[研究開発公募の成果]

Taylor 級数演算ライブラリの使用方法

平山　弘　舘野裕文　浅野直之　川口隆史

神奈川工科大学 工学部

概要　数値計算では、微分演算は差分近似で計算するのがこれまでの常識である。差分近似は精度があまり良くないため、数値計算では微分を使う計算法はあまり使われてこなかった。Taylor展開ライブラリは、１変数の関数に限られるが微分を通常の数値計算と同程度の精度で計算する手段を与えるものである。このため、微分演算を数値計算で使わない理由はなくなる。

このライブラリでは、プログラムとして定義された関数のTaylor展開、常微分方程式の解のTaylor展開ができる。逆関数は常微分方程式の解として表現できるので、逆関数のTaylor展開もできる。これらのプログラムの考え方、使い方を説明する。

1. Taylor級数プログラムとは

Taylor級数のライブラリとは、C++言語[3]用のライブラリでプログラムの形で与えられた関数を指定された次数のTaylor級数に展開するためのプログラムである。以下で、

（１）　関数のTaylor展開

（２）　常微分方程式の解のTaylor展開

について、簡単な例をあげて説明する。

関数がTaylor展開できると、その関数の微分積分が容易にできると同時に、微分係数も容易に計算できる。数値計算では、微分係数は、hを小さな数値とすると

$$f'(x) \cong \frac{f(x+h)-f(x)}{h} \tag{1.1}$$

と近似して計算することになる。この計算には打ち切り誤差を含むためあまり精度の良い計算にはならない。このため、これまでの数値計算では微分を含む計算は行うべきでないとされてきた。Taylor展開ライブラリでは、このような計算を行っていないため、高精度で計算できる。いわゆる自動微分のアルゴリズムを使って計算している。

たとえば、$f(x)=\sqrt{1+x+x^2+x^3}$ の $x=1$ における微分係数を求めるには、$f(x)$ を $x=1$ でTaylor展開する。まず、平方根の中をTaylor展開する。

$$\begin{aligned} 1+x+x^2+x^3 &= 1+\{1+(x-1)\}+\{1+(x-1)\}^2+\{1+(x-1)\}^3 \\ &= 4+6(x-1)+4(x-1)^2+(x-1)^3 \end{aligned} \tag{1.2}$$

このようにTaylor展開すれば、打ち切り誤差を含む極限操作を行なわないので、精度良くTaylor展開できる。このTaylor展開の係数は、微分係数を階乗で割ったものである

から、階乗を掛けることによって、微分係数を求めることができる。次に $f(x)$ のTaylor展開を求める。$f(x)$ を微分すると

(1.3) $$f'(x)=\frac{6+8(x-1)+3(x-1)^2}{2\sqrt{4+6(x-1)+4(x-1)^2+(x-1)^3}}=\frac{6+8(x-1)+3(x-1)^2}{2f(x)}$$

が得られる。分母に $f(x)$ を掛けると

(1.4) $$f(x)f'(x)=3+4(x-1)+\frac{3}{2}(x-1)^2$$

となる。ここで、

(1.5) $$f(x)=f_0+f_1(x-1)+f_2(x-1)^2+\cdots+f_n(x-1)^n+\cdots$$

と仮定して、(1.4)に代入し、展開する。Taylor展開の係数が等しいと置くと、次の関係式が得られる。定数項を等しいと置くと

(1.6) $$f_0f_1=3$$

が得られる。１次の係数が等しいと置くと

(1.7) $$2f_0f_2+f_1^2=4$$

さらに、２次の係数が等しいと置くと

(1.8) $$3f_0f_3+2f_1f_2+f_2f_1=\frac{3}{2}$$

が得られる。一般に $m(m>3)$ 次の係数を等しいと置くと

(1.9) $$\sum_{k=0}^{m}(m+1-k)f_kf_{m+1-k}=0$$

が得られる。$f(1)=2$ であるから $f_0=2$ となる。これから、

(1.10) $f_1=\frac{3}{2}$、$f_2=\frac{7}{16}$、$f_3=-\frac{5}{64}$、$f_4=\frac{11}{1024}$、$f_5=\frac{37}{4096}$、$f_6=-\frac{349}{32768}$

が得られる。すなわち

(1.11) $$f(x)=2+\frac{3}{2}(x-1)+\frac{7}{16}(x-1)^2-\frac{5}{64}(x-1)^3+\frac{11}{1024}(x-1)^4+\cdots$$

となる。このTaylor展開式から微分係数を求めると

０次の係数　$f(1)$　　１次の係数　$f'(1)$

２次の係数　$\frac{f''(1)}{2!}$　　３次の係数　$\frac{f'''(1)}{3!}$

であるから、$f(1)=2$、$f'(1)=\frac{3}{2}$、$f''(1)=\frac{7}{8}$、$f'''(1)=\frac{15}{32}$ となる。４階微分以降も同様に計算できる。

ここでは、計算に分数を使ったが、これは最初の定数 f_0 が有理数ならば、それ以降の計算には四則演算しか行われないので、有理数で厳密に表現できる。このことを示す

ために、ここでは有理数計算を使っている。通常、この計算は倍精度浮動小数点数を使い高速に計算する。(1.4)の微分方程式を解いてこの関数のTaylor展開を求めたが、(1.4)の代わりに

(1.12) $$f(x)^2 = 4 + 6(x-1) + 4(x-1)^2 + (x-1)^3$$

として、(1.5)の式を代入しても同様な結果が得られる。このように、Taylor展開の係数は打ち切り誤差の入らない計算法で計算できる。この例では平方根の計算には誤差が入らないが、一般には平方根の計算には打切り誤差が入る。この打切り誤差は、C++言語の標準関数ならば丸め誤差程度となるように最適化されている場合が多いので、計算全体としては丸め誤差程度であると考えられる。

1.1 関数のTaylor級数展開

以下のプログラムは、$x = 2$ における関数

(1.13) $$f(x) = \sqrt{x+1}\sin(x^2+1)$$

の値を計算するC++言語プログラムである。

List 101 通常の数値計算プログラム

```
 1 : #include <iostream> // th11.cpp
 2 : #include <cmath>
 3 : using namespace std ;
 4 : int main()
 5 : {
 6 :     double x, y, z ;
 7 :     x = 2.0 ;
 8 :     cout << x << endl ;
 9 :     y = sqrt(x+1.0)*sin(x*x+1.0) ;
10 :     cout << y << endl ;
11 : }
```

このプログラムを実行すると

```
  2
-1.66091
```

となる。このプログラム(List 101)を書き換えて、$x = 2$ におけるTaylor級数を計算する。書き換えたプログラムは次のようになる。

List 102 Taylor展開式を計算するプログラム

```
 1 : #include "taylor_template.h" // th12.cpp
 2 : typedef taylor_template<double>  taylor ;
 3 : using namespace std ;
 4 : int main()
 5 : {
 6 :     taylor x, y, z ;
 7 :     x = taylor( 2.0, 2.0, 1.0 ) ;
```

```
 8 :     cout << x << endl ;
 9 :     y = sqrt(x+1.0)*sin(x*x+1.0) ;
10 :     cout << y << endl ;
11 : }
```

書き換えた部分には、下線を引いてある。1と2行目の宣言部分と6行目の変数宣言、7行目の変数xの値を設定する部分である。このプログラム実行するとｘとｙが、次の結果が出力される。

```
2+(x-2)
-1.66091+1.68845*(x-2)+14.1292*(x-2)^2+3.66819*(x-2)^3
-20.7686*(x-2)^4-17.9645*(x-2)^5+5.83217*(x-2)^6
```

式の表示は、C++言語の式と同じ形式になるように記述しているが、べき乗は関数powを使わないで記号^を使っている。

上の出力は見やすくするため、2行目の３次の項の後に改行を入れ、６次項以降を省略している。このような計算ができるのがTaylor級数ライブラリである。上のプログラム(List 102)を説明する。

1行目　Taylor級数の定義しているファイルを読み込む。

2行目　Taylor級数ライブラリはtemplateとして定義されている。係数の型をdouble型にする。係数の型をdoubleにしたものをtaylor型と定義する。このように再定義して使うことを前提にこのライブラリは作られている。もしこのように再定義しない場合、taylor_template<double>が名前になり大変長い型名になり面倒である。このtemplateのクラス名が長いのは、クラス名を長い名前にして、ユーザの使いやすい名前と重ならないようにするためである。

6行目　taylor型の変数x,y,zを定義する。

7行目　$f(x)=x$ を $x=2$ で $f(x)=2+(x-2)$ とTaylor展開し、それを変数 x に代入する。

taylor（p, a0, a1）

は、展開位置がpで、定数項がa0、１次の係数がa1であるTaylor級数を定義する。すなわち

$$a_0+a_1(x-p)$$

を意味する。

以降の説明にあるように、Taylor級数演算ライブラリは、Taylor級数の演算だけ定義している。計算を始めるには、上の例のように最初はユーザがTaylor展開の係数を指定する必要がある。

8行目　x を $x=2$ で展開した式を出力する。

9行目　関数 $f(x)=\sqrt{x+1}\sin(x^2+1)$ を $x=2$ で展開する。

10行目　$f(x)$ を $x=2$ で展開した式を出力する。

Taylor級数ライブラリを使うと、このようなTaylor展開式の計算が通常の数値計算と同程度の速さで計算できる。数式処理と似たような計算であるが通常の数値計算と同じく高速である。Templateライブラリなので Taylor級数の係数として、複素数やlong double

などが使える。分数や高精度数などのクラスが使えるならば、このような型も使える。係数の型として分数や高精度数を使えば通常の数式処理と同じように計算に時間がかかり、高速性も失われる。

Taylor級数に展開できれば、Taylor級数の係数が微分係数の定数倍で表せるので、これから微分係数が求められる。この計算には、差分を使わない、いわゆる自動微分の公式を使って計算している。この計算には打ち切り誤差が入らないので、高精度の微分係数が容易に計算できる。当然ながら、丸め誤差があるので、常に高精度で計算できるわけではない。

1.2 Taylor級数法による微分方程式の解法

Taylor級数を使う方法の利点のもう一つは、常微分方程式の解をTaylor展開できることである。逆関数の計算も逆関数を満たす微分方程式をTaylor展開することによって求められる。微分方程式をTaylor展開する研究としては、Corless等[2]の研究がある。FORTRANのソースプログラムを書き換える方法で行うものである。ここでは、C++言語のオーバーロード機能を使って書き換えと同様な機能を実現している。Fortran 90以降のFortran言語でも同様に記述できる。

次のように$\frac{dy}{dx}$について解かれた形になっている常微分方程式について考える。

(1.14) $$\frac{dy}{dx} = f(x,y) \qquad \text{初期条件} \quad y(0) = a_0$$

ここで、yおよびfは、一般にベクトルである。解yはxについて、Taylor級数展開できるものとする。この微分方程式の解は、Picardの逐次計算法[7]

(1.15) $$y_0 = a_0$$

(1.16) $$y_n = a_0 + \int_0^x f(x,y_{n-1})dx$$

によって計算できる。

例として、次の微分方程式を解く。

(1.17) $$\frac{dy}{dx} = \sin(y), \quad y(0) = 1$$

解は区間[0,1]で求める。この問題にはPicardの逐次計算法を利用すれば、解をTaylor展開できる。このプログラムは次のようになる。

List 103 微分方程式の解のTaylor展開

```
1 : #include "taylor_template.h" // th13.cpp
2 : typedef taylor_template<double>  power ;
3 : void func( const power& x, const power& y, power& dy )
4 : {
5 :       dy = sin(y) ;         //  方程式の右辺を計算する関数
```

```
 6 : }
 7 : int main()
 8 : {
 9 :    double  x0, y0 ;
10 :    power   x, y, dy ;
11 :    x0 = 0 ;    y0 = 1 ;          //  初期条件
12 :    y = power( x0, y0 ) ;
13 :    x = power( x0, x0, 1.0 ) ;    //  x を x0＋1.0*(x-x0) と展開する。
14 :    for( int i=0 ; i<7 ; i++ )    //   7次まで計算する。
15 :    {
16 :       y.set_degree(i+1) ;        //  高次の項の計算を省略するために指定
17 :       func( x, y, dy ) ;         //  方程式の右辺の計算を行う
18 :       y = y0+integrate(dy) ;     //  ピカールの逐次反復公式
19 :       cout << y << endl ;
20 :    }
21 : }
```

