常微分方程式の数値解法

例えば、微分方程式

$$\frac{\mathrm{d}y}{\mathrm{d}t} = ay$$

の解析解は

$$y(t) = C\exp(at), \qquad C \text{ is constant}$$

と分かっている。後はこの関数が通る点を一点してやればCを一意に定めることが出来、具体的な解が得られる。

線形微分方程式はたいてい容易に解析的に解くことが出来るが、非線形な微分方程式は大抵容易には解析的に解くことが出来ない。特に、化学で扱うようなたくさんの粒子が相互作用するような多体問題はだいたい難しい。

そのような時には数値的に解くことで微分方程式の数値解を出すのが便利である。

微分方程式の一般形として、

$$\frac{dy}{dt} = f(t, y)$$

という形に落とし込むことが出来れば、Euler 法を用いて数値的に解くことが出来る。

例えば、先の$\frac{\mathrm{d}y}{\mathrm{d}t} = ay$の場合は$f(t, y) = ay$である。この表式は一階の微分方程式しか当てはめることが出来ないが、二階の微分方程式にも拡張可能である。

1 Euler 法

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y) \tag{1}$$

という形の微分方程式を考える。一階の微分方程式を解くには初期値が一つ必要なので、

$$y(t_0) = y_0$$

とする。これを解くには、yのテーラー展開を考える。

$$y(t_n + h) = y_n + \left.\frac{\mathrm{d}y}{\mathrm{d}t}\right|_{t_n} h + \left.\frac{1}{2}\frac{\mathrm{d}y^2}{\mathrm{d}^2 t}\right|_{t_n} h^2 + \left.\frac{1}{6}\frac{\mathrm{d}y^3}{\mathrm{d}^3 t}\right|_{t_n} h^3 + \cdots$$

これを一次までで打ち切り、さらに式 1 を代入して、$y(t_n + h)$をy_{n+1}とすると、

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{2}$$

となる。これを実装すると、以下のようになる。

```
def euler(t0, y0, f, h, step):
    ts = [t0]
    ys = [y0]
    for _ in range(step):
        y = ys[-1] + h * f(ts[-1], ys[-1])
        t = ts[-1] + h
        ts.append(t)
        ys.append(y)
    return np.array(ts), np.array(ys)
```

では、例として、この方法を微分方程式

$$\frac{\mathrm{d}y}{\mathrm{d}t} = 0.5y$$

に対して適用してみよう。

```
def f_exp(t, y):
    return 0.5 * y
```

この微分方程式の解析解は

$$y(t) = C\exp(0.5t), \qquad C \text{ is constant}$$

であるが、このうち$x = 0$で$y = 1$を通る解を求めてみよう。このような解析解は
$\mathrm{y} = \exp(0.5t)$であるので、解析解と数値解を比較してみよう。

```
ts, ys = euler(0, 1, f, 0.1, 100)
plt.plot(ts, ys)
plt.plot(ts, np.exp(0.5 * ts))
```

また、ステップ幅を増やしたり減らしたりして確認してみよう。

2　二階の微分方程式に対する Euler 法

Euler 法を使うには方程式を$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y)$の形にすることが必要である。しかし、例えば二階の微分方程式の場合はどうだろうか？

$$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = -\omega^2 y \tag{3}$$

この微分方程式の解は

$$\mathrm{y(t)} = C_1 \cos(\omega t) + C_2 \sin(\omega t)$$

である。ただし、ここでC_1とC_2は任意の複素数の定数である。

これを Euler 法で解くには式を連立方程式化するのが良い。式 3 は新しい変数$\dot{y}$を導入し、

$$\begin{cases} \dfrac{\mathrm{d}y}{\mathrm{d}t} = \dot{y} \\ \dfrac{\mathrm{d}\dot{y}}{\mathrm{d}t} = -\omega^2 y \end{cases}$$

と記述できる。これを$\frac{dy}{dt} = f(t, y)$の形式で書き表すならば、

$$\mathbf{y} = \begin{pmatrix} y \\ \dot{y} \end{pmatrix}$$

として、

$$\mathbf{f}(t, \mathbf{y}) = \begin{pmatrix} \dot{y} \\ -\omega^2 y \end{pmatrix}$$

と記述できる。式(2)はそのままベクトルに拡張され、

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(t, \mathbf{y}_n)$$

となる。このようなベクトルの場合でも先の Euler 法の実装は流用できるので、そのまま使ってみよう。

```
def f_sin(t, yyd):
    y, yd = yyd
```

```
    return np.array([yd, -0.5 * y])
```

```
ts, ys = euler(0, np.array([1, 0]), f_sin, 0.01, 10000)
plt.plot(ts, ys[:, 0])
plt.plot(ts, np.cos(np.sqrt(0.5) * ts))
```

ただし、ここで、ys はyと$\dot{y}$からなる行列であり、yのみをプロットしている。

3 Runge-Kutta 法

Euler 法はそれなりに良い微分方程式の解を表現するが、exp の場合も sin の場合も曲率が足りず、元の関数からずれていく。これは Euler 法が一次のテーラー展開を用いたものであることに由来する。

