数値積分と数値微分（基礎）

重田 出

講義・演習の目標

関数の積分を台形則・中点則・シンプソン則・モンテカルロ法で解く。また，オイラー法・ルンゲクッタ法で常微分方程式の初期値問題を解く。

1　台形法による数値積分

関数 $f(x)$ の定積分を，微分積分学の教科書に行うように解析的に（数式として）求めるのではなく，微小区間に分割してそれぞれの面積を求め，それらを足し合わせることにより近似値を求める方法を数値積分という。これは，

$$\int_b^a f(x)dx = \sum_{i=1}^{n} \Delta A(x_i) \tag{1}$$

で一般的に表すことができる。ここで，i，x は $[a,b]$ を n 個に分割したときの i 番目の点を表し，$\Delta A(x_i)$ は点 x_i を代表点として求めた微小面積を表す。

数値積分のうちで最も単純な方法として，台形法による数値積分がある。これは，図 1 のように，関数 $f(x)$ の区間 $[a,b]$ を n 等分して刻み幅 $h=(b-a)/n$ の小区間に分け，各等分点における関数の値を直線で結んで小区間ごとに台形を作り，この台形の面積を積分区間内で足し合わせる方法である。

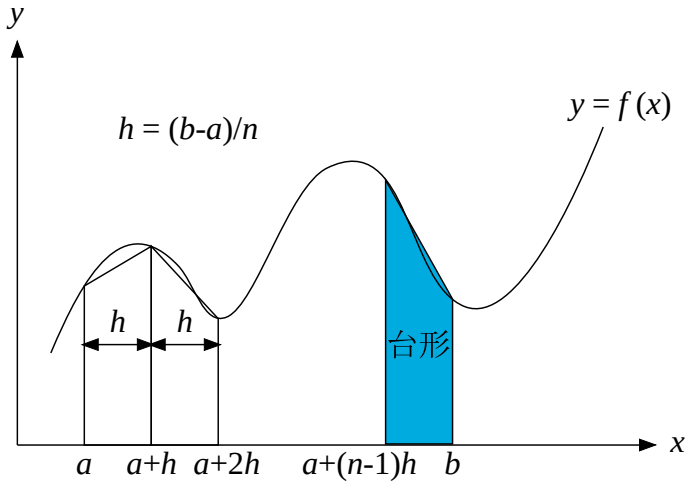

図 1: 台形法による数値積分。

x 軸の刻みの幅を $h = (b-a)/n$ とすると，ひとつの小区間 $[x_i, x_{i+1}] = [a+i{\cdot}h, a+(i+1){\cdot}h]$ における台形の面積 $\Delta A(x_i)$ は，

$$\Delta A(x_i) = \frac{h}{2}\{f(x_i) + f(x_{i+1})\} = \frac{h}{2}\{f(a+i{\cdot}h) + f(a+(i+1){\cdot}h)\}, \qquad (2)$$

で求められ、a から b までの範囲についての積分結果は，

$$\sum_{i=1}^{n} \Delta A(x_i) = \sum_{i=1}^{n} \frac{h}{2}\{f(x_i) + f(x_{i+1})\} \qquad (3)$$

$$= h\left\{\frac{1}{2}f(a) + \sum_{i=1}^{n-1} f(a+i{\cdot}h) + \frac{1}{2}f(b)\right\} \qquad (4)$$

となる。この方法は，非常に簡単で理解し易く，プログラミングも作り易いが，図 1 から分かるように，関数が上に凸の範囲では数値計算の結果が解析的な積分結果より常に小さく，関数が下に凸の範囲では数値計算の結果が解析的な積分結果より常に大きくなるので，精密な計算が必要な場合には適切な方法ではない。