このプログラムを実行すると、１回反復ごとに１次だけ精度が上がることがわかる。ここでは、５次の解まで表示する。

```
1+0.841471*x
1+0.841471*x+0.227324*x^2
1+0.841471*x+0.227324*x^2-0.0583626*x^3
1+0.841471*x+0.227324*x^2-0.0583626*x^3-0.0615375*x^4
1+0.841471*x+0.227324*x^2-0.0583626*x^3-0.0615375*x^4-0.0079143*x^5
```

このような反復計算を行い微分方程式の解のTaylor級数を計算する。ライブラリにはこの計算を行う関数picardが準備されている。

このTaylor展開式を利用して、要求する精度からステップ幅を決めることもできる。ステップ幅が決まれば、このステップ幅だけ進んだ x の値とその点における y の値がTaylor展開式を利用して計算することができる。

計算されたTaylor級数の有効範囲を次のように定義する。Taylor級数の有効範囲とは、εを要求精度を与えたとき、このTaylor級数を使って要求精度以上で計算できる範囲を言う。これは、Taylor級数の収束半径から類推したものである。有効範囲の半径を r とすると、最高次の n 次の項の係数を a_n としたとき、Taylor級数の有効範囲では、

$$\left|a_n r^n\right| \leq \varepsilon \tag{1.18}$$

を満たす必要がある。第n項が十分小さく、それ以上の項が無視できると仮定している。この仮定が成り立つならば、この範囲内であれば、Taylor級数を使って要求精度以上で計算できる。最高次の項が0の場合や偶然最高次の項が他の項に比べ異常に小さいような場合、この考え方はうまくいかない。高次の項が無視できるほど小さいという仮定が成り立たないからである。最高次の項だけでなく、高次の２、３項を使い、(1.18)が成

り立つようにすれば、より信頼性ある有効範囲が計算できる。(1.18)の式から

(1.19) $$r = \sqrt[n]{\frac{\varepsilon}{|a_n|}}$$

と有効範囲の半径が計算できる。複数の項でこれらの半径を求めた場合、それらの最小値が、求める半径となる。展開点から半径 r 内の数は要求精度以上で計算できるから、次のステップを r だけ進めれば要求精度内で次のステップにおける関数値を計算できる。以下でその例を示す。(1.14)の微分方程式を精度1.0e-10で解く。プログラムは、以下のようになる。

List 104 Taylor展開による微分方程式の解法

```
 1 : #include "taylor_template.h" // th14.cpp
 2 : typedef taylor_template<double> power ;
 3 : void func( const power& x, const power& y, power& dy )
 4 : {
 5 :       dy = sin(y) ;
 6 : }
 7 : int main()
 8 : {
 9 :   double y0, x0, h, eps, r, t ;
10 :    power x, y, dy, p, q ;
11 :    eps = 1.0e-12 ;                //   要求精度
12 :    x0 = 0 ;  y0 = 1 ;             //   初期条件
13 :    cout << x0 << "  " << y0 << endl ;
14 :    while( x0 < 1.0 )
15 :    {
16 :       y = power(x0, y0) ;
17 :        x = power( x0, x0, 1.0 ) ;   //  x を x0＋1.0*(x-x0) と展開する。
18 :       for( int i=0 ; i<15 ; i++ )
19 :       {
20 :          y.set_degree(i+1) ;    // 高次の項の計算を省略するために指定
21 :          func( x, y, dy ) ;
22 :          y = y0+integrate(dy) ;    //      ピカールの逐次反復公式
23 :       }
24 :       int m = y.cur_deg ;
25 :       h = 100000.0 ;
26 :       for( int j=0 ; j<4 ; j++ )
27 :       {
28 :          t = y.coeff[m-j] ;
29 :          if(t==0.0) continue ;
```

```
30 :            r = pow( eps/abs(t), 1.0/(m-j) ) ; // 有効範囲の計算
31 :            if( r < h ) h = r ;
32 :        }
33 :        x0 += h ;              //  eval関数は多項式を評価する関数
34 :        cout << y << endl ;
35 :        y0=subst(y,x0) ;         //  次のステップのｙの値の計算
36 :        cout << x0 << "  " << y0 << endl ;
37 :    }
38 : }
```

これを計算すると次の結果が得られる。Taylor級数は15次まで計算しているが、ここでは、見やすくするため、５次まで表示している。xで展開された級数を右に表示する。次のxの値まで、このTaylor展開を使えば精度よく計算できる。Taylor級数を微分すれば、微分係数も精度よく計算できる。微分を何回もすれば、それだけTaylor展開式も次数が低くなり、徐々に精度が悪くなる。

x	xのおけるTaylor級数
0.00000	1+0.841471*x+0.227324*x^2-0.0583626*x^3-0.0615375*x^4-0.0079143*x^5
0.19618	1.1733+0.922031*(x-0.19618)+0.178466*(x-0.19618)^2 -0.107614*(x-0.19618)^3-0.0609886*(x-0.19618)^4+0.0103187*(x-0.19618)^5
0.420174	1.38742+0.983234*(x-0.420174)+0.0896444*(x-0.420174)^2 -0.152975*(x-0.420174)^3-0.0358615*(x-0.420174)^4+0.0335572*(x-0.420174)^5
0.60402	1.57023+1*(x-0.60402)+0.000281095*(x-0.60402)^2 -0.166667*(x-0.60402)^3-0.000117123*(x-0.60402)^4+0.0416666*(x-0.60402)^5
0.820579	1.78513+0.977118*(x-0.820579)-0.103916*(x-0.820579)^2 -0.148118*(x-0.820579)^3+0.0409477*(x-0.820579)^4+0.0307986*(x-0.820579)^5
1.00471	

この式を利用して、一定間隔刻みの関数値も容易に計算できる。

yを計算するとき、Taylor展開式をPadé展開に変換して計算することによってA安定な計算法[7]となる。これをプログラムするとList 104の34～35行目を、次のように置き換えたものになる。

```
34:     pade(y,p,7,q,7) ;
35:     y0=subst(p,x0)/subst(q,x0) ;   //     次のステップのｙの値の計算
```

このプログラムを実行すると、以下のような解が得られる。

0	1
0.19618	1.1733
0.420174	1.38742
0.60402	1.57023
0.820579	1.78513
1.00471	1.960650

この例題ではあまり大きな違いはないが、長時間計算を行うとその違いが徐々にでてくる。

1.3 Taylor級数法による連立微分方程式の解法

Taylor級数は、１階微分方程式だけでなく、高階微分方程式や連立微分方程式も容易に解ける。高階微分方程式は、１階連立微分方程式に変形して解く。これはRunge-Kutta法など他の数値解法と同様である。

ここでは、例題としてよく取り上げられるvan der Pol方程式を解く。van der Pol方程式

(1.20) $\dfrac{d^2y}{dt^2}-K(1-y^2)\dfrac{dy}{dt}+y=0$　　初期条件　$t=0$ のとき $y=1$、$\dfrac{dy}{dt}=1$

を解く。この方程式は次のように1階連立微分方程式

(1.21) $$\begin{cases}\dfrac{dy}{dt}=z\\ \dfrac{dz}{dt}=K(1-y^2)z-y\end{cases}$$　初期条件　$t=0$ のとき $y=1$、$z=1$

に変形できる。以下にこの方程式（K=0.5）を解くプログラムを示す。ここでは微分方程式の解をTaylor展開する関数picardを使っている。

List 105 Taylor展開によるvan der Pol方程式の解法

```
 1 : #include "taylor_template.h" // th15.cpp
 2 : typedef taylor_template<double> power ;
 3 : void func( const power& x, const power* y, power* dy, int n )
 4 : {
 5 :         dy[0] = y[1] ;
 6 :         dy[1] = 0.5*(1.0-y[0]*y[0])*y[1]-y[0] ;
 7 : }
 8 : int main()
 9 : {
10 :     double y0[2], x0, h, eps, r, t, xp, xh ;
11 :       power x, y[2], dy[2] ;
12 :       int     m, n, d ;
13 :       n = 2 ;                                  //   ２元連立方程式
14 :       m = 15 ;                                 //   15次まで展開
15 :       eps = 1.0e-12 ;                          //     要求精度
16 :       x0 = 0 ; y0[0] = 1 ; y0[1] = 1 ;   //     初期条件
17 :       xp =0 ; xh = 0.1 ;                       //     xpからxh間隔で出力
18 :       while( x0 < 1.0 )
19 :       {
20 :         picard( &func, x0, y0, y, n, m ) ; //  初期値x0,y0からm次の
                                                  //  展開式を計算
```

```
21 :            for( int i=0 ; i<2 ; i++ )
22 :            {
23 :                d = y[i].cur_deg ;
24 :                h = 100000.0 ;
25 :                for( int j=0 ; j<3 ; j++ )
26 :                {
27 :                    t = y[i].coeff[d-j] ;
28 :                    if(t==0.0) continue ;
29 :                    r = pow( eps/abs(t), 1.0/(d-j) ) ; // 有効範囲の計算
30 :                    if( r < h ) h = r ;
31 :                }
32 :            }
33 :            x0 += h ;
34 :            while( xp < x0 ) // 等間隔に出力
35 :            {
36 :                y0[0]=subst(y[0],xp) ;
37 :                y0[1]=subst(y[1],xp) ;
38 :                cout << xp << "    " << y0[0] << "    " << y0[1] << endl ;
39 :                xp += xh ;
40 :            }
41 :            y0[0]=subst(y[0],x0) ;          //    次のステップのy0の値の計算
42 :            y0[1]=subst(y[1],x0) ;
43 :        }
44 : }
```

このプログラムを実行すると、４回のTaylo展開で

0	1	1
0.1	1.09468	0.890551
0.2	1.17756	0.76476
0.3	1.24723	0.627022
0.4	1.30273	0.482038
0.5	1.34355	0.334276
0.6	1.36962	0.187541
0.7	1.38119	0.0447226
0.8	1.37876	-0.0922892
0.9	1.36296	-0.222537
1	1.33448	-0.34585

の計算結果が得られる。表示されている数値は、別の方法を使い、確認している。

2. Taylor級数の演算

Taylor級数は、プログラムとしては、係数を配列として表現する。Taylor級数を$x=a$で展開したときの式を

(2.1) $$f(x) = f_0 + f_1(x-a) + f_2(x-a)^2 + f_3(x-a)^3 + \cdots$$

とする。この中の低次の係数m個を取り、m次を越える高次係数を省略する。係数

(2.2) $$f_0,\ f_1,\ f_2,\ f_3,\ f_4,\ \cdots,\ f_m$$

を配列として表現する。Taylor級数は、この係数配列と展開位置aで定義する。Taylor級数は、次のような、構造体として定義する。C++言語では、

```
template <typename T>
class   taylor_template
{
    int      m  ; // 宣言されている係数の配列大きさ－１(alloc_siz)
    int      n  ; // Taylor展開の次数（cur_deg）
    T         a ; // 展開位置(pos)
    T        *f ; // 係数のポインター(coeff)
} ;
```

