbit.append(list(x))
8          x1 = x
9          # python3 requests list() as below:
```

```
        k1 = list(map(lambda v: v(t, *x1), vectorfield))
        x2 = x
        for i in range(width):
            x2[i] += dt / 2.0 * k1[i]
        k2 = list(map(lambda v: v(t + dt / 2.0, *x2), vectorfield))
        x3 = x
        for i in range(width):
            x3[i] += dt / 2 * k2[i]
        k3 = list(map(lambda v: v(t + dt / 2.0, *x3), vectorfield))
        x4 = x
        for i in range(width):
            x4[i] += dt * k3[i]
        k4 = list(map(lambda v: v(t + dt, *x4), vectorfield))
        for i in range(width):
            x[i] += dt / 6.0 * (k1[i] + 2.0 * k2[i] + 2.0 * k3[i] + k4[i])
        t += dt
    return(orbit)
```

演習 1.4　先の diff_euler.py にならい、Runge-Kutta 法で Lorenz 方程式を数値計算する diff_runge.py を実行してみなさい。T= 50, dt = 0.01 としたときの軌道要素を 3 次元プロットしなさい。

ただし、Lorenz 系は次の関数で定義されるとし、Euler 法と同じく軌道要素を求めるとする。

```
# lorenz system
p, r, b = 10.0, 28.0, 8.0 / 3.0
def v1(t, x, y, z):
        return(-p * x + p * y)
def v2(t, x, y, z):
        return(-x * z + r * x - y)
def v3(t, x, y, z):
        return(x * y - b * z)
....
....
vector = [v1, v2, v3]
x0 = [0.1, 0.1, 0.1]
dt = 0.01
T = 20
orbit = runge_kutta(x0, T, dt, vector)
```

2　微分方程式をモジュール Scipy を使って解く

Python の拡張モジュール Scipy は Numpy と並んで広く科学技術計算で使われてる。Matplotlib を含めて標準モジュールではないが、

Python のグラフィックライブラリ matplotlib は 3 次元プロットを描くことができる。図 1 は、Runge-Kett 法で書いたスクリプトを lorenz_3d.py を差分刻み $dt = 0.005$、経過時間 $T = 20$ で実行した結果である。5 行目の通常のライブラリ matplotlib.pyplot に加えて、4 行目で mpl_toolkits.mplot3d から Axes3D をインポートしている。Axes3D は 74,75 行目で次のようにオブジェクト ax で使われている。なお、75 行目は 80(79) 行目で 3 次元プロットするために必要である。

lorenz_3d.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

# vector fields f1, f2, f3 of Lorenz eq as functions of t, x, y, z.
p, r, b = 10.0, 28.0, 8.0/3.0
```

