数値積分と数値微分（発展）

重田 出

講義・演習の目標

ルンゲクッタ法で常微分方程式の初期値問題を解く。

6　ルンゲクッタ法で微分方程式を解く

常微分方程式の初期値問題を解く場合に広く使われている方法として，ルンゲクッタ（Runge-Kutta）法がある。Eular 法では，細かい刻みに分けたとき，各刻みの始点での傾きを用いて終点の値を決めている。これに対し，Runge-Kutta 法では，一旦求めた値を使って傾きの修正を行う操作を組み込み，終点の決定精度を上げている。以下に，Runge-Kutta 法の手順を示す。ある量 A(スカラーでもベクトルでも可) の時間発展の微分方程式が，既知関数 $f(t,x)$ により，

$$\frac{dx}{dt} = \lim_{\Delta t \to 0} \frac{x(t+\Delta t) - x(t)}{\Delta t} = f(t,x) \tag{14}$$

で与えられているとし，t での x の値，すなわち初期値 $x(t)$ を与えて，$t+dt$ での x の値 $x(t+dt)$ を計算するとする。Euler 法では，図 7 の点線のように，t での傾き（1 次微係数）$f(t,x(t))$ を使い，$x(t)+f(t+dt,x(t+dt))dt$ として一次直線でいきなり $t+dt$ での値 $x(t+dt)$ を計算している。式 (14) を変形した $x(t+\Delta t) =$

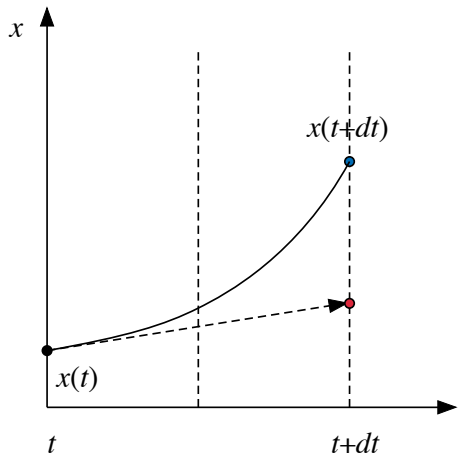

図 7: オイラー法。

$x(t) + \Delta t \cdot f(t, x) = x(t) + x'(t)\Delta t$ と x の t の周りで Tayler 展開

$$x(t + \Delta t) = x(t) + x'(t)\Delta t + \frac{1}{2!}x''(t)\Delta t^2 + \frac{1}{3!}x'''(t)\Delta t^3 + \cdots \tag{15}$$

とを比較すると，Eular 法が x の t の周りでの Teyler 展開の 1 次の項までになっている。従って，Eular 法による微分方程式の解は，真の解に対して 2 次以上の項が計算誤差となっていることがわかる。

これに対し，Runge-Kutta 法では，図 8 のように，一旦，Euler 法と同じやり方で $t+dt$ における x の値 k_1 を求め，この値を使って t と $t+dt$ の中間点 $t+dt/2, x(t)+k_1/2$ での x の傾き（一次微係数）$f(t+dt/2, x(t)+k_1/2)$ を求め直し，それを用いて $x(t+dt)$ を計算し直している。これを式で書くと，

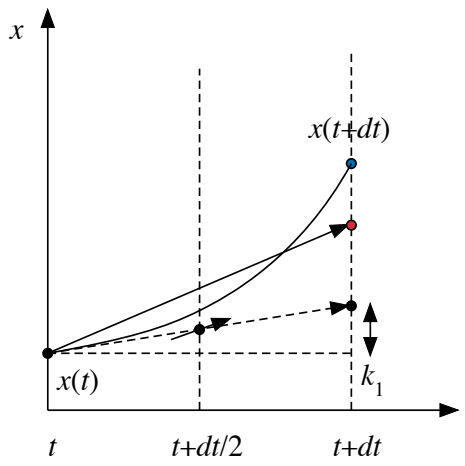

図 8: 2 次の Runge-Kutta 法。

$$k_1 = f(t, x(t))dt \tag{16}$$

$$x(t + dt) = x(t) + f(t + dt/2, x(t) + k_1/2)dt \tag{17}$$

というようになる。この解法は，2 次の Runge-Kutta 法と呼ばれる。Taylor 展開と比較すると dt の 2 次まで精度となっている。

Runge-Kutta 法には更に高次の解法がある。図 9 は，4 次の Runge-Kutta 法と呼ばれる数値積分法を表している。この方法は，式 (16) と式 (17) の手順をもう一回繰り返したものである。さらに具体的に書くと，

