数値流体解析・やってみたらこうなった

金野 祥久

はじめに

この文書の目的

この文書では，理工系大学の 4 年生程度の知識を前提に，数値流体解析のプログラムを一から作り，研究レベルで使えるような数値流体解析プログラムにまで発展させるまでの段階を解説します．

数値流体解析の手法は以下の書籍にある方法をそのまま使います．

> 梶島岳夫著「乱流の数値シミュレーション 改訂版」養賢堂，2014．ISBN978–4–8425–0526–8.
> ☞ Amazon.co.jp へのリンク

この文書は数値流体解析そのものの知識よりも，それをプログラムとして実現（「実装」と表現されます）するための方法を学ぶことを目的としています．ですので，いきなり完成形のプログラムを提示はしません．わざと遠回りして，初歩的なプログラムから書き起こしますし，適切でない実装方法も意図的に取り上げます．

今回対象とするのは数値流体解析のプログラムですが，実際にプログラムを組むときには数値計算の部分だけでなく，ファイルの入出力やエラー処理など，周辺の部分に時間を取られます．この文書では遠回りしてプログラミングの周辺知識を学ぶことで，読者の皆さんがプログラミングする際に本質ではない部分で足を取られる危険をなるべく減らし，本質的な部分に集中できるようにすることを目指します．

想定する環境

この文書の中では，以下の環境で実行できるサンプルプログラムを作成します．

- Ubuntu 14.04 LTS amd64 (64bit OS)
- GNU C++ バージョン 4.8
- 解析結果の可視化に，ParaView を用いる．

- バージョン管理に BitBucket を用いる

これ以外の環境は，少なくとも当面は対象としません．

記号の説明

✎ 鉛筆マーク（✎）は言葉の説明などを表しています．

☞ 人差し指のマーク（☞）は外部参照です．この文書の中にすべての説明を記述するのは労力に対して得るものが少なく，現実的ではないので，Wikipedia などのサイトにある説明を活用しています．

⚠ 注意マーク（⚠）がある部分は，注意する点や間違いやすい点を記しています．この文書の中では，効率の悪いプログラムやアルゴリズムを意図的に使っているところがありますが，そこではこのマークで注意を促しています．

目次

第 1 章

バージョン管理しよう

「はじめに」の章で「わざと遠回りして」と書いた割にはいきなり知識の天下りになりますが，ソースファイルをバージョン管理しましょう．つまり，バージョン管理システムを導入して，そのシステムにプログラムのソースファイルを登録し，変更履歴を管理する，ということです．

✐ ソースファイルとは，ソースコードを記述したファイルのことです．☞ ソースコード（Wikipedia）

☞ バージョン管理システム（Wikipedia）

バージョン管理システムを導入することで，そのファイルに誰が，いつ，どのような変更をしたのかがわかるようになります．チームでひとつのプログラムを作成している時に特に力を発揮しますが，個人で作成している時でも，過去のバージョンとの違いを調べたり，いつ変更したかが分かると便利です．いつでも元に戻せるという安心感があるので，プログラムをどんどん変更し，いろいろ試すことができます．

第 2 章

2 次元キャビティ流れ・問題設定

2.1　計算手法の選択

「はじめに」でも述べましたが，この文書では下記の書籍に基づいて CFD プログラムを作っていきます．（以降ではこの本のことを「教科書」と呼びます．）

> 梶島岳夫著「乱流の数値シミュレーション 改訂版」養賢堂，2014．ISBN978–4–8425–0526–8.