次はもっと精度が高く、かつそれほど複雑でない例として古典ルンゲクッタ法(RK4)を紹介しよう。RK4 の式は Euler 法の場合と同じように、微分方程式を

$$\frac{dy}{dt} = f(t, y)$$

のように表現して利用する。Euler 法と違い、RK4 では以下の更新式を用いる。

$$t_{n+1} = t_n + h$$

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

ただし、k_1~k_2は

$$k_1 = f(t_n, y_n)$$

$$k_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right)$$

$$k_3 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right)$$

$$k_4 = f(t_n + h, y_n + hk_3)$$

である。実装してみよう。

```
def rk4(t0, y0, f, h, step):
    ts = [t0]
    ys = [y0]
    for _ in range(step):
        k1 = f(ts[-1], ys[-1])
        k2 = f(ts[-1] + 0.5 * h, ys[-1] + 0.5 * h * k1)
        k3 = f(ts[-1] + 0.5 * h, ys[-1] + 0.5 * h * k2)
        k4 = f(ts[-1] + h, ys[-1] + h * k3)
        y = ys[-1] + h * (k1 + 2 * k2 + 2 * k3 + k4) / 6
        t = ts[-1] + h
        ts.append(t)
        ys.append(y)
    return np.array(ts), np.array(ys)
```

ではこれを先ほどと同じように sin や exp に使ってみよう。

```
ts, ys = rk4(0, 1, f_exp, 0.1, 100)
plt.plot(ts, ys)
plt.plot(ts, np.exp(0.5 * ts))
```

```
ts, ys = rk4(0, np.array([1, 0]), f_sin, 0.01, 10000)
plt.plot(ts, ys[:, 0])
plt.plot(ts, np.cos(np.sqrt(0.5) * ts))
```

どうだろうか？ 同じ h の Euler 法の場合と比較してみよう。

$\frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$の係数の乗せ方は数値積分の章で扱った台形公式や紹介にとどめたシンプソン公式と深い関わりがあるが、今回はその詳細に立ち入ることはしない。

Euler 法も RK4 も基本的にテーラー展開に従って次の値を決めている。

$$y(t_n + h) = y_n + \left.\frac{\mathrm{d}y}{\mathrm{d}t}\right|_{t_n} h + \left.\frac{1}{2}\frac{\mathrm{d}y^2}{\mathrm{d}^2 t}\right|_{t_n} h^2 + \left.\frac{1}{6}\frac{\mathrm{d}y^3}{\mathrm{d}^3 t}\right|_{t_n} h^3 + \cdots$$

しかし、RK4 の公式はyの二次以降の微分をあらわに用いることなく、テーラー展開にして 4 次の精度を持つように設計されている。（Euler 法が 1 次の精度であったことを考えると驚くべき精度の向上である。）

ルンゲクッタ法には今回紹介した RK4 の他にもさまざまな方法が存在する。

大きく分けて陽的ルンゲクッタ法と陰的ルンゲクッタ法という二種類の方法がある。

陽的ルンゲクッタの方が速いが、発散してしまいやすい。

良く知られている陽的ルンゲクッタの中で最も精度が良いのは Dormand-Prince 法と呼ばれる 5 次の方法である。

4 調和振動子

一次元調和振動子を解いてみよう。ポテンシャル

$$V(x) = \frac{1}{2}kx^2$$

を考えよう。ただし k はバネ定数である。このポテンシャル中で働く力は

$$F(x) = -kx$$

である。さて、このポテンシャルの中を動く粒子の座標をxとする。また、粒子の速度をvとしよう。速度の定義より、

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v$$

運動方程式$F = ma$より、

$$\frac{\mathrm{d}v}{\mathrm{d}t} = a = \frac{F}{m} = -\frac{k}{m}x$$

が成り立つ。従って、Euler 法は

$$\begin{cases} x_{n+1} = x_n + hv_n \\ v_{n+1} = v_n - h\dfrac{k}{m}x_n \end{cases}$$

となる。

$$\mathbf{xv}_n = \begin{pmatrix} x_n \\ v_n \end{pmatrix}$$

として、

$$\mathbf{f}(\mathbf{xv}_n) = \begin{pmatrix} x_n + hv_n \\ v_n - h\dfrac{k}{m}x_n \end{pmatrix}$$

とすればよい。

```
m = 1
k = 2

def acceleration(x):
    return - k / m * x

def f(t, xv):
    x, v = xv
    return np.array([v, acceleration(x)])
```

では計算してみよう。

```
ts, xvs = rk4(0, np.array([2, 0]), f, 0.01, 10000)
```

さて、xvs は座標と速度がまとまったベクトルであり、xvs[:, 0]が座標からなるベクトルであるので、これを保存しよう。open で保存する方法もあるが、今回は np.save 関数を紹介しよう。

```
np.save("harmonics1d", ys[:, 0])
```

この関数を用いることで numpy の配列をそのまま保存することが出来る。保存したファイルをアニメーションする Python スクリプト renderer1.py を用意したので、ダウンロードして実行してみて欲しい。このスクリプトは三つの引数を取り、以下のように実行する。

```
>>>python renderer1.py harmonics1d.npy 2 0.1
```