と表す。（）の中の名前はプログラムで使用している変数名である。Taylor級数の計算には、１次や２次の低次の式がかなりの頻度で現れるので、n(cur_deg)を利用して高次の係数の計算を省き、計算の高速化をはかっている。m(alloc_siz)は、Taylor級数変数ごとに必要な記憶容量を取れるようにするための変数で、記憶容量の節約をはかる。このようにTaylor級数型を定義すると同時に、これらの型と言語に備わっている通常の数値型の演算を定義する。演算子の再定義（オペレータ・オーバーロード）では、掛け算の交換法則などは自動的に定義されないので、Taylor級数と倍精度実数の積と倍精度実数とTaylor級数の積を別々に定義しなければならない。

Taylor級数は、一種の数値型変数で通常の数値と同様に計算[5-8]できる。四則演算だけでなく、C++で準備されている多くの関数が使える。平方根、指数対数関数、三角関数、逆三角関数などが使える。

2.1 Taylor級数の四則演算

Taylor級数の四則計算のプログラムは、以下のように簡単に作ることができる。平行移動によって、展開位置を原点に移すことができるので一般性を失うことなしに、原点で展開した式だけを扱うことができる。この級数を次のように定義する。

(2.3) $$f(x) = f_0 + f_1 x + f_2 x^2 + f_3 x^3 + \cdots$$

(2.4) $$g(x) = g_0 + g_1 x + g_2 x^2 + g_3 x^3 + \cdots$$

(2.5) $$h(x) = h_0 + h_1 x + h_2 x^2 + h_3 x^3 + \cdots$$

四則演算は、以下のように定義できる。これらの公式は簡単なものであるがまとめて、記載されている文献があまりないので以下に記載する。ここで、mは、演算の対象と

なっているTaylor級数の次数である。

（1）和差　$\mathrm{h}(x) = \mathrm{f}(x) \pm \mathrm{g}(x)$

係数は次の式によって計算することができる。

(2.6)　$$\mathrm{h}_n = \mathrm{f}_n \pm \mathrm{g}_n \qquad (n = 0, \cdots, m)$$

（2）乗算　$\mathrm{h}(x) = \mathrm{f}(x)\,\mathrm{g}(x)$

乗算の係数は次の式によって計算することができる。

(2.7)　$$\mathrm{h}_n = \sum_{k=0}^{n} \mathrm{f}_k \mathrm{g}_{n-k} \qquad (n = 0, \cdots, m)$$

この計算では、mが100以上の高次のTaylor展開の乗算のとき、高速フーリエ変換を使った計算法が高速であるが、ここでは、100次以下のTaylor級数を想定しているので、本ライブラリには、高速フーリエ変換を利用した乗算ルーチンは組み込まれていない。

（3）除算　$\mathrm{h}(x) = \dfrac{\mathrm{f}(x)}{\mathrm{g}(x)}$

除算の係数は次の式によって計算することができる。式からわかるように、$\mathrm{g}_0 = 0$ のときは、計算することはできない。ただし、$\mathrm{f}_0 = \mathbf{g}_0 = 0$ の場合は、分子と分母を x で割り、$\mathrm{g}_0 \neq 0$ にできる場合がある。極限を計算する場合にはこの操作を行う必要がある。この操作を行うには、$\mathrm{g}_0 = 0$ かどうか判定する必要がある。浮動小数点数ではこの判定は厳密にできないので、この判定を行う関数is_zeroを準備してある。必要に応じてこの関数を変更してゼロを定義できる。何も変更しない場合には、絶対値が 10^{-10} より小さい場合ゼロと判定している。この操作で、$\mathrm{g}_0 \neq 0$ になれば、以下の式で除算を行うことができる。

(2.8)　$$\mathrm{h}_n = \frac{1}{\mathrm{g}_0}\left(\mathrm{f}_n - \sum_{k=0}^{n-1} \mathrm{h}_k \mathrm{g}_{n-k}\right) \qquad (n = 0, \cdots, m)$$

この公式は、$g(x)h(x) = f(x)$ とおいて、(2.3)、(2.4)、(2.5)の式を代入して、展開し、各次数の係数が等しいと置いて得られる。

（4）逆数　$h(x) = \mathrm{invers}(f(x)) = \dfrac{1}{f(x)}$

逆数の係数は次の式によって計算することができる。除算と同じ方法で得られる。除算と同じように、$\mathrm{g}_0 = 0$ のときは、計算することはできない。逆関数を求める関数inv_funcとは名前が似ているので注意する必要がある。

(2.9)　$$\mathrm{h}_0 = \frac{1}{\mathrm{f}_0}, \qquad \mathrm{h}_n = -\frac{1}{\mathrm{f}_0}\sum_{k=0}^{n-1} \mathrm{h}_k \mathrm{f}_{n-k} \qquad (n = 0, \cdots, m)$$

（5）二乗 $\mathrm{h}(x) = \mathrm{square}(\mathrm{f}(x)) = \mathrm{f}(x)^2$

二乗の係数には次のような関係式が成り立つ。この式によって、二乗の計算を乗算の約半分の時間で計算することができる。

$$\mathrm{h}_0 = \mathrm{f}_0{}^2$$

$$\mathrm{h}_n = \begin{cases} 2\sum_{k=0}^{\lfloor n/2 \rfloor} \mathrm{f}_k \mathrm{f}_{n-k} & n:odd \\ (\mathrm{f}_{n/2})^2 + 2\sum_{k=0}^{\left\lfloor \frac{n-1}{2} \right\rfloor} \mathrm{f}_k \mathrm{f}_{n-k} & n:even \end{cases} \qquad (n = 1, \cdots, m) \tag{2.10}$$

ただし、記号「$\lfloor x \rfloor$」は x を超えない最大の整数を示す。この計算も乗算と同じように高速フーリエ変換を使えば次数が100次を超えるようなTaylor展開式に対しては高速に計算できるが、本ライブラリは主に100次以下のTaylor級数を対象としているので、高速フーリエ変換を使ったルーチンは組み込まれていない。

2.2 Taylor級数の関数計算

関数の計算は、常微分方程式を級数法で解くアルゴリズムを使って計算することができる。この計算方法は、簡単な常微分方程式のTaylor級数による解法の例になっている。

（6）べき乗 $\mathrm{h}(x) = \mathrm{f}(x)^a$ （a は定数）

この関数は、つぎの微分方程式を満たす。

$$\mathrm{f}(x)\frac{d\,\mathrm{h}(x)}{dx} = a\,\mathrm{h}(x) \tag{2.11}$$

この式の両辺に、(2.3)、(2.4)、(2.5)の式を代入して、各次数の x の係数を等しいと置いて、次の関係式が得られる。

$$\mathrm{h}_0 = \mathrm{f}_0{}^a,$$

$$\mathrm{h}_n = \frac{1}{n\mathrm{f}_0}\sum_{k=1}^{n}\{(a+1)k - n\}\mathrm{f}_k \mathrm{h}_{n-k} \qquad (n = 1, \cdots, m) \tag{2.12}$$

$a = \frac{1}{2}$ とおけば、平方根を計算するためのプログラムになる。上の式を単純に計算すると、$f_0 = 0$ のとき、計算ができなくなるが、$f_0 = 0$ であっても、$f_k = 0 \quad (k < p)$、$f_p \neq 0$、$a > 0$ で ap が整数ならば、計算可能で、計算結果はTaylor級数になる。たとえば、

$$\sqrt{x^2 + x^3} = x + \frac{1}{2}x^2 - \frac{1}{8}x^3 + \cdots \tag{2.13}$$

となる。本ライブラリでは、このような場合でも問題なく計算できるようになっている。

このような計算をするために、ap が整数かどうか判定する必要がある。浮動小数点数ではその判定は厳密にはできない。本ライブラリでは、is_integerの関数を準備して、その判定を行っている。何も変更していない場合、最寄の整数との差の絶対値が 10^{-10} より小さい場合、整数と判定している。

（7）指数関数　$\mathrm{h}(x) = e^{\mathrm{f}(x)}$

この指数関数は、次の微分方程式を満たす。

$$\frac{d\,\mathrm{h}(x)}{dx} = \mathrm{h}(x)\frac{d\,\mathrm{f}(x)}{dx} \tag{2.14}$$

この式から、べき乗計算の場合と同様な方法で、次のような関係式が得られる。

$$\mathrm{h}_0 = e^{\mathrm{f}_0}, \qquad \mathrm{h}_n = \frac{1}{n}\sum_{k=1}^{n} k\,\mathrm{h}_{n-k}\,\mathrm{f}_k \qquad (n = 1,\cdots,m) \tag{2.15}$$

（8）対数関数　$\mathrm{h}(x) = \mathrm{logf}(x)$

この関数は、次の微分方程式を満たす。

$$\mathrm{f}(x)\frac{d\,\mathrm{h}(x)}{dx} = \frac{d\,\mathrm{f}(x)}{dx} \tag{2.16}$$

この式から、べき乗計算の場合と同様な方法で、次のような関係式が得られる。

$$\mathrm{h}_0 = \mathrm{logf}_0, \quad \mathrm{h}_n = \frac{1}{n\,\mathrm{f}_0}\left(n\,\mathrm{f}_n - \sum_{k=1}^{n-1} k\,\mathrm{h}_k\,\mathrm{f}_{n-k}\right) \tag{2.17}$$

$(n = 1,\cdots,m)$

（9）三角関数　$\mathrm{g}(x) = \sin\mathrm{f}(x), \quad \mathrm{h}(x) = \cos\mathrm{f}(x)$

この二つの関数は、次の連立微分方程式を満たす。

$$\begin{aligned} \frac{d\,\mathrm{g}(x)}{dx} &= \mathrm{h}(x)\frac{df(x)}{dx}, \\ \frac{d\,\mathrm{h}(x)}{dx} &= -\mathrm{g}(x)\frac{df(x)}{dx} \end{aligned} \tag{2.18}$$

この式から、係数に対する次のような関係式が得られる。

$$\begin{aligned} &\mathrm{g}_0 = \sin\mathrm{f}_0, && \mathrm{h}_0 = \cos\mathrm{f}_0 \\ &\mathrm{g}_n = \frac{1}{n}\sum_{k=1}^{n} k\,\mathrm{h}_{n-k}\,\mathrm{f}_k, && \mathrm{h}_n = -\frac{1}{n}\sum_{k=1}^{n} k\,\mathrm{g}_{n-k}\,\mathrm{f}_k \qquad (n = 1,\cdots,m) \end{aligned} \tag{2.19}$$

三角関数は、このように sin と cos を同時に計算すると、計算式が単純で見易い公式となる。sinとcosを同時に計算する関数sin_cos(x,s,c)が準備されている。 tan はこのようにして得られた sin と cos のTaylor級数をわり算することによって得る。この事情は、以下の双曲線関数 sinh と cosh の場合も同様である。

(10) 双曲線関数

$$g(x) = \sinh f(x), \quad h(x) = \cosh f(x)$$

この二つの関数は、次の微分方程式を満たす。

$$\begin{aligned} \frac{d\,g(x)}{dx} &= h(x)\frac{df(x)}{dx}, \\ \frac{d\,h(x)}{dx} &= g(x)\frac{df(x)}{dx} \end{aligned} \tag{2.20}$$

この式から、係数に対する次のような関係式が得られる。

$$\begin{aligned} g_0 &= \sinh f_0, & h_0 &= \cosh f_0 \\ g_n &= \frac{1}{n}\sum_{k=1}^{n} k\, h_{n-k} f_k, & h_n &= \frac{1}{n}\sum_{k=1}^{n} k\, g_{n-k} f_k \end{aligned} \qquad (n = 1,\cdots,m) \tag{2.21}$$

