計算天文学 II 第6回 常微分方程式の初期値問題 (1)

牧野淳一郎

2001 年 11 月 26 日

1 常微分方程式

今週と来週は常微分方程式の話をする。計算天文学 I でも常微分方程式についてはかなり詳しく扱っているはずなので、ここでは I でカバーしていないいくつかの話題をとりあげることにする。

2 ルンゲ・クッタ（復習）

計算天文学 I では、ルンゲ・クッタ (RK) 法とそのバリエーションを扱った。ルンゲクッタ法とは、非常に一般的には以下のような形に書ける方法である。

$$\begin{aligned} y_{n+1} &= y_n + h\sum_{i=1}^{s} b_i k_i \\ k_i &= f(y_n + h\sum_{j=1}^{s} a_{ij}k_j, x_n + c_i h) \end{aligned} \tag{1}$$

自然数 s を段数（number of stages）という。a_{ij}, b_i, c_i はパラメータであるが、a と c は普通

$$c_i = \sum_{j=1}^{s} a_{ij} \tag{2}$$

となるようにとる。これは、一般にそうでないような公式は不可能ではないがあまりいいことがない（精度がよくならない）からである。

と、こう、式に書いてしまうとすぐにはわからないが、例えば $s = 2$ の場合に書き下してみると

$$\begin{aligned} x_{n+1} &= x_n + h(k_1 b_1 + k_2 b_2) \\ k_1 &= f(x_n + h(a_{11}k_1 + a_{12}k_2), t_n + c_1 h) \\ k_2 &= f(x_n + h(a_{21}k_1 + a_{22}k_2), t_n + c_2 h) \end{aligned} \tag{3}$$

と、まあ、こんな感じになる。こちらを良く見ればすぐにわかるように、

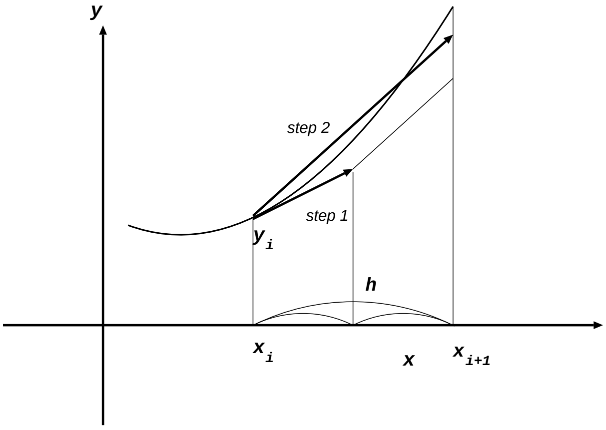
1. a_{ij} $(j \geq i)$ がすべて 0 ならば、k_1から順に計算していくことができる。つまり、「陽的」公式になっている。

2. $a_{ij}\quad(j>i)$ が 0 のときは、各 k_i についての式に k_i だけが入ってくる。これを半陰的 (semi-implicit) 公式という。この場合には、まず k_1 についての方程式をといて、次に k_2 についてのを解いて、、、と順番に計算出来る。
3. 上のような制約が全くない時は、「陰的」公式ということになる。このときは、すべての k_i に対する (一般には非線形な) 方程式を一度に解く必要がある。

今日はとりあえず陽的な公式だけを考える。形式的には、もっとも簡単な前進オイラー法もルンゲ・クッタ公式の一つということになるが、まあ、普通もっとも簡単な RK 法というと、2 次の公式

$$
\begin{aligned}
k_1 &= \frac{h}{2} f(y_i, x_i) \\
y_{i+1} &= x_i + h f(y_i + k_1, x_i + h/2) \qquad (4)
\end{aligned}
$$

である。この方法がどのように働くかについては、図的な説明というものが可能である。元の点からまずオイラー法と同様に接線を引く。が、これを次の時刻まで延ばすのではなく、ステップの半分のところで止める。で、ここでもう一回微分方程式の右辺を評価する。ここでの導関数の値を使って、もとのところ (t_i, x_i) から直線を引くわけである。