1. (t, x) での勾配で，dt だけ進んだときの x の増分 k_1 を求める。
2. k_1 を用いて，t と $t + dt$ の中点での勾配 $f(t + dt, x + k_1/2)$ を求め，この勾配で $x(t)$ から dt だけ進んだときの x の増分 k_2 を求める。
3. k_2 を用いて，t と $t + dt$ の中点での勾配 $f(t + dt, x + k_2/2)$ を求め，この勾配で $x(t)$ から dt だけ進んだときの x の増分 k_3 を求める。

4. k_3 を用いて，t と $t+dt$ の中点での勾配 $f(t+dt, x+k_3/2)$ を求め，この勾配で $x(t)$ から dt だけ進んだときの x の増分 k_4 を求める。

という手順で x の増分の暫定値 k_1，k_2，k_3，k_4 を求め，これらの加重平均を計算して $x(dt)$ を求めている。これを式で書くと，

$$k_1 = f(t, x(t))dt \tag{18}$$
$$k_2 = f(t + dt/2, x(t) + k_1/2)dt \tag{19}$$
$$k_3 = f(t + dt/2, x(t) + k_2/2)dt \tag{20}$$
$$k_4 = f(t + dt, x(t) + k_3)dt \tag{21}$$
$$x(t + dt) = x(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{22}$$

となる。この計算は，実質的にEular法で用いた点の他に，t+dt∗(1/6)，t+dt∗(1/2)，$t + dt * (5/6)$ の3点の情報を用いていることになっている。4次のRunge-Kutta法は，Taylor展開の4次の項までを計算に用いていることに相当し，誤差は dt の5次以上の項である。通常，Runge-Kutta法と呼んでいるのは4次のRunge-Kutta法である。

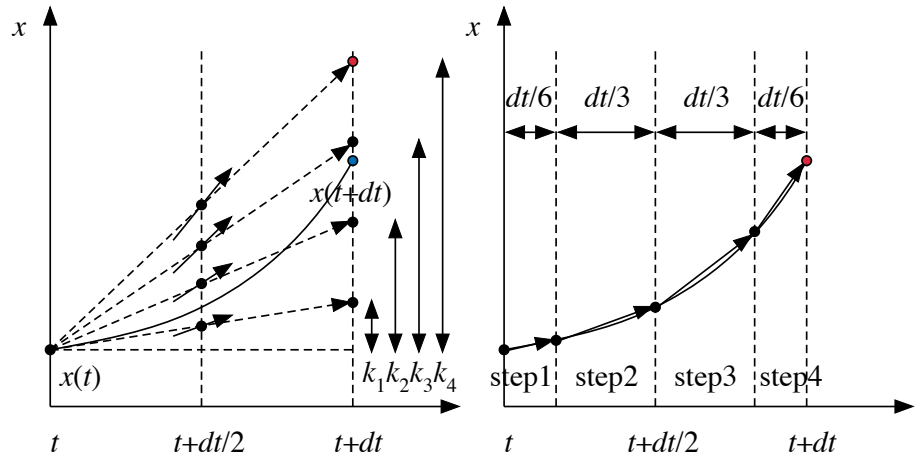

図 9: 4次のRunge-Kutta法。

以下に，$dy/dx = y$ をRunge-Kutta法で計算するプログラムを示す。この中で，Runge-Kutta法で計算する部分を `S_RungK` という関数にしている。

