計算天文学 II 第3回

牧野淳一郎

2002 年 11 月 10 日

1 放物型方程式

物理・天文で出てくる偏微分方程式は大抵は 2 階である。で、普通は空間 3 次元と時間 1 次元の 4 次元空間で定義されるわけだが、この講義ではいくつかの特殊な場合を除いて空間 1 次元時間 1 次元の 2 次元で考える。これは、そうしておかないとプログラムを書くのも、また計算機のほうも大変になるからである。

放物型方程式ってのは何か？というのは皆さん知っているはずなので（だよね？）厳密な定義は省くが、要するに以下の形に書ける（移流）拡散方程式のことである。

$$\frac{\partial u}{\partial t} = -K\frac{\partial u}{\partial x} + D\frac{\partial^2 u}{\partial x^2} \tag{1}$$

ここで x, t はそれぞれ空間、時間を表す変数であり、 u は方程式が記述する量である。拡散方程式として見ればなにかの濃度ということになるし、熱伝導の方程式としてみれば温度なりエンタルピーなりということになる。K は一次の係数で、これは普通の拡散方程式や熱伝導方程式では 0 である。これが 0 でないのはどういう場合かというのは後でちょっと考える。 D が普通の拡散係数ということになる。今は D が空間・時間に依存しない場合を考える。

依存しない場合はもちろん変数分離で解けるので、数値計算することに意味があるのは D が空間・時間に依存する場合だが、まあ、とりあえずは答がわかる場合を計算してみることにしよう。

放物型方程式の場合には、初期条件と境界条件を与えないと解が定まらない。以下では、

$$\begin{aligned} \frac{\partial u}{\partial t} &= D\frac{\partial^2 u}{\partial x^2} \\ 0 < x < 1, \quad 0 < t, \quad u(0,t) &= u(1,t) = 0, \quad u(x,0) = u_0(x) \end{aligned} \tag{2}$$

という固定境界の場合を考える。

2 差分法

偏微分方程式の数値計算の方法は基本的には有限要素法と差分法に大別される。あ、あとスペクトラル法というのもあるが、これはちょっとおいておく。有限要素法の考え方はだいぶややこしいので、まず差分法を考える。

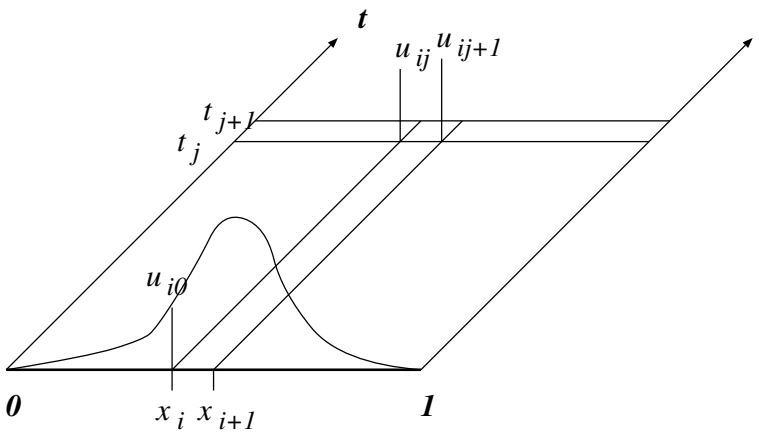

図 1: 差分法の考え

差分法とは、基本的には微分方程式に出てくる空間微分や時間微分の項を、差分で置き換えるものである。差分というのはそもそも何かというのを説明しないといけない。

例えば区間 $[0,1]$ を n 等分して、 x_0 から x_n まで $(x_i = i\Delta x, \Delta x = 1/n)$ で表すとする。時間も同様に、 $t_j = j\Delta t$ として、これから求める近似解 $\hat{u}$ を簡単のため単に u_{ij} と書く。だいぶ記号が繁雑になるので、図 1 に様子を書いておく。

ここで、差分とは例えば $u_{i+1,j} - u_{i,j}$ といった具合に隣との差をとることである。微分を差分で近似するには、例えば

$$\frac{\partial u}{\partial x} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x} + O(\Delta x^2) \tag{3}$$

