

計算物理演習レポート 2 数値計算

1 概要

2 体問題の数値計算を行った。数値計算の手法としては、Euler 法、Leapfrog 法、古典 4 次 Runge-Kutta 法を用いた。また、各値の計算には C++ 14、プロットには gnuplot を使用した。

2 Euler 法

2.1 内容

2 体問題の数値計算を Euler 法で行った。

2.2 ソースコード

euler/n.cpp

```
1 #include <string>
2 #include "../util.h"
3
4 using namespace std;
5
6 int main() {
7     const double dt = 0.1, G = 1.0, M =
8         1.0, m = 1.0;
9     int n = 2000;
10    double x[n], y[n], v_x[n], v_y[n];
11    x[0] = 0.5, y[0] = 0.0, v_x[0] =
12        0.0, v_y[0] = 1.63;
13    for (int i = 0; i < n - 1; i++) {
14        x[i + 1] = x[i] + v_x[i] * dt;
15        y[i + 1] = y[i] + v_y[i] * dt;
16        double r = sqrt(pow(x[i], 2) +
17            pow(y[i], 2));
18        v_x[i + 1] = v_x[i] - G * M * m
19            * x[i] / (pow(r, 3) * m) *
20            dt;
21        v_y[i + 1] = v_y[i] - G * M * m
22            * y[i] / (pow(r, 3) * m) *
23            dt;
24    }
25    string s;
26    for (int i = 0; i < n; i++) {
27        s += to_string(x[i]) + " " +
28            to_string(y[i]) + "\n";
29    }
30 }
```

```
22     string fpath = "../out/euler/n.dat";
23     write_to_file(s, fpath);
24
25     // gnuplot で
26     // ''set size ratio -1
27     // plot 'n.dat' with points pointtype
28         0''
29     // でプロット
30     return 0;
31 }
```

util.cpp

```
1 // #include <random>
2 #include "Sample.h"
3
4 using namespace std;
```

util.h

```
1 #pragma once
2
3 #include <fstream>
4 #include <random>
5
6 using namespace std;
7
8 inline double get_random_0_to_1() {
9     // 乱数生成器
10    static mt19937_64 mt64(0);
11
12    // [0.0, 1.0) の一様分布実数生成器
13    uniform_real_distribution<double>
14        get_rand_uni_real(0.0, 1.0);
15    // 乱数を生成
16    return get_rand_uni_real(mt64);
17 }
18
19 inline int write_to_file(const string& s,
20     const string& fpath) {
21     ofstream f;
22     f.open(fpath);
23     f << s;
24     f.close();
25 }
```

```
23 return 0;
24 }
```

2.3 結果

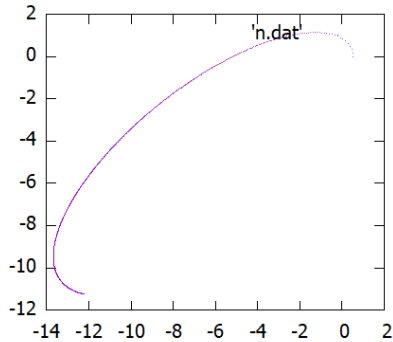


図1 $dt = 0.1, n = 1000$ とした場合

dt は 0.1 のまま、 n をもう少し大きくしたとき (5000 秒先まで追ったとき)

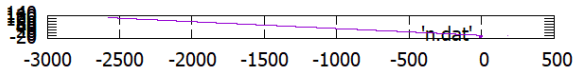


図2 $dt = 0.1, n = 50000$ とした場合

dt は 0.1 のまま、 n を少しだけ大きくしたとき (200 秒先まで追ったとき)

