シミュレーション概論ノート第10講

14 ラプラス方程式

14.1 全般的注意

今回はラプラス方程式を扱う。ラプラス方程式は電磁気学の静電場を記述する方程式として現れたが、むしろその真骨頂は複素解析における重要な役割を強調すべきであろう。その解は調和関数として知られて古くから膨大な研究がなされているだけでなく、2次元完全流体等、多くの物理系でもお馴染みの方程式となっている。

ラプラス方程式は2次元以上でなくては無意味である。特に2次元の場合は調和解析と結び付く特別な意味合いがあるのでここでの解析は2次元系に限定するものとしよう。方程式は

$$(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2})u(x,y)=0 \tag{1}$$

である。領域は簡単のために $0\le x\le 1$,$0\le y\le 1$ とする。[1]ここで境界条件は Dirchlet 型の固定境界のものを考える。周期境界が意味を持たないのはほぼ明らかであろう。具体的には

$$u(x,0) = \phi_1(x),\quad u(x,1)=\phi_2(x)\quad (0\le x\le 1) \tag{2}$$
$$u(0,y) = \psi_1(y),\quad u(1,y)=\psi_2(y)\quad (0\le y\le 1) \tag{3}$$

としよう。境界条件の整合性のために$\phi_1(0)=\psi_1(0)$, $\phi_1(1)=\psi_2(1)$, $\phi_2(0)=\psi_1(1)$, $\phi_2(1)=\psi_2(1)$ が必要である。

(1) の差分解を求めるよう。正方格子を考えて x,y方向に N格子を考える。刻みを h として $Nh=1$ としよう。また $x_i=ih$, $y_j=jh$ として微分解 $u(x_i,y_j)$ に対応する差分解を $U_{i,j}$と表す。中心差分を行なえば

$$4U_{i,j}=U_{i+1,j}+U_{i-1,j}+U_{i,j+1}+U_{i,j-1} \tag{4}$$

となることは直ちにわかる。上式はある格子点での値は最近接する格子点での平均値で置き換えられており、近接格子点の中に最大値、最小値があるという最大値の原理をラプラス方程式が満たしていることの反映である。また境界条件は

$$U_{i,0} = \phi_1(x),\quad U_{i,N}=\phi_2(x)\quad (i=0,1,\cdots,N) \tag{5}$$
$$U_{0,j} = \psi_1(y),\quad U_{N,j}=\psi_2(y)\quad (j=0,1,\cdots,N) \tag{6}$$

となる。

さて (4) は端から順番に計算できる形になっていないので直ちにプログラムを書くことは難しい。ここで $N=4$ のときに具体的に (4) を連立一次方程式に書き直すと次の様になることが容易に確認できる。

$$\begin{pmatrix} A & B & 0\\ B & A & B\\ 0 & B & A\end{pmatrix}\begin{pmatrix}\mathbf{U}_1\\ \mathbf{U}_2\\ \mathbf{U}_3\end{pmatrix}=\begin{pmatrix}\mathbf{f}_1\\ \mathbf{f}_2\\ \mathbf{f}_3\end{pmatrix} \tag{7}$$

[1]一般に複雑な形状を持つ境界条件下では差分法は有効ではない。その場合は有限要素法などを用いるか、等角写像などを用いて簡単な座標系で計算したものを元の座標に移すという方法が取られる。

と書ける。但し A,Bはそれぞれ 3×3 の小行列であり

$$A=\begin{pmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix}, B=\begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \tag{8}$$

で与えられ、$\mathbf{U}_j$, $\mathbf{f}_j$は

$$\begin{aligned} \mathbf{U}_j &= (U_{1,j}, U_{2,j}, U_{3,j})^T \\ \mathbf{f}_1 &= (U_{0,1}+U_{1,0}, U_{2,0}, U_{4,1}+U_{3,0})^T \\ \mathbf{f}_2 &= (U_{0,j}, 0, U_{4,j})^T \\ \mathbf{f}_3 &= (U_{0,3}+U_{1,4}, U_{2,4}, U_{4,3}+U_{3,4})^T \end{aligned} \tag{9}$$

となる。ここで Tは転置を表す。

上の結果を一般の Nに拡張することは容易である。(7) は

$$\begin{pmatrix} A & B & & & & \\ B & A & B & & & \\ & \ddots & \ddots & \ddots & & \\ & & & B & A & B \\ & & & & B & A \end{pmatrix} \begin{pmatrix} \mathbf{U}_1 \\ \mathbf{U}_2 \\ \vdots \\ \mathbf{U}_{N-2} \\ \mathbf{U}_{N-1} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_{N-2} \\ \mathbf{f}_{N-1} \end{pmatrix} \tag{10}$$