というようなことになる。

なお、差分のとり方は一意的ではないことに注意して欲しい。例えば、$(u_{i,j} - u_{i-1,j})/\Delta x$ でもいいし、$(u_{i+1,j} - u_{i-1,j})/2\Delta x$ といったものも考えられる。さらに、もっと沢山の点を使うことも原理的には可能である。

さて、拡散方程式なので 2 次微分がいる。これはどう作ればいいかというわけだが、これはむしろ一般に高階微分を差分で表す、つまり数値微分の方法を説明しておくほうが簡単であろう。

$x_{i-k}, x_{i-k+1}, ...x_i, ...x_{i+l}$ の $k+l+1$ 点での関数値 $u_j (i-k \leq j \leq i+l)$ をつかって、点 x_i での m 階導関数の近似値を求める考え方は以下のようなものである。

1. 各点で $p(x_j) = u_j$ を満たす $k+l$ 次補間多項式 $p(x)$ を作る。
2. $p(x)$ を m 回微分する。
3. 微分した結果の式に x_i を代入する。

2 階導関数を作るためには最低 3 点いるので、まずもっとも簡単な以下の形を使ってみる。

$$\frac{\partial^2 u}{\partial x^2} \sim \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} \tag{4}$$

で、時間微分についても、もっとも安直な形

$$\frac{\partial u}{\partial t} = \frac{u_{i,j+1} - u_{i,j}}{\Delta t} \tag{5}$$

を使ってみる。

3 プログラム

今回は Fortran と C++ と両方出してみよう。初期条件は $x = 0.5$ のところにだけ値がある δ 関数的なものということにする。

なお、面倒なので $D = 1$ ということのする。時間の単位が $1/D$ と思えば同じことである。

```
c
c     program parabolic1
c
c     Copyright 2000 J. Makino

      subroutine initparam(u, nx, dx, dt, niter, gmode)
      real*8 u(*), dx, dt, tmax
      integer nx, niter, gmode, i
      write(*,*)'Enter nx, tmax, dt:'
      read(5,*) nx, tmax, dt
      dx = 1d0/nx
      niter = (tmax+dt/2)/dt
      write(*,*)'Enter graphics mode 0: no G'
      write(*,*)'                    1: animation'
      write(*,*)'                    2: no-erase)'
      read(5,*) gmode
      write(6,*) 'nx = ', nx, ' dt = ', dt, ' niter = ', niter
      write(6,*) 'gmode = ', gmode
      do i = 1, nx+1
         u(i) = 0
      enddo
      u((nx+2)/2) = nx
      end

      subroutine push_system(u, unew, nx, dx, dt)
      real*8 u(*), unew(*), dx, dt, lambda
      integer nx, i
      lambda = dt/(dx*dx)
      do i = 2, nx
         unew(i) = u(i) + lambda*(u(i-1)-2*u(i)+u(i+1))
      enddo
      do i = 2, nx
         u(i) = unew(i)
      enddo
      end

      subroutine initgraph
      integer pgopen
      if(pgopen('?') .LE. 0) stop
      call pgask(.false.)
      end
```

```
 subroutine show_graphics(u, nx, dx, gmode,
$     xmin, xmax, ymin,ymax, first)
 real*8 u(*), dx
 real*8 xmin, xmax, ymin, ymax
 integer nx, gmode, i, first
 if (gmode .eq. 0) return
 call pgbbuf
 if (first .eq. 1) then
    call pgpage
    call pgenv(real(xmin), real(xmax), real(ymin), real(ymax),0,-2)
 endif
 if (gmode .eq. 1) then
    call pgeras
 endif
 call pgbox('bcnst', 0.0, 0, 'bcnst', 0.0, 0)
 call pglab('X', 'Y', ' ')
 call pgmove(0.0, real(u(1)))
 do i = 2, nx+1
    call pgdraw(real((i-1)*dx), real(u(i)))
 enddo
 call pgebuf
 end

 program parabolic
 integer nmax
 parameter (nmax=10001)
 real*8 u(nmax), unew(nmax), dx, dt
 integer nx, niter, gmode, iter
 call initparam(u, nx, dx, dt, niter, gmode)
 if (gmode .ne. 0) call initgraph
 call show_graphics(u, nx, dx, gmode,
$        0d0, 1d0, 0d0, 10d0, 1)
 do iter = 1, niter
    call push_system(u, unew, nx, dx, dt)
    call show_graphics(u, nx, dx, gmode,
$        0d0, 1d0, 0d0, 10d0, 0)
 enddo
 end
```