この公式を利用して、sinhとcoshを同時に計算する関数sinh_cosh(x,s,c)が準備されている。

(11)微分 $h(x) = \dfrac{d\,f(x)}{dx}$

この定義式から、次のような関係式が得られる。

$$f_m = 0, \qquad h_n = (n+1) f_{n+1} \qquad (n = 0,\cdots,m-1) \tag{2.22}$$

このように、最高次数の係数f_mは、0となる。

(12)積分 $h(x) = \int_0^x f(t)dt$

この定義式から、次のような関係式が得られる。

$$h_0 = 0, \qquad h_n = \frac{1}{n} f_{n-1} \qquad (n = 1,\cdots,m) \tag{2.23}$$

定数項は、積分定数なので、任意で良いが、ここで作成したプログラムでは、0としてある。

2.3 その他のTaylor級数の関数

上記以外の関数で、C++言語で標準定義されている関数を、これまで定義した関数を利用して、簡単に定義することができる。

(13)関数のべき乗 $h(x) = f(x)^{g(x)}$

この関数は、

$$h(x) = e^{g(x)\log f(x)} \tag{2.24}$$

として計算することができる。

(14)逆三角関数

これらの関数は、

(2.25) $$\sin^{-1}\mathrm{f}(x)=\sin^{-1}\mathrm{f}_0+\int_0^x\frac{\mathrm{f}'(t)}{\sqrt{1-\mathrm{f}(t)^2}}\,dt$$

(2.26) $$\cos^{-1}\mathrm{f}(x)=\cos^{-1}\mathrm{f}_0-\int_0^x\frac{\mathrm{f}'(t)}{\sqrt{1-\mathrm{f}(t)^2}}\,dt$$

(2.27) $$\tan^{-1}\mathrm{f}(x)=\tan^{-1}\mathrm{f}_0+\int_0^x\frac{\mathrm{f}'(t)}{1+\mathrm{f}(t)^2}\,dt$$

によって計算することができる。

2.4 Taylor級数の比較

Taylor級数の比較には、いろいろな定義が考えられるが、ここでは、数値計算を０次のTaylor級数として含む形にし、計算の流れを数値計算と同じ手順になるように定義する。このような拡張を行えば、既に開発されている多く数値計算用のプログラムを、Taylor級数の計算に利用することができるからである。

このようにするには、Taylor級数の展開位置の近傍で、Taylor級数の大きさを比較すればよいことがわかる。すなわち、

(2.28) $$s=\lim_{x\to 0}\left\{\mathrm{f}(x)-\mathrm{g}(x)\right\}$$

の値によって、大きさを決める。これによって、

(2.29) $s>0$　ならば　$\mathrm{f}(x)>\mathrm{g}(x)$

(2.30) $s=0$　ならば　$\mathrm{f}(x)=\mathrm{g}(x)$

(2.31) $s<0$　ならば　$\mathrm{f}(x)<\mathrm{g}(x)$

と判断する。このように定義すると、定数項の比較だけで判定できるので、単純で大変効率的であり、また、数値計算用のプログラムを簡単にTaylor級数の計算用プログラムとして利用できる。しかしながら、この定義では、常微分方程式の解法のように、まず定数項を決定し、次ぎに１次の係数を決定し、次々に次数を上げていくような計算では、収束の判定がうまくできないなどの不都合も生じる。このような場合には、定数項以外の項を含めた、比較を定義する必要がある。

3. Taylor級数操作関数

Taylor 級数を効率的に利用するために、Taylor 級数を操作する関数が準備されている。

3.1 Taylor級数のコンストラクタ（taylor(p,x0,x1,x2)）

Taylor 級数を直接定義する。

```
template <typename T> taylor() ; //  デフォールトコンストラクタ
```

template <typename T> taylor(T x0) ; //定数で展開位置を指定しない
template <typename T> taylor(T p, T x0) ; //定数で展開位置を指定
template <typename T> taylor(T p, T x0, T x1) ; // $x_0 + x_1(x-p)$
template <typename T> taylor(T p, T x0, T x1, T x2);// $x_0 + x_1(x-p) + x_2(x-p)^2$

p は、展開位置、x0,x1,x2 はそれぞれ定数項、1次係数、2次係数を意味する。3次以上の式は、2次まで上のコンストラクタを使い、3次の係数を次の方法で入れる。

$1 + 2x + 3x^2 + 4x^3$ を変数yに入れるには

```
y = taylor( 0.0, 1.0, 2.0, 3.0 ) ;
y[3]=4.0 ;
```

と分けて書く。直接 y.coeff[3]=4.0 として高速化しても良いが、y.cur_deg などをそれに応じて変更する必要があるので、y[3]=4.0 とするのが簡単で間違いが少ない。

3.2 代入関数（subst(p,x,[n])）

Taylor 級数pに x を代入する。この関数は

```
template <typename T> T subst( taylor_template<T> & p, T x) ;
template <typename T> T subst( taylor_template<T> & p, T x, int n) ;
template <typename T> T subst( taylor_template<T> & p, taylor_template<T>& x) ;
template <typename T> T subst( taylor_template<T> & p, taylor_template<T>& x, int n ) ;
```

これによって Taylor 級数の値が容易に求められる。n を指定することによって、何次まで計算するかを指定できる。指定しない場合は、Taylor 級数pの次数となる。変数 x は Taylor 級数であってもよい。

三角関数 $\sin x$ を原点で Taylor 展開し、$x = \frac{30}{180}\pi$(30 度)ときの値を計算する。計算は、3、5、7、9次の Taylor 展開式で計算した値を求める。以下にそのプログラムを示す。

List 301　三角関数の計算

```
1 : #include "taylor_template.h"  // th31.cpp
2 : typedef taylor_template<double> taylor ;
3 : int main()
4 : {
5 :         taylor s, x ;
6 :         x = taylor( 0.0, 0.0, 1.0 ) ;   //  xを原点で Taylor 展開
7 :         s = sin(x) ;                          // sin(x)の Taylor 展開式を計算
8 :         for( int i=1 ; i<10 ; i+=2 )
9 :         {
10 :            cout << i << "    " << setprecision(14) ;
11 :            cout << subst( s, 30.0/180.0*3.141592653589793, i ) << endl ;
12 :        }
13 : }
```

これを実行すると

```
1    0.5235987755983
3    0.49967417939436
5    0.50000213258879
7    0.49999999186902
9    0.50000000002028
```

が出力される。

3.3 微分方程式の解のTaylor展開(picard(f,x0,y0,y,[n]))

関数fで定義されている微分方程式を x=x0 において y=y0 である解を n 次まで Taylot 展開し、それをyに返す。n を省略した場合、set_degree 関数で指定した次数となる。

picard 関数は微分方程式

(3.1)　$\frac{dy}{dx} = f(x, y)$　　初期条件　$x = x0$ のとき $y = y0$

の解の Taylor 展開を求める関数である。この微分方程式の解は、Picard の逐次近似法

(3.2)
$$y_0 = y0$$
$$y_{n+1} = y0 + \int_{x0}^{x} f(x, y_n)dx$$

で計算できる。y および f は、n 元連立微分方程式の場合、n 元のベクトルとなる。x 、y_n は、$x = x0$ で Taylor 展開されている必要がある。初期値 $y0$ は定数なので、展開位置の意味がないが、Picard の計算を行うには、展開位置を設定していなければならない。関数 f は次のような形式の関数になる。スカラー方程式の場合

```
void f( const taylor &x, const taylor &y, taylor &dy) ;
```

n 元の方程式の場合

```
void f(const taylor &x, const taylor *y, taylor *dy, int n) ;
```

の形式となる。スカラー方程式の場合、通常の関数にもできるが、n元連立方程式の場合と形式を合わせた。スカラーおよびベクトルそれぞれ、Taylor 級数の次数を指定する版と指定しない版の合計4つのルーチンを準備した。

スカラー・計算結果の次数指定

```
template <typename pwtype>
void picard( void (*func)( const taylor_template<pwtype> &xx,
const taylor_template<pwtype> &yy, taylor_template<pwtype> &dyy),
const pwtype& x0, const pwtype& y0, taylor_template<pwtype> &y,
const int n )
```

スカラー・計算結果の次数指定なし

```
template <typename pwtype>
void picard( void (*func)( const taylor_template<pwtype> &xx,
const taylor_template<pwtype> &yy, taylor_template<pwtype> &dyy),
const pwtype& x0, const pwtype& y0, taylor_template<pwtype> &y )
```

ベクトル・計算結果の次数指定

```
template <typename pwtype>
void picard( void (*func)( const taylor_template<pwtype> &xx,
const taylor_template<pwtype> *yy, taylor_template<pwtype> *dyy, const int k ),
const pwtype& x0, const pwtype *y0,
taylor_template<pwtype> *y, const int m, const int n )
```

ベクトル・計算結果の次数指定なし

```
template <typename pwtype>
void picard( void (*func)( const taylor_template<pwtype> &x,
const taylor_template<pwtype> *y, taylor_template<pwtype> *dy, const int k ),
const pwtype& x0, const pwtype *y0,
taylor_template<pwtype> *y, const int m )
```

を準備した。整数 k,m は共に方程式の元数を意味する。これによって微分方程式の解の Taylor 級数が求められる。n を指定することによって、何次まで計算するかを指定できる。Picard の計算は反復計算を行うために、同じ計算を何回も行う。これを避けるには、途中の計算結果を保存しておけば、同じ計算を何回も行わないで済む。このプログラムでは、途中結果を保存しておく領域を準備していないので、同じ計算を何回もすることになる。この計算によって計算速度は、大体半分程度になるが、中間結果を保存する領域が要らないなどの利点もあり、計算手順が簡単なので、このような仕様にしている。

次の微分方程式の解の Taylor 展開を求める。

(3.3) $$\frac{dy}{dx} = -\frac{1}{3}x^2y^2 \quad \text{初期条件} \quad y(2) = 1$$

この方程式を倍精度で解くためのプログラムは次のようになる。

List 302 picard 関数を利用した微分方程式の解

```
1 : #include "taylor_template.h"  // th32.cpp
2 : typedef taylor_template<double> taylor ;
3 : void func( const taylor& x, const taylor& y, taylor& dy )
4 : {
5 :       dy = -1.0/3.0*x*x*y*y ;
6 : }
7 : int main()
8 : {
9 :       taylor   y, p, q ;
10 :      double     x0, y0 ;
11 :      int        n ;
12 :      x0 = 2 ; y0=1 ; n=10 ;
13 :      picard( &func, x0, y0, y, n ) ;
14 :      cout << y << endl ;
15 : }
```

このプログラムでは、10 次までと制限しているので、解は 10 次の Taylor 級数となる。

1-1.33333*(x-2)+1.11111*(x-2)^2-0.703704*(x-2)^3+0.345679*(x-2)^4-0.115226*(x-2)^5
+0.00137174*(x-2)^6+0.0365798*(x-2)^7-0.0368846*(x-2)^8+0.0246406*(x-2)^9
-0.0123287*(x-2)^10

もし有理数のクラスが利用できるならば、(3.3)の問題は厳密に計算でき、次のようなプログラムで簡単に計算出来る。

List 303　有理数を利用した微分方程式の解

```
1 : #include "taylor_template.h" // th33.cpp
2 : #include "rational.h"
3 : typedef taylor_template<rational> dtaylor ;
4 : void func( const dtaylor& x, const dtaylor& y, dtaylor& dy )
5 : {
6 :       dy = rational(-1)/rational(3)*x*x*y*y ;
7 : }
8 : int main()
9 : {
10 :       dtaylor   y, p, q ;
11 :       rational x0, y0 ;
12 :       int         n ;
13 :       x0 = 2 ; y0=1 ; n=10 ;
14 :       picard( &func, x0, y0, y, n ) ;
15 :       cout << y << endl ;
16 : }
```