✎ CFD とは Computational Fluid Dynamics の頭文字をとったもので，日本語では「**数値流体力学**」とか「計算流体力学」と訳されます．コンピュータを使って流れの解析をする方法のことです．☞ 数値流体力学（Wikipedia）

この章からしばらくの間は，上記の書籍に紹介されている計算方法のうち最も簡単な方法で，数値流体解析を行うプログラムを作ることにします．具体的には，次のような手法を使います．

- 等間隔直交格子を使う．
- SMAC 法に基づく有限差分法を採用する．
- 空間の離散化は，対流項，粘性項ともに 2 次中心差分．
- 時間の離散化（時間進行の計算）は，1 次の陽的 Euler 法
- 変数配置はスタガード変数配置とする．
- 線型方程式の解法には LU 分解を用いる．

✎ CFD に関連する専門用語がいくつか出てきました．「直交格子」「有限差分法」「SMAC 法」「離散化」「対流項」「粘性項」「2 次中心差分」「陽的 Euler 法」「スタガード配置」など．この文書の中では，これらの用語を説明していません．上記の教科書で勉強してく

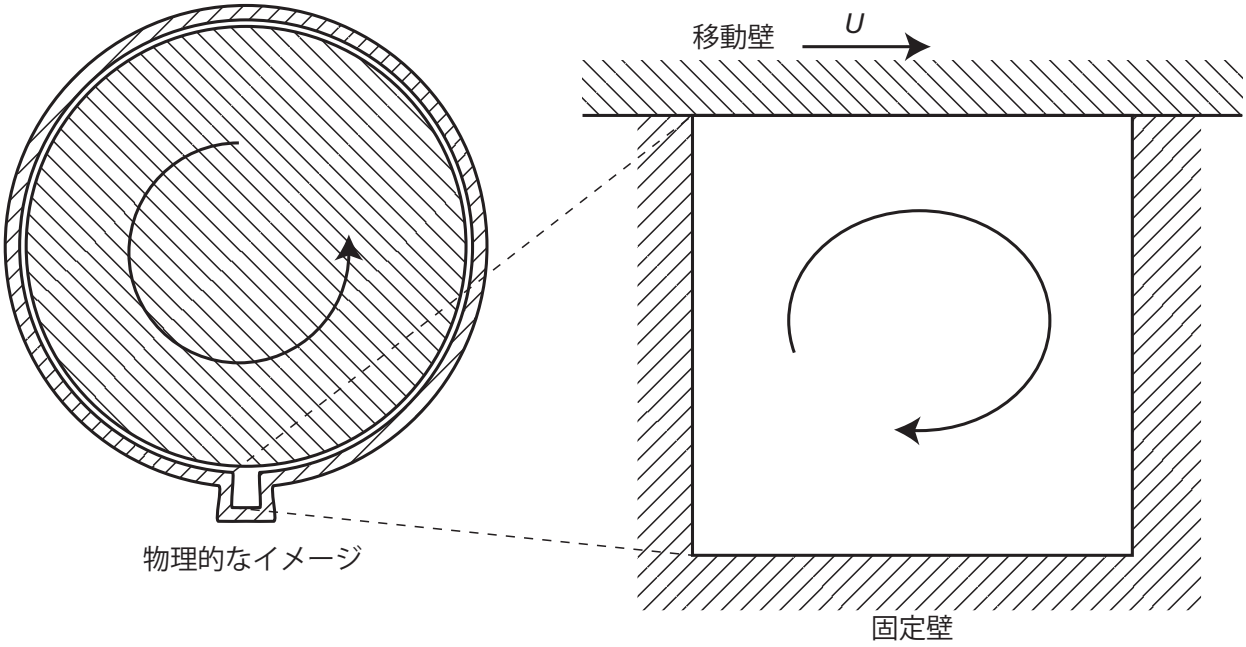

図 2.1　2 次元キャビティ流れ

ださい.

⚠「LU 分解」は，実用で用いるには遅すぎる解析手法です．ここではあくまでサンプルとして取り上げますが，後により効率の良い方法に置き換えます.

2.2　計算対象

計算する対象は，2 次元キャビティ流れとします.

2 次元キャビティ流れは，次のような特徴から CFD プログラムの最初の計算対象に適しています.

- 計算領域が真四角なので，プログラムの中で計算領域を表すのがかんたん.
- まわりが全部壁なので，流体の流入や流出を考えなくてよい.
- 先人たちが計算した結果が公開されているので，それと比較することで，自分のプログラムを検証することができる．Ghia らの論文 [2] が特に有名.