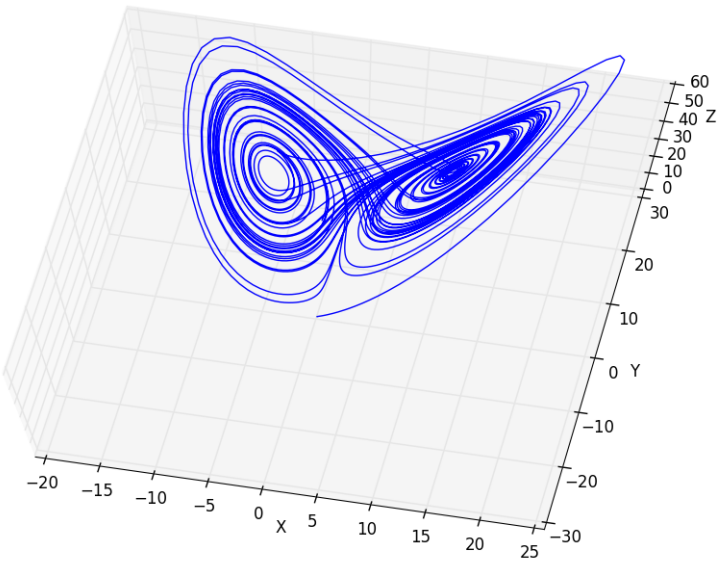

図 1　`lorenz_3d.py` の実行結果。初期値 $\boldsymbol{x}0 = (0.1, 0.1, 0.1)$, 差分刻み $dt = 0.005$、経過時間 $T = 20$. viewpoint はマウスで適宜変化できる。

```
9   def f1(t, x, y, z):
10      return -p * x + p * y
11  def f2(t, x, y, z):
12      return -x * z + r * x - y
13  def f3(t, x, y, z):
14      return x * y - b * z
15
16  def euler(x0, T, dt, f):
17      width = len(x0)
18      x = x0
19      t = 0
20      orbit = []
21      orbit.append(list(x))
22      while t <= T:
23          fx = map(lambda a: a(t, *x), f)
24          for i in range(width):
25              x[i] += dt * fx[i]
26          orbit.append(list(x))
27          #print x,
28          t += dt
29
30  def runge_kutta(x0, T, dt, f):
31      width = len(x0)
32      x = x0
33      t = 0
34      orbit = []
35      while t <= T:
36          orbit.append(list(x))
37          x1 = x
38          f1 = map(lambda a: a(t, *x1), f)
39          x2 = x
40          for i in range(width):
41              x2[i] += dt/2 * f1[i]
42          f2 = map(lambda a: a(t + dt/2, *x2), f)
43          x3 = x
44          for i in range(width):
45              x3[i] += dt/2 * f2[i]
46          f3 = map(lambda a: a(t + dt/2, *x3), f)
47          x4 = x
48          for i in range(width):
49              x4[i] += dt/2 * f3[i]
50          f4 = map(lambda a: a(t + dt/2, *x4), f)
51          for i in range(width):
```

```
            x[i] += dt/6 * (f1[i] + 2 * f2[i] + 2 * f3[i] + f4[i])
        t += dt
    return orbit


func = [f1, f2, f3]
x0 = [0.1, 0.1, 0.1]
dt = 0.005
T = 20

####  orbit = [[x0,y0,z0], [x1,y1,z1],[x2,y2,z2], ...]
#orbit = euler(x0, T, dt, func)
orbit = runge_kutta(x0, T, dt, func)

xorbit = []
yorbit = []
zorbit = []
for i in range(len(orbit)):
    xorbit.append(orbit[i][0])
    yorbit.append(orbit[i][1])
    zorbit.append(orbit[i][2])

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
#ax.scatter(xorbit, yorbit, zorbit, zdir='y')
ax.plot(xorbit, yorbit, zorbit, zdir='y')
plt.show()
```

3　数値解の誤差

時間刻み Δt を小さくすることが望ましいことは明かである。正しい結果は $\Delta t \to 0$ の極限で「のみ」もたらされるのであって、微分方程式の数値解から得られ結果は¡span class=”red”¿真の解¡/span¿ではない。しかし、Δt を小さくすればするほど、同じ計算法でも時間経過は遅くなり計算量が増す（それでも有限の時間刻みで計算している以上、誤差の存在から逃れられない）。

また、時間刻み Δt を一定にする積極的な理由もない。ベクトル場 $\boldsymbol{f}(\boldsymbol{x}, t)$ が急峻なときには、わずかな Δt であっても大きな変化 $\boldsymbol{x}(t + \Delta t) - \boldsymbol{x}(t)$ をもたらすため、そのような箇所では時間刻みを細かくすることが望ましい。たとえば、万有引力や Coulomb 力のように距離の二乗に反比例する力が働くとき場合である。仮に、¡strong¿衝突解¡/strong¿がもたらされる場合、その原因が数値誤差に起因するか、実際に起こり得る現象化を吟味することは容易いことではない。

常微分方程式の数値解法は、ここで取り上げた中間点を考慮して精度を向上させる Runge-Kutta 法など実用上広く用いられている方法がいくつかあるが、誤差限界は存在している。非線形方程式では、たとえば Lorenz 方程式のような単純な場合でも真の解の挙動が知られていないために、それ自体が研究対象になり得る（Smale(1998) の Mathematical Problems for the Next Century（Mathematical Intelligencer 20 (2): 7 - 15）の問題 14)。

実際、精度保証付き数値計算の研究は W. Tucker(2002) の A Rigorous ODE Solver and Smale’s 14th Problem によって非線形常微分方程式については一応の解決をみたが（参考：「精度保証付き数値計算の力学系への応用について)、偏微分方程式においては依然として大きな研究テーマであり続けている。明日に天気予報についても断定できるほどの確度はまだない。

4　Lorenz 方程式を研究する

初期条件を与えればそれ以降の挙動は決定論的に（解の存在と一意性があれば）一意的に定まってしまう微分方程式（力学系ともいう）の研究において、Edward N. Lorenz の Deterministic Nonperiodic Flow J.

