# Chapter 1

# 差分法(Euler法)

## 1.1 Euler法

物理の基本的な式はみな微分方程式です。ニュートンの運動方程式も量子力学のシュレディンガー方程式もそうです。それで、ここではコンピュータを使ってどうやって微分方程式を計算するのかを簡単に紹介したいと思います。

さて、最も簡単な形としてオイラー（Euler）法というものがあります。まずもっとも簡単な微分方程式として例えば

$$\frac{\mathrm{dy}}{\mathrm{dx}} = y$$

のようにyのxでの微分がx,yの関数でかけているとき

$$\frac{\mathrm{dy}}{\mathrm{dx}} = f(x, y)$$

を考えていきます。このままでは当然コンピュータには計算させられません。そのためにはなんとか電卓で計算できる形、つまり四則演算の形にしてやらなければいけません。さて、それにはなにより微分のところがどうにもなりませんね。そもそも微分とは

$$\frac{\mathrm{dy}}{\mathrm{dx}} = \frac{y(x + \Delta x) - y(x)}{\Delta x}, \Delta x \to 0$$

でした。これから0にはできないけども出来うる限り小さい$\Delta x$を使えば

$$\frac{y(x + \Delta x) - y(x)}{\Delta x} = f(x, y) \tag{1.1}$$

$$y(x + \Delta x) = y(x) + f(x, y)\Delta x \tag{1.2}$$

となりこの計算をn回繰り返していけば$y(x + n\Delta x)$といったようにyをどんどん求めていくことができます。これが「Euler法」です。実に簡単ですね。ためしに

$$\frac{\mathrm{dy}}{\mathrm{dx}} = y, y(0) = 1$$

で計算してみましょう。この式の解は

$$y = e^x$$

なので、今$x = 0$から1まで計算するとなると本当の答えは$y(1) = e = 2.71828$となるはずです。さて、この計算のためのC言語でのプログラムのソースはこちらです。

```
/*Euler法の練習*/

#include <stdio.h>
```

```
int main(void){
 double y=1.0; /*y の初期値は 1 です*/
 double x=1000; /*この逆数がメッシュの大きさです*/
 long i;

 for(i=1;i <= x;i++){
  y += (1/x) * y;
 }
 printf("%f\n",y); /*結果を表示します*/
 return 0;
}
```

このソースでは基本的に

$$y_{n+1} = y_n + y_n \Delta x, (y_n = y(n\Delta x))$$

を繰り返しです。$\Delta x = 0.001$ にして、x=0 から 1 までなので上の式を 1,000 回繰り返したのですが、ここで問題なのが $\Delta x$ の値（以下メッシュという）をいくつにするかです。理論的にはメッシュをできるだけ小さくしたほうが実際の微分に近づくわけだからそのほうがいいんですが、当然 0 にはできないしまた、メッシュを小さくすればするほど計算回数が多くなり時間がかかってしまいます。$\Delta x = 0.001$ のこのソースでは結果は 2.716924 と誤差は 0.05 %というところです。メッシュの大きさと真値との誤差をグラフにしたのが次の図です。