普通は、RK 法というといわゆる古典的 RK 公式

$$
\begin{aligned}
y_{n+1} &= y_n + h(k_1/6 + k_2/3 + k_3/3 + k_4/6) \\
k_1 &= f(y_n, t_n) \\
k_2 &= f(y_n + hk_1/2, x_n + h/2) \\
k_3 &= f(y_n + hk_2/2, x_n + h/2) \\
k_4 &= f(y_n + hk_3, x_n + h) \qquad (5)
\end{aligned}
$$

のことをさす。これは、いろいろ良い性質をもつ。例えば

1. a_{ij} が $i-j=1$ 以外すべて 0 なので、右辺の計算が楽である。
2. 次数が 4 次であり、4 段陽的公式で到達可能な最高次数を達成している
3. 係数が簡単な有理数なので、プログラムしやすい。また丸め誤差を小さくできる。

この公式が 4 次であることを示すのは、それほど簡単ではない。腕力に自信があるひとは挑戦してみて欲しい。

4 次よりも高精度な公式というのも非常によく研究されていて無数にいろんなものが知られている。ただし、4 次よりも高い次数では、次数よりも必要な段数が大きくなる。この 2 つの関係が一般にどうなるかは未解決の問題である。というようなこともあるので、よほど変わったことをしたいというのでない限り、専門家が作った公式を使うのが無難である。

Dormand と Prince は、最近、段数 / 次数のさまざまな組合せについて、離散化誤差を非常に小さくした公式を求めた。これらは、

`http://www.unige.ch/math/folks/hairer/software.html`

から入手できる (Fotran と C がある)。

普通の (というのがどういう意味かは今のところ明らかではないが) 常微分方程式を手軽に解くにはこれらの RK 公式を使えば十分である。

実用的には、こちらが欲しい計算精度をどうやって達成するかという問題がある。この問題を解決するためには、誤差を推定しながら積分の刻み幅を自動的に調節する必要がある。これについては次回にちょっと触れる。

3 ルンゲ・クッタ以外の方法

さて、「普通は」RK で十分というのは、もちろん時と場合によっては違う方法を使うべきであるという意味である。なぜそういう場合があるかというのは、以下の理由による。

- RK 法は (次数のいかんにかかわらず) ある問題を解くのにもっとも計算量が少ない方法ではないことが多い。
- 特に、問題の種類によっては、特別な RK 法を使うと非常にうまくいくことがある。
- 問題の種類によっては、普通の RK 法ではどんなに計算時間をかけてもまともな答がでない。

まあ、このへんは詳しい話をやりだすときりがないので、簡単に。以下、上の 3 つを順番に扱う。

4 線形多段階法

RK 法は、「一段階」法である。これはどういう意味かというと、微分方程式

$$\frac{dy}{dx} = f(x, y) \tag{6}$$

の $x = x_i$ での近似解 y_i があったとして、次の x の値 $x_{i+1} = x_i + \Delta x_i$ での近似解 y_{i+1} を求めるのに、x_i, y_i と微分方程式そのものだけで十分であるということである。中間のややこしい値を計算する必要はあるが、それは RK 法のほうが勝手にやることで、使う方が入力を与える必要はない。

これに対して、一段階ではない方法、つまり多段階法というのは、y_{i+1} を計算するのに、y_i 以外の情報、例えば y_{i-1} や、そこで計算した導関数の値 f_{i-1} 、さらにもっと昔の情報を使うやりかたのことである。これは、プログラムとしてはもちろん RK 法に比べれば面倒になる。昔の値をとっておかないといけないし、また、一番最初に計算を始める時にどうするかという問題もあるからである。

さらに、「一段階でない」というだけなので、可能な計算法にあまりに多様な可能性がある。例えば、以下のような方法が考えられる。

$$y_{i+1} = y_{i-1} + 2\frac{f_i}{\Delta x} \tag{7}$$

これは、陽的中点公式といわれるもので、式としては偏微分方程式の解法の時に空間微分の1次の項に使った中心差分と同じ形になっている。従って、理屈としては2次の精度をもった公式になっている。