2.3　プログラムの設計

計算には SMAC 法を使うと決めたので，用意する変数は以下のようになります.

表す物理量	本書での変数名	教科書での変数名
速度	u, v	$\boldsymbol{u}^n$
圧力	p	P^n
速度の予測値	up, vp	$\boldsymbol{u}^P$
圧力の修正量	phi	ϕ
時間刻み	dt	Δt

表 2.1 本章で用いる変数名と，教科書 [1] 内での変数名との対応

1. 計算領域の大きさを表す変数 Lx，Ly
2. 計算領域の分割数 Nx，Ny
3. 各微小部分（これ以降では「セル」と呼びます.）の縦横の大きさ dx，dy．今回は等間隔格子を想定しているので，上記の変数とは dx = Lx/Nx，dy = Ly/Ny の関係があります.
4. 天井の横方向速度 U
5. Reynolds 数 Re．流体の動粘性係数を ν とすると，$\mathrm{Re} = \dfrac{\mathrm{U}\cdot\mathrm{Lx}}{\nu}$ と表される.
6. 各セルの左右辺の，横方向の流速 u．2 次元の計算なので 2 次元配列にする.
   ⚠ 2 次元計算だから 2 次元の配列を使うというのは，一見分かりやすそうですが，良い方法とは言えません．後に改善します
7. 各セルの上下辺の，縦方向の流速 v．u と同じく 2 次元配列にする.
8. 各セル中心の圧力 p．u，v と同じく，2 次元配列にする.（実際には圧力そのものではなく，動圧力 p/ρ に対応.）
9. 流速 u の予測値 up．SMAC 法を用いるので，はじめに速度の予測値を計算し，後に修正する．u 同じく 2 次元配列で，個数も u と同じ.
10. 流速 v の予測値 vp
11. 圧力 p の修正量 phi.
12. 時間刻み dt.
13. 圧力の修正量 phi に関する Poisson 方程式の，係数行列 M
14. Poisson 方程式の右辺ベクトルを格納する配列 b

これらの変数と，教科書 [1] 中で使われている変数名との対応を表 2.1 に示しています.

2.4 配列の大きさ・境界条件の扱い方

変数 u と up，v と vp，p と phi は，それぞれ同じ要素数の配列になります．では，具体的な要素数はいくつになるでしょうか.