プログラム 1　台形公式による積分

```
#include <stdio.h>
#define X_MIN 0.0              // 積分範囲の最小値
#define X_MAX 100.0            // 積分範囲の最大値
#define DIV_NUM 120            // 積分範囲の分割数

// 積分対象関数.
float function (float x)
{
    float result;
    result = x;                // 一次直線
    return result;
}

int main()
{
    double integral;           // 積分結果
    double h;                  // 積分範囲を n 個に分割したときの幅
    double x, dA;
    int i;

    printf ("積分範囲=[%f,%f] 分割数=%d\n", X_MIN, X_MAX, DIV_NUM);

    // === 台形公式による積分（開始） ===
    h = (X_MAX - X_MIN) / DIV_NUM;  // 分割幅
    integral = 0.0;            // 積分結果の変数を初期化
    x = X_MIN;                 // 積分範囲の変数を初期化
    for (i=0; i<DIV_NUM; i++) {
        // 微小範囲の面積Δ A
        dA = (function(x)+function(x+h))*h/2.0;
        integral += dA;        // Σ
```

```
        x += h;
    }
    // === 台形公式による積分（終了） ===

    printf ("積分結果= %lf\n", integral);
    printf ("解析的に求めた結果(底辺100，高さ100の直角三角形)=%lf\n",
(X_MAX-X_MIN)*function(X_MAX)/2.0);

    return(0);
}
```

(問題) 上記の台形公式による積分プログラムは式 (3) をもとにしてプログラムを組み上げているが，このやり方では $f(x+h)$ の値は次の繰り返しでは $f(x)$ に相当しており，同じ値に対する計算を 2 回ずつ行っていることになっている。このプログラムを，同じ計算を 2 回行わないようにして，高速化せよ。

2 中点法による数値積分

面積を求める方法として，幅の狭い台形を敷き詰めていく方法に変えて，図 2 のように微小区間の中点を高さの代表値とする幅の狭い長方形で敷き詰める手法を中点法という。

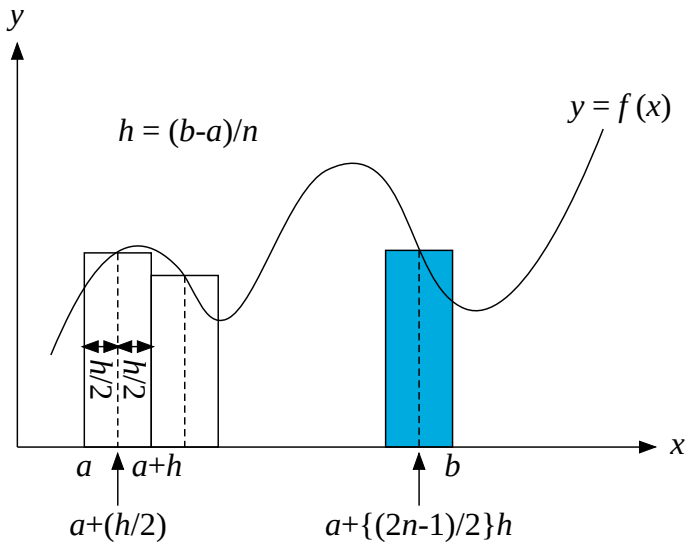

図 2: 中点法による数値積分。

関数 $f(x)$ の区間 $[a,b]$ を n 等分して刻み幅 $h=(b-a)/n$ の小区間に分け，あるひとつの小区間 $[x_i, x_{i+1}] = [a+i\cdot h, a+(i+1)\cdot h]$ において，その幅の中点 $x_i + h/2$ における関数の値を高さとする幅 h の長方形を作ると，この長方形の面積 $\Delta A(x_i)$ は，

$$\Delta A(x_i) = h \cdot f\left(x_i + \frac{1}{2}h\right) = h \cdot f\left(a + \frac{2i-1}{2}h\right), \tag{5}$$

で求められ，a から b までの範囲についての積分結果は，

$$\sum_{i=1}^{n} \Delta A(x_i) = h \sum_{i=1}^{n} f\left(x_i + \frac{1}{2}h\right) = h \sum_{i=1}^{n} f\left(a + \frac{2i-1}{2}h\right), \tag{6}$$

となる。

（課題） プログラム 1「台形法による積分」を，「中点法」で計算するように変更せよ。

3　シンプソン法による数値積分

台形法では小区間において関数 $f(x)$ を直線で近似して数値積分を計算しているのに対し，シンプソン（Simpson）法ではこの区間において関数 $f(x)$ を 2 次曲線で近似し，積分の精度を上げている。図 3 の右側にシンプソン法の基本概念を示す。この図において，$(x_0, f(x_0))$，$(x_1, f(x_1))$，$(x_2, f(x_2))$ の 3 点を通る二次曲線の方程式 $g(x)$ は，

$$\begin{aligned} g(x) = &\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_0) + \frac{(x-x_2)(x-x_0)}{(x_1-x_2)(x_1-x_0)} f(x_1) \\ &+ \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} f(x_2) \end{aligned} \tag{7}$$