及び

$$A=\begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{pmatrix}, B=\begin{pmatrix} -1 & & & & \\ & -1 & & & \\ & & \ddots & & \\ & & & -1 & \\ & & & & -1 \end{pmatrix} \tag{11}$$

で与えられ、$\mathbf{U}_j$, $\mathbf{f}_j$は

$$\begin{aligned} \mathbf{U}_j &= (U_{1,j}, U_{2,j}, \cdots, U_{N-1,j})^T \\ \mathbf{f}_1 &= (U_{0,1}+U_{1,0}, U_{2,0}, \cdots, U_{N,1}+U_{N-1,0})^T \\ \mathbf{f}_2 &= (U_{0,j}, 0, 0, \cdots, U_{N,j})^T \\ \mathbf{f}_3 &= (U_{0,N-1}+U_{1,N}, U_{2,N}, \cdots, U_{N-2,N}, U_{N,N-1}+U_{N-1,N})^T \end{aligned} \tag{12}$$

となる。

(10) の左辺の行列では、小行列が 3 重の尾日で対角に並んでいた。このような行列はブロック 3 重対角行列と呼ばれ、殆んどの要素が 0 である。このような問題を効率よく解く方法は必ずしも今までのような LU 分解や掃き出し法などに頼ったものとは異なりある種の反復緩和法である場合がある。ここではそうした考え方に基づいたアルゴリズムとして SOR (successive overrelaxation method) を紹介する。これは Gauss-Seidel method の一般化になっている。

こうした反復緩和法は (4) に着目して、それが正しく等式が成立するように緩和するまで反復を繰り返すのである。従って k回目の反復で得られた $U_{i,j}$を $U_{i,j}^{(k)}$と記すと

$$U_{i,j}^{(k+1)} - U_{i,j}^{(k)} = \omega\{(U_{i+1,j}^{(k)} + U_{i-1,j}^{(k)} + U_{i,j+1}^{(k)} + U_{i,j-1}^{(k)})/4 - U_{i,j}^{(k)}\} \tag{13}$$

として左辺の絶対値が小さくなると収束するというアルゴリズムである。ωは加速パラメータと呼ばれ、この場合$\omega = 1.73$ であることが知られている。なお$\omega = 1$ としたものが Gauss-Seidel method である。

アルゴリズムは以下の通りである。

- システムサイズの設定： N, 初期設定,l_{max}を適当に与える。

  ϵを与える。

  - 全ての i, jに対して $U_{i,j} = 0$ とする（但し default でそうなっている）。
  - 境界条件：全ての i, jに対して

$$U_{i,0} = \phi_1(x), \quad U_{i,N} = \phi_2(x) \quad (i = 0, 1, \cdots, N)$$

$$U_{0,j} = \psi_1(y), \quad U_{N,j} = \psi_2(y) \quad (j = 0, 1, \cdots, N)$$

    とする。

- $l = 1, 2, \cdots, l_{max}$の順に

$$sum = 0, \quad error = 0$$

  - $i = 1, 2, \cdots, N-1$ の順に
    - $j = 1, 2, \cdots, N-1$ の順に

$$UU_{i,j} = \frac{\omega}{4}(U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}) + (1-\omega)U_{i,j}$$

$$sum = sum + |UU_{i,j}|$$

$$error = error + |UU_{i,j} - U_{i,j}|$$

$$U_{i,j} = UU_{i,j}$$

      を繰り返す。

    を繰り返す。

  if $error < \epsilon sum$ then goto next step

  を繰り返す

- $U_{i,j}$の値を答える。

といったものになろう。例題として

$$\epsilon = 10^{-10}, N = 20, U_{i,0} = sin(\pi i/20), U_{i,N} = U_{0,j} = U_{N,j} = 0$$

という境界条件を与えてラプラス方程式を解いてみる。$\omega = 1$ とした場合、収束に 766 回かかって、$\omega = 1.73$ の場合は収束に 75 回かかる。但し正しい収束回数及び収束結果を得るためには倍精度の計算が必要である。サンプルプログラムでは収束回数を書きだしていないが書き出すように工夫してみよう。