です。縦軸に誤差の%の対数、横軸がメッシュの幅の対数です。これからともに対数を取ったものは直線に

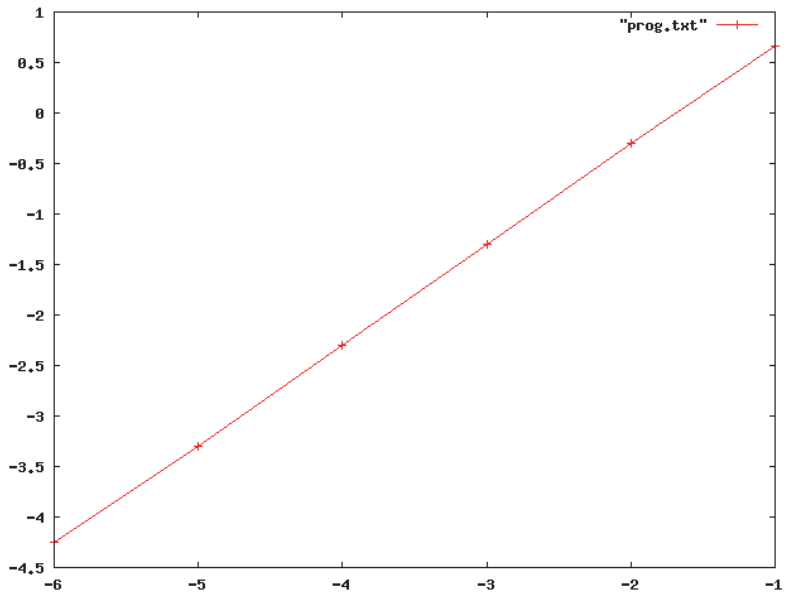


Figure 1.1: メッシュの大きさと誤差

なっていることがわかると思います。この場合はかなり幅の小さいメッシュにしても時間はそれほどかかりませんが、もっと複雑な式の場合は誤差と時間の兼ね合いを考えてメッシュの幅を決める必要があるでしょう。

## 1.2 二次の微分方程式

さて、代表的な物理の方程式といえばニュートンの運動方程式

$$\mathbf{F} = m\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{dt}^2}$$

ですが、これは二回微分の式です。これに限らず二回微分の方程式は物理では大変多いです。一般に二回微分の式

$$\ddot{y} = f(y, x)$$

を Euler 法で計算するにはまず、ｙの一回微分を新たに $v(y, x) = \dot{y}$ という関数だと考え

$$\dot{v} = f(y, x)$$
$$\dot{y} = v(y, x)$$

をいうふたつの一回微分の式に分解してそれぞれ計算していけば、求めることができます。この方法を使えば基本的に何次元の微分方程式でも計算可能です。さて試しにオイラー法で運動方程式を解いてみましょう。計算する式として重力による惑星の軌道を考えてみましょう。簡単のため重力定数や質量など仮に１だとした式を考えると

$$\frac{\mathrm{d}^2\mathrm{x}}{\mathrm{dt}^2} = -\frac{x}{r^3}$$
$$\frac{\mathrm{d}^2\mathrm{y}}{\mathrm{dt}^2} = -\frac{y}{r^3}$$

となります。ただし

$$x(0) = 1, y(0) = 0$$
$$\frac{\mathrm{dx}(0)}{\mathrm{dt}} = 0, \frac{\mathrm{dy}(0)}{\mathrm{dt}} = 1$$
$$r = \sqrt{x^2 + y^2}$$

です。さて、この解は楕円軌道だと仮定し

$$x = a\cos\omega t + C_1, y = b\sin\omega t + C_2$$

を代入して計算してやればすぐに半径１、中心０の円であることがわかります。ではさっそくこの式をオイラー法で計算しましょう。ソースは変数が増えるのでややこしくなりますがこうです。

```
/*Euler 法 メッシュの幅は０．２とｔ＝２０まで計算する*/

#include <stdio.h>
#include <math.h>

int main(void){
 double x =1;
 double u[2] ={0};
 double y =0;
 double v[2] ={1}; /*u,v はそれぞれ x,y の一回微分*/
 int k;

 for(k=0;k < 1000; k++){
  u[1] =u[0]+ -0.02 * x / sqrt(y*y+x*x);
  v[1] =v[0]+ -0.02 * y / sqrt(y*y+x*x);
/*u,v は二回使うので一時的な保存先として u(v)[1] を使う*/
  x += 0.02 * u[0];
  y += 0.02 * v[0];
  u[0]=u[1];
  v[0]=v[1];

  if(k%20==0) printf("%f %f\n",x,y); /*20 回に一回だけ結果を表示させる*/
```

```
 }
 return 0;
}
```

このソースでの計算は

$$
\begin{aligned}
v_{n+1} &= v_n - \frac{x_n}{r^3}\Delta t \\
u_{n+1} &= u_n - \frac{y_n}{r^3}\Delta t \\
x_{n+1} &= x_n + v_n\Delta t \\
y_{n+1} &= y_n + u_n\Delta t
\end{aligned}
$$

を繰り返すだけです。メッシュの幅は０．２、ｔ＝２０まで計算しました。さて、この結果は次のようになります。
赤い線が理論値で緑が計算結果です。これははっきりいって悲惨ですね。もちろんメッシュを小さくすればマ