これを適当なファイルに落して、コンパイル・実行する。コンパイルには

```
  f77  -o parabolic1  parabolic1.f -L/home/makino/utils/pgplot5.2 \
          -lpgplot -L/usr/X11R6/lib -lX11
```

とすると、 parabolic1 という名前の実行ファイルができる。

ここでは、 PGPLOT というわりと広く使われているグラフィックライブラリを使う。これは画面、PS ファイル等様々な出力形態を同じプログラムで切替えられるので割合に便利である。

中身の解説は、見ての通りという気もする。次に C++ のプログラムをつける

4 C++ でのプログラム

```
//     program parabolic1

#include <iostream>
using namespace std;
#include "cpgplot.h"

void initparam(double  u[],
               int & nx,
               double &dx,
               double & dt,
               int & niter,
               int &gmode)
{
    cerr << "Enter nx, tmax, dt:";
    double tmax;
    cin >> nx >> tmax >> dt;
    dx = 1.0/nx;
    niter = (tmax+dt/2)/dt;
    cerr << "Enter graphics mode 0: no G\n"
         << "                    1: animation\n"
         << "                    2: no-erase)\n";
    cin >> gmode;
    cout << "nx = " << nx<< " dt = " << dt << " niter = "<< niter<< "\n";
    cout << "gmode = " <<  gmode << "\n";
    int i;
    for (i=0;i<=nx; i++) u[i] = 0;
    u[(nx+1)/2] = nx;
}

void push_system(double u[],
                 double unew[],
                 int nx,
                 double dx,
                 double dt)
{
      double lambda = dt/(dx*dx);
      int i;
      for (i=1; i<nx; i++){
          unew[i] = u[i] + lambda*(u[i-1]-2*u[i]+u[i+1]);
      }
      for(i=1; i<nx; i++){
         u[i] = unew[i];
```

```
      }
}

void initgraph()
{
    if(cpgopen("?") !=1 ) exit(-1);
    cpgask(0);
}

void show_graphics(double u[],
                   int nx,
                   double dx,
                   int gmode,
                   double xmin,
                   double xmax,
                   double ymin,
                   double ymax,
                   int first)
{
    if (gmode == 0) return;
    cpgbbuf();
    if (first == 1){
         cpgpage();
         cpgenv(xmin, xmax, ymin, ymax,0,-2);
    }
    if (gmode == 1) {
         cpgeras();
    }
    cpgbox("bcnst", 0.0, 0, "bcnst", 0.0, 0);
    cpglab("X", "Y", " ");
    cpgmove(0.0, u[0]);
    int i;
    for(i=1;i<nx+1;i++){
        cpgdraw(i*dx, u[i]);
    }
    cpgebuf();
}

int main()
{
#define NMAX 10001
    static double u[NMAX], unew[NMAX];
    double dx, dt;
    int nx, niter, gmode, iter;
```

```
    initparam(u, nx, dx, dt, niter, gmode);
    if (gmode !=  0) initgraph();
    show_graphics(u, nx, dx, gmode, 0.0, 1.0, 0.0, 10.0, 1);
    for(iter = 1; iter <=niter; iter++){
         push_system(u, unew, nx, dx, dt);
         show_graphics(u, nx, dx, gmode,0.0, 1.0, 0.0, 10.0, 0);
    }
    return 0;
}
```

これをちょっと動かしてみよう。

コンパイルには

```
g++  -I/usr/local/include  -o cparabolic1 cparabolic1.C \
         -L/usr/local/lib -lcpgplot -lpgplot \
         -L/usr/X11R6/lib -lX11 -lg2c -lm
```

でいいはずである。 g++ のバージョンによっては、 -lg2c が -lf2c でないといけないかもしれない。また、 PGPLOT をインストールした場所が違うとそのあたりが変わってくる。