で与えられる（グレゴリー・ニュートンの差分公式で 2 次の項までを使った場合に

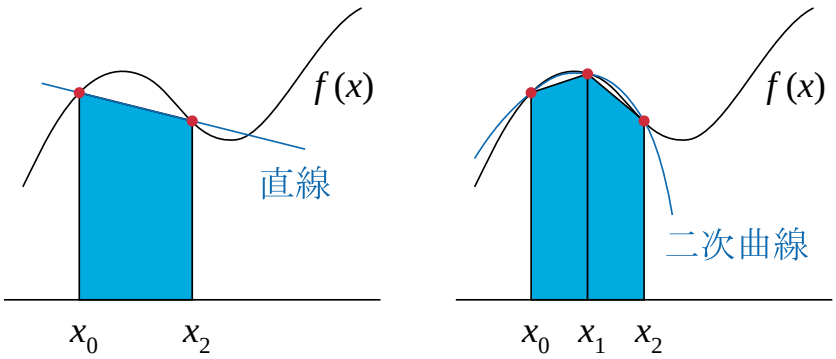

図 3: 台形法 (左) とシンプソン法 (右) による数値積分の比較。

相当)。この二次曲線 $g(x)$ の区間 $[x_0, x_2]$ の範囲での積分値は，部分積分を用いて，

$$\begin{aligned}\int_{x_0}^{x_2} g(x)dx =&\left[\frac{(x-x_1)(x-x_2)}{2(x_0-x_1)(x_0-x_2)}f(x_0) - \frac{(x-x_1)^3}{6(x_0-x_1)(x_0-x_2)}f(x_0)\right.\\ &+\frac{(x-x_2)(x-x_0)}{2(x_1-x_2)(x_1-x_0)}f(x_1) - \frac{(x-x_2)^3}{6(x_1-x_2)(x_1-x_0)}f(x_1)\\ &\left.+\frac{(x-x_0)(x-x_1)}{2(x_2-x_0)(x_2-x_1)}f(x_2) - \frac{(x-x_0)^3}{6(x_2-x_0)(x_2-x_1)}f(x_2)\right]_{x_0}^{x_2}\\ =&\frac{(x_2-x_0)^2(x_2-x_1)}{2(x_2-x_0)(x_2-x_1)}f(x_2) - \frac{(x_2-x_0)^3}{6(x_2-x_0)(x_2-x_1)}f(x_2)\\ &+\frac{(x_0-x_1)(x_0-x_2)^2}{2(x_0-x_1)(x_0-x_2)}f(x_0) + \frac{(x_0-x_2)^3}{6(x_0-x_1)(x_0-x_2)}f(x_0)\\ &+\frac{(x_0-x_2)^3}{6(x_1-x_2)(x_1-x_0)}f(x_1) \qquad (8)\end{aligned}$$

と計算できる。$x_1 - x_0 = x_2 - x_1 = (x_2 - x_0)/h$ とすると，(8) 式は，

$$\begin{aligned}\int_{x_0}^{x_2} g(x)dx =&\frac{(2h)^2h}{2(2h)h}f(x_2) - \frac{(2h)^3}{6(2h)h}f(x_2) - \frac{(-h)(-2h)^2}{2(-h)(-2h)}f(x_0)\\ &+\frac{(-2h)^3}{6(-h)(-2h)}f(x_0) + \frac{(-2h)^3}{6(-h)h}f(x_1)\\ =&hf(x_2) - \frac{4}{6}hf(x_2) + hf(x_0) - \frac{4}{6}hf(x_0) + \frac{8}{6}hf(x_1)\\ =&\frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)] \qquad (9)\end{aligned}$$