プログラム 6　Runge-Kutta法による微分方程式の初期値問題解析

```
// === Runge-Kutta法による微分方程式の初期値問題解析プログラム ===
#include <stdio.h>
#include <math.h>
#define EPS 1.0e-8
#define N 50    // 分割数
```

```
#define X0 0.0  // X の初期値
#define XN 5.0  // X の終了値
#define Y0 1.0  // Y の初期値

// 関数が主プログラム main より後ろに書かれているときは，main の前に
// このように定義を書いておく。そうしないと，関数が見つからないという
// エラーが表示される。
void S_RungK( double, double *, double, int, double(*)() );
double FUNC( double, double );

// 主プログラム
void main(){
    FILE *fout;
    int i;
    double x, h, y[101];

    fout=fopen("RK.csv","w");

    h=(XN-X0)/(double)N;
    x=X0;
    y[0]=Y0;
    printf(" X0=%f XN=%f Y0=%f h=%f \n",x,XN,y[0],h);

    S_RungK(x,y,h,N,*FUNC);
    for(i=0; i<=N; i++) {
        printf(" i=%f, y=%f \n", h*i, y[i]);
        fprintf(fout,"%f, %f\n", h*i, y[i]);
    }

    fclose(fout);
}

/** Function to calculate Runge-Kutta method **
// 入力; y[0] = y の初期値
// h = x の刻み
// N =計算ステップ数（刻み h で N 回計算）
// x = x の初期値
// FUNC = dy/dx の関数
// 出力; y[N+1}=計算結果（N+1 個の配列に入れられている）
void S_RungK( double x, double y[], double h, int n,
double (*FUNC)()){
    int i;
    double k1,k2,k3,k4;

    for(i=0; i<N; i++){
        k1=h*(*FUNC)(x, y[i]);                // step1
        k2=h*(*FUNC)(x+0.5*h, y[i]+0.5*k1);  // step2
        k3=h*(*FUNC)(x+0.5*h, y[i]+0.5*k2);  // step3
        k4=h*(*FUNC)(x+h,y[i]+k3);            // step4
        x=x+h;
        y[i+1]=y[i]+(k1+2.0*(k2+k3)+k4)/6.0;
    }
}
```

```
// dy/dx の関数をここに書く
// 以下では，dy/dx=y の場合を与えている
double FUNC( double x, double y ){
    return ( y );
}
```

（参考文献）http://www.hm5.aitai.ne.jp/~minemura/Csimu/index.html

この計算の結果を，図 10 に示す。

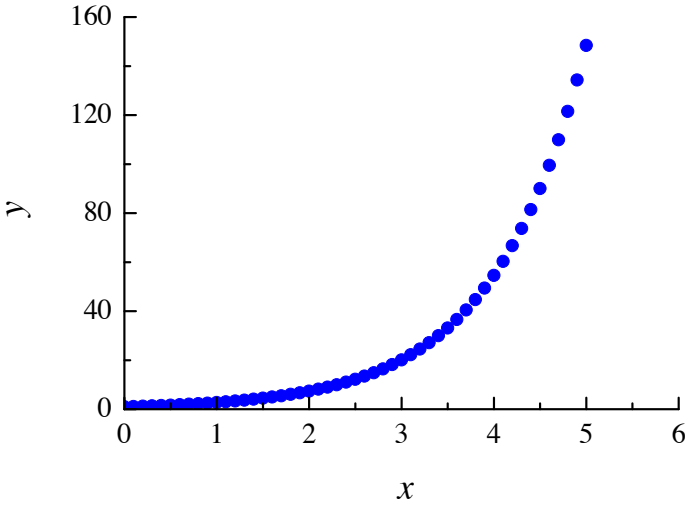

図 10: $dy/dx = y$ を 4 次の Runge-Kutta 法で計算した結果。

以下には，ケプラー運動を計算するプログラムを示す（全文は未掲載）。この中で，Runge-Kutta 法の各中間点での微係数から qx, qy, px, py の暫定値を計算する関数を RungeKutta() としている。

プログラム 7　Runge-Kutta 法によるケプラー運動

```
// === Runge-Kutta 法によるケプラー運動を計算 ===
void RungeKutta( double qx, double qy, double px, double py,
double *qxk, double *qyk, double *pxk, double *pyk )
{
    double q, qi3;
    q = hypot( qx, qy );
    qi3 = 1.0/(q*q*q);
    *qxk = px/M*dT;
    *qyk = py/M*dT;
    *pxk = -GM*M*qx*qi3*dT;
    *pyk = -GM*M*qy*qi3*dT;
}
```