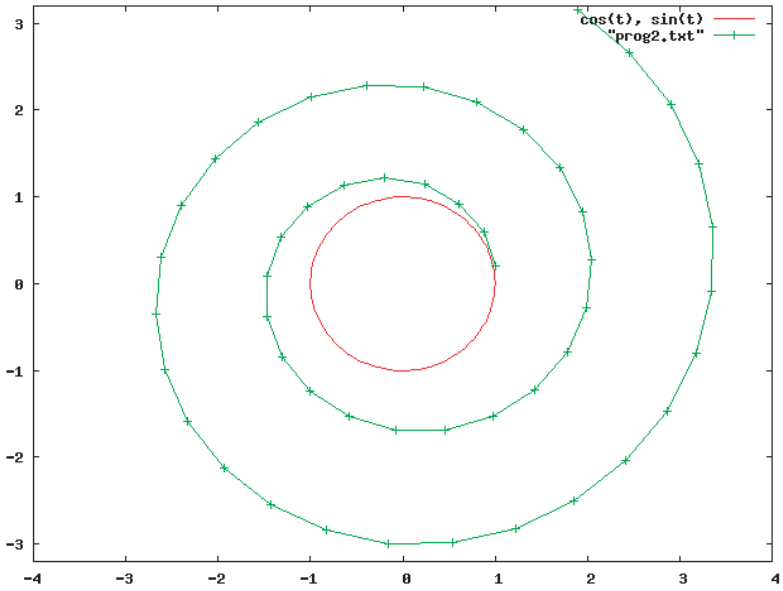


Figure 1.2: Euler 法による惑星運動

シになりますがだんだん外に膨らんでいくのは変わりません。これはオイラー法の欠陥なのです。これで地球の 100 年後を計算すれば太陽系から離脱してしまいます。そこで次にもっと精度の高い計算方法を紹介します。

# Chapter 2

# 高次の差分法(Runge-kutta法)

## 2.1 Toyler法

一般に

$$\frac{\mathrm{dy}}{\mathrm{dx}} \cong \frac{y(x+\Delta x)-y(x)}{\Delta x}$$

という形に近似して計算する方法を差分法といいます。もっともシンプルな形はオイラー法ですがこれをもっと精度のよい近似にするには

$$\frac{y(x+\Delta x)-y(x)}{\Delta x} = \frac{\mathrm{dy}}{\mathrm{dx}} + \frac{1}{2!}\frac{\mathrm{d^2y}}{\mathrm{dx^2}}\Delta x + \frac{1}{3!}\frac{\mathrm{d^3y}}{\mathrm{dx^3}}\Delta x^2 + \cdots$$

と Toyler（テイラー）展開したものを使えばいいのです。これを Toyler 法といいます。たとえば２次までとると

$$y(x+\Delta x) = y(x) + \frac{\mathrm{dy}}{\mathrm{dx}}\Delta x + \frac{1}{2!}\frac{\mathrm{d^2y}}{\mathrm{dx^2}}\Delta x^2$$

です。ここで

$$\frac{\mathrm{dy}}{\mathrm{dx}} = f(x,y)$$

から

$$\frac{\mathrm{d^2y}}{\mathrm{dx^2}} = \frac{\mathrm{df(x,y)}}{\mathrm{dx}} = \frac{\partial f(x,y)}{\partial x} + \frac{\partial f(x,y)}{\partial y}\frac{\mathrm{dy}}{\mathrm{dx}}$$

というふうに次々にｙのｎ回微分が求まるので基本的に何次でも計算できます。ただ、実はこのテイラー法はあまり使われません。何でかというと $f(x,y)$ が簡単なときはいいですが複雑な場合計算がすごく面倒になります。そしてもうひとつは、そんな面倒な計算をしないで済む方法が別にあるからです。

## 2.2 Runge-kutta法

さて、上で言った別の方法というのがこの Runge-kutta（ルンゲ-クッタ）法です。この方法の目的は $f(x,y)$ の微分を使わずに次数を上げることです。そのためにちょっと数式をいじりますががんばってついてきてください。これから２次の Runge-kutta（ルンゲ-クッタ）法を導きますが以下