と書き表せる。このやり方を一般化して，関数 $f(x)$ の区間 $[a, b]$ に適用すると，

$$\begin{aligned}\sum_{i=1}^{n}\Delta A(x_i) =&\frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)] + \frac{h}{3}[f(x_2) + 4f(x_3) + f(x_4)] + \cdots\\ &+\frac{h}{3}[f(x_{2n-2}) + 4f(x_{2n-1}) + f(x_{2n})]\\ =&\frac{h}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots\\ &2f(x_{2n-2}) + 4f(x_{2n-1}) + 2f(x_{2n})]\\ =&\frac{h}{3}[f(x_0) + 4f(x_1) + f(x_3) + \cdots + 2f(x_2) + f(x_4) + \cdots + f(x_n)]\\ =&\frac{h}{3}\left[f(a) + 4\sum_{i=1}^{n} fa + (2i-1)h + 2\sum_{i=1}^{n-1} fa + 2ih + f(b)\right] \qquad (10)\end{aligned}$$

となる。この式を見るとわかるように，実際のプログラミングにあたっては，奇数項の合計と偶数項の合計を求めるようにすれば良い。

プログラム 2　Simpson 法による積分

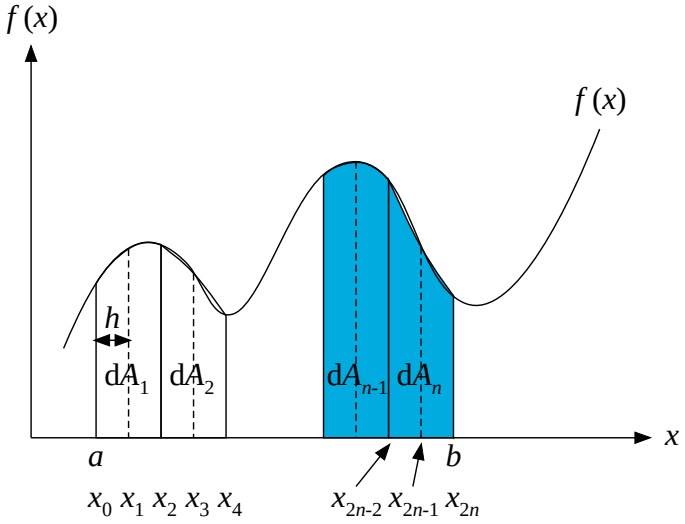

図 4: シンプソン法による数値積分。

```
// === Simpson 法による積分 (開始) ===
    h = (X_MAX - X_MIN) / (2.0*DIV_NUM);  // 分割幅
    x = X_MIN; // 積分範囲の変数を初期化
    integral = function(x);               // 積分結果の変数の初期値

    for (i=1; i<DIV_NUM; i++) {
        dA = 4.0*function(x+h) + 2.0*function(x+2.0*h);
        integral += dA;                   // Σ
        x += 2.0*h;
    }

    integral += ( 4.0*function(x+h) + function(x+2.0*h) );
    integral *= h/3.0;
// === Simpson 法による積分 (終了) ===
```

4　モンテカルロ法による数値積分

多次元の体積 V 中で一様に N 個の点 $x_1, x_2, \cdots, x_n$ をランダムにとったとすると，多次元体積 V についての関数 f の積分は，

$$\int f dV \approx V \cdot \langle V \rangle \tag{11}$$

で推定できる。ここで，$\langle\ \rangle$ は，体積 V のうち関数 f の中に含まれる領域の割合を表す。モンテカルロ（Monte Carlo）法により，円の半径を与えて円の面積を求めるプログラムは以下のようになる。乱数は，C 言語に用意されている `rand()` を用いている。