Runge-Kutta 法の扱いは少々面倒だが，Euler 法より遥かに精度の高い計算ができる。事実，このプログラムは惑星が太陽を 10 回まわるまで計算しても，ずっと同じ楕円軌道を通ることが確認できている。一方，Euler 法では近日点移動のような動き（今回の課題では真実ではない動き）が起こってしまう。

7　放物線運動の数値解法（オイラー法による微分方程式の数値解法）

一様重力場での質点の 2 次元平面内の運動を例として，微分方程式の数値計算について説明する。質点の質量を m [kg]，位置を q_x, q_y，重力が $-q_y$ の方向に向いているとすると，運動方程式は，

$$\frac{dq_x^2}{dt^2} = 0,\ \frac{dq_y^2}{dt^2} = -g \tag{23}$$

で表される。この解は，容易に解析的に（＝紙と鉛筆で式を変形して，解の数式を書き表すことができる）に求めることができるが，ここではあえて計算機により数値計算に解いてみることとする。

この式は，2 階の微分方程式なので，x 方向および y 方向の運動量 p_x [kg·m/s]，p_y [kg·m/s] を導入して，

$$\frac{dq_x^2}{dt^2} = \frac{dp_x}{dt} = 0, \frac{dq_y^2}{dt^2} = \frac{dp_y}{dt} = -g, \tag{24}$$

$$\frac{dq_x}{dt} = \frac{p_x}{m}, \frac{dq_y}{dt} = \frac{p_y}{m}, \tag{25}$$

と書き，単位時間ステップ dT ごとに，運動量 p_x，p_y の 1 階微分方程式を解き，その値を使ってさらに位置 p_x [m]，p_y [m] の 1 階の連立微分方程式を解くように変形する。質点が地面より上にある間の軌跡を Eular 法で計算するプログラムを下に示す。

プログラム 8　Eular 法による弾道運動

```
// 弾道運動を Eular 法で解く
#include <stdio.h>
#define G 9.80665  // 重力加速度 [m/s/s]
#define M 1.0      // 質量 [kg]
#define N 256      // 計算ステップ数
#define T0 0.0     // 時間の初期値 [s]
#define TN 5.0     // 時間の終了値 [s]
#define QX0 0.0    // X 方向の位置の初期値 [m]
#define QY0 0.0    // Y 方向の位置の初期値 [m]
#define PX0 1.0    // X 方向の運動量の初期値 [kgm/s]
#define PY0 2.0    // Y 方向の運動量の初期値 [kgm/s]

//---- main function
int main(void)
{
    FILE *fout;
    int i;
    double qx, qy, px, py, dT;

    fout = fopen("kekka.csv","w");  // 出力ファイルを開く

    printf(" 初期値 T0=%4.1f QX0%4.1f QY0=%4.1f PX0=%4.1f PY0=%4.1f
    N=%5d\n",T0,QX0,QY0,PX0,PY0,N);
```

```
    dT = (TN-T0)/(double)N;
    qx = QX0; qy=QY0; px=PX0; py=PY0;

    for(i=0; i<N; i++) {   // qy が正（地面より上）の間は計算を繰り返す
        printf("qx=%lf yx=%lf\n",qx,qy);   // ファイルに qx,qy を書く
        fprintf(fout,"%lf, %lf\n",qx,qy);  // ファイルに qx,qy を書く
        qx += px/M*dT;      // x 方向の位置 (各ステップ)
        qy += py/M*dT;      // y 方向の位置 (各ステップ)
        px += M*0*dT;       // x 方向の運動量 (各ステップ)
        py += -M*G*dT;      // x 方向の運動量 (各ステップ)
        if(qy<0.0) break;  // y 方向の位置が 0 以下のときに計算をやめる
    }

    fclose(fout);           // 出力ファイルを閉じる
    return(0);
}
```