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C  Laplace equation solver via SOR method
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

         implicit real*8 (a-h,o-z)

         parameter (N=20,lm=1000,w=1.0d0,eps=1.0d-10)
         real*8 u(0:N,0:N)

         pi=4.0*datan(1.0d0)
         h=1.0d0/dble(n)

         do 10 i=0,n
            do 1 j=0,n
              u(i,j)=0.0d0
 1          continue
 10       continue

         do 2 i=1,n-1
            u(i,0)=dsin(pi*dble(i)/dble(n))
 2       continue

         do 300 l=1,lm
            sum=0.d0
            error=0.d0
           do 30 i=1,n-1
             do 3 j=1,n-1
        uuu=(u(i+1,j)+u(i-1,j)+u(i,j+1)+u(i,j-1))*w/4.d0+(1.d0-w)*u(i,j)
c                uuu=dum*w/4.0+(1.0-w)*u(i,j)
                sum=sum+dabs(uuu)
                error=error+dabs(uuu-u(i,j))
                u(i,j)=uuu
 3            continue
 30         continue

c
           if (error.lt.eps*sum) goto 999

 300       continue

 999       do 400 i=0,n
             write(6,500) (dble(i)*h,dble(j)*h,u(i,j),j=0,n)
```

```
          write(6,600)
 400      continue

 500      FORMAT(5x,3f10.4)
 600      format()

          end
```

一方 C のプログラムは

```
************************************************************
  シミュレーション概論

  Laplace equation solver via SOR method

  コンパイルは、
  gcc talaplace.c -lm

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

#define N    20
#define LM   1000
#define W    1.0
#define EPS  1.0e-10

main()
{
  int i, j, l;
  double pi, h, sum, error, dum, uuu;
  double U[N+1][N+1];

  pi = 4.0 * atan(1.0);
  h = 1.0 / (double)N;

  for(i = 0; i <= N; i++){
    for(j = 0; j <= N; j++){
      U[i][j] = 0.0;
    }
  }

  for(i = 1; i <= N - 1; i++){
```

```
    U[i][0] = sin(pi * (double)i / (double)N);
  }

  for(l = 1; l <= LM; l++){
    sum = 0.0;
    error = 0.0;
    for(i = 1; i <= N - 1; i++){
      for(j = 1; j <= N - 1; j++){
dum = U[i+1][j] + U[i-1][j] + U[i][j+1] + U[i][j-1];
uuu = dum * W / 4.0 + (1.0 - W) * U[i][j];
sum += fabs(uuu);
error += fabs(uuu - U[i][j]);
U[i][j] = uuu;
      }
    }
    if(error < EPS * sum) break;
  }

  for(i = 0; i <= N; i++){
    for(j = 0; j <= N; j++){
      printf("\t%10.4f\t%10.4f\t%10.4f\n",
     (double)i * h, (double)j * h, U[i][j]);
    }
    printf("\n");
  }
}
```

となる。

既に述べた通り、最大値、最小値は必ず境界に現れる。従って発散の可能性はなく、拡散方程式や波動方程式のように数値不安定性を気にする必要はない。図として数値解を与えておく。

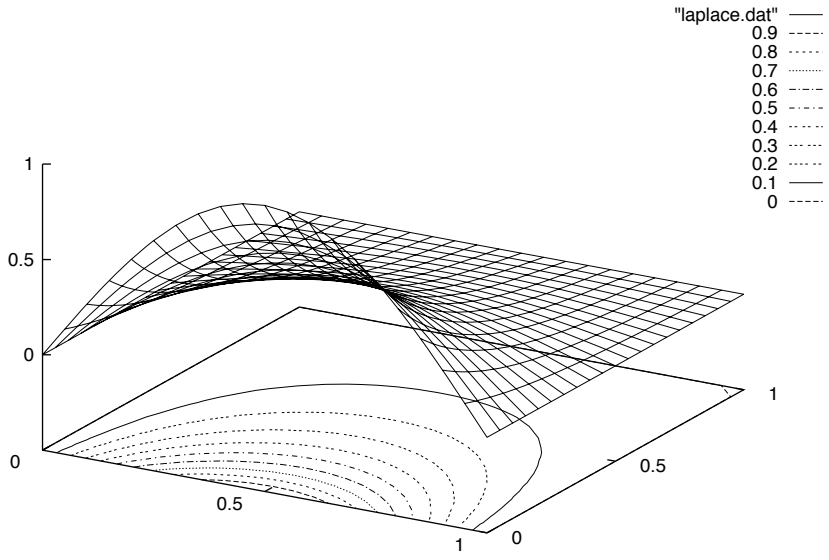

図 1: ラプラス方程式の解