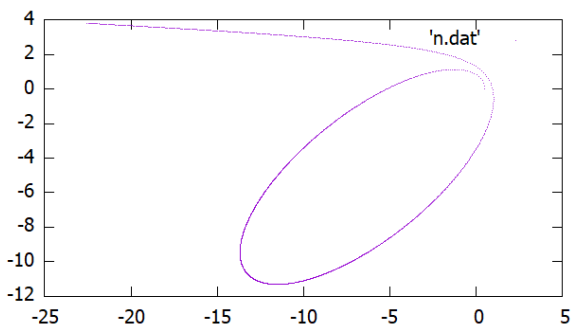


図3 $dt = 0.1, n = 2000$ とした場合

dt をもう少し小さくしたとき ($dt = 0.01$ で時刻は同じ 200 秒まで)

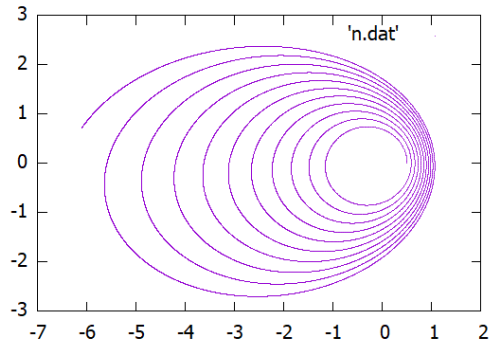


図4 $dt = 0.01, n = 20000$ とした場合

2.4 考察

徐々に半径が変わっているが、これは Euler 法がエネルギーを保存しないためと考えられる。また、 dt を小さくすればより精度が高くなっているが、これは $\frac{\Delta x}{\Delta t}$ を小さくすればより $\frac{dx}{dt} = \lim_{\Delta t \rightarrow 0} \frac{\Delta x}{\Delta t}$ に近づくためと考えられる。

3 Leapfrog 法

3.1 内容

同じ 2 体問題の数値計算を Leapfrog 法でも行った。

3.2 ソースコード

leap-frog/n.cpp

```
1 #include <string>
2 #include "../util.h"
3
4 using namespace std;
5
6 int main() {
7     const int tmax = 100;
8     const int n = 1000;
9     const double dt = double(tmax) /
10         double(n), G = 1.0, M = 1.0, m
11         = 1.0;
12     double x[2 * n], y[2 * n], v_x[2 *
13         n], v_y[2 * n];
14     x[0] = 0.5, y[0] = 0.0, v_x[0] =
15         0.0, v_y[0] = 1.63;
16
17     double r = sqrt(pow(x[0], 2) + pow(y
18         [0], 2));
19     v_x[1] = v_x[0] - G * M * m * x[0]
20         / (pow(r, 3) * m) * dt * 0.5;
21     v_y[1] = v_y[0] - G * M * m * y[0]
22         / (pow(r, 3) * m) * dt * 0.5;
23     for (long i = 0; i < n - 1; i++) {
24         x[2 * (i + 1)] = x[2 * i] + v_x
```

```

    [2 * i + 1] * dt;
18     y[2 * (i + 1)] = y[2 * i] + v_y
        [2 * i + 1] * dt;
19     r = sqrt(pow(x[2 * (i + 1)], 2)
        + pow(y[2 * (i + 1)], 2));
20     v_x[2 * i + 3] = v_x[2 * i + 1]
        - G * M * m * x[2 * (i +
        1)] / (pow(r, 3) * m) * dt;
21     v_y[2 * i + 3] = v_y[2 * i + 1]
        - G * M * m * y[2 * (i +
        1)] / (pow(r, 3) * m) * dt;
22 }
23 string s;
24 for (int i = 0; i < n; i++) {
25     s += to_string(x[2 * i]) + " "
        + to_string(y[2 * i]) + "\n";
26 }
27 string fpath = "../out/leap-frog/n.
    dat";
28 write_to_file(s, fpath);
29
30 // gnuplot で
31 // ``set size ratio -1
32 // plot 'n.dat' with points pointtype
    0``
33 // でプロット
34     return 0;
35 }