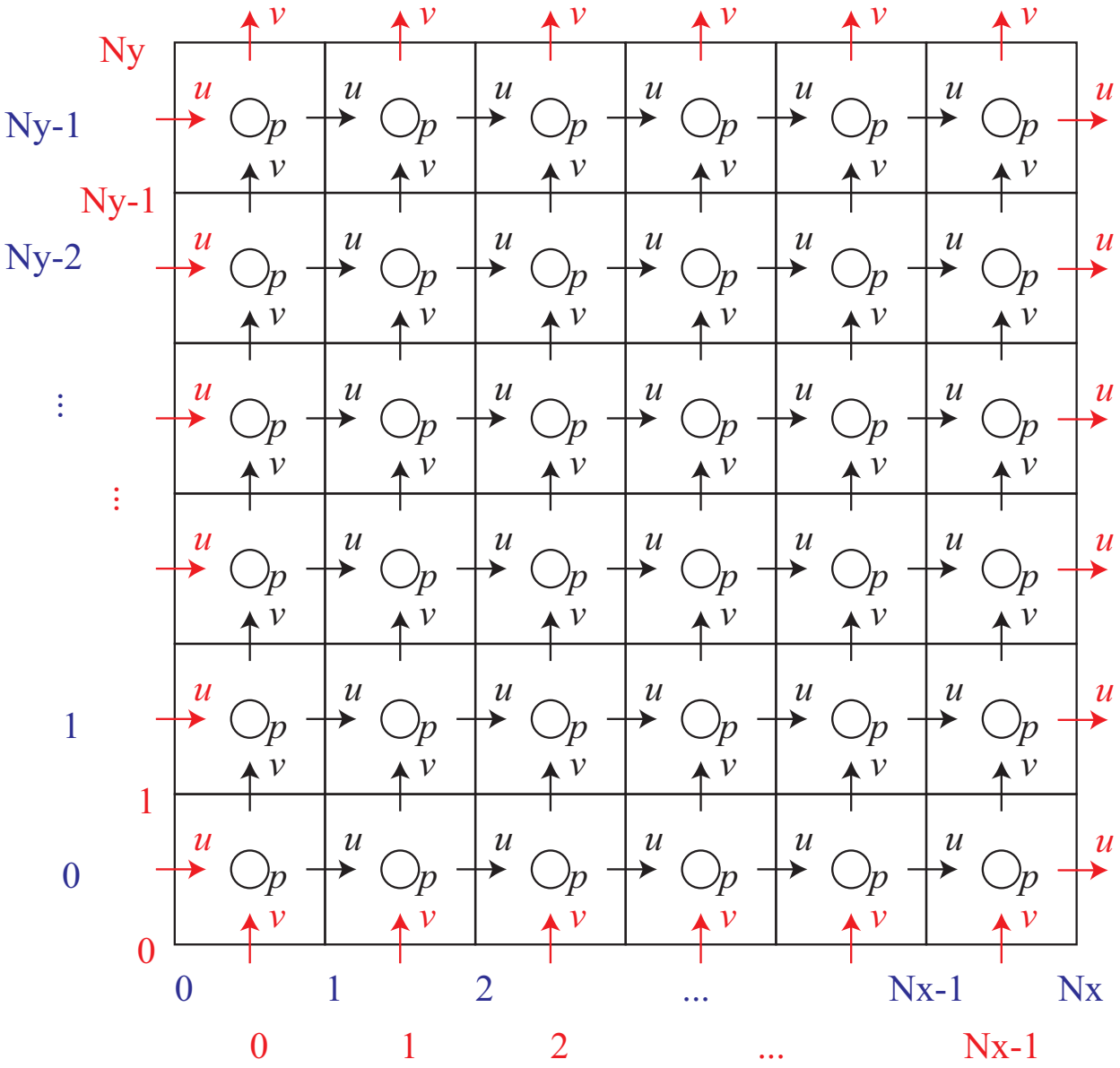

図 2.2　スタガード変数配置

図 2.2 を見てください．スタガード変数配置では，u，v，p は置かれている位置が違うので，変数の個数にも差があります．境界条件まで含めると，次のようになります．

- u: (Nx + 1) × Ny
- v: Nx × (Ny + 1)
- p: Nx × Ny

⚠ 境界条件のところまで変数を用意するのは，メモリの無駄です．ここではプログラムの作成を簡単にするために，境界条件まで変数として持つことにしますが，後に改善し

ます．

なおC言語およびそれから派生した言語では，配列の番号（インデックス）が0から始まります．例えば要素数Nの配列a[N]では，a[0]，a[1]，…，a[N-1]の要素がある，ということです．図2.2ではこのルールに基づいて番号を振ってあります．

係数行列Mは，圧力の修正量phiを計算するための線型方程式の係数行列ですから，（phiの数）行×（phiの数）列の正方行列です．ですから要素数は(Nx·Ny)×(Nx·Ny)です．また右辺ベクトルの大きさはNx·Nyです．

⚠ 実際には，係数行列Mのほとんどの部分は0です．圧力の修正量の方程式は，各セルごとの方程式を，接しているセルとの関係から導きます．2次元直交格子なら接しているセルは4つなので，係数行列は1行あたり5箇所（自分自身と，接している要素のぶん）しか係数が入らず，残りは全て0の，スカスカの行列です．このような行列を**疎行列**（sparse matrix）と呼びます．
2次元であれば，0でない要素は5×(Nx·Ny)しかありません．それなのに(Nx·Ny)×(Nx·Ny)も要素を用意するのは大きな無駄です．ここは後に改善します．