このプログラムでは、10 次まで解が次のように得られる。この解は誤差のない厳密解である。

(1)+(-4/3)*(x-(2))+(10/9)*(x-(2))^2+(-19/27)*(x-(2))^3+(28/81)*(x-(2))^4
+(-28/243)*(x-(2))^5+(1/729)*(x-(2))^6+(80/2187)*(x-(2))^7+(-242/6561)*(x-(2))^8
+(485/19683)*(x-(2))^9+(-728/59049)*(x-(2))^10

このような計算は、解のチェックを行うときに役立つ。出力は一般的な出力方法で出しているため、不要な括弧が表示されあまり見やすい形式にはならない。

次の複素微分方程式の解の Taylor 展開を求める。初期条件を複素数にした方程式である。

(3.4)　$\dfrac{dy}{dx} = -\dfrac{1}{3}x^2y^2$　初期条件　$y(2) = 1 + i$

この方程式を倍精度複素数で解くためのプログラムは次のようになる。

List 304　複素微分方程式の解の Taylor 展開

```
1 : #include "taylor_template.h" // th34.cpp
2 : #include <complex>
3 : typedef complex<double> dcomplex ;
4 : typedef taylor_template<dcomplex> taylor ;
5 : void func( const taylor& x, const taylor& y, taylor& dy )
6 : {
7 :       dy = -dcomplex(1)/dcomplex(3)*x*x*y*y ;
```

```
 8 : }
 9 : int main()
10 : {
11 :        taylor   y ;
12 :        dcomplex     x0, y0 ;
13 :        int          n ;
14 :        x0 = dcomplex(2) ; y0=dcomplex(1,1) ; n=10 ;
15 :        picard( &func, x0, y0, y, n ) ;
16 :        cout << y << endl ;
17 : }
```

この微分方程式の解は、

(1,1)+(0,-2.66667)*(x-(2,0))+(-3.55556,2.22222)*(x-(2,0))^2+(5.92593,3.33333)*(x-(2,0))^3
+(0.0987654,-11.1605)*(x-(2,0))^4+(-16.0988,8.72428)*(x-(2,0))^5
+(25.3032,16.1783)*(x-(2,0))^6+(3.13123,-49.1632)*(x-(2,0))^7
+(-73.0511,34.5411)*(x-(2,0))^8+(107.579,77.4256)*(x-(2,0))^9
+(25.7126,-215.885)*(x-(2,0))^10

となる。このように容易に求められる。

3.4 逆関数（inv_func(x) ）

逆関数の計算は、方程式の零点を求めるのに便利であり、任意次数の逆関数が計算できるので、これを利用すれば、任意の次数の零点を求める公式が得られる。逆数を計算する関数 invers 関数とは取り違えないように注意する必要がある。

逆関数の計算は、$y = f(x)$ の逆関数 z が $x = f(z)$ すなわち $z(x) = f^{-1}(x)$ となることを利用し、この両辺を微分して、

$$\frac{dz}{dx} = \frac{1}{f'(z)} = v(z) \tag{3.5}$$

という逆関数の微分方程式を得る。これを初期条件 $z(y_0) = f^{-1}(y_0) = x_0$ と Picard の逐次近似法

$$z_n = x_0 + \int_{y_0}^{y} v(z_{n-1})\,dx \tag{3.6}$$

を使うと解が得られる。この繰り返し演算を行い、逆関数を n 次までの近似関数を求める。

展開点 x_0 における n 次までの Taylor 級数

$$y(x) = f(x) = y_0 + y_1(x - x_0) + y_2(x - x_0)^2 + \cdots + y_n(x - x_0)^n \tag{3.7}$$

が与えられたとき、関数 inv_func(y)によって

$$z(x) = f^{-1}(x) = x_0 + x_1(x - y_0) + x_2(x - y_0)^2 + \cdots + x_n(x - y_0)^n \tag{3.8}$$

と逆関数が求められる。

数値例として、次の関数を考える。

(3.9)　　$f(x) = e^x - x^2 + \log x - 3$

この関数を$x = 2$でTaylor展開すると

1.0822+3.88906*(x-2)+2.56953*(x-2)^2+1.27318*(x-2)^3+0.292252*(x-2)^4

となる。これの逆関数のTaylor展開式を計算すると

2+0.257132*(x-1.0822)-0.0436838*(x-1.0822)^2+0.0092772*(x-1.0822)^3
-0.0019049*(x-1.0822)^4

となる。逆関数のTaylor展開式にゼロを代入することによって、零点が1.6562と得られる。この点でさらにTaylor展開し、逆関数を計算して再度零点を計算する。この計算を2回繰り返すと倍精度の限界までの精度の結果が得られる。

List 305　逆関数の計算

```
 1 : #include "taylor_template.h"  // th35.cpp
 2 : typedef taylor_template<double>  taylor ;
 3 : using namespace std ;
 4 : int main()
 5 : {
 6 :       taylor::set_degree(4) ;  // 計算は4次までとする。
 7 :       taylor x, y, z ;
 8 :       x = taylor( 2.0, 2.0, 1.0 ) ;
 9 :       y = exp(x)-square(x)+log(x)-3.0 ;
10 :       cout << y << endl ;
11 :       z = inv_func(y) ;
12 :       cout << z << endl ;
13 :       cout << subst(z,0.0) << endl ;
14 : }
```

3.5 Padé展開（pade(a,p,m,q,n)

常微分方程式の解は、簡単にTaylor展開の形で得られる。このTaylor展開式を、有理関数展開であるPadé展開することによって、A安定な計算法が得られる。

Padé展開とは、以下のようにTaylor展開式を、有理関数に変形したものである。

(3.10)　　$$a_0 + a_1 x + a_2 x^2 + \cdots \quad = \quad \frac{p_0 + p_1 x + \cdots + p_M x^M}{1 + q_1 x + \cdots + q_L x^L}$$

pade(a,p,m,q,n)関数は、Taylor級数aを(3.10)式のように、m次の分子、n次の分母に変換する。この変換した右辺にステップ幅を代入することによって、次のステップにおける初期値を計算する。この操作は、A安定な計算になる。

(3.10)式の両辺に右辺の分母を掛け、$M+L$次の係数まで一致するように、有理関数の係数を決定することによって得られる。この条件は、次の様な式で表される。

(3.11)　　$$a_l + \sum_{k=1}^{m} a_{l-k} q_k = p_l \quad (l = 0, \cdots, M)$$

ただし、mは、lがＬ以下ならば、$m=l$とし、lがＬを越えるならば、$m=L$とする。(3.11)と同じ関係式であるが、lがMを越えているいる場合、次の関係式が得られる。

(3.12) $$a_l+\sum_{k=1}^{L} a_{l-k}q_k=0 \quad (l=M+1,\cdots,M+L)$$

(3.10)の連立方程式を解き、有理式の分母の係数$(q_0,q_1,\cdots,q_L)$を決定し、その係数を(3.9)式に代入して、分子の係数$(p_0,p_1,\cdots,p_M)$を求めることができる。

Padé展開は、一般に、同じ次数のTaylor展開式より高精度で、収束の良い式を与えることができる場合が多いので、常微分方程式の解のTaylor展開式をPadé展開することは、効率的な常微分方程式の計算法を与えると期待できる。

Taylor展開式を、Padé展開することによって、高精度な計算法が得られるだけでなく、A安定な計算法になる。

テスト方程式

(3.13) $\dfrac{dy}{dx}=\lambda y$　$x=0$のとき、$y=y_0$とする。λは一般に複素定数である。

を解くことを考える。この方程式の解のTaylor展開は、良く知られているように、

(3.14) $$y=y_0e^{\lambda x}=y_0\sum_{k=0}^{\infty}\frac{(\lambda x)^k}{k!}$$

となる。A安定とは、$\mathrm{Re}\,\lambda x<0$のとき、(3.14)の右辺の級数部分の絶対値が１より小さくなるときを言う。(3.14)の無限級数を使う計算法は、無限級数が$e^{\lambda x}$の展開式であることから１より小さくなるのでA安定である。実際の計算では、有限項で打ち切った式を使うので、Taylor展開式を使う限りA安定な計算法は存在しない。多くの項をとれば、A安定な式に近づくので、次数を上げることによって、安定性のよりよい公式を作れる。Corliss等[1]は、30次などのかなり高次の公式を使うことによって、安定化を計っている。

(3,14)の指数関数$e^{\lambda x}$のPadé展開については、良く研究されており、次のような結果が得られている。

定理：　指数関数$e^{\lambda x}$を(3.1)のように、分子M次、分母L次式にPadé展開したとき、$M\le L\le M+2$のとき、A安定である。

証明：E. Hairer and G. Wanner[4]のⅣ章（60頁）を参照のこと。

分母の次数が分子の次数と同じか１または２次高い式に変形すれば、A安定な公式となる。このように変形すれば常微分方程式の初期値問題に対する任意次数のA安定な公式が得られたことになる。従来のA安定な計算法は、非線型の方程式を解く必要がある[14]のに対し、本方法では、連立一次方程式を解くだけで計算できるため、計算方法としても、単純で高速である。

本方法は、任意の次数まで計算できるので、問題にあった次数で計算することができる。これは、従来のA安定な計算法にはない特徴である。 Padé展開したとき、分母の

次数を分子より高い次数にすれば、L安定な公式ともなる。

以下に、指数関数 e^x を12次まで展開し、そのTaylor展開から分子５次、分母５次のPadé展開するプログラムを示す。

List 306　Padé 展開の計算

```
1 : #include "taylor_template.h" // th36.cpp
2 : typedef taylor_template<double>  taylor ;
3 : using namespace std ;
4 : int main()
5 : {
6 :       taylor::set_degree(12) ;
7 :       taylor x, y, p, q ;
8 :       x = taylor( 0, 0, 1 ) ;
9 :       y = exp(x) ;
10 :      pade( y, p, 5, q, 5 ) ;
11 :      cout << y << endl ;
12 :      cout << p << endl ;
13 :      cout << q << endl ;
14 : }
```

指数関数の展開式は

1+x+0.5*x^2+0.166667*x^3+0.0416667*x^4+0.00833333*x^5+0.00138889*x^6
+0.000198413*x^7+2.48016e-05*x^8+2.75573e-06*x^9+2.75573e-07*x^10
+2.50521e-08*x^11+2.08768e-09*x^12

５次の分子は

1+0.5*x+0.111111*x^2+0.0138889*x^3+0.000992063*x^4+3.30688e-05*x^5

５次の分母は

1-0.5*x+0.111111*x^2-0.0138889*x^3+0.000992063*x^4-3.30688e-05*x^5

となる。任意次数の零点の計算は、逆関数を利用して計算することができるが、このpade関数を利用しても計算できる。分子を１次式、分母を高次の式に展開する。すなわち

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots = \frac{p_0 + p_1 x}{1 + q_1 x + \cdots + q_L x^L}$$

と展開する。 $f(x) = 0$ は、(L+1)次の精度で $p_0 + p_1 x = 0$ の方程式となる。これを解くことによって高次の計算が可能である。

3.6 割り算関数（div(f,g,q,r)）

関数 div(f,g,q,r)は Taylor 級数f、gを多項式とみなし、f を g で割り、商を q と余り r を求める関数である。これらの級数（多項式）には、次の関係式が成り立つ。