$$h = \Delta x$$
$$x_n = x_0 + nh$$

とします。まず、計算する式を今

$$\frac{\mathrm{dy}}{\mathrm{dx}} = f(x,y), y(x_0) = y_0$$

だとすると

$$y_{n+1} = y_n + hF(x_n, y_n, h)$$

と差分法にするにあたって

$$F(x, y, h) = \alpha f(x_n, y_n) + \beta f(x_n + ph, y_n + qhf(x_n, y_n))$$

という適当な $f(x, y)$ の足し合わせにしてやります。なぜこの形かと言えばこうするとうまくいくからなんです（笑）さて, まず $f(x_n + ph, y_n + qhf(x_n, y_n))$ を展開します

$$f(x_n + ph, y_n + qhf(x_n, y_n)) = f(x_n, y_n) + ph\frac{\partial f(x_n, y_n)}{\partial x} + qhf(x_n, y_n)\frac{\partial f(x_n, y_n)}{\partial y} + O(h^2)$$

これを元の式にいれ

$$\begin{aligned} y_{n+1} &= y_n + hF(x_n, y_n, h) \\ &= y_n + h\left[\alpha f(x_n, y_n) + \beta\left(f(x_n, y_n) + ph\frac{\partial f(x_n, y_n)}{\partial x} + qhf(x_n, y_n)\frac{\partial f(x_n, y_n)}{\partial y} + O(h^2)\right)\right] \\ &= y_n + h(\alpha + \beta)f(x_n, y_n) + h^2(\beta p\frac{\partial f(x_n, y_n)}{\partial x} + \beta q f(x_n, y_n)\frac{\partial f(x_n, y_n)}{\partial y}) + O(h^3) \end{aligned}$$

さて、ここで $y_{n+1}$ の本当の展開は

$$\begin{aligned} y_{n+1} &= y_n + h\frac{\mathrm{dy_n}}{\mathrm{dx}} + \frac{h^2}{2!}\frac{\mathrm{d^2y_n}}{\mathrm{dx^2}} + O(h^3) \\ &= y_n + h\frac{\mathrm{df}}{\mathrm{dx}} + \frac{h^2}{2!}[\frac{\partial f}{\partial x} + f\frac{\partial f}{\partial y}] + O(h^3) \end{aligned}$$

よって２次まで一致させるには

$$\alpha + \beta = 1, \beta p = \beta q = \frac{1}{2}$$

よって

$$\alpha = 1 - \beta, p = q = \frac{1}{2\beta}$$

ここで $\beta$ は任意です。そのため同じ２次のルンゲ-クッタでもいくつか存在します。ふつう $\beta = \frac{1}{2}, 1$ です。

具体的に書くと、$\beta = \frac{1}{2}, \alpha = \frac{1}{2}, p = q = 1$ の場合

$$y_{n+1} = y_n + \frac{1}{2}h\left[f(x_n, y_n) + f(x_n + h, y_n + hf(x_n, y_n))\right]$$

なので実際の計算手順は

$$\begin{aligned} \bar{y}_{n+1} &= y_n + hf(x_n, y_n) \\ y^*_{n+1} &= y_n + hf(x_{n+1}, \bar{y}_{n+1}) \\ y_{n+1} &= \frac{\bar{y}_{n+1} + y^*_{n+1}}{2} \end{aligned}$$

となります。これをつかって早速、オイラー法でいまいちだった重力の問題

$$\begin{aligned} &\frac{\mathrm{d^2x}}{\mathrm{dt^2}} = -\frac{x}{r^3} \\ &\frac{\mathrm{d^2y}}{\mathrm{dt^2}} = -\frac{y}{r^3} \\ &x(0) = 1, y(0) = 0, \frac{\mathrm{dx}(0)}{\mathrm{dt}} = 0, \frac{\mathrm{dy}(0)}{\mathrm{dt}} = 1 \\ &r = \sqrt{x^2 + y^2} \end{aligned}$$