第一引数の「harmonics1d.npy」は保存したファイルの名前であり、第二引数の 2 はシミュレーションに用いたバネ定数。第三引数の 0.1 はアニメーションの速度を示す。第三引数が小さい程高速に再生される。

5　二次元重力系

二次元の重力系を考える。中心に質量 M の星が存在する時に周りの物質が受ける加速度は

$$\boldsymbol{a} = -\frac{M}{4\pi|r|^3}\mathbf{r}$$

```
def central_acceleration(M, r):
    return - r * M / (np.linalg.norm(r) ** 3)
```

となる。ただし、$\mathbf{r} = (x \quad y)$である。また、速度を$\mathbf{v}$とすると、

$$\begin{cases} \dfrac{\partial \mathbf{r}}{\partial t} = \mathbf{v} \\ \dfrac{\partial \mathbf{v}}{\partial t} = -\dfrac{M}{4\pi|r|^3}\mathbf{r} \end{cases}$$

となる。これを numpy の一次元配列で表現するために、

$$\mathbf{rv} = \begin{pmatrix} x \\ y \\ v_x \\ v_y \end{pmatrix}$$

としよう。$\mathbf{rv}$が従う微分方程式を

$$\frac{d\mathbf{rv}}{dt} = f(t, \mathbf{rv})$$

の形式で書き表すと、

$$f(t, \mathbf{rv}) = \begin{pmatrix} v_x \\ v_y \\ -\dfrac{Mx}{4\pi(x^2+y^2)^{3/2}} \\ -\dfrac{My}{4\pi(x^2+y^2)^{3/2}} \end{pmatrix} = \begin{pmatrix} \mathbf{v} \\ -\dfrac{M}{4\pi|r|^3}\mathbf{r} \end{pmatrix}$$

のようになる。すこしややこしいが、これをコードに落とすと以下のようになる。

```
def f_central(t, rv):
    n = len(rv)
    r = rv[:n//2]
    v = rv[n//2:]
    return np.concatenate([v, central_acceleration(M, r)])
```

さて、この関数を使って、微分方程式を解いてみよう。

```
m = 1
M = 10


r0 = np.array([5.0, 0.0])
v0 = np.array([0.0, 0.5])
ts, rs = rk4(0, np.concatenate([r0, v0]), f_central, 0.01, 10000)
```

では、保存しよう。ただし、得られて rs の前半は位置に関する部分なので、ここだけを保存しよう。

```
np.save("central.npy", rs[:, :rs.shape[1]//2])
```

これを再生するために、renderer2.py を用意した。保存して実行して見て欲しい。

```
>>> python renderer2.py central.npy 100 0.1
```

第一引数と第三引数の意味は renderer2.py と同じであるが、第二引数は軌跡のステップ数を示す。では実行してみて欲しい。また、初期値の r0 と v0 を適宜変更して様子を見て欲しい。

6 多粒子の二次元重力系

これまで解いてきた系は全て解析的に解ける系であった。しかし、これから取り扱う多粒子の系は解析的に扱うことが出来ず、それ故に Runge-kutta 法が威力を発揮する系である。原点に限らず多粒子が存在する場合の各質点にかかる加速度は

$$\mathbf{a}_i = -\sum_{j \neq i} \frac{M_j}{4\pi \left|\mathbf{r}_i - \mathbf{r}_j\right|^3} \left(\mathbf{r}_i - \mathbf{r}_j\right)$$

で表される。この式は以下のコードに落とし込まれる。

```
def central_accelerations(M, r):
```

```
    a = np.zeros_like(r)
    for i in range(len(M)):
        for j in range(len(M)):
            if i != j:
                a[i, :] += central_acceleration(M[j], r[i, :] - r[j, :])
    return a
```

また、微分方程式は

$$\mathbf{rv} = \begin{pmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \\ \vdots \\ \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \end{pmatrix}$$

として、

$$\frac{d\mathbf{rv}}{dt} = f(t, \mathbf{rv}) = \begin{pmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{a}_1 \\ \mathbf{a}_2 \\ \vdots \end{pmatrix}$$

で表される。これは以下のようなコードに落とし込まれる。

```
def f(t, rv):
    n = len(rv)
    r = rv[:n//2]
    v = rv[n//2:]
    return np.concatenate([v, central_accelerations(M, r.reshape([-1,
2])).flatten()])
```

concatenate は二つの配列を結合する関数である。では、実行してみて欲しい。

```
M = [100.0, 10.0, 1.0]
r0 = np.array([[0.0, 0.0],
               [0.0, 5.0],
               [0.0, 5.5]
               ]).flatten()
```

```
v0 = np.array([[-0.4, 0.0],
               [ 4.0, 0.0],
               [8.0, 0.0]
               ]).flatten()
ts, rs = rk4(0, np.concatenate([r0, v0]), f, 0.01, 4000)
np.save("satellite.npy", rs[:, :rs.shape[1]//2])
```

では、これもアニメーションを作成してみよう。

```
>>> python renderer2.py central.npy 100 0.1
```