この方法は、しかし、立派な名前までついているのにもかかわらず、実は使えない公式である。

どのような問題があるのかを示すために、以下の線形方程式

$$y = -ky \tag{8}$$

に陽的中点公式を適用したらなにが起きるかを考えてみる。刻みは固定のΔxとすると、離散化したものは

$$y_{i+1} = y_{i-1} + 2\frac{-ky_i}{\Delta x} \tag{9}$$

になる。これは線形差分方程式なので、前にもやったように固有値を調べればいい。今$\alpha = k/\Delta x$として整理すると、固有方程式は

$$\lambda^2 + 2\alpha\lambda - 1 = 0 \tag{10}$$

となって、これは実解を2つ持つ。それらをλ_1, λ_2とすれば、$\lambda_1\lambda_2 = 1$なので$\alpha \neq 0$ならどちらかは必ず絶対値が1より大きい。

ちょっと式を見ればわかるように、絶対値が1より大きい固有値は$\alpha > 0$、つまり$k > 0$なら負である。したがって、必ず振動的に発散することになる。

なお、上のような、安定な線形微分方程式について振舞いを調べるというのが、常微分方程式の安定性解析の基本になる。非線形な方程式では違うとかいろんなことがあるかもしれないわけだが、まあ、少なくとも線形安定でないと話にならないし、それ以上のことは一般論としていうのは難しいからである。

4.1 アダムス法

さて、中点公式はともかくとして、使える線形多段階法はじゃあどんなものかというわけだが、これも無限にいろんな作り方がある。そのへんの詳しい話はそういう本に譲ることにして、ここではもっとも広く使われているアダムス法について説明する。

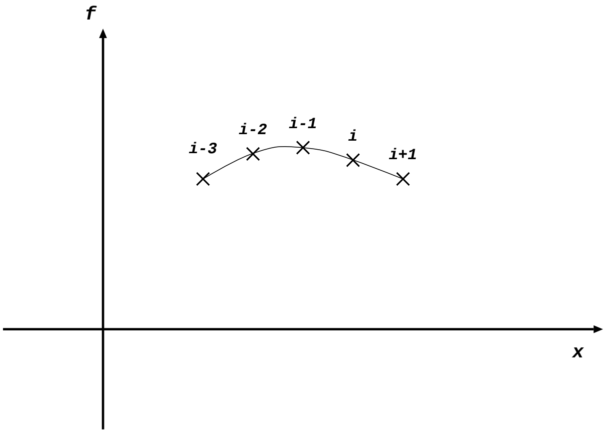
原理は、いくつかのステップでの導関数（微分方程式の右辺）fの値を憶えておいて、それを通る補間多項式を作り、それを積分して解を求めようというものである。

上の図に概念を示す。ここでは、ラグランジュ補間（ニュートン補間）をして多項式を作る。で、その作った多項式を積分する。例えば、点iから$i+1$まで積分するのに、点$i-p$からiまでの関数値を使うとすれば、p次の多項式で

$$P(x_j) = f_j = f(x_j, y_j) \quad (i-p \leq j \leq i) \tag{11}$$

を満たすものを作る。で、$i+1$での解y_{i+1}は

$$y_{i+1} = y_i + \int_{x_i}^{x_{i+1}} P(x)dx \tag{12}$$



で与えられる。刻み h が定数であるとすれば、p を決めれば上の式を

$$y_{i+1} = y_i + h\sum_{l=0}^{p} a_{pl} f_{i-l} \tag{13}$$

の形に書き直せる。

簡単な例として、$p = 1$ の場合を考えてみよう。この時、補間多項式は一次であって

$$P(x) = f_i - \frac{f_{i-1} - f_i}{h}(x - x_i) \tag{14}$$

となって、これを積分すれば、結局

$$y_{i+1} = y_i + \frac{h}{2}(3f_i - f_{i-1}) \tag{15}$$

となる。

一般に、アダムス法では任意段数の公式が構成でき、その次数は段数に等しいことがわかっている。これは、ルンゲ・クッタなどに比べればはるかによい性質をもっているということでもある。

4.2 出発公式

アダムス法はいくらでも高次の公式が作れ、計算量もあまり多くないということがわかっているが、必ずしも広く使われているというわけでもない。その理由はいろいろあるが、一つは、