を解いてみましょう。このソースはこちらです

```
#include <stdio.h>
#include <math.h>

int main(void){
 double x[3] ={1,0,0};
 double u[3] ={0};
 double y[3] ={0};
 double v[3] ={1,0,0};
 int k;

 for(k=1;k <= 100; k++){
  u[1] =u[0]+ -0.2 * x[0] / sqrt(y[0]*y[0]+x[0]*x[0]);
  x[1] =x[0]+ 0.2 * u[0];
  v[1] =v[0]+ -0.2 * y[0] / sqrt(y[0]*y[0]+x[0]*x[0]);
  y[1] =y[0]+ 0.2 * v[0];
  u[2] =u[0]+ -0.2 * x[1] / sqrt(y[1]*y[1]+x[1]*x[1]);
  x[2] =x[0]+ 0.2 * u[1];
  v[2] =v[0]+ -0.2 * y[1] / sqrt(y[1]*y[1]+x[1]*x[1]);
  y[2] =y[0]+ 0.2 * v[1];
  u[0] =(u[1]+u[2])/2;
  x[0] =(x[1]+x[2])/2;
  v[0] =(v[1]+v[2])/2;
  y[0] =(y[1]+y[2])/2;

  if(k%2==0) printf("%f %f\n",x[0],y[0]);
 }
 return 0;
}
```

その結果はこちら
結果は明らかですね。オイラー法に比べるとメッシュの幅が一緒なのにこれほど違うんです。実際の物理で

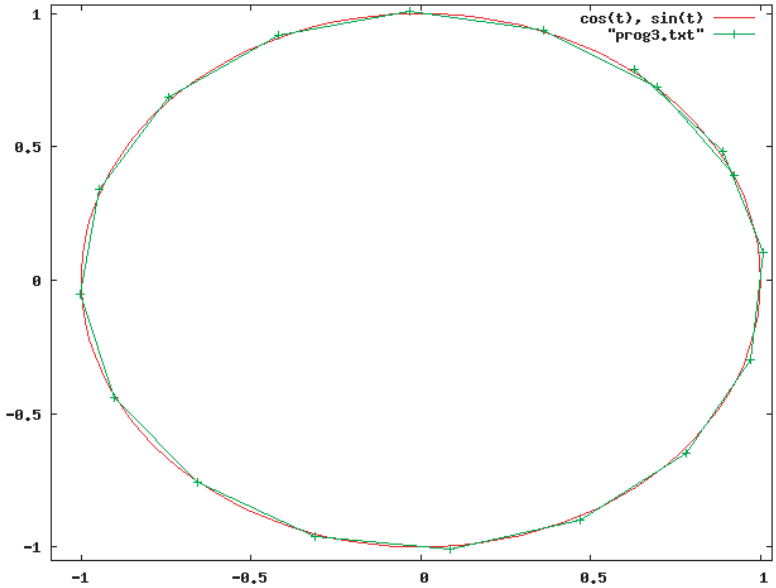


Figure 2.1: 2次の RK 法

も RK 法はよく使われているそうです。

# Chapter 3

# 高次のRunge-Kutta法

２次より高次の RK 法は同じように

$$y_{n+1} = y_n + hF(x_n, y_n, h)$$
$$F = \sum_{i=1}^{m} \alpha_i k_i$$
$$k_i = f(x_n + \sum_{j=1}^{m} P_{ij}, y_n + \sum_{j=1}^{m} P_{ij} k_j)$$

というふうに $f(x, y)$ を適当に組み合わせることで得られます。ここで以下

$$\begin{pmatrix} P_{11} & P_{12} & \cdots & P_{1m} \\ P_{21} & P_{22} & \cdots & P_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ P_{m1} & \cdots & \cdots & P_{mm} \\ \alpha_1 & \cdots & \cdots & \alpha_m \end{pmatrix}$$

といふうにm＋１行m列の行列の形（Stetter の行列表現）で高次の RK 法を紹介していきます。

## 3.1 ２次の RK 法

先ほどやったやつですがこの行列で表すと

$$\begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 1/2 & 1/2 \end{pmatrix}$$

また、さきほど $\beta = 1$ の場合は

$$\begin{pmatrix} 0 & 0 \\ 1/2 & 0 \\ 0 & 1 \end{pmatrix}$$

となります。

### 3.1.1 ３次の RK 法

$$\begin{pmatrix} 0 & 0 & 0 \\ 1/2 & 0 & 0 \\ -1 & 2 & 0 \\ 1/6 & 4/6 & 1/6 \end{pmatrix}$$

が３次の RK 法です。もちろん係数は代表的なものであって、これが唯一ではありません。ただ、３次はあまり使われません。それは手間や精度などを考慮すると４次のほうがお得だからです。

## 3.2　４次の RK 法

４次では例えば

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1/6 & 2/6 & 2/6 & 1/6 \end{pmatrix}$$