(3.15)　　$f(x) = q(x)g(x) + r(x)$

この演算を使って、次の多項式のスツルム(Strum)列を計算する。

(3.16) $$f(x) = 3.22x^6 + 4.12x^4 + 3.11x^3 - 7.25x^2 + 1.88x - 7.84$$

このときのプログラムは、

List 307　多項式の割り算

```
 1 : #include "taylor_template.h"   // th37.cpp
 2 : typedef taylor_template<double>   taylor ;
 3 : using namespace std ;
 4 : int main()
 5 : {
 6 :       taylor q, r, y, f[20] ;
 7 :       //  yに-7.84+1.88*x-7.25x^2+3.11x^3+4.12x^4+3.22x^6 を入れる
 8 :       y= taylor(0.0, -7.84, 1.88, -7.25 ) ;
 9 :       y[3] = 3.11 ;   y[4] = 4.12 ; y[6] = 3.22 ;
10 :       f[0] = y ;
11 :       f[1] = diff(y) ;
12 :       for( int i=2; ; i++ )
13 :       {
14 :             div( f[i-2], f[i-1], q, r ) ;
15 :             f[i] = -r ;
16 :             if( f[i].cur_degree() == 0 ) break ;
17 :       }
18 :       for( int i=0; ; i++ )   //  スツルム列を出力
19 :       {
20 :             cout << f[i] << endl ;
21 :             if( f[i].cur_degree() == 0 ) break ;
22 :       }
23 : }
```

このときの出力は

```
-7.84+1.88*x-7.25*x^2+3.11*x^3+4.12*x^4+3.22*x^6
1.88-14.5*x+9.33*x^2+16.48*x^3+19.32*x^5
7.84-1.56667*x+4.83333*x^2-1.555*x^3-1.37333*x^4
123.003-120.748*x+89.6995*x^2-109.245*x^3
-4.81953+0.147843*x-4.1486*x^2
-23.3193-9.22173*x
31.7217
```

となる。この結果から、正根 1 個、負根1個の2つの実根があることがわかる。

3.7 計算次数制御関数（set_degree(n), degree()）

Taylor 展開の計算次数を制御する。計算は最大この次数で行われる。3次と4次の Taylor 級

数 t3 と t4 があるとする。この set_degree 関数で指定した次数(変数名 deg)を 6 とする。この場合、加減算は、4次になるので問題なく計算できる。乗算の場合、一般に7次の式になるが、ここで指定した次数が 6 であるため、6次になる。平方根や三角関数の場合の計算結果もこの次数に制限されて6次になる。

この次数は、変数宣言には、宣言される Taylor 級数の次数になる。一度宣言されたら、その変数が消去されて再び宣言されるまで、次数は変わらない。

宣言される Taylor 級数の次数を固定することもできる。この場合、プログラムの定数 FIX_LENGTH を入れ、MAX_DEG の値を設定する。これを指定すると MAX_DEG 次の Taylor 級数が宣言される。この次数を見るには、degree 関数を使用する。これらの関数(set_degree、degree()は static 関数として宣言されているので、

```
taylor_template<double>::set_degree(5) ;
a.set_degree(5) ; // a は taylor_template<double>クラスの変数
```

のように使える。

この次数制御によって、必要のない高次の計算を避け、計算効率を上げることができる。微分方程式の解を taylor 展開する関数 picard の中で、この関数を使用し高速化をはかっている。

このような static な数値を使っているため、並列化とくにマルチスレッド化が難しくなっている。これを避けるためにいろいろな案を考えているが、使い易い方法が見つからないためこの方法を使用している。

3.8 いろいろな情報を設定、読み取る関数（set_cur_degree(n),cur_degree(), alloc_size()）

Taylor 級数のいろいろな情報を設定したり、読み取るための関数を準備している。

```
a.set_cur_degree(7) ; // 変数 a の次数を 8 に設定する。
a.cur_degree() ;      // 変数 a の次数を得る。
a.alloc_size();       // 変数 a の格納可能最大次数を得る。
```

これらの数値は、直接

```
n=a.cur_deg ;
n=a.alloc_siz ;
```

として、読み出すこともできる。直接代入もできるが、変数の構造を理解した上で行う必要がある。

3.9 Taylor級数の係数の読み書きするための演算子関数（[]、()）

Taylor 級数変数を a とすると、n 次の係数は、

```
t=a[n] ;
t=a(n) ;
```

として、読み出すことができる。内部の構造を理解しているならば同じ内容を

```
t=a.coeff[n] ;
```

と書ける。この違いは、n がtの次数を超えた数値のとき、a.coeff[n]はどんな値を返すか分からないことである。a[n]と a(n)は 0 を返す。()を使うと[]を使うよりも高速なので、読み出しには、()を使うのが効率的である。[]は代入にも使える。

a[n]=t ;

として、a の係数を設定できる。直接

a.coeff[n]=t ;

と書くこともできる。この場合高速であるが、この代入で Taylor 展開式の次数が変化したような場合、内部データに矛盾が生じる可能性があるので注意が必要である。

3.10 Taylor級数を文字列で指定する関数(formula)

関数 formula は、係数が倍精度型専用の関数で、文字列として与えられた数式を Taylor 展開するものである。この関数は次のように宣言されている。

```
taylor_template<double> formula( const char* str ) ;
taylor_template<double> formula( const double p, const char *str ) ;
taylor_template<double> formula( const char *str, const double p ) ;
```

文字列strに文字列として式を与えたとき、その式を x=p で Taylor 展開する。展開位置 p が与えられていない場合は原点となる。式の中で使える変数は「x」のみで、すべてxの式でなければならない。

Taylor 級数を定義するには、コンストラクタを使って定義できる。係数が倍精度の場合、次のようにも書ける。

たとえば、変数tに$1+x+4x^2+4.2x^3$を代入することを考える。これは formula 関数を使うと

t=formula("1+x+4*x^2+4.2*x^3") ;

と書くことができる。展開位置を指定することもできる。展開位置を x=2.1 とすると

t=formula("1+x+4*x^2+4.2*x^3",2.1) ;

または

t=formula(2.1,"1+x+4*x^2+4.2*x^3") ;

と書くことができる。代入された変数 t を出力すると、

t=1+(x-2.1)+4*(x-2.1)^2+4.2*(x-2.1)^3

となる。展開位置を省略すると展開位置は原点となる。式の文字列に展開位置を直接入れても展開位置は変わらない。たとえば、

t=formula("1+(x-2.1)+4*(x-2.1)^2+4.2*(x-2.1)^3") ;

と書くと、式は次のように展開され、原点で展開された式になる。変数 t の内容は

t=-22.3562+39.766*x-22.46*x^2+4.2*x^3

となる。

関数 formula の式には四則演算や関数などを書くことができる。その式を Taylor 展開し、その式が返される。xの3乗を x^3 と書いたが、C++言語や C 言語と同じように pow(x,3)と書くことも出来る。

t=formula("(2*x^3+4.1*pow(x,4)+3*x-3)/(sin(x)+exp(x))") ;

と書くと、7次まで表示すると変数 t には、

```
t=-1.5+3*x-2.625*x^2+2.875*x^3-0.1375*x^4-0.63125*x^5
 +0.696354*x^6-0.578646*x^7
```

が入る。

展開位置を指定すると展開位置が決まるが、逆関数 inv_func を使った場合には展開位置が変更されるので注意する必要がある。展開位置は、元の Taylor級数 の定数項となる。例として指数関数 e^x の逆関数を求める。このときの式は

t=formula(“inv_func(exp(x))”) ;

となる。この式を実行して7次まで表示すると

t=(x-1)-0.5*(x-1)^2+0.333333*(x-1)^3-0.25*(x-1)^4+0.2*(x-1)^5
-0.166667*(x-1)^6+0.142857*(x-1)^7

となる。$\log(x)$ の $x=1$ における Taylor 展開であることがわかる。展開位置は原点であるものが、$x=1$ に変更されている。展開位置は e^x の定数項の1になっていることがわかる。

以下に Legendre 多項式を計算するプログラムを示す。Legendre 多項式 $P_n(x)$ は次のように定義される多項式である。

$$P_n(x)=\frac{1}{2^n n!}\frac{d^n}{dx^n}(x^2-1)$$

この式を使って、n=10 までの多項式を計算する。$P_3(x)$ と $P_4(x)$ の内積

$$\int_{-1}^{1} P_3(x)P_4(x)dx=0$$

を計算し、直交することを確かめる。また $P_5(x)$ 同士の内積を計算し

$$\int_{-1}^{1} p_n(x)^2\,dx=\frac{2}{2n+1}$$

の $n=5$ の場合を計算する。

List 308　Legendre 多項式の計算

```
 1 : #include "taylor_template.h"   // th38.cpp
 2 : typedef taylor_template<double>   taylor ;
 3 : int main()
 4 : {
 5 :       taylor   p[11], q, r, s ;
 6 :       double d = 1 ;
 7 :       q = formula("x^2-1") ;    //  qに x^2-1 を入力
 8 :       for( int n=0 ; n<=10 ; n++ )
 9 :       {
10 :             r = diff( pow( q, n ), n ) ;
11 :             p[n] = r/d ;
12 :             d *= 2*(n+1);
13 :             cout << n << " : " << p[n] << endl ;
14 :       }
15 :       s = integrate( p[3]*p[4] ) ;    //  3次と4次の多項式の内積を計算
16 :       cout << subst( s, 1.0)-subst(s, -1.0) << endl ; //直交するので 0 となる。
17 :       s = integrate( p[5]*p[5] ) ;    //  同じ5次の多項式の内積を計算
```

```
18 :        cout << subst( s, 1.0 )-subst(s, -1.0 ) << endl ;  //  2/11 となる。
19 : }
```

計算結果は

```
 0 : 1
 1 : x
 2 : -0.5+1.5*x^2
 3 : -1.5*x+2.5*x^3
 4 : 0.375-3.75*x^2+4.375*x^4
 5 : 1.875*x-8.75*x^3+7.875*x^5
 6 : -0.3125+6.5625*x^2-19.6875*x^4+14.4375*x^6
 7 : -2.1875*x+19.6875*x^3-43.3125*x^5+26.8125*x^7
 8 : 0.273438-9.84375*x^2+54.1406*x^4-93.8438*x^6+50.2734*x^8
 9 : 2.46094*x-36.0938*x^3+140.766*x^5-201.094*x^7+94.9609*x^9
10 : -0.246094+13.5352*x^2-117.305*x^4+351.914*x^6-427.324*x^8+180.426*x^10
  0
  0.181818
```

となる。この計算を行うには途中 20 次以上になるので、最大次数は 20 以上にして計算しなければならない。通常このような計算には、漸化式があるのでそれを利用する。計算効率の良い計算になる。

4. Taylor級数のプログラムの例

4.1 gamma関数のTaylor展開

Taylor級数を求める計算は、数値計算の一部にはなっていない。これは、Taylor級数を求める一般的な計算方法が複雑で実用的でないと見なされていたからではないかと思われる。Taylor級数の計算は、通常の数値計算とあまり変わらないことを以下に示す。

例として gamma 関数の Taylor 展開式を計算する。この関数の Taylor 級数展開式は、手計算では簡単には計算することができないが、gamma関数の数値計算プログラムが存在するならば、簡単に計算できることを示す。

計算のための公式[1]は、通常の数値計算方法と同様に、漸近展開式

$$\log(\Gamma(x)) \approx \left(x-\frac{1}{2}\right)\log(x)-x+\frac{1}{2}\log(2\pi)+\frac{1}{12x}-\frac{1}{360x^3}+\frac{1}{1260x^5}-\cdots \cdots+\frac{B_{2n}}{2n(2n-1)x^{2n-1}}+\cdots \tag{4.1}$$