このプログラムの計算結果を図 11 に示す。

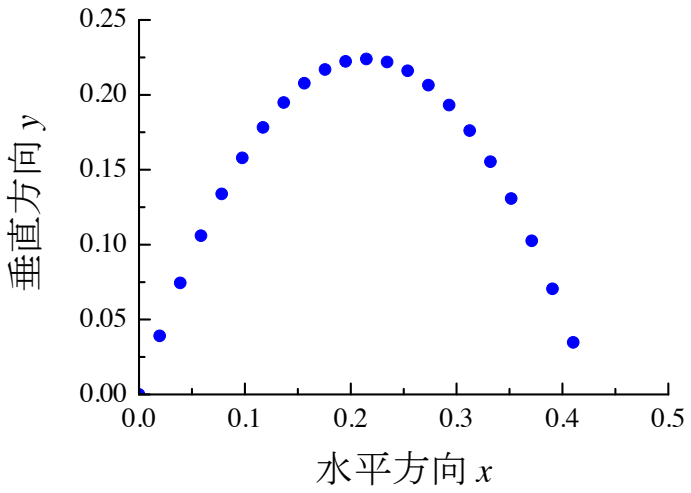

図 11: 弾道運動 (放物線運動) を Eular 法で解く。

（演習） 演習上記のプログラムを Runge-Kutta 法で解くプログラムに改良せよ。

8 ケプラー運動の数値解法（ルンゲクッタ法による微分方程式の数値解法）

ニュートンの万有引力の法則を元に惑星の起動を計算してみる。簡単のため中心の太陽は動かないとして，惑星は 1 つだけを考える。この条件では，惑星の動きは 2 次元平面内に限定される。解析的にこの問題を扱う場合，極座標で考えるのが普通であるが，単純にデカルト座標で計算することにする。放物運動との違いは

働く力が場所によって変わることである。太陽を原点として，惑星の位置 (q_x, q_y)，運動量 (p_x, p_y) の微分方程式は以下のようになる。

$$p_x = m\frac{dq_x}{dt},\ p_y = m\frac{dq_y}{dt} \tag{26}$$

$$\frac{dp_x}{dt} = -\frac{GMm}{q^2}\cos\phi = -\frac{GMmq_x}{q^3},\ \frac{dp_y}{dt} = -\frac{GMm}{q^2}\cos\phi = -\frac{GMmq_y}{q^3}, \tag{27}$$

このまま計算すると非常に大きな値を使うことになるので，数値計算では単位系を整理して数値が 1 前後になるようにする。ここでは，

- 惑星の円軌道の半径を 1 天文単位。
- 公転周期を 1 年。

とする「長さと時間の単位系」を使う。このようにすると，円運動の釣り合いを表す以下の式

$$\frac{GMm}{r} = mr\omega^2,\ r = 1,\ T = 1,\ \omega = \frac{2\pi}{T} = 2\pi \tag{28}$$

において，重力定数 G に太陽の質量 M をかけた値 GM は $4\pi^2$ となることがわかる。

- 惑星の質量は（運動には関係しないので）任意に選べるので，ここでは 1。
- 時間刻 dT は 1 年の 256 分の 1。

とする。この式を Runge-Kutta 法で解いてみる。

以下に，ケプラー運動を計算するプログラムを示す。この中で，Runge-Kutta 法の各中間点での微係数から惑星の位置 q_x，q_y 及び惑星の運動量 p_x，p_y の暫定値を計算する関数を `RungeKutta()` としている。