「どうやって計算を始めるべきかよくわからない」

ということである。つまり、初期値問題としてはもちろん x_0 における y_0 しか知らないのに、多段階法ではその前の時刻での解が必要になるわけである。これに対する対応策はいくつかあるが、時間刻み一定の場合には、基本的にはルンゲ・クッタなどの別な方法で解を求めておくというやりかたが普通である。

というわけで結局プログラムを書く手間が 2 倍以上になるというのが、多段階法の実用上の問題である。

4.3　陰的アダムス法

さて、前に述べた公式では、補間多項式を陽的に求めた。すなわち、時刻 i とそれより以前の値だけを使っていた。これに対し、陰的な補間多項式、つまり t_{i+1}での関数の値を使った公式というものも考えられる。刻み h が定数であるとすれば、p を決めれば前と同様に

$$x_{i+1} = x_i + h \sum_{l=-1}^{p-1} b_{pl} f_{i-l} \tag{16}$$

の形に書けることになる。また手をぬいて $p = 1$ の場合を考えれば、これは単に台形公式

$$x_{i+1} = x_i + \frac{h}{2}(f_i + f_{i+1}) \tag{17}$$

となる。

陰的公式の場合、例によってどうやって代数方程式を解くかが問題になる。通常の方法は、

- 初期値として同じ次数の陽的アダムス法の解を持ちいる。
- そのあとは直接代入法で反復する。

というものである。ただし、特に線形多段階法の場合、反復を繰り返さないことが多い。反復回数を 1 回とか 2 回に固定してしまうのである。なぜそれでいいか、また、そもそもなぜ陰的公式を使うかというあたりはレポート課題ということで。

なお、このやり方を、予測子・修正子法と呼ぶ。線形多段階法はほとんどこの形で使われるため、線形多段階法のことをさして予測子・修正子法と呼ぶ人もいる。

5　構造体とクラス

今回はちょっと C++ 言語の話題というか、今回以降の話で便利な機能について。

偏微分方程式とか、あるいは連立常微分方程式の数値解法をプログラムするのに、普通は配列を使う。しかし、これは割合に面倒だし、いろいろ妙な間違いをする可能性も高い。例えば、$\boldsymbol{x}$ に $\Delta\boldsymbol{x}$ を足すのは、数式としては$\boldsymbol{x} + \Delta\boldsymbol{x}$ で済むのに、例えば C では

```
for(i=0;i<n;i++) x[i] += dx[i];
```

Fortran なら

```
      do i = 1, n
          x(i) = x(i) + dx(i)
      enddo
```

という具合で、数式なら 4 文字で済むところをその何倍も書かないといけない。これは、C にしても Fortran にしても、配列 (あるいはベクトルとか行列) といった、数学では基本的な要素であるものに対する演算を直接に表現する記法を持っていないからである。

まあ、Fortran 95 とかそういったものを使うとそういう記法があるという話もないわけではないが、あまり普及していない。普及していない理由はいろいろあるが、その一つは C++ を使えば同様なことが実現できるからである。

C++ 言語自体は、配列に対する演算というものを用意しているわけではない。しかし、C++ では、プログラムの中で新しい 型」(実数型とか整数型というのと同じ意味での) を定義して、さらにそれに対する演算を定義することができる。これでベクトル型とかいったものを自分の使いやすいように定義すればいいことになる。

例えば、非常に基本的なベクトル型の定義は以下のようなものになる。ここでは加算と入出力くらいしか定義していないが、他の必要な演算も同様に定義できる。一応使いそうな演算を定義したものが