2.5 プログラムの大枠を書く

ここまで決めてきた変数を含むプログラムを，実際に書いてみましょう．

ソースコード2.1 プログラムの全体像

```
1  #include <iostream>
2
3  // 計算空間のサイズと天井の速度
4  double const Lx = 1.;
5  double const Ly = 1.;
6  double const U = 1.;
7
8  // Reynolds 数
9  double const Re = 1.;
10
11 // 計算空間の分割数
12 int const Nx = 16;
13 int const Ny = 16;
14 int const Ncells = Nx*Ny;          // number of cells
```

```
15
16 // 流れの物理量
17 double u[Nx+1][Ny];
18 double v[Nx][Ny+1];
19 double p[Nx][Ny];
20
21 // SMAC法の予測値
22 double up[Nx+1][Ny];
23 double vp[Nx][Ny+1];
24
25 // 圧力の修正量
26 double phi[Nx][Ny];
27
28 // 解析時間と時間刻み
29 double t_end = 10.;
30 double dt = 0.0001;
31
32 // 係数行列
33 double M[Ncells][Ncells];
34
35 // 右辺ベクトルを格納する配列. 線型方程式を解き終わった時には, 根が格
36 // 納される.
37 double b[Ncells];
38
39 int main()
40 {
41   // 各セルの大きさ
42   double dx = Lx / Nx;
43   double dy = Ly / Ny;
44
45   // 動粘性係数をReから計算
46   double nu = U*Lx/Re;
47
48   double t = dt;
```

```
49
50    // 時間進行
51    while（t <= t_end）{
52      // ステップ1: 流速の予測値の計算
53      // ステップ2: 圧力の修正量を計算
54      // ステップ3: 圧力と流速の修正
55      // 計算結果の出力
56      t = t + dt;
57    }
58
59    return 0;
60  }
```

ご覧のとおり，変数の宣言がしてあるだけで，計算の中身をまだ全然書いていません．次の章から，中身を作っていきますが，まずはここまでのところを説明します．

- 定数は double const または int const と宣言しています．プログラムの中で，間違って Lx や U に数値を代入しても，定数として宣言してあればコンパイラが見つけて，エラーを出してくれます．**人間は必ずミスをしますので，ミスを見つけやすくすることはとても重要です．**
  - ⚠ そう言っておきながら，上のプログラムでは C++ の配列を多用しています．C 言語系の配列は要素数の範囲チェックをしません．だから例えば要素数が 20 個の配列の 25 番目の要素の値を読んだり，値を代入したりしても，一見，エラーにならずに動いてしまうことがあります．これはミスを見つけやすいとは言えません．ここは後に改善します
- 計算は t = 0 からではなく，t = dt からはじめています．初期状態が t = 0 で，最初のステップでは t = dt の状態を計算するからです．

第 3 章

差分のプログラミング・2 次中心差分

この章では，SMAC 法の予測段階のプログラムを作ります．つまり，現在の流れ場の状態を表す変数 u，v，p を使って，流速の予測値 up，vp を計算する，ということです．

SMAC 法の予測段階では，対流項，圧力項，粘性項を計算しますので，これらを計算する方法を説明していきます．運動量保存則の式は，教科書 [1] の式 (1.46) にあるように，

$$\frac{\partial u_i}{\partial t} + \frac{\partial (u_i u_j)}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j} + f_i$$

と表されます．これを添え字を使わないで書くと，2 次元の場合は，

$$\frac{\partial u}{\partial t} + \frac{\partial (uu)}{\partial x} + \frac{\partial (uv)}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu \left(\frac{\partial^2 u}{\partial x \partial x} + \frac{\partial^2 u}{\partial y \partial y} \right) + f_x \tag{3.1}$$

$$\frac{\partial v}{\partial t} + \frac{\partial (vu)}{\partial x} + \frac{\partial (vv)}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial y} + \nu \left(\frac{\partial^2 v}{\partial x \partial x} + \frac{\partial^2 v}{\partial y \partial y} \right) + f_y \tag{3.2}$$