5 安定性

ここまでで、拡散方程式に対するもっとも簡単な差分近似である、空間微分

$$\frac{\partial^2 u}{\partial x^2} \sim \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} \tag{6}$$

と、時間微分

$$\frac{\partial u}{\partial t} = \frac{u_{i,j+1} - u_{i,j}}{\Delta t} \tag{7}$$

の組合せをプログラムにしてみた。これを動かしてみた人は気がついたと思うが、この方法は $\Delta t/\Delta x^2 > 0.5$ になると「うまく動かない」。

具体的には、拡散方程式なので本来は次第に滑らかになっていくべきなのに、どんどんガタガタになってしまう。しかも、これは $\Delta t/\Delta x^2 > 0.5$ であれば Δx や Δt をどんなに小さくしても起こる。

まず、なぜそういう振動が起きるのかを考え、それから振動を起こさない方法があるのかどうかを議論していこう。

6 線形安定性解析

さて、まず、なぜ振動が起きるかということだが、これは原理的には差分化された方程式の固有値を求めればわかる。

つまり、u_j を時刻 $t_j = j\Delta t$ での数値解をベクトルとして書いたものだとすると、差分近似は、

$$u_{j+1} = Au_j \tag{8}$$

という形に書ける。ここで A は $(n-1)$ 次元正方行列で、その各要素は差分近似の形から決まる。この場合には、対角要素とその両側一列以外の全ての要素が 0 である 3 重対角行列といわれる形をしている。要素の値は

$$a_{ii} = 1 - 2\delta \quad (0 < i < n) \tag{9}$$

$$a_{i,i-1} = a_{i-1,i} = \delta \tag{10}$$

ここで、 $\delta = \Delta t/\Delta x^2$ である。

この行列は実対称行列なので固有値分解ができて、適当なユニタリ行列 U を持ってきて

$$A = U^{-1}\Lambda U \tag{11}$$

と書ける。ここで Λ は固有値行列、つまり対角要素が固有値でそれ以外が 0 の行列である。

このように分解できるので、 $v = Uu$ とすれば最初の差分近似式は

$$v_{j+1} = \Lambda v_j \tag{12}$$

つまり、

$$v_{i,j+1} = \lambda_i v_{ij} \tag{13}$$

という $n-1$ 個の独立な方程式になって、解も自明に求まる。いうまでもないが、 U の各列は A の固有ベクトルになっている。つまり、U の i 行目を x_i と書くと、

$$\lambda x_i = Ax_i \tag{14}$$

である。

不安定性が起きないための条件は、全ての固有値 λ_i の絶対値が 1 を超えないということである。

6.1 固有値を求める

さて、そういうわけで、固有値 λ がどうなっているか調べてみよう。式 (14) をもともとの差分近似に入れてちょっと変形すると、

$$(\lambda + 2\delta - 1)u_i = \delta(u_{i+1} + u_{i-1}) \tag{15}$$

となる。 x がいつのまにか u に戻っているが深い意味はない。ここで $\delta = \Delta t/\Delta x^2$ である。さらに、

$$\alpha = \frac{\lambda + 2\delta - 1}{\delta} \tag{16}$$

とおいてもうちょっと変形すると、結局

$$u_{i+1} - \alpha u_i + u_{i-1} = 0 \tag{17}$$

となる。

断るまでもないと思うが、これは元の偏微分を変数分離して空間方向の関数についての常微分方程式を導くのとほとんど同じ操作になっている。これを u_i についての差分方程式と見て解を求めることを考える。

線形差分方程式なので、解は $u_i = Cp^i$ の形である。これを代入すると

$$p^2 - \alpha p + 1 = 0 \tag{18}$$

解は

$$p = \frac{\alpha \pm \sqrt{\alpha^2 - 4}}{2} \tag{19}$$

となる。

真面目にやるには境界条件を満たす固有ベクトルをもとめないといけないが、面倒なので無限遠境界の場合を考える。この時、 p が実数のものはどちらかで無限大に発散するのでよろしくないので、複素数の場合を考える。この時は $|p| = 1$ になっているので無限遠でも発散しない。

なお、有限の固定境界の場合も結局境界条件を満たすような解を作るためには p が複素数でないといけないことがすぐにわかる。このため、 $|\alpha| < 2$ だけを考えればよい。