プログラム 3　モンテカルロ法で円の面積を求める

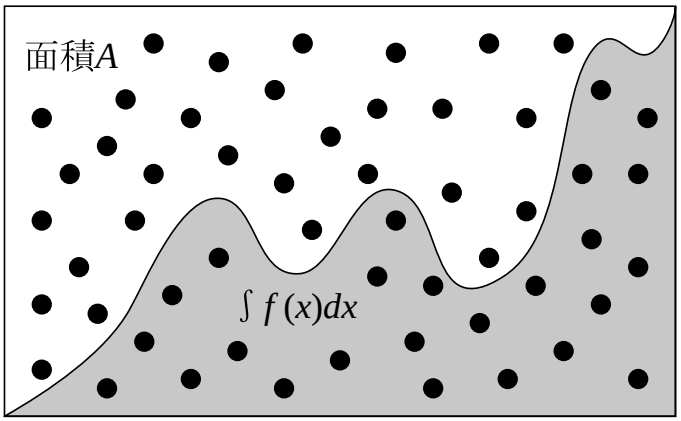

図 5: モンテカルロ積分。領域 A でランダムに点をとる。関数 f の積分は，ランダムな点が曲線 f の下側に落ちる割合と A の面積との積で推定できる。

```
#include <stdio.h>
#include <stdlib.h>   // rand を使うときのヘッダ
#define N 400         // 繰り返し回数
#define D 10.0        // 正方形の辺の長さ

main()
{
    unsigned long i;
    unsigned long in=0,out=0;
    float ratio,x,y,r,square;

    square=D*D;      // 正方形の面積
    r=D/2.0;         // 正方形に内接する円の半径

    for(i=0; i<N; ++i) {
        // x 座標を一様乱数で求める
        x= (double)rand()/(double)RAND_MAX*r;
        // y 座標を一様乱数で求める
        y= (double)rand()/(double)RAND_MAX*r;
        // (x,y) が円の内側か外側かを判定←判定条件
        if ((x*x+y*y)<=r*r) ++in;
        else ++out;
        printf("回数=%d 内側の回数=%d 外側の回数=%d x=%5.2f y=%5.2f\n",
i,in,out,x,y);
    }
    // (x,y) が円の内側のときと外側のときの比
    ratio=(float)in/(float)(in+out);

    printf("範囲内の点数= %d 範囲外の点数= %d\n",in,out);
    printf("正方形の面積＝%f\n",square);
    printf("円の面積（＝正方形の面積× r の内外の比）＝ %f\n",square*ratio);
    printf("円周率を使って求めた円の面積（3.14*r*r）＝%f\n",3.14*r*r);
}
```

5　オイラー法で微分方程式を解く

（参考資料） http://www.sm.rim.or.jp/~shishido/bibun1.html

積分などの解析的な方法では解くのが困難，もしくは解けないことがある。しかし，微分方程式とは言ってみれば変数の「変化」の仕方を表した方程式なので，単純にその方程式にしたがって変数を変化させていけば，大体の様子はわかるはずである。さらにそのように数値計算で解くのは,「計算機」たるコンピュータの最も得意とする分野とも言える。例えば，

$$\frac{dy}{dx} = 2x, \quad x = 1 \text{ で } y = 1 \tag{12}$$

という微分方程式があったとする。これは積分すれば解析的にも簡単に解ける（$y = x^2$) が，初期値を入れた後，y を傾き $dy/dx = 2x$ に従って変化させていっても大体の値は求まる。

まず，$x = 1$ での y は 1，傾きはその時点での $2*x$ の値, すなわち 2 であり，これを使って $x = 1.1$ での y を求めてみる。x の増加分 (0.1) に $x = 1$ のときの傾きをかけて y の増加分が求まり，これを $x = 1$ のときの y の値に加えれば，$x = 1.1$ のときの y の値は大まかには求まる。実際に y の変化量を計算すると，$2 * 0.1 = 0.2$ であり，これに y の元の値 1 を加えて 1.2 になる。実際の値（微分方程式を積分して解析的に解いた値）は，$(1.1)2 = 1.21$ であり，わりと近い値と言える。この説明の中で，$x = 1$ のときの傾きをかけて という部分は，

$$y(x + \Delta x) = y(x) + \left.\frac{dy}{dx}\right|_x \Delta x \tag{13}$$

ということであり，y をテーラー展開して Δx の 2 次以降の項を無視したものである。このようにして微分方程式を解く方法をオイラー法という。オイラー法のように数値を入れながら数値計算で解いていく方法だと，数値を代入できる“1 階”の微分方程式の形に表された問題なら，どんな問題でも（とりあえず）解くことができる。ただし，これで求まる解はあくまでも近似値であり，条件によっては誤差がかなり大きくなるので注意が必要である。さらに，数値計算の解は数値であって一般解や特殊解のような「式」は得られないので，解の「意味」をとらえにくい面もある。