プログラム 9　Runge-Kutta 法による惑星の軌道運動（ケプラー運動）

```
// 惑星の軌道運動（ケプラー運動）を Runge-Kutta 法で解く
#include <stdio.h>
#include <math.h>
// ---- physical setting
#define M_PI 3.1415926535
#define GM 4*M_PI*M_PI
#define M 1.0           // 惑星の質量
#define dT 1.0/256      // 時間ステップ
#define QX0 1.0         // x 方向の位置の初期値
#define QY0 0.0         // y 方向の位置の初期値
#define PX0 0.0         // x 方向の運動量の初期値
#define PY0 sqrt(GM)*M  // y 方向の運動量の初期値

// ---- main function
int main(void)
{
    FILE *fout;

    double qx, qy;
    double px, py;
```

```
    double T;
    double qxk1, qyk1, pxk1, pyk1;
    double qxk2, qyk2, pxk2, pyk2;
    double qxk3, qyk3, pxk3, pyk3;
    double qxk4, qyk4, pxk4, pyk4;
    double tqx, tqy, tpx, tpy;
    double q, qi3;

    fout = fopen("orbit.csv","w");  // 出力ファイルを開く

    qx = QX0; qy = QY0; px = PX0; py = PY0;
    for( T = 0.0; T < 10.00; T += dT ){
        // 位置(qx,qy)と運動量(px,py)のk1を求める
        tqx = qx; tqy = qy; tpx = px; tpy = py;
        q = hypot( tqx, tqy ); qi3 = 1.0/(q*q*q);
        qxk1 = tpx / M * dT;
        qyk1 = tpy / M * dT;
        pxk1 = -GM * M * tqx * qi3 * dT;
        pyk1 = -GM * M * tqy * qi3 * dT;

        // 位置(qx,qy)と運動量(px,py)のk2を求める
        tqx = qx+0.5*qxk1; tqy = qy+0.5*qyk1; tpx = px+0.5*pxk1;
tpy = py+0.5*pyk1;
        q = hypot( tqx, tqy ); qi3 = 1.0/(q*q*q);
        qxk2 = tpx / M * dT;
        qyk2 = tpy / M * dT;
        pxk2 = -GM * M * tqx * qi3 * dT;
        pyk2 = -GM * M * tqy * qi3 * dT;

        // 位置(qx,qy)と運動量(px,py)のk3を求める
        tqx = qx+0.5*qxk2; tqy = qy+0.5*qyk2; tpx = px+0.5*pxk2;
tpy = py+0.5*pyk2;
        q = hypot( tqx, tqy ); qi3 = 1.0/(q*q*q);
        qxk3 = tpx / M * dT;
        qyk3 = tpy / M * dT;
        pxk3 = -GM * M * tqx * qi3 * dT;
        pyk3 = -GM * M * tqy * qi3 * dT;

        // 位置(qx,qy)と運動量(px,py)のk4を求める
        tqx = qx+qxk3; tqy = qy+qyk3; tpx = px+pxk3; tpy = py+pyk3;
        q = hypot( tqx, tqy ); qi3 = 1.0/(q*q*q);
        qxk4 = tpx / M * dT;
        qyk4 = tpy / M * dT;
        pxk4 = -GM * M * tqx * qi3 * dT;
        pyk4 = -GM * M * tqy * qi3 * dT;

        // 暫定値k1,k2,k3,k4 の加重平均
        qx += (qxk1 + 2*qxk2 + 2*qxk3 + qxk4)*(1.0/6);
        qy += (qyk1 + 2*qyk2 + 2*qyk3 + qyk4)*(1.0/6);
        px += (pxk1 + 2*pxk2 + 2*pxk3 + pxk4)*(1.0/6);
        py += (pyk1 + 2*pyk2 + 2*pyk3 + pyk4)*(1.0/6);

        printf("qx=%f qy=%f px=%f py=%f \n",qx,qy,px,py);
```

```
        fprintf(fout,"%f, %f\n",qx,qy); // ファイルに qx,qy を書く
    }

    fclose(fout);  // 出力ファイルを閉じる
    return(0);
}
```

図 12 に計算結果を示す。この結果は円運動になているが，初期条件を変えると楕円運動とすることができる。Runge-Kutta 法の扱いは少々面倒だが，Euler 法より遥かに精度の高い計算ができる。事実，このプログラムは惑星が太陽を 10 回まわるまで計算しているが，ずっと同じ楕円軌道を通っている。Euler 法では近日点移動 (?) のような動きが発生してしまう。