このとき、式 (16) は

$$\lambda = 1 - (2 - \alpha)\delta \tag{20}$$

と書き直せる。$\delta > 0$ なので、$|\alpha| < 2$ なるある α について $|\lambda| < 1$ であるためには、結局

$$\delta < \frac{2}{2 - \alpha} \tag{21}$$

を満たせば良く、任意の α についてなり立つためには

$$\delta < \frac{1}{2} \tag{22}$$

つまり

$$\frac{\Delta t}{\Delta x^2} < \frac{1}{2} \tag{23}$$

であればいいことになる。

上でやったことは、結局空間方向をフーリエ級数展開して、各空間波長に対する時間発展が安定である (減衰していく) ことを要求しているのと同じである。このようなやり方を von Neumann の方法による安定性解析という。

6.2 「直観的」説明

上の議論からわかるように、不安定条件に関係する α の値は実際には $\alpha = -2$ だけで、それ以外の α からはもっと緩い条件しかでない。 $\alpha = -2$ は $p = -1$ に対応するので、これは結局 u_i と u_{i+1} で値が逆転するようなパターンである。そのようなパターン (モード) が 1 ステップ計算するとどうなるかをもう一度もとの差分式に戻って書き直してみると、結局

$$u_{i,j+1} = (1 - 4\Delta t/\Delta x^2)u_{ij} \tag{24}$$

となる。括弧内の絶対値が 1 より大きいと、パターンがどんどん成長していってめちゃくちゃな答になるわけである。括弧内の絶対値が 1 より小さいためには、

$$\frac{\Delta t}{\Delta x^2} < \frac{1}{2} \tag{25}$$

であればよく、前節の結果と同じである。このように、偏微分方程式の場合には大抵もっとも波長の短いモードが減衰できるかどうかで安定性が決まる。

7 陰解法

前節の議論から、初めに作ったプログラムでまともな答が求まるためには、$\Delta t/\Delta x^2 < 1/2$ でないといけないということがわかった。これは結構厳しい条件である。というのは、空間刻みの数を増やすと、その 2 乗に比例して時間刻みの数を増やさないといけないので、全体としては空間刻みの数の 3 乗に比例して計算時間がかかるということになるからである。まあ、最近の計算機は速いので待てなくもないかもしれないが、もうちょっとなんとかならないかという気もする。

実は、なんとかなる方法というのはある。これが陰解法 (implicit method) というものである。拡散方程式の場合、もっとも簡単な陰解法は、空間微分に u_j ではなく u_{j+1} のほうを使う、つまり、

$$\frac{\partial^2 u}{\partial x^2} \sim \frac{u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1}}{\Delta x^2} \tag{26}$$

というものを使うというだけである。時間微分は同じものですます。差分公式としてまとめて書くと、

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1}}{\Delta x^2} \tag{27}$$

となる。最初の方法では、 $u_{i,j+1}$ は左辺に単独で現れ、右辺は時刻 j の項だけだったので、$u_{i,j+1}$ を直接計算できた。このように、直接計算できる方法のことを陽解法 explicit method という。これに対し、上に書いた方法では時刻 $j+1$ の項が右辺に複数現れているので、これは全体として u_{j+1} についての線形連立方程式になっている。このように、新しい時刻での解が代数方程式を解かないと求められない方法のことを陰解法という。

線形連立方程式を解かないといけないのはもちろん面倒なわけだが、それだけのことはある。先ほどと全く同じように δ とか λ を定義すると、両者の関係が今度は

$$\lambda = \frac{1 + \alpha\delta}{1 + 2\delta} \tag{28}$$

となる。したがって、 $|\alpha| < 2$ なる任意の α と $\delta > 0$ なる任意の δ について、 $|\lambda| < 1$ が満たされているのである。つまり、どんなに Δt を大きくとっても、決して不安定にならない。この性質を無条件安定という。この場合には、安定性ではなくて計算精度の観点から Δt を決められることになり、だいぶましである。