```

util.h、util.cpp は 2 と同一である。

3.3 結果

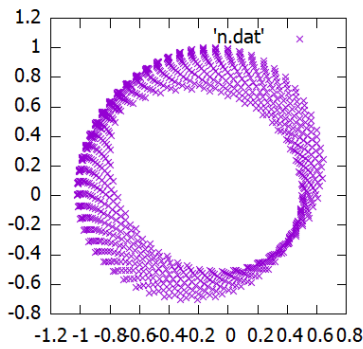


図5 $dt = 0.1, n = 1000, v_y = 1.63$ とした場合

dt は 0.1 のまま、 n をもう少し大きくしたとき (5000 秒先まで追ったとき)

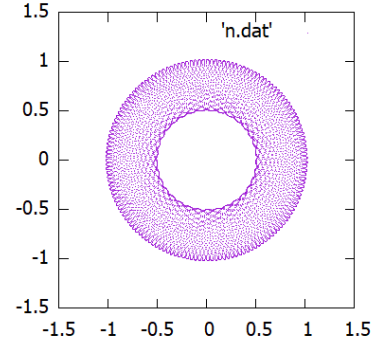


図6 $dt = 0.1, n = 50000, v_y = 1.63$ とした場合

3.4 考察

Euler 法と違い Leapfrog 法はエネルギーを保存するので、円の半径はほぼ一定に保たれているように見える。

4 古典 4 次 Runge-Kutta 法

4.1 内容

テキストにはないが、同じ 2 体問題の数値計算を古典 4 次 Runge-Kutta 法 (RK4) でも行った。

4.2 計算方法

運動方程式

$$m \frac{d^2 x}{dt^2} = -\frac{GMmx}{r^3} \quad (1)$$

$$m \frac{d^2 y}{dt^2} = -\frac{GMmy}{r^3} \quad (2)$$

は次のような x 、 y 、 v_x 、 v_y についての連立常微分方程式で書ける。

$$\frac{dx}{dt} = v_x \quad (3)$$

$$\frac{dy}{dt} = v_y \quad (4)$$

$$\frac{dv_x}{dt} = -\frac{GMmx}{mr^3} \quad (5)$$

$$\frac{dv_y}{dt} = -\frac{GMmy}{mr^3} \quad (6)$$

ここで、各方程式の右辺をそれぞれ

$$\frac{dx}{dt} = f(t, x, y, v_x, v_y) \quad (7)$$

$$\frac{dy}{dt} = g(t, x, y, v_x, v_y) \quad (8)$$

$$\frac{dv_x}{dt} = h(t, x, y, v_x, v_y) \quad (9)$$

$$\frac{dv_y}{dt} = i(t, x, y, v_x, v_y) \quad (10)$$

と置くと、 t_{i+1} 、 x_{i+1} 、 y_{i+1} 、 $v_{x_{i+1}}$ 、 $v_{y_{i+1}}$ は t_i 、 x_i 、 y_i 、 v_{x_i} 、 v_{y_i} を使って次のように求められる。

$$t_{i+1} = t_i + dt \quad (11)$$

$$x_{i+1} = x_i + \frac{dt}{6} (k_0 + 2k_1 + 2k_2 + k_3) \quad (12)$$

$$y_{i+1} = y_i + \frac{dt}{6} (l_0 + 2l_1 + 2l_2 + l_3) \quad (13)$$

$$v_{x_{i+1}} = v_{x_i} + \frac{dt}{6} (m_0 + 2m_1 + 2m_2 + m_3) \quad (14)$$

$$v_{y_{i+1}} = v_{y_i} + \frac{dt}{6} (n_0 + 2n_1 + 2n_2 + n_3) \quad (15)$$

ただし、

$$k_0 = f(t_i, x_i, y_i, v_{x_i}, v_{y_i}) \quad (16)$$

$$l_0 = g(t_i, x_i, y_i, v_{x_i}, v_{y_i}) \quad (17)$$

$$m_0 = h(t_i, x_i, y_i, v_{x_i}, v_{y_i}) \quad (18)$$

$$n_0 = i(t_i, x_i, y_i, v_{x_i}, v_{y_i}) \quad (19)$$

$$k_1 = f\left(t_i + \frac{dt}{2}, x_i + \frac{dtk_0}{2}, y_i + \frac{dtl_0}{2}, v_{x_i} + \frac{dtm_0}{2}, v_{y_i} + \frac{dtn_0}{2}\right) \quad (20)$$