と書くことができます．この式に基づいて離散化していきますが，外力項 f_x と f_y は今回の計算では出てきません．

✎ 教科書 [1] の式 (1.46) には “Einstein の総和規則” が使われています．同じ添字が 2 回出てきたら，それはその添字の範囲で繰り替えし使って足し合わせることを表す，という規則のことです．
例えば $\frac{\partial (u_i u_j)}{\partial x_j}$ では，j が 2 回使われています．2 次元なので $j = 1, 2$ です．だから $j = 1$ のときの式と，$j = 2$ のときの式を足しあわせます．

$$\frac{\partial (u_i u_j)}{\partial x_j} = \frac{\partial (u_1 u_1)}{x_1} + \frac{\partial (u_1 u_2)}{x_2}$$

ここでは $u_1 = u$，$u_2 = v$，$x_1 = x$，$x_2 = y$ を表しています．粘性項でも同じ規則が使われています．

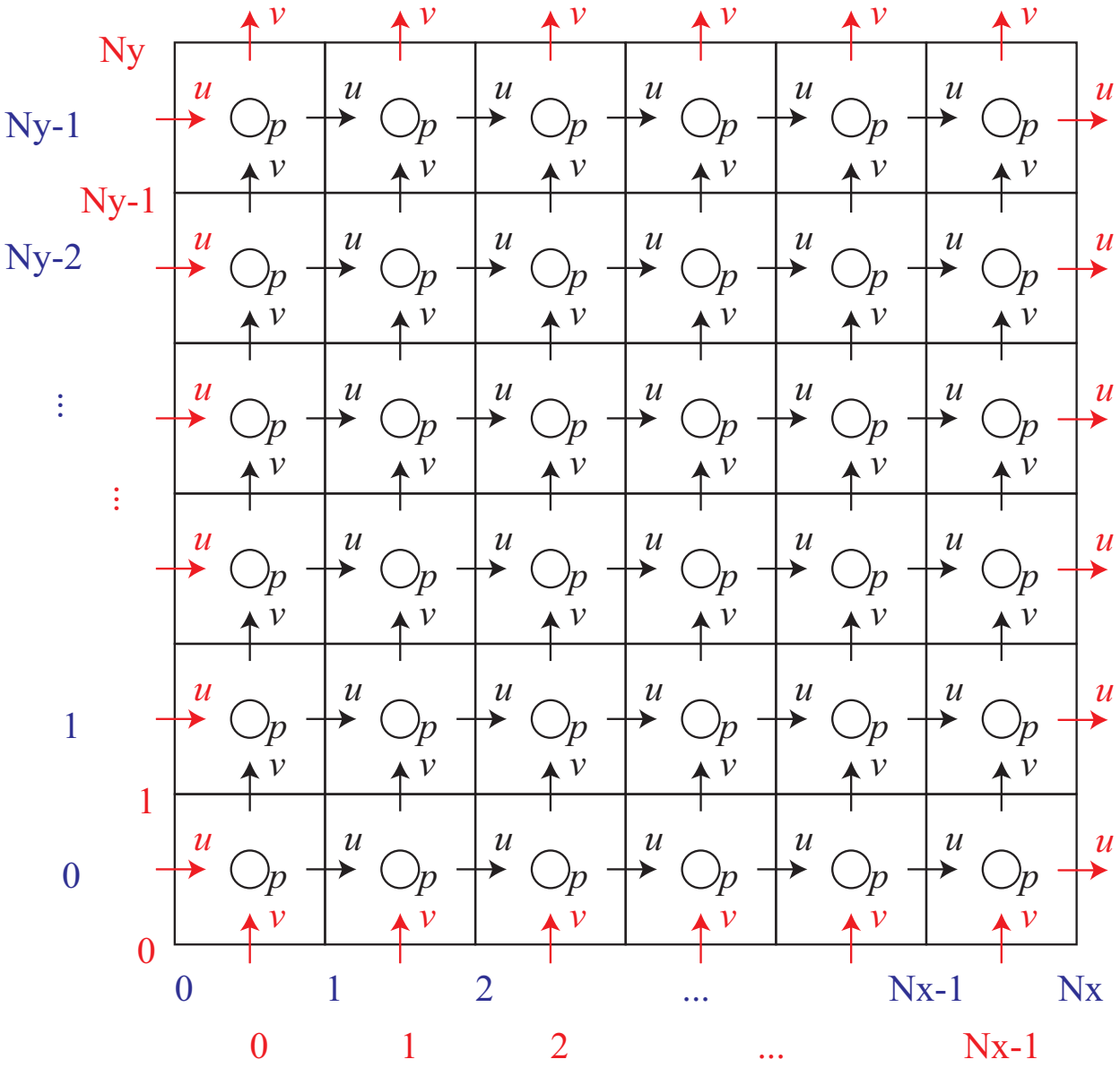

図 3.1　再掲・スタガード変数配置．青字が u の番号を，赤字が v の番号を表している．

3.1　繰り返し計算の範囲

図 3.1 を見てください．図 2.2 を再度掲載したものです．赤い矢印は境界での速度（境界条件）を表しています．壁面なので実際にはゼロですが，図 3.1 では分かりやすくするために長さを持った矢印で表しました．

この図から分かるように，up と vp では，計算する範囲が異なります．横方向のループのインデックスを i，縦方向のループのインデックスを j と書くと，

- up: $1 \leqq \mathtt{i} \leqq \mathtt{Nx}-1$，$0 \leqq \mathtt{j} \leqq \mathtt{Ny}-1$