7.1 3 重対角行列を解く

陽解法と同じように行列で書いてみると、

$$Au_{j+1} = u_j \tag{29}$$

で、

$$a_{ii} = 1 + 2\delta \tag{30}$$

$$a_{i,i-1} = a_{i-1,i} = -\delta \tag{31}$$

ということになる。A は δ の符号が違うだけで前と同じ三重対角行列になっている。行列 (以下、線形連立方程式のことを単に行列ということがある) を解くのは例えばガウスの消去法では計算量が一般には次元の 3 乗になるので嬉しくないが、三重対角の場合には計算量が次元数に比例する。前進消去で消さないといけない項が常に 1 つだけだからである。

具体的には、a_{ii} を ad, a_{ii-1} を al, a_{ii+1} を ad、右辺の値を b と表して、前進消去の部分が、

```
        double c = al[i]/ad[i-1];
        ad[i] -= c*au[i-1];
        b[i] -=  c*b[i-1];
```

というだけである。後退代入も、 au のところだけを消せばいいので

```
        b[i] = (b[i]-au[i]*b[i+1])/ad[i];
```

というふうに一つ下の値を引くだけになる。

一応、3 重対角行列を解くプログラム全体を与えておくと、

```
void solve_tridiagonal(double ad[],
                       double al[],
                       double au[],
                       double b[],
                       int n)
{
    int i;
    for(i=1;i<n;i++){
        double c = al[i]/ad[i-1];
        ad[i] -= c*au[i-1];
        b[i] -=  c*b[i-1];
    }
    b[n-1] /= ad[n-1];
    for(i=n-2;i>=0;i--){
        b[i] = (b[i]-au[i]*b[i+1])/ad[i];
    }
}
```

となる。

係数の ad 等を計算して、1 ステップ進める関数全体は、

```
void push_system(double u[],
                 double ad[],
                 double au[],
                 double al[],
                 int nx,
                 double dx,
```

```
                double dt)
{
      double delta = dt/(dx*dx);
      int i;
      for(i=0;i<nx-1;i++){
          ad[i] = 1 + 2*delta;
          al[i] = au[i] = -delta;
      }
      solve_tridiagonal(ad, al, au, u+1, nx-1);
}
```

である。

8 高精度の方法

さて、前節の簡単な陰解法は無条件安定で大変結構なのであるが、時間刻みを大きくとれるとなると、今度は計算精度のほうが気になってくる。空間方向の微分は 2 次精度で、テイラー展開の 2 次の項までが正しく入っているのに対し、時間微分は 1 次精度にしかなっていないからである。

なお、上の 1 次の陽解法のことを 1 次の前進差分、陰解法のことを 1 次の後退差分ということがある。

時間方向の精度をあげる一つの方法は、これまでに見てきた 2 つの方法を混ぜて使うことである。つまり、

$$\frac{u_{i,j+1}-u_{i,j}}{\Delta t} = \frac{u_{i+1,j+1}-2u_{i,j+1}+u_{i-1,j+1}}{2\Delta x^2} + \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{2\Delta x^2} \tag{32}$$

この方法をクランク・ニコルソン法という。これは時間方向に対称な形になっている。この方法では、時間方向も 2 次精度にすることができる。

9 練習

1. メインプログラムを付け加えて、1 次の陰解法のプログラムを完成し、$\Delta t/\Delta x$ をいろいろ変えて安定に計算できることを確認せよ。
2. クランク・ニコルソン法が無条件安定であることを証明せよ。
3. クランク・ニコルソン法のプログラムを作り、上と同様に安定であることを確認せよ。
4. クランク・ニコルソン法の誤差が $\Delta t, \Delta x$ のそれぞれ何乗になっているか、 $\Delta t, \Delta x$ をいろいろ変えてプログラムを走らせて結果を厳密解と比べることで調べよ。
5. クランク・ニコルソン法が時間方向 2 次精度であることを証明せよ。
6. 前進差分、クランク・ニコルソン法、後退差分は、 時刻 j と $j+1$ での空間微分の混ぜ方を変えているだけである。これらすべてを統一的に計算できるプログラムを作成し、正しく動くことを確認せよ。

7. さらに、周期境界の場合も扱えるようにプログラムを拡張せよ。3 重対角行列の処理の部分をどう変えればいいか？

10　次週予告

楕円型方程式を扱う。