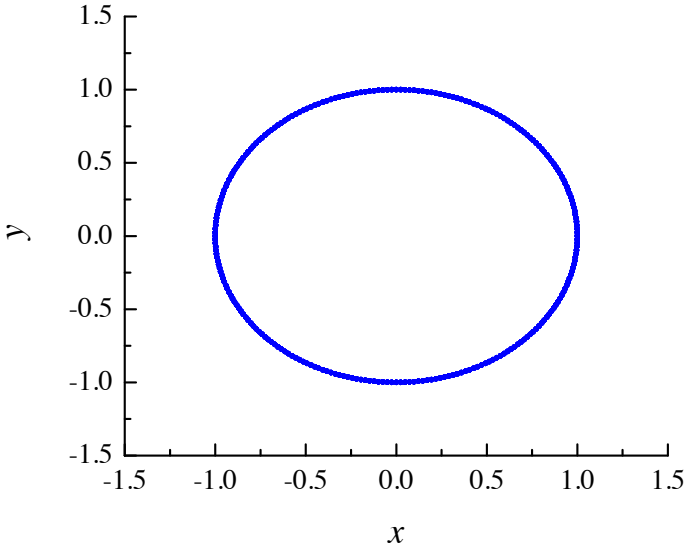

図 12: Keplar 運動を Runge-Kutta 法で解く。

9 偏微分方程式の数値解法

ここまでは，常微分方程式の数値解法について述べたが，以下では，熱拡散，流体運動，波動など，多くの分野で扱われる偏微分の数値解法について述べる。特に，場の状態の時間発展（非定常状態）などを扱うときには偏微分方程式の数値解法は必須であり，この意味で計算機シミュレーションの王道といえる。とはいっても基本的な考え方は常微分方程式と同じであり，方程式を「差分近似」して離散化して計算している。物理の問題としては 2 階あるいは 1 階の偏微分方程式がよく出現する。2 階の偏微分方程式はその形から，(1) 楕円型（ラプラス方程式），(2) 双曲型（波動方程式），(3) 放物型（拡散方程式）に分類され，それぞれに適当な数値解法が考案されている。ここでは，放物型方程式のうちで最も簡単な方程式である空間に対して一次元の拡散方程式を解いてみる。これは，細い棒の端に熱を加えたときに，どのように熱が伝わっていくかという問題（熱伝導方程式）などに現れる。一次元の拡散方程式は，時間を t，棒の位置を x，時間 t，位置 x に

おける状態（例えば，温度）を $u(x,t)$ とすると，

$$\frac{\partial u(x,t)}{\partial t} = \kappa \frac{\partial^2 u(x,t)}{\partial x^2} \tag{29}$$

と表される。κ は定数であり，熱伝導方程式においては κ を熱伝導係数に相当する。

9.1 方程式の簡略化（規格化）

計算機で行う計算には，打ち切り誤差や丸め誤差といった誤差があり，極端に値の違う数値同士の計算（特に，足し算，引き算）を行うのは不得意である。微分方程式を数値的に解く場合，数値計算として必要な精度を保つために，我々が普通に使う単位系（SI とか CGS とか）をそのまま計算することはせず，変数変換を行って各変数の変動範囲をほぼ同じ数値にそろえる。たとえば，時間の変動範囲を T，空間的な広がりを X とすると，$\mathrm{t}/T \to \tau$，$x/X \to \xi$ というように変換する。このとき，各項の係数の値をできるだけ 1 にするように変換を行う。このような操作は規格化と呼ばれ，数値解法において重要な手順のひとつである。拡散方程式も問題に応じて適当に規格化すれば，拡散係数 $\kappa = 1$ にすることができる。従がって，ここで解くべき方程式は，

$$\frac{\partial u(x,t)}{\partial t} = \frac{\partial^2 u(x,t)}{\partial x^2} \tag{30}$$

と表される。で良い。

9.2 差分近似（離散近似）

時間 t の範囲は $t > 0$，場所 x の範囲は $0 < x < L$（L は棒の長さを規格かしたもの）とし，初期状態では，

- 棒の内部 $0 < x < L$ の初期状態 $u(x,0) = 1$。
- 棒の端 $x = 0$ 及び $x = L$ において，常に 0，すなわち，$u(0,t) = 0$，$t > 0$。

とする。この条件で，計算をスタートさせ，各時間，各場所の値う $u(x,t)$ を求める。時間，空間の変数 t，x を離散化して，それぞれ Δt，Δx ごとに分割すると，拡散方程式は，

$$\frac{\partial u(x,t)}{\partial t} \to \frac{u(j\cdot\Delta x,(n+1)\cdot\Delta t) - u(j\cdot\Delta x, n\cdot\Delta t)}{\Delta t} \tag{31}$$

$$\frac{\partial^2 u(x,t)}{\partial x^2} \to \frac{u((j+1)\cdot\Delta x, n\cdot\Delta t) - 2u(j\cdot\Delta x, n\cdot\Delta t) + u((j-1)\cdot\Delta x, n\cdot\Delta t)}{\Delta x^2} \tag{32}$$