- vp: $0 \leqq \mathtt{i} \leqq \mathtt{Nx}-1$，$1 \leqq \mathtt{j} \leqq \mathtt{Ny}-1$

したがってこれを C++ の for 文で書くと，次のようなプログラムになるはずです．

ソースコード 3.1　予測計算のループ

```
for (int i = 1; i <= Nx - 1; ++i)
  for (int j = 0; j <= Ny - 1; ++j) {
    // upの計算
  }

for (int i = 0; i <= Nx - 1; ++i)
  for (int j = 1; j <= Ny - 1; ++j) {
    // vpの計算
  }
```

3.2 u の対流項，圧力勾配項，粘性項

up と vp では，変数が配置されている場所が違うので，検査空間の取り方も異なり，対流項と粘性項を計算するために使う変数の範囲（ステンシル）も異なります．図 3.2 と図 3.3 に，それぞれ u と v の検査空間とステンシルを示しています．赤い破線で囲った黄色い四角が検査空間を表し，青い破線で囲った範囲の変数がステンシルを表しています．

はじめに図 3.2 を使って，u の対流項，粘性項，圧力勾配項をすべて 2 次中心差分で定式化します．検査空間とステンシルの図を描いて考えれば，さほど難しくありません．

x 方向の対流項，圧力勾配項，粘性項は，式 (3.1) に示したように，次のように表されます．

- 対流項: $\frac{\partial(uu)}{\partial x} + \frac{\partial(uv)}{\partial y}$
- 圧力勾配項: $-\frac{1}{\rho}\frac{\partial p}{\partial x}$
- 粘性項: $\nu\left(\frac{\partial^2 u}{\partial x \partial x} + \frac{\partial^2 u}{\partial y \partial y}\right)$