$$l_1 = g\left(t_i + \frac{dt}{2}, x_i + \frac{dtk_0}{2}, y_i + \frac{dtl_0}{2}, v_{x_i} + \frac{dtm_0}{2}, v_{y_i} + \frac{dtn_0}{2}\right) \quad (21)$$

$$m_1 = h\left(t_i + \frac{dt}{2}, x_i + \frac{dtk_0}{2}, y_i + \frac{dtl_0}{2}, v_{x_i} + \frac{dtm_0}{2}, v_{y_i} + \frac{dtn_0}{2}\right) \quad (22)$$

$$n_1 = i\left(t_i + \frac{dt}{2}, x_i + \frac{dtk_0}{2}, y_i + \frac{dtl_0}{2}, v_{x_i} + \frac{dtm_0}{2}, v_{y_i} + \frac{dtn_0}{2}\right) \quad (23)$$

$$k_2 = f\left(t_i + \frac{dt}{2}, x_i + \frac{dtk_1}{2}, y_i + \frac{dtl_1}{2}, v_{x_i} + \frac{dtm_1}{2}, v_{y_i} + \frac{dtn_1}{2}\right) \quad (24)$$

$$l_2 = g\left(t_i + \frac{dt}{2}, x_i + \frac{dtk_1}{2}, y_i + \frac{dtl_1}{2}, v_{x_i} + \frac{dtm_1}{2}, v_{y_i} + \frac{dtn_1}{2}\right) \quad (25)$$

$$m_2 = h\left(t_i + \frac{dt}{2}, x_i + \frac{dtk_1}{2}, y_i + \frac{dtl_1}{2}, v_{x_i} + \frac{dtm_1}{2}, v_{y_i} + \frac{dtn_1}{2}\right) \quad (26)$$

$$n_2 = i\left(t_i + \frac{dt}{2}, x_i + \frac{dtk_1}{2}, y_i + \frac{dtl_1}{2}, v_{x_i} + \frac{dtm_1}{2}, v_{y_i} + \frac{dtn_1}{2}\right) \quad (27)$$

$$k_3 = f(t_i + dt, x_i + dtk_2, y_i + dtl_2, v_{x_i} + dtm_2, v_{y_i} + dtn_2) \quad (28)$$

$$l_3 = g(t_i + dt, x_i + dtk_2, y_i + dtl_2, v_{x_i} + dtm_2, v_{y_i} + dtn_2) \quad (29)$$

$$m_3 = h(t_i + dt, x_i + dtk_2, y_i + dtl_2, v_{x_i} + dtm_2, v_{y_i} + dtn_2) \quad (30)$$

$$n_3 = i(t_i + dt, x_i + dtk_2, y_i + dtl_2, v_{x_i} + dtm_2, v_{y_i} + dtn_2) \quad (31)$$