以下にプログラム例を示す。この例では，x が 1.0 から 6 までの値を，初期値 $x = 1.0$ で $y = 1.0$，幅 $h = 0.02$ として求めている。

プログラム 4　オイラー法で $\frac{dy}{dx} = 2x$ を求める。

```
#include <stdio.h>
#define XSTART 1.0    // x の初期値
#define XEND 6.0      // x の最終値
#define YSTART 1.0    // y の初期値
#define H 0.02        // x の刻み幅

main()
{
    double x, y, dy;

    y = YSTART;
    x = XSTART;
```

```
    while(x<6.0) {
        dy = 2.0*x*H;
        y += dy;
        x += H;
        printf("x=%lf y=%lf\n",x,y);
    }

    return(0);
}
```

実行結果は，

```
x=1.020000 y=1.040000
x=1.040000 y=1.080800
x=1.060000 y=1.122400
x=1.080000 y=1.164800
x=1.100000 y=1.208000
 ........
x=5.940000 y=35.184800
x=5.960000 y=35.422400
x=5.980000 y=35.660800
x=6.000000 y=35.900000
x=6.020000 y=36.140000
```

となる。このプログラムに，第 8 回目から第 10 回目までで学んだ図形の描画の方法を利用して，グラフにしてみると，

プログラム 5　オイラー法で $\frac{dy}{dx} = 2x$ を求める＋描画

```
#include <stdio.h>
#include <glut.h>   // OpenGL を使うためのヘッダファイル

#define XSTART 1.0      // x の初期値
#define XEND 6.0        // x の最終値
#define YSTART 1.0      // y の初期値
#define H 0.02          // x の刻み幅

void display(void)      // 図形表示を行う関数。
{
    double x, y, dy;

    glClear(GL_COLOR_BUFFER_BIT);  // キャンバスの初期化

    glBegin(GL_LINE_STRIP);  // ペンを下げる (描き始め)
    y = YSTART;
    x = XSTART;
    while(x<6.0) {      // Eular 法で解く
        dy = 2.0*x*H;
        y += dy;
        x += H;
        printf("x=%lf y=%lf\n",x,y);
        glVertex2d(x*0.2-0.9, y*0.04-0.9);  // 描画
    }
```

```
    glEnd();              // ペンを上げる（描き終わり）

    glBegin(GL_LINE_STRIP);  // ペンを下げる（描き始め）
    glVertex2d(-0.9, 0.9);   // 軸のプロット
    glVertex2d(-0.9, -0.9);
    glVertex2d( 0.9, -0.9);
    glEnd();              // ペンを上げる（描き終わり）

    glFlush();            // キャンバスへ表示
}
```

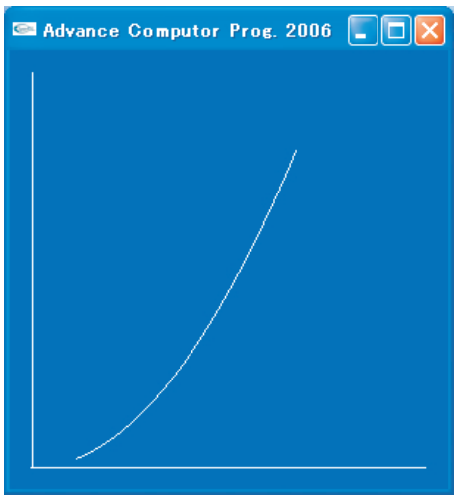

図 6: オイラー法による結果。

（問題） 上に示したオイラー法で求めているのは 1 階の微分方程式であるが，2 階の微分方程式を解くにはどうすればよいか。
（ヒント） 2 階微分は導関数の 1 階微分で表される。例えば，加速度は速度の 1 階微分であり，位置の 2 階微分である。

6 ルンゲクッタ法で微分方程式を解く

ここで使用するルンゲクッタ（Runge-Kutta）法の原理は以下のようなものである。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，

$$\frac{dA}{dt} = f(A) \tag{14}$$

で与えられているとする。ここで，$f(A)$ は既知関数である。$t = 0$ での A の値，すなわち初期値 $A(0)$ を与えて，$t = dT$ での A の値 $A(dT)$ を計算する場合，Euler 法では図 7 のようになる。すなわち，$t = 0$ での傾き（1 次微係数）$f(A(0))$ を使って，点線のように一次直線でいきなり dT での値 $A(dT)$ を $f(A(0))dT$ として計算している。