`http://grape.astron.s.u-tokyo.ac.jp/˜makino/pcphysics/programs/vector.h`

にあるので、実際にプログラムを作る時にはこれを使ってもよい。牧野が書いたプログラムは信用できないという向きは自分で書くこと。

```
/-------------------------------------------------------
/  vector  --  a class for VLEN-dimensional vectors
/  VLEN  must be defined as constant before this header
/-------------------------------------------------------
#ifndef  _VECTOR_H
#  define  _VECTOR_H
class vector
{
private:
    double element[VLEN];
public:
    vector(double c=0 ) {for (int i=0;i<VLEN;i++)element[i]=c;}
    double & operator [] (int i)       {return element[i];}
    friend const vector operator + (const vector &,
                                    const vector &);
    vector& operator += (const vector& b)
    {for(int i=0; i<VLEN;i++)element[i] += b.element[i];
    return *this;}
    friend ostream & operator <<(ostream & , const vector & );
    friend istream & operator >> (istream & , vector & );
};

inline ostream & operator <<(ostream & s, const vector & v)
{
    for(int i=0; i<VLEN;i++)s <<v.element[i] <<"  " ;
    return s;
}
inline istream & operator >> (istream & s, vector & v)
{
    for(int i=0; i<VLEN;i++)s >> v.element[i];
    return s;
}
```

```
inline const vector operator + (const vector &v1,
                                const vector & v2)
{
    vector v3;
    for(int i=0; i<VLEN;i++)
        v3.element[i] = v1.element[i]+ v2.element[i];
    return v3;
}
typedef const vector (vfunc)(const vector &);
#endif
```

実際にこれを使うには、

```
const int VLEN = 2;
#include  "vector.h"

double k;
vector dxdt(vector & x)
{
    vector d;
    d[0] = x[1];
    d[1] = -k*x[0];
    return d;
}


    .....
    double h;
    vector kx1;
    kx1 = dxdt(x)*h;
    ......
```

というような具合に、ベクトルの大きさを指定する（これはここでは固定である）。もちろん、可変にするとかいろんなことができるが、繁雑になるのでここでは固定サイズにする。

ここで、注意してほしいのは、 + という演算子や [] という、これも「演算子」が、新しく定義されていることである。これらは、もちろん+ なら実数や整数に対してすでに定義されているが、ここではベクトル同士の演算に対して新しい意味を持つように拡張されていることになる。[] も同様で、配列の他にここで定義したベクトル型について新しい意味を持つようになったわけである（といっても、こちらは普通の配列に対してとまったく同じように働くが）。

これは、偉そうにいうと C++ の演算子多重定義 (overload) という強力な機能である。まあ、落ち着いて考えてみると、この機能の実現はそんなに難しいわけではなくて、例えば + という演算子が出てきたところで、それが適用されているデータ型を見て、そのデータ型に対して定義されている演算を呼ぶようにするというだけのことである。これは、コンパイラにそういう機能を付け加えるだけで実現できる。

C++ の場合には、このような、ある意味での機能拡張は、「クラス」というものを使って実現されている。これはもともとは「オブジェクト指向」とかそういった難しい機能を実現するためのものであるが、ここではとりあえずあんまりそういうことは考えないでベクトル型を使うことにする。

あ、皆さんが使っているコンパイラが十分新しいと、上のプログラムがそのままでは動かないかもしれない。これは C++ の新しい標準では vector という語に別の意味を与えているからである。こういうときはしょうがないので vector を myvector とかそういう別の語に置き換えて使って欲しい。

6 練習

1. 一次元調和振動子
$$\frac{d^2x}{dt^2} = -kx \tag{18}$$
について、初期条件
$$x(0) = 1, \quad \frac{dx}{dt}|_{t=0} = 0 \tag{19}$$
からの、$t = 2\pi$ までの数値解を、古典的ルンゲ・クッタ公式を使って求めよ。刻み幅をいろいろ変えて、精度が刻みにどのように依存するか、また、どこまで高い精度が実現可能か調べ、なぜそうなるかを考察せよ。
なお、牧野が書いたプログラムが

`http://grape.astron.s.u-tokyo.ac.jp/~makino/pcphysics/programs/index.html`

の hermonic5a.C にあるので、今回はこれをコピーしてコンパイル・実行するだけでも一応いいことにする。
2. 刻み幅が等しい場合について、以下の公式を導け
   - 4 次の陽的アダムス公式
   - 4 次の陰的アダムス公式
   - 4 次の陽的シュテルマー公式
3. 上で導いた 4 次の陽的・陰的アダムス公式を使って、1 と同じ単振動の方程式について同様な解析を行なえ。出発公式にはルンゲ・クッタを使うか、あるいは厳密解をつかってもよい。
4. 陰的線形多段階法について、直接代入法が収束する速さが何で決まるかを考察せよ。

7 次週予告

来週は、常微分方程式の続き。