のようなものが有名です。４次の RK 法はかなりよく使われるものです。ためしにこれであの

$$\frac{\mathrm{d}^2\mathrm{x}}{\mathrm{dt}^2} = -\frac{x}{r^3}$$
$$\frac{\mathrm{d}^2\mathrm{y}}{\mathrm{dt}^2} = -\frac{y}{r^3}$$
$$x(0) = 1, y(0) = 0, \frac{\mathrm{dx}(0)}{\mathrm{dt}} = 0, \frac{\mathrm{dy}(0)}{\mathrm{dt}} = 1$$
$$r = \sqrt{x^2 + y^2}$$

を解いてみましょう。具体的な計算手順は

$$k_1 = f(x_n, y_n)$$
$$k_2 = f(x_n + h/2, y_n + k_1 h/2)$$
$$k_3 = f(x_n + h/2, y_n + k_2 h/2)$$
$$k_4 = f(x_n + h, y_n + k_3 h)$$
$$y_{n+1} = y_n + (k_1 + 2k_2 + 2k_3 + k_4)h/6$$

となります。このソースは

```
/*四次 RK 法*/

#include <stdio.h>
#include <math.h>

int main(void){
 double x =1;
 double kx[5] ={0};
 double u =0;
 double ku[5] ={0};
 double y =0;
 double ky[5] ={0};
 double v =1;
 double kv[5] ={0};
 double h;
 int k,t;

 for(k=0;k < 100; k++){
  for(t=0;t <=3;t++){
   if(t<=2) h=0.1;
   else h=0.2;
```

```
   ku[t+1] =- (x+h *kx[t]) / sqrt((y+h *ky[t])*(y+h*ky[t])+(x+h *kx[t])*(x+h *kx[t]));
   kx[t+1] = u+h *ku[t];
   kv[t+1] = -(y+h *ky[t]) / sqrt((y+h *ky[t])*(y+h*ky[t])+(x+h *kx[t])*(x+h *kx[t]));
   ky[t+1] =  v+h *kv[t];
  }

  u =u+(ku[1]+2*ku[2]+2*ku[3]+ku[4])*0.2/6;
  x =x+(kx[1]+2*kx[2]+2*kx[3]+kx[4])*0.2/6;
  v =v+(kv[1]+2*kv[2]+2*kv[3]+kv[4])*0.2/6;
  y =y+(ky[1]+2*ky[2]+2*ky[3]+ky[4])*0.2/6;

  if(k%2==0) printf("%f %f\n",x,y);
 }
 return 0;
}
```

です。この結果が
 です。点の位置はほぼ理論値と一致していますね。確かにこの４次の RK 法がよく使われる理由がわかって

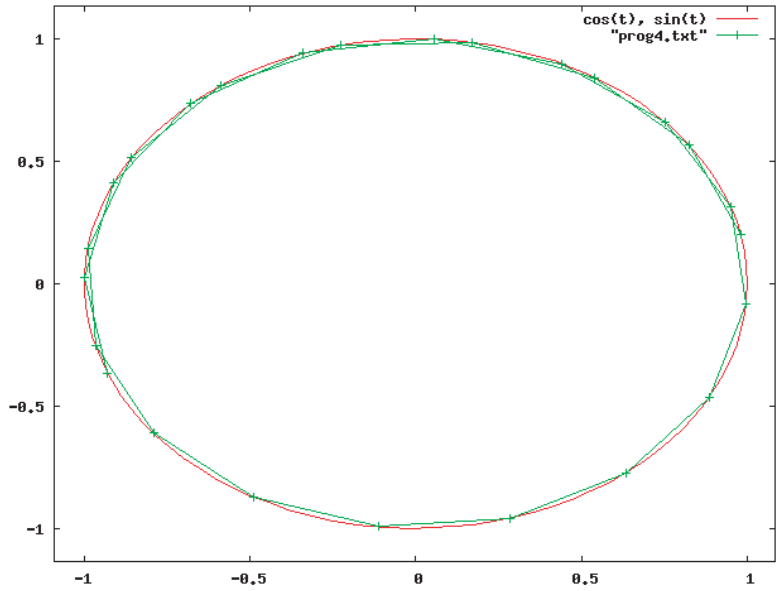


Figure 3.1: ４次の RK 法

いただけるかと思います。