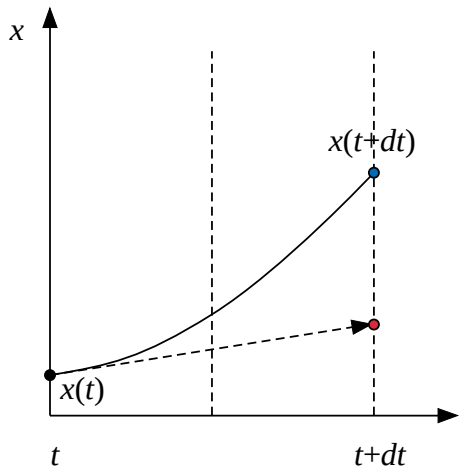

図 7: オイラー法。

図 8 は，2 次の Runge-Kutta 法と呼ばれる数値積分法を表したものである。この方法では，一旦，Euler 法と同じやり方で dT における A の値 k_1 を求め，この値を使って 0 と dT の中間点 $dT/2$ での A の傾き（一次微係数）$f(A(0)) + k_1/2$ を求め直し，それを用いて $A(dT)$ を計算し直している。これを式で書くと，

$$k_1 = f(A(0))dT \tag{15}$$

$$A(dT) = A(0) + f(A(0) + k_1/2)dT \tag{16}$$

というようになる。この解法は，Taylor 展開と比較すると dT の 2 次まで精度があることがわかる。Runge-Kutta 法は更に高次の解法がある。図 9 は，4 次の Runge-Kutta 法による数値積分法を表したものである。この方法は，(15) 式と (16) 式の手順をもう一回繰り返したものである。4 次の Runge-Kutta 法では，3 点の中間点を順次使って求めた $A(dT)$ の暫定値 k_1, k_2, k_3, k_4 の加重平均で $A(dT)$ を計算している。Taylor 展開と比較すると dT の 4 次まで精度があることがわかる。

$$k_1 = f(A(0))dT \tag{17}$$

$$k_2 = f(A(0) + k_1/2)dT \tag{18}$$

$$k_3 = f(A(0) + k_2/2)dT \tag{19}$$

$$k_4 = f(A(0) + k_3/2)dT \tag{20}$$

$$A(dT) = A(0) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{21}$$

通常，Runge-Kutta 法と呼んでいるのは 4 次の Runge-Kutta 法である。

以下に，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 6　Runge-Kutta 法によるケプラー運動を計算

```
// === Runge-Kutta 法によるケプラー運動を計算 ===
```

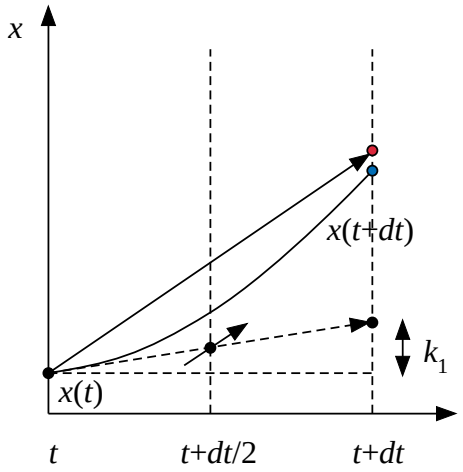

図 8: 2 次の Runge-Kutta 法。

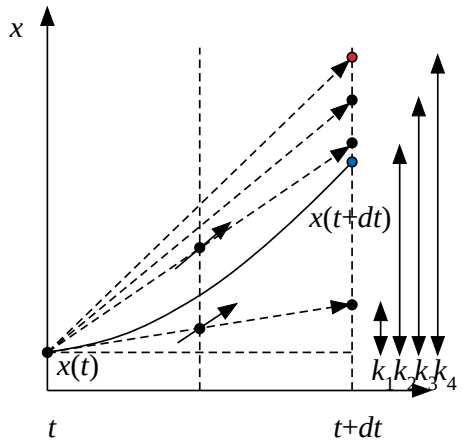

図 9: 4 次の Runge-Kutta 法。

```
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。