である。

4.3 ソースコード

runge-kutta/n.cpp

```
1 #include <string>
2 #include "../util.h"
3
4 using namespace std;
5
6 #define tmax 100
7 #define n 1000
```

```
8 #define dt double(tmax) / double(n)
9 #define G 1.0
10 #define M 1.0
11 #define m 1.0
12
13 double f_dx_dt(double t, double _x,
14                 double _y, double _v_x, double _v_y)
15 {
16     return _v_x;
17 }
18
19 double g_dy_dt(double t, double _x,
20                 double _y, double _v_x, double _v_y)
21 {
22     return _v_y;
23 }
24
25 double h_dv_x_dt(double t, double _x,
26                  double _y, double _v_x, double _v_y)
27 {
28     double r = sqrt(pow(_x, 2) + pow(_y, 2));
29     return -G * M * m * _x / (pow(r, 3) * m);
30 }
31
32 double i_dv_y_dt(double t, double _x,
33                  double _y, double _v_x, double _v_y)
34 {
35     double r = sqrt(pow(_x, 2) + pow(_y, 2));
36     return -G * M * m * _y / (pow(r, 3) * m);
37 }
38
39 int main() {
40     double t[n], x[n], y[n], v_x[n], v_y[n];
41
42     t[0] = 0.0, x[0] = 0.5, y[0] = 0.0, v_x[0] = 0.0, v_y[0] = 1.63;
43
44     for (int i = 0; i < n - 1; i++) {
45         double k0 = f_dx_dt(t[i], x[i], y[i], v_x[i], v_y[i]);
46         double l0 = g_dy_dt(t[i], x[i], y[i], v_x[i], v_y[i]);
47         double m0 = h_dv_x_dt(t[i], x[i], y[i], v_x[i], v_y[i]);
48         double n0 = i_dv_y_dt(t[i], x[i], y[i], v_x[i], v_y[i]);
49         double k1 = f_dx_dt(t[i] + dt / 2.0, x[i] + dt / 2.0 * k0,
50                             y[i] + dt / 2.0 * l0, v_x[i] + dt / 2.0 * m0, v_y[i] + dt / 2.0 * n0);
```

```

41 double l1 = g_dy_dt(t[i] + dt / (10 + 2.0 * l1 + 2.0 * l2 +
    2.0, x[i] + dt / 2.0 * k0, 13);
    y[i] + dt / 2.0 * l0, v_x[i] 55 v_x[i + 1] = v_x[i] + (dt /
    ] + dt / 2.0 * m0, v_y[i] + 6.0) * (m0 + 2.0 * m1 + 2.0
    dt / 2.0 * n0); * m2 + m3);
42 double m1 = h_dv_x_dt(t[i] + dt 56 v_y[i + 1] = v_y[i] + (dt /
    / 2.0, x[i] + dt / 2.0 * k0 6.0) * (n0 + 2.0 * n1 + 2.0
    , y[i] + dt / 2.0 * l0, v_x * n2 + n3);
    [i] + dt / 2.0 * m0, v_y[i] 57 }
    + dt / 2.0 * n0); 58 string s;
43 double n1 = i_dv_y_dt(t[i] + dt 59 for (int i = 0; i < n; i++) {
    / 2.0, x[i] + dt / 2.0 * k0 60 s += to_string(x[i]) + " " +
    , y[i] + dt / 2.0 * l0, v_x to_string(y[i]) + "\n";
    [i] + dt / 2.0 * m0, v_y[i] 61 }
    + dt / 2.0 * n0); 62 string fpath = "../out/runge-kutta/n
44 double k2 = f_dx_dt(t[i] + dt / .dat";
    2.0, x[i] + dt / 2.0 * k1, 63 write_to_file(s, fpath);
    y[i] + dt / 2.0 * l1, v_x[i] 64
    ] + dt / 2.0 * m1, v_y[i] + 65 // gnuplot で
    dt / 2.0 * n1); 66 // ``set size ratio -1
45 double l2 = g_dy_dt(t[i] + dt / 67 // plot 'n.dat' with points pointtype
    2.0, x[i] + dt / 2.0 * k1, 0``
    y[i] + dt / 2.0 * l1, v_x[i] 68 // でプロット
    ] + dt / 2.0 * m1, v_y[i] + 69 return 0;
    dt / 2.0 * n1); 70 }
46 double m2 = h_dv_x_dt(t[i] + dt
    / 2.0, x[i] + dt / 2.0 * k1
    , y[i] + dt / 2.0 * l1, v_x
    [i] + dt / 2.0 * m1, v_y[i]
    + dt / 2.0 * n1);
47 double n2 = i_dv_y_dt(t[i] + dt
    / 2.0, x[i] + dt / 2.0 * k1
    , y[i] + dt / 2.0 * l1, v_x
    [i] + dt / 2.0 * m1, v_y[i]
    + dt / 2.0 * n1);
48 double k3 = f_dx_dt(t[i] + dt, x
    [i] + dt * k2, y[i] + dt *
    l2, v_x[i] + dt * m2, v_y[i]
    + dt * n2);
49 double l3 = g_dy_dt(t[i] + dt, x
    [i] + dt * k2, y[i] + dt *
    l2, v_x[i] + dt * m2, v_y[i]
    + dt * n2);
50 double m3 = h_dv_x_dt(t[i] + dt,
    x[i] + dt * k2, y[i] + dt
    * l2, v_x[i] + dt * m2, v_y[
    i] + dt * n2);
51 double n3 = i_dv_y_dt(t[i] + dt,
    x[i] + dt * k2, y[i] + dt
    * l2, v_x[i] + dt * m2, v_y[
    i] + dt * n2);
52 t[i + 1] = t[i] + dt;
53 x[i + 1] = x[i] + (dt / 6.0) *
    (k0 + 2.0 * k1 + 2.0 * k2 +
    k3);
54 y[i + 1] = y[i] + (dt / 6.0) *