を使う。この公式は、漸近展開式なので、x が十分に大きくないと、精度のよい計算ができないので、x が小さな数値の場合、整数を加えて20より大きな値にし、その値の関数値を(4.1)の公式を使って精度よく計算する。得られた計算値を、公式

(4.2)　　$$\Gamma(x)=\frac{\Gamma(x+1)}{x}$$

を何回か使って、もとのxに戻し、目的の値の関数値を計算する。この計算法は、gamma関数の計算によく使われる方法である。この計算方法をTaylor級数に適用すれば、gamma関数のTaylor級数が得られる。ここでは、数学公式集[1]と比較するために$\frac{1}{\Gamma(x)}$の展開式を計算しその係数を出力した。以下にそのC++言語のプログラムを示す。ループや条件判断があるような計算式でも、問題なくTaylor展開できることがわかる。

List 401　ガンマ関数の Taylor 展開

```
 1 : #include <stdio.h> // th41.cpp
 2 : #include "taylor_template.h"
 3 : typedef taylor_template<double> power ;
 4 : power inv_gamma( const power& x ) ;
 5 : int main()
 6 : {
 7 :         power xs = power( 0, 0, 1 ) ;
 8 :         power p = inv_gamma(xs) ;
 9 :         for( int i= 1 ; i<21 ; i++ )
10 :         {
11 :               printf("%2d     %25.18e¥n", i, p(i)) ;
12 :         }
13 : }
14 : //
15 : //         Γ(x) のテイラー級数の計算
16 : //
17 : power inv_gamma( const power& x )
18 : {
19 :           double Bernoulli[31] = { // Bernoulli 数
20 :                                        1.0/1.0, -1.0/2.0, 1.0/6.0,
21 :                                0.0, -1.0/30.0, 0.0, 1.0/42.0, 0.0,
22 :                                      -1.0/30.0, 0.0, 5.0/66.0, 0.0,
23 :                                  -691.0/2730.0, 0.0, 7.0/6.0, 0.0,
24 :                                  -3617.0/510.0, 0.0, 43867.0/798.0,
25 :                                           0.0, -174611.0/330.0, 0.0,
26 :                        854513.0/138.0, 0.0, -236364091.0/2730.0,
27 :                                             0.0, 8553103.0/6.0, 0.0,
28 :                                           -23749461029.0/870.0, 0.0,
29 :                                             8615841276005.0/14322.0
30 :                                    } ;
```

```
31 :        power f, z, z2 ;
32 :        power p ;
33 :        double PI = 4*atan(1.0) ;
34 :        const int BERN = 30 ;
35 :      int limit = 20 ;
36 :      int m ;
37 :        int k = 0 ;
38 :        z = x ;
39 :        if( x(0)<limit )
40 :        {
41 :                    k = limit - x(0) ;
42 :                    z = x + double(k) ;
43 :        }
44 :        z2 = z*z ;
45 : // ---
46 :        f = (z-0.5)*log(z) - z + 0.5*log(2*PI) ;
47 :        for ( m=2 ; m<=BERN ; m+=2 )
48 :        {
49 :                    f += Bernoulli[m] / (double(m*(m-1))*z) ;
50 :                    z *= z2 ;
51 :        }
52 :        f = exp(f) ;
53 : // ---
54 :        p = 1 ;
55 :        for ( m=0 ; m<k ; m++ )
56 :        {
57 :                    p *= x+double(m) ;
58 :        }
59 :        p /= f ;
60 :        return p ;
61 : }
```

このプログラムを実行すると、Taylor級数の係数が得られる。１次から20次までの係数を出力すると、次のようになる。定数項は、0である。

次数	係数
1	9.999999999999922284e-01
2	5.772156649015286467e-01
3	-6.558780715202487954e-01
4	-4.200263503409576438e-02
5	1.665386113822918124e-01
6	-4.219773455554559627e-02

```
 7    -9.621971527875607805e-03
 8     7.218943246661659215e-03
 9    -1.165167591857371211e-03
10    -2.152416741166731219e-04
11     1.280502823894988514e-04
12    -2.013485478167396749e-05
13    -1.250493481667219391e-06
14     1.133027231746540948e-06
15    -2.056338415757610432e-07
16     6.116095034085128226e-09
17     5.002007686592098620e-09
18    -1.812745943542406640e-09
19     1.043426834208811565e-10
20     7.782257803057486477e-12
```

この係数は、公式集[1]の値と比較すると14桁程度一致することが解る。また、この結果から、$\Psi(x)$ などのpolygamma関数[1]も容易に計算できる。

4.2. みかけの特異点及びその近くでの関数値の計算

Taylor級数の計算方法を利用すれば、つぎの関数の $x=0$ における関数値の計算のように、見かけの特異点における関数値を桁落ちなしで計算することができる。

$$f(x)=\frac{x}{e^x-1} \tag{4.3}$$

この関数の $x=0$ における値は、通常の計算方法では、精度良く計算することは困難である。$x=0$ を代入すると、ゼロによる割り算が起こるので、0に非常に近い値で代用して計算するが、桁落ちが生じるため、精度の良い計算値は期待できない。

関数値を計算する位置（$x=0$）で分子及び分母をTaylor展開すれば、分子と分母の定数項が 0 となるので、分子及び分母を x で割れば、関数値を桁落ちなしで計算できる。このように、関数の解析的な性質を使うと、大きな桁落ちが避けることができる場合が多い。このような問題は、数値解析にはよく現れるので、このような部分を如何に精度よく計算できるかが、重要である。この値を計算するC++言語のプログラムを以下に示す。

List 402　みかけの特異点近くでの関数計算

```
1 : #include "taylor_template.h" // th42.cpp
2 : typedef taylor_template<double> power ;
3 : int main()
4 : {
5 :       power x, y ;
6 :       x = power( 0.0, 0.0, 1.0 ) ;        // 0+1*x を設定する。
7 :       y=x/(exp(x)-1.0) ;                  // Taylor 級数を計算する。
```

```
 8 :     cout << y << endl ;                 // 計算した Taylor 級数を出力する。
 9 :     cout << subst(y,0.0) << endl ;   // x=0 を代入する。
10 : }
```

6行目で(4.3)のTaylor級数を計算する。このTaylor級数は、８次まで示すと

```
1-0.5*x+0.0833333*x^2-6.93889e-18*x^3-0.00138889*x^4
-2.1684e-19*x^5+3.30688e-05*x^6+1.35525e-20*x^7-8.2672e-07*x^8
```

となる。厳密な計算では、３次以上の奇数次の係数は、0になるが、この計算では、誤差が生じているため0にならない。7行目で、この式に0を代入し、１の結果が得られる。0に近い値を入れても、高精度結果が得られる。

4.3. 多変数のTaylor展開の計算

１変数のTaylor級数のみを扱ってきたが、Taylor級数の係数をTaylor展開することによって、あまり効率的でないが多変数関数のTaylor展開することができる。１階偏微分係数を集めれば、Jacobi行列も容易に計算できる。

２変数関数のTaylor展開を行う。例として、次の関数をTaylor展開する。

$$f(x,y)=\sqrt{x^2+y^2-x+2y+1} \tag{4.4}$$

を$(x,y)=(1,2)$でTaylor展開する。この点でx、yをTaylor展開すると

$$\begin{aligned} x &= 1+(x-1) \\ y &= 2+(y-2) \end{aligned} \tag{4.5}$$

と展開できる。この式を定義し、それを(4.6)に代入すれば、Taylor展開式が得られる。次数を大きくすると多変数の場合、係数がたくさん出てくるので、ここでは２次までと制限する。２変数関数$P(x,y)$が次のようにTaylor展開されているとする。

$$\begin{aligned} P(x,y) &= (p_{0,0}+p_{0,1}(y-b)+p_{0,2}(y-b)^2+\cdots) \\ &+ (p_{1,0}+p_{1,1}(y-b)+p_{1,2}(y-b)^2+\cdots)(x-a) \\ &+ (p_{2,0}+p_{2,1}(y-b)+p_{2,2}(y-b)^2+\cdots)(x-a)^2 \\ &+ \ddots \end{aligned} \tag{4.6}$$

プログラムでは、$p_{i,j}$の係数はp[i][j]と書けるので、(4.7)式から

```
x[0][0] = 1.0 ;   x[1][0] = 1.0 ;   x.set_position(1.0) ;
y[0][0] = 2.0 ;   y[0][1] = 1.0 ;   y[0].set_position(2.0) ;
```

として、変数x、yを展開位置でTaylor展開する。それを(4.6)に代入し、$f(x,y)$をTaylor展開する。以下にそれを実行するプログラムを示す。

List 403　２変数関数の Taylor 展開

```
 1 : #include "taylor_template.h"   // th43.cpp
 2 : typedef taylor_template<double>   taylor ;
 3 : typedef taylor_template<taylor>   dtaylor ;
```

```
 4 : using namespace std ;
 5 : int main()
 6 : {
 7 :       taylor::set_degree(2) ;  //  最大次数を二次に設定
 8 :       dtaylor::set_degree(2) ;
 9 :       dtaylor x, y, z ;
10 :       x[0][0] = 1.0 ; x[1][0] = 1.0 ;  //  x を(x,y)=(1,2)で Taylor 展開
11 :       x.set_position(1.0) ;          //   x の展開位置を指定
12 :       y[0][0] = 2.0 ; y[0][1] = 1.0 ;  //  y を(x,y)=(1,2)で Taylor 展開
13 :       y[0].set_position(2.0) ;        //   y の展開位置を指定
14 :       cout << x << endl ;           //  確認のため x を出力
15 :       cout << y << endl ;            //  確認のため y を出力
16 :       z = sqrt(x*x+y*y-x+taylor(2)*y+taylor(1)) ;  // Taylor 展開を計算
17 :       cout << z << endl ;           //  展開した結果を出力
18 :       cout << z(0)(0) << endl ;   // f(1,2)の値
19 :       cout << z(1)(0) << endl ;   // x で偏微分した時の値
20 :       cout << z(0)(1) << endl ;   // y で偏微分した時の値
21 : }
```

これを実行すると、次のように出力される。（式を整形しコメントを付加している。）

```
(1)+(1)*(x-(1))                x=1+(x-1)を意味する。
(2+2*(y-2))                    y=2+(y-2)を意味する。
(3+2*(y-2))+(0.166667-0.111111*(y-2)+0.0740741*(y-2)^2)*(x-(1))
          +(0.162037-0.101852*(y-2)+0.0617284*(y-2)^2)*(x-(1))^2
3                              f(1,2)の値
0.166667                       xで偏微分した時の値
2                              yで偏微分した時の値
```

この方法を使うことによって、多変数関数の微分を行うことができる。それを使えばJacobi行列も計算できる。変数は最初にxを使い、次にy、z、u,、v、w、x006、x007、…、x999が使われる。このような使い方によって偏微分係数を計算することが可能であるが、この方法はあまり実用的ではない。多変数専用のプログラムを作るべきである。特に係数だけでなく展開位置もTaylor展開されるので、不自然な利用方法になる。

4.4. 数値積分

十分に速く収束する級数に展開できる関数は、効率良く数値積分を行うことができる。たとえば

(4.7) $$\int_1^2 e^x \log x dx$$

を計算してみる。被積分関数を積分区間の中点 $x = 1.5$ でTaylor展開し、それを積分すれば、(4.7)の関数の不定積分のTaylor展開が得られる。その関数に2と1を代入すれば、積分の値が得られる。以下の例では10次から21次のTaylor級数を使った計算を示す。