Atmos. Sci., 20(11963), 130 - 141) は極めて大きな役割を果たした。ほんの僅かな初期条件の違いによって、その後の挙動が極めて鋭敏に依存して全く異なってしまうという**初期条件鋭敏性**（sensitive dependence on initial conditions）によって、事実上挙動が予想できないという**カオス**（chaos）研究の先鞭をつけた。

数値軌道を調べてみよう。ベクトル場のパラメータは以降 $p = 10.0$, $r = 28.04$, $b = 8.0/3$ とする。パラメータ依存性の研究では、$p = 10, b = 8/3$ を固定して、r を変化させるのが通例である。

4.1 Poincare の切断面の方法

この $1 < r$ のときには、原点 $(0, 0, 0)$ 以外に 2 つの不動点 $C_\pm$

$$C_\pm = (\pm\sqrt{b(r-1)}, \pm\sqrt{b(r-1)}, r-1)$$

が現れることが分かっている（z 成分は 2 点とも $r-1$ で、同じ高さにある）。真の軌道 $(x(t), y(t), z(t))$ は平面 $z = r-1$ で定まる xy-平面 $\Sigma_{z=r-1}$ を「下から上に」あるいは「上から下へ」と**横断**（transverse）する。たとえば、「下から上に」平面 $\Sigma_{z=r-1}$ を横断する xy-座標を求めてみよう。¡/p¿ ¡p¿ Lorenz 系の 3 つの軌道要素 $\boldsymbol{x} = (x, y, z)$ を持つ数値軌道リスト $[\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N]$ において、各要素の z-成分の列 $[z_0, z_1, \ldots, z_N]$ に注目する。各 z 要素から平面 $\Sigma_{z=r-1}$ の高さを引いたリスト

$$\begin{aligned}&[z_0 - (r-1), z_1 - (r-1), \ldots, z_N - (r-1)]\\ =&[s_0, s_1, \ldots, s_N], \qquad s_j = z_j - (r-1)\end{aligned}$$

を考える。

k 番目の点 $\boldsymbol{x}_k$ が平面 $\Sigma_{z=r-1}$ を「下から上に」横断するとは、$k+1$ 番目の点 z_{k+1} が次のように続いているときである：

$$z_k - (r-1) < 0 \text{ and } z_{k+1} - (r-1) < 0 \Leftrightarrow s_k > 0 \text{ and } s_k s_{k+1} < 0$$

ベクトル場が滑らかで時間刻みが十分に小さければ、そのような z-成分を持つ点 $\boldsymbol{x}_k$ は平面 $\Sigma_{z=r-1}$ にごく近いと見なすことができる。こうした z-成分を持つ点 $\boldsymbol{x}_k$ について (x, y) は軌道横断する平面 $\Sigma_{z=r-1}$ における交点 P であるとみなす。Lorenz 系の軌道を観察してみると、軌道は 2 つの不動点 $C_\pm$ の周りを回り続けており、平面 $\Sigma_{z=r-1}$ を何度でも横断していることがわかる。

周期軌道に点 $\boldsymbol{x}^*$ で横断的な平面 Σ を考える。平面 Σ 上にある点 $\boldsymbol{x}^*$ の十分近くから出発した軌道は再び Σ を同じ方向に横切る。このとき、微分方程式が定める流れは平面 Σ から Σ への写像

$$P : \Sigma \to \Sigma$$

を定めているという。周期軌道と Σ との交点 $\boldsymbol{x}^*$ は写像 P の不動点 $P(\boldsymbol{x}^*) = \boldsymbol{x}^*$ である。周期軌道に近傍で定義される平面 Σ 上の写像を **Poincare 写像**という。こうして、n 次元の微分方程式の研究は、周期軌道が存在するとき、それに横断的に横切る $n-1$ 次元横断面 Σ 上の Poincare 写像の研究に帰着される。

次のスクリプト `lorenz_section.py` は、時刻 $T = 100$, 差分刻み $dt = 0.05$ の Runge-Kutta 法で Lorenz 系の x, y, z-成分ごとの軌道リスト `xorbit`, `yorbit`, `zorbit` において、z-成分が平面 $\Sigma_{z=r-1}$ を「上から下に」横断すると見なせる時の x, y-成分の系列をそれぞれリスト `xsection`, `ysection` に求めて、xy-平面にプロットする（図 2）。

```
label=lorenz\_section.py]
import matplotlib.pyplot as plt
...
...
xorbit = []
yorbit = []
zorbit = []
for i in range(len(orbit)):
    xorbit.append(orbit[i][0])
    yorbit.append(orbit[i][1])
    zorbit.append(orbit[i][2])
```

```
12
13  xsection = []
14  ysection = []
15  for i in range(len(zorbit)-1):
16      if zorbit[i]-(r-1) > 0 and zorbit[i+1] -(r-1) < 0:
17          xsection.append(xorbit[i])
18          ysection.append(yorbit[i])
19
20  plt.xlabel('x')
21  plt.ylabel('y')
22  plt.plot(xsection, ysection,'rx')
23  plt.show()
```