$J_{max} = 1/\Delta x$ とすると，$j = 0, 1, 2, \cdots, J_{max}$ となる。また，$n = 0, 1, 2, \cdots$ となる。n の上限値は必要な時間分だけ大きくとることになる。これを用いて，

$$\frac{u(j\cdot\Delta x,(n+1)\cdot\Delta t) - u(j\cdot\Delta x, n\cdot\Delta t)}{\Delta t} \tag{33}$$

$$= \frac{u((j+1)\cdot\Delta x, n\cdot\Delta t) - 2u(j\cdot\Delta x, n\cdot\Delta t) + u((j-1)\cdot\Delta x, n\cdot\Delta t)}{\Delta x^2} \tag{34}$$

両辺に Δt をかけて，

$$u(j\cdot\Delta x,(n+1)\cdot\Delta t)-u(j\cdot\Delta x,n\cdot\Delta t) \tag{35}$$

$$=\frac{\Delta t}{\Delta x^2}\{u((j+1)\cdot\Delta x,n\cdot\Delta t)-2u(j\cdot\Delta x,n\cdot\Delta t)+u((j-1)\cdot\Delta x,n\cdot\Delta t)\} \tag{36}$$

となる。$\Delta t/\Delta x^2$ を r と置くと，解くべき離散化された拡散方程式は，

$$u(j\cdot\Delta t,(n+1)\cdot\Delta x)=r\cdot u((j+1)\cdot\Delta x,n\cdot\Delta t)$$

$$+(1-2r)\cdot u(j\cdot\Delta x,n\cdot\Delta t)+r\cdot u((j-1)\cdot\Delta x,n\cdot\Delta t) \tag{37}$$

$$r=\frac{\Delta t}{\Delta x^2} \tag{38}$$

したがって，プログラムは次のようになる。

プログラム 10　一次元拡散方程式

```
// 一次元拡散方程式（熱拡散方程式）を解く
#include <stdio.h>
#define JMAX 10   // 空間の刻みの数
#define NMAX 250  // 時間の最大値
#define NINT 25   // 計算結果を出力する時間間隔

void main()
{
    FILE *fout;

    double u[JMAX+1],unew[JMAX+1];  // 場の状態 u，計算途中の変数 unew
    double delx,delt,r;             // Δ x, Δ t
    int j,n;                        // 空間位置 j，時間の進み n

    delx = 1.0/(double)JMAX;  // 空間の刻みΔ x
    delt = 0.004;             // 時間の刻みΔ t
    r = delt/delx*delx;       // 係数 r

    // 初期状態の設定
    u[0] = 0.0; for(j=1; j<JMAX; j++) u[j]=1.0; u[JMAX]=0.0;
    for(j=0; j<=JMAX; j++) unew[j]=u[j];

    fout = fopen("diffuse.csv","w");  // 出力ファイルを開く

    for( n=0; n<=NMAX; n++) {          // 時間発展の繰り返し
        // 計算結果を時間ステップ NINT 回ごとに出力
        if( (n%NINT) == 0) {
            for(j=0; j<JMAX; j++) fprintf(fout, "%lf, ", u[j]);
            fprintf(fout,"%lf\n",u[JMAX]);
        }

        for(j=1; j<JMAX; j++) unew[j] = r*u[j+1]+(1.0-r*2.0)*u[j]
+r*u[j-1];
```

```
                    // 時間1ステップ分の計算
                    // 端は解散していないことに注意
        for(j=0; j<=JMAX; j++) u[j] = unew[j];  // 計算結果を保存
    }

    fclose(fout);   // 出力ファイルを閉じる
}
```

拡散方程式の場合，上記の離散化を行うと時間方向の積分を安定に行う為には，

$$r < \frac{1}{2} \tag{39}$$

である必要があることが知られている（安定性条件）。このため，空間を精度よく解こうと思って，Δx を小さくすると，時間刻み幅 Δt を非常に小さくとらなくならなければならないので，非常に計算量が多くなるという欠点がある。これを改善するため，クランクーニコルソン法などの方法が考えられている。