List 404　数値積分

```
 1 : #include "taylor_template.h"  // th44.cpp
 2 : typedef taylor_template<double>  taylor ;
 3 : using namespace std ;
 4 : int main()
 5 : {
 6 :       taylor x, y ;
 7 :       x = taylor(1.5,1.5,1.0);//変数 x を x=1.5 で Taylor 展開する。1.5+(x-1.5)
 8 :       y = integrate(exp(x)*log(x)) ;  // 不定積分の Taylor 展開を計算
 9 :       for( int i=11 ; i<22 ; i++ )
10 :       {
11 :             cout << setprecision(14) ;  // 精度 14 桁表示に指定
12 :             cout << i << " : " << subst( y, 2.0, i) - subst( y, 1.0, i ) << endl ;
13 :       }
14 : }
```

これを使って計算すると、利用したTaylor級数の次数と積分値は

```
11 : 2.0625868739345
12 : 2.0625868739345
13 : 2.0625868633058
14 : 2.0625868633058
15 : 2.0625868624124
16 : 2.0625868624124
17 : 2.0625868623347
18 : 2.0625868623347
19 : 2.0625868623278
20 : 2.0625868623278
21 : 2.0625868623272
```

の結果が得られる。積分値は2.062586862327であることがわかる。この例題では積分区間を分割しないでも計算ができたが、一般には台形公式などのように積分区間を分割し区間毎にこの計算法を適用しなければならない。

4.5. 無限級数の和

無限級数を計算する方法として、Euler-Maclaurinの総和公式がある。この公式は

(4.8) $$\sum_{k=0}^{n} f(k) = \int_0^n f(x)dx + \frac{1}{2}(f(0)+f(n)) + \sum_{j=1}^{m-1} \frac{B_{2j}}{(2j)!}(f^{(2j-1)}(n) - f^{(2j-1)}(0)) + \Omega_{2m}$$

(4.9) $$|\Omega_{2m}| \le \frac{2^{2m-1}}{2^{2m-1}} \frac{|\mathbf{B}_{2m}|}{(2m)!} \int_0^n |f^{(2m)}(t)|dt$$

である。ここで、B_nはBernoulli数である。この公式には、関数の微分係数を使っており、いままで数値計算法としてはあまり利用されなかった公式である。この公式に関しては数式処理を使って無限級数の計算に使われている。(4.8)において、nを大きくしたとき、関数値、その微分係数および(4.9)で示された項が非常に小さくなるとき、

(4.10) $$\sum_{k=0}^{\infty} f(k) \cong \int_0^{\infty} f(x)dx + \frac{1}{2}f(0) - \sum_{j=1}^{m-1} \frac{B_{2j}}{(2j)!} f^{(2j-1)}(0)$$

と書くことができる。この級数は一般には漸近級数である。例として次の無限級数の和を計算する。

(4.11) $$S = \sum_{k=1}^{\infty} \frac{1}{k^2}$$

厳密値 $$S = \frac{\pi^2}{6} = 1.64493406684822643647241516664602518921894990$$

この式に直接(4.10)を適用すると、非常に収束が遅く、精度良く計算するのが困難なので、次のように二つの和に変形する。

(4.12) $$S = \sum_{k=1}^{20} \frac{1}{k^2} + \sum_{k=21}^{\infty} \frac{1}{k^2}$$

最初の項は直接計算し、次の項の計算に(4.10)を適用する。このとき、$f(k) = \frac{1}{(k+21)^2}$ となる。積分項は

(4.13) $$\int_0^{\infty} f(x)dx = \int_0^{\infty} \frac{dx}{(x+21)^2} = \left[-\frac{1}{x+21}\right]_0^{\infty} = \frac{1}{21}$$

となる。厳密に計算できない場合には、数値積分を使うこともある。これを計算するプログラムは

List 405　無限級数の和

```
1 : #include "taylor_template.h" // th45.cpp
2 : typedef taylor_template<double> taylor ;
3 : taylor func( const taylor& x )
```

```
 4 : {
 5 :       return 1.0/((x+21.0)*(x+21.0)) ;
 6 : }
 7 : int main()
 8 : {
 9 :       double Bernoulli[31] = { // Bernoulli 数
10 :          1.0/1.0,-1.0/2.0,1.0/6.0,0.0,-1.0/30.0,0.0,
11 :          1.0/42.0,0.0,-1.0/30.0,0.0,5.0/66.0,0.0,
12 :          -691.0/2730.0,0.0,7.0/6.0,0.0,-3617.0/510.0,
13 :          0.0,43867.0/798.0,0.0,-174611.0/330.0,0.0,
14 :          854513.0/138.0,0.0,-236364091.0/2730.0,0.0,
15 :          8553103.0/6.0,0.0,-23749461029.0/870.0,0.0,
16 :          8615841276005.0/14322.0
17 :       } ;
18 :       double s = 0 ;
19 :       for( int k=1 ; k <=20 ; k++ )
20 :       {
21 :             s += 1.0/(k*k) ;
22 :       }
23 :       taylor x = taylor( 0, 0, 1 ) ;
24 :       taylor p = func(x)   ;
25 :       s += 1.0/21.0 + 0.5/(21.0*21.0) ;
26 :       for( int j= 1 ; j<6 ; j++ )
27 :       {
28 :             s -= Bernoulli[2*j]*p(2*j-1)/(2*j) ;
29 :             cout << j << setprecision(15) ;
30 :             cout << ″    ″ << s << endl ;
31 :       }
32 : }
```

このプログラムを実行すると、

```
1    1.64493407499678
2    1.64493406683505
3    1.64493406684827
4    1.64493406684823
5    1.64493406684823
```

となり、4回の補正でほぼ厳密値と一致することがわかる。(4.8)の公式は、刻み幅を 1 にした台形公式の誤差評価式と見なすことも出来るので、これを利用すると台形公式で積分値を計算しそれに誤差を考慮して高精度の計算が可能である。

5. 機能のまとめ

Taylor級数に対して、通常のdouble型とほぼ同じプログラムが可能なように作成している。通常のdouble型の演算でTaylor級数の演算として意味のあるものはほぼすべて網羅している。

5.1. 使用できる演算子

double型の数値と同じように使えるように演算子を再定義してある。

=、+=、-=、*=、/=、-（符号）、+（符号）、＋、-、*、/

この他に、比較演算子が定義してある。

==、!=、>、<、>=、<=

Taylor級数も係数を読み書きするために括弧も裁定されている。Taylor級数変数をtとすると

```
t[3]=2.6    ;        tの３次の係数を書き込む
v=t[4]      ;        tの４次の係数をvに代入
                       （このように書けるが（）を使った方が高速）
v=t(4)      ;          tの４次の係数をvに代入
```

出力のために、<<演算子が定義されている。

```
cout   <<   t      ;   tを出力する。
```

5.2. Taylor級数の制御する関数

つぎのような関数が使える。

```
a.set_degree(n) ;                                  // Taylor級数の最大次数をnにする。
                                                   //  この次数は、宣言時の次数となる。
taylor_template<double>::set_degree(n) ;//    set_degree関数はstatic関数なので
                                                  //  このように指定可能
n=a.degree() ;                  // Taylor級数の最大次数を読み取りnに代入する。
a.set_cur_degree(n) ;           // Taylor級数の次数をnにする。
n=a.degree() ;                  // Taylor級数の次数を読み取りnに代入する。
n=a.aloc_size() ;     // Taylor級数aの収容可能最大次数を読み取りnに代入する。
```

5.3. Taylor級数の関数

つぎのような関数が使える。

sqrt(x)、exp(x)、log(x)、log10(x)、sin(x)、cos(x)、tan(x)、
sinh(x)、cosh(x)、asin(x)、acos(x)、atan(x)、pow(x,y)

が利用可能である。係数の型として指定した型がこれらの演算が可能でなければならない。たとえば、C++言語に標準で定義されている複素数（complex）は逆三角関数などが定義されていないので、逆三角関数は計算できなくなる。有理数のこれらの関数も定義されていないので計算できない。有理数の場合、特定の値の場合厳密に計算できるの

で、そのときだけ値を返すようにすると、公式集にのっているようなTaylor級数が計算できる場合がある。

Taylor級数用につぎのような関数が準備されている。

square(x)	2乗を計算する。
invers(x)	逆数を計算する。
inv_func(x)	逆関数を計算する。
subst(x,y)	xにｙを代入する。
diff(x,n)	xをn回微分する。
diff(x)	xを微分する。
integrate(x)	積分する。
sin_cos(x,s,c)	sin(x)とcos(x)を同時に計算する。
sinh_cosh(x,s,c)	sinh(x)とcosh(x)を同時に計算する。
left_shift(x,n)	係数を配列と見て、左（低次の方）にnだけ移動させる。
left_shift(x)	係数を配列と見て、左（低次の方）に1だけ移動させる。
picard(f,x0,y0,y)	y'=f(x,y)を初期値x=x0のときy=y0の解をTaylor展開する。

係数が倍精度型のTaylor級数用には、つぎのような関数が準備されている。

formula(form)	formで指定した式を原点でTaylor展開する。
formula(form, a)	formで指定した式をx=aでTaylor展開する。
formula(a, form)	formで指定した式をx=aでTaylor展開する。

6. templateライブラリの使用法について

templateライブラリは、引数として型をとるが、その型以外の型をプログラムに使うと誤りになる。これを無くすには、引数型以外の型との演算や関数を定義すればよいが、引数型として、引数の型ではないとして使った型を使うと二重定義などの誤りになってしまう。

たとえば、templateライブラリでは、taylor_template<T>とT型の加算は定義されている。taylor_template<T>と整数の加算を行う演算を定義した場合、Tがdouble型なら問題なく動作するが、Tが整数型なら、同じ演算が２度定義されることになり、誤りになる。このようなことを避けるため、taylor_template<double>と整数型の加算は定義していない。

Templateクラスの演算は、引数の型とtempateクラスの型だけの演算だけしか定義されていないので、それ以外は、キャストなどを利用して、型を合わせる必要がある。

7. あとがき

Taylor級数のtemplateライブラリを作成し提供することになった。はじめはdouble型専用のTaylor級数のライブラリを考えていたが、複素数も使いたいとの要求もあったので、templateライブラリ型にした。このようなtemplateライブラリは、見本となるのはC++言語に標準で入っているSTLや次期のSTLと呼ばれるBoostライブラリなどがあるが、ここでは、単純なtemplateの使用にとどめた。

このプログラムでは、ある程度使いやすさ優先させることにした。このため、グロー

バルな変数（deg）を使用した。このため、マルチスレッドのプログラムはあまり速くならないことになった。これを避けることは可能であるが、著しく使いやすさを犠牲にすることになるため、今回は、マルチスレッド環境での高速演算を諦めることにした。

このライブラリに関して、改善点などの意見・感想をお寄せください。

参考文献

[1] Abramowitz M. and Stegun I.A., Handbook of Mathmatical Functions, Dover, New York, 1970

[2] Corliss G. and Chang Y. F., Solving Ordinary Differential Equations Using Taylor Series, ACM Trans. Math. Soft., 8(1982), 114-144

[3] Ellis M. A. and Stroustrup B., The Annotated C++ Reference Manual, Addison-Wesley, New York, 1990

[4] Hairer E., Wanner G., Solving Ordinary Differential Equations II, Springer-Verlag, 1991

[5] Henrici P., Applied and Computational Complex Analysis, Vol. 1, Chap. 1, John Wiley & Sons, New York, 1974

[6] 平山弘, C++言語によるべき型特異点をもつ関数の数値積分, 日本応用数理学会論文誌, 5(1995), 257-266

[7] 平山, 小宮、佐藤、”Taylor級数法による常微分方程式の解法”, 日本応用数理学会、Vol 12. No.1, pp.1-8,2002

[8] Rall,L. B., Automatic Differentiation Technique and Applications, Lecture Notes in Computer Science, Vol. 120, Springer Verlag, Berlin-Heidelberg-New York, 1981