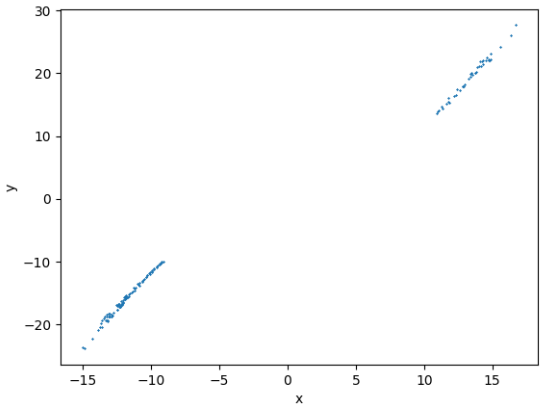

図 2　Lorenz 系の Poincare 横断面: 時刻 $T = 100$ までを差分刻み $dt = 0.05$、初期値 $(0.1, 0.1, 0.1)$ を出発する軌道をモジュール SciPy を使って求め、平面 $\Sigma_{z=r-1}$ を下から上に横切る横断面上の点列をプロット。

図 2 が示しているように、Lorenz が研究したパラメータで指定される Lorenz 系の軌道は '薄い'Lorenz バタフライに引き込まれていることがわかる。これを **Lorenz アトラクター**という。

演習 4.1　スクリプト `lorenz_section.py` を実行して、平面 $\Sigma_{z=r-1}$ を解軌道が上から下に横断する Poincare 断面をプロットしてみなさい。

4.2　Lorenz プロット

E. N. Lorenz は Deterministic Nonperiodic Flow J. Atmos. Sci., 20(11963), 130 - 141) において、今日では¡span class=”bold blue”¿Lorenz プロット¡/span¿として知られる力学系の対象次元をさらに減らす別の方法を提案した。原点近くを出発した軌道は、今のパラメータの設定値では、双曲不安定点である原点を離れて暫くすると（移行期を過ぎると)、二つの $z = r - 1$ の高さにある 2 つの不動点 $C_\pm$ の回りを不規則的に交互に周回する（何回かを不動点 C_+ または C_- の回りを何回転かしてから他方の不動点の回りを回り、これを交互に不規則に繰り返す)。

このように、解軌道 $\boldsymbol{x}(t) = (x_1(t), x_2(t), \ldots, x_n(t)$ が有限領域に留まって動いているとしよう。今、ある座標成分 x_i-成分 $x_i(t)$ の変化に着目すると、ある時刻 t_k があって、時刻 t_k をはさむ近傍時間 $t \in (t_k - \varepsilon, t_k + \varepsilon)$ で $x(t) < x(t_k)$ となるような時刻の系列 $(t_i)_{i\in\mathbb{N}} = t_1, t_2, \ldots, t_k, \ldots$、言い換えると、$x_i(t)$ が時刻 t_k ごとに局所的に極大値 x_k^* となる系列

$$x_1^*, x_2^*, \ldots, x_k^*, \ldots$$

を考えることができる。このとき、平面上に $(x_1^*, x_2^*), (x_2^*, x_3^*), \ldots, (x_{k-1}^*, x_k^*), (x_k^*, x_{k+1}^*), \ldots)$ をプロットして得られる様子を **Lorenz プロット**という。

Lorenz は、3 つの軌道要素 $\boldsymbol{x} = (x, y, z)$ を持つ軌道リスト $[\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N]$ において、z-成分の局所極大

値列に注目した。z 成分の列 $(z_0, z_1, \ldots, z_N)$ の隣り合う値の差 $dz_i = z_{i+1} - z_i$ の列

$$z_1 - z_0, z_2 - z_1, \ldots, z_{i+1} - z_i, \ldots$$

において、

$$z_{i-1} < z_i \text{ and } z_i > z_{i+1}$$

となる z_i 値を局所極大値 z_i^{max} とし、その極大値列を求め、その Lorenz プロット（たとえば z-成分）を取った。

その結果、局所的極大値の系列 (z_i) が、適当にスケーリングするとあたかも区間 $(0, 1]$ 上の傾き 2 のテント写像

$$T(x) = \begin{cases} 2x, & x \in (0, 1/2) \\ 2 - 2x, & x \in [1/2, 1) \end{cases}$$