```

util.h, util.cpp は 2 と同一である。

4.4 結果

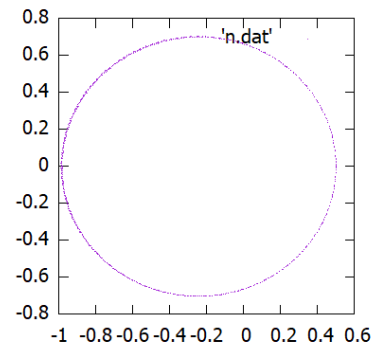


図 7 $dt = 0.1, n = 1000, v_y = 1.63$ とした場合

dt は 0.1 のまま、 n をもう少し大きくしたとき (5000 秒先まで追ったとき)

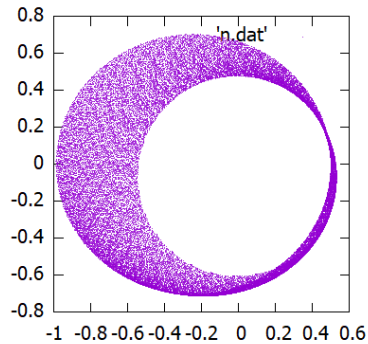


図 8 $dt = 0.1, n = 50000, v_y = 1.63$ とした場合

4.5 考察

古典 4 次 Runge-Kutta 法は 4 次精度なので、Euler 法や Leapfrog 法に比べ同じ $dt = 0.1$ でもかなり精度がよくなっており、軌道がほとんどずれていない。

ただし、5000 秒先まで追うとどんどん軌道半径が小さくなってしまった。これは、ただ軌道半径がほぼ一定に保たれて位置がずれるだけの Leapfrog 法とは違って、古典 4 次 Runge-Kutta 法がエネルギーを保存しないためと思われる。

参考文献

- [1] ルンゲクッタ法による常微分方程式の数値解法 埼玉工業大学
<https://www.sit.ac.jp/user/konishi/JPN/L.Support/SupportPDF/Runge-KuttaMethod.pdf>
- [2] ルンゲ = クッタ 法 - Wikipedia
<https://ja.wikipedia.org/wiki/ルンゲ = クッタ法>
- [3] N 体シミュレーションの基礎 道越秀吾 国立天文台 CfCA https://www.cfca.nao.ac.jp/cfca/hpc/mu-v/text/michikoshi_12.pdf
- [4] suti-sekibun-bibun.pdf
https://www.sci.kagoshima-u.ac.jp/fujii/data_kougi/suti-sekibun-bibun.pdf
- [5] 古典 4 次 Runge-Kutta 法の精度確認 - Qiita
<https://qiita.com/kaityo256/items/e3428deb394b3ad1e739>