に従っているかのように挙動していることを数値的な大発見をした。

次のスクリプト `lorenz_plot.py` は、時刻 $T = 100$, 差分刻み $dt = 0.05$ の Runge-Kutta 法で Lorenz 系の x, y, z-成分ごとの軌道リスト `xorbit`, `yorbit`, `zorbit` において z 成分のリスト `zorbit` に着目した Lorenz プロットを描く。点列 `zorbit` $= [z_0, z_1, \ldots, z_N]$ における隣り合う値の差 $\text{zdif}_i = z_{i+1} - z_i$ を計算しながら、局所極大 $z_{i-1} < z_i$ and $z_i > z_{i+1}$ となるような z_i 値の列 `zextremum` を求めている。ただし、Lorenz アトラクターに引き込まれるまでのトランジット期の挙動を対象から除くために、ステップ数 `transit = 2000` 以降についての局所極大値を求めている。

lorenz_plot.py

```
# -*- coding: utf-8 -*-
# numerila solution fo Lorenz system using scipy.integrate
# Lorenz section

import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# lorenz system
p, r, b = 10.0, 28.0, 8.0 / 3.0
def lorenz_system(x, t):
    vxt = -p * x[0] + p * x[1]
    vyt = -x[0] * x[2] + r * x[0] - x[1]
    vzt = x[0] * x[1] - b * x[2]
    return([vxt, vyt, vzt])

x0 = [0.1, 0.1, 0.1]# initial points
t0 = 0
T = 100
dt = 0.01

# numerila solution using scipy.integrate
times = np.arange(t0, T, dt)
orbit = odeint(lorenz_system, x0, times)
#print(orbit)
# Poincare surface of section
# z = r-1 の x-y 平面を（増加して、または減少して）横切るときの (x,y) を plot
xsectionlist = []
ysectionlist = []

zhight = [0, 0, r-1]
for k in range(len(orbit)-1):
    if (orbit[k] - zhight)[2] < 0 and (orbit[k] - zhight)[2] * (orbit[k+1] - zhight)[2] < 0:
#    if (orbit[k] - zhight)[2] * (orbit[k+1] - zhight)[2] < 0:
        xsectionlist.append(orbit[k,0])
```

```
36          ysectionlist.append(orbit[k,1])
37
38
39  #plot using matplotlib
40  ax = plt.axes()
41  plt.xlabel('x')
42  plt.ylabel('y')
43  plt.plot(xsectionlist, ysectionlist, "x", markersize=1)
44  plt.show()
```

上のスクリプト `lorenz_plot.py` で、21-23 行で z-成分の局所極大値の列

`zextremum` $= [z_{max,1}, \ldots, z_{max,k-1}, z_{max,k}, z_{max,k+1}, \ldots, z_{max,N_k}]$ を求めて、26 行から 29 行は

$$\texttt{zpair} = [\ldots, (z_{max,k-1}, z_{max,k}), (z_{max,k}, z_{max,k+1}), \ldots]$$

のようにして、隣り合う 2 点を組にして $\{(z_{max,k}, z_{max,i+k})\}$ がプロットされるように、ライブラリ matplotlib の使い方にあわせるために、その x-成分系列と y-成分系列をそれぞれ `zpairx`, zpairy のリストにしてから Lorenz プロットしている（図 3）。

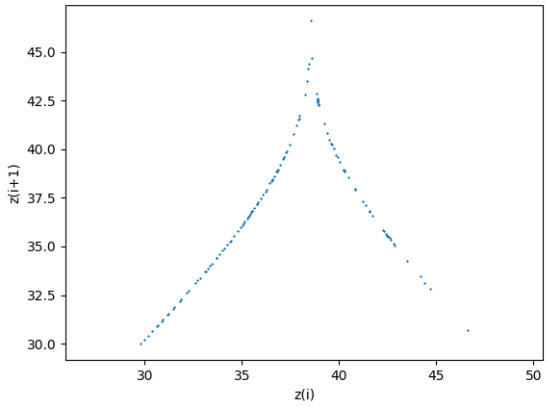

図 3　Lorenz プロット: $T = 100$, $dt = 0.005$, $(x_0, y_0, z_0) = (0.1, 0.1, 0.1)$

図 3 は 1 次元区間 I から I 自身へのテント型写像 L

$$L : I \to I$$

が存在することを強く示唆する。言い換えると、もしこの写像が存在するならば、$z_0 \in I$ からテント型関数によって、$z_{n+1} = Lz_n = L^n z_0$ のように L を反復適用して得られる点列 $z_0, z_1, z_2, \ldots$ が Lorenz 方程式の z 方向の局所極大値として得られることになる。このような写像の存在が明らかになれば、Lorenz 方程式の研究は 1 次元区間 I 上の写像 L の研究に帰着されることになる。

演習 4.2　上のスクリプト `lorenz_plot.py` を実行し、Lorenz プロットを描いてみなさい。