uu は単純に $u \cdot u$ のことで，uv は $u \cdot v$ です．

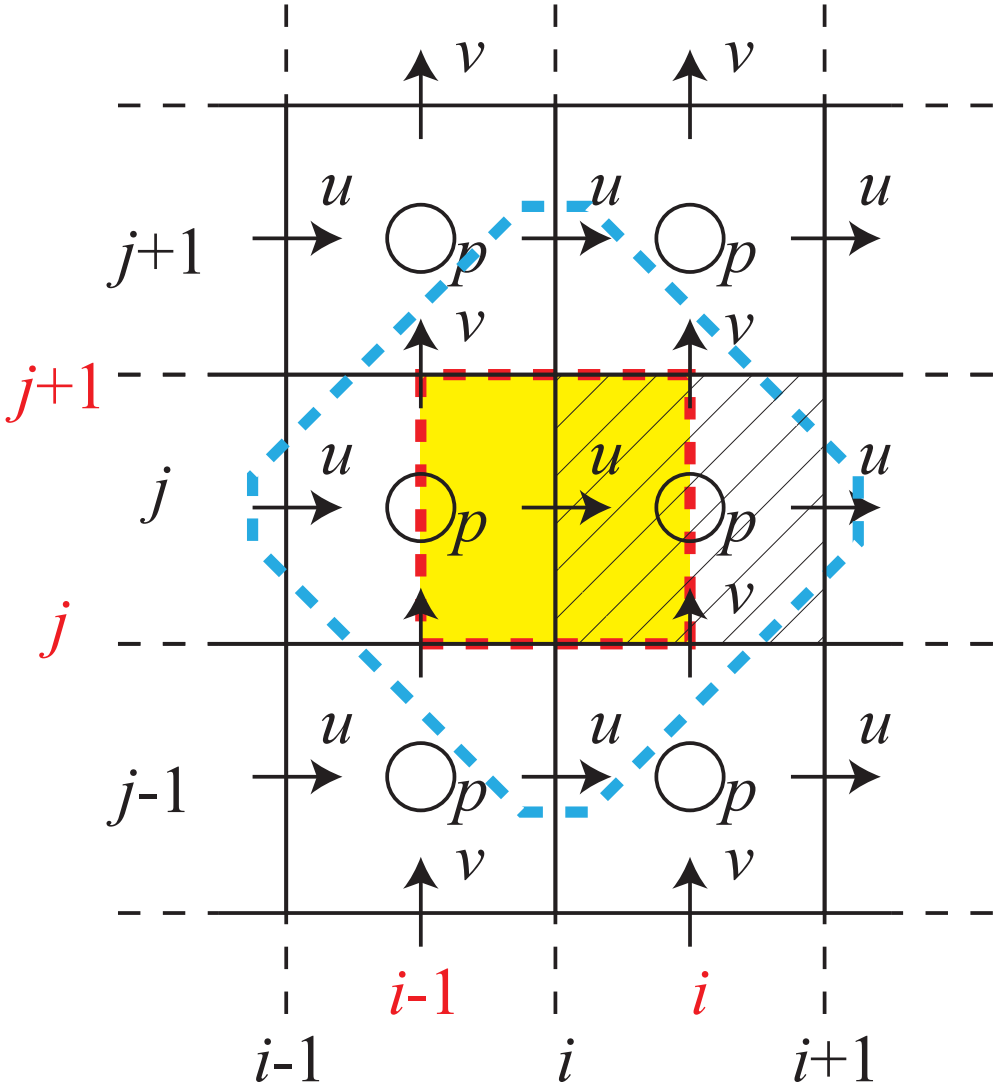

図 3.2　u の検査空間とステンシル（赤い破線で囲った黄色い四角が検査空間，青い破線で囲った範囲の変数がステンシル）．ハッチングしてあるセルは (i, j) 番目のセルで，そのセルの左の u が u[i][j]，下の v が v[i][j]，中心の p が p[i][j] である．

3.2.1　対流項の計算

対流項 $\frac{\partial(uu)}{\partial x}$ から行きます．この項は x に関する偏微分なので，検査空間の x 方向に対する uu の傾き（勾配）を計算します．最も単純な計算方法は，

$$\frac{\partial(uu)}{\partial x} \approx \frac{(\text{検査空間の右側の } uu) - (\text{検査空間の左側の } uu)}{\Delta x}$$

と表されます．

通常の等号 = ではなく波うった等号 ≈ を使ったのは，この式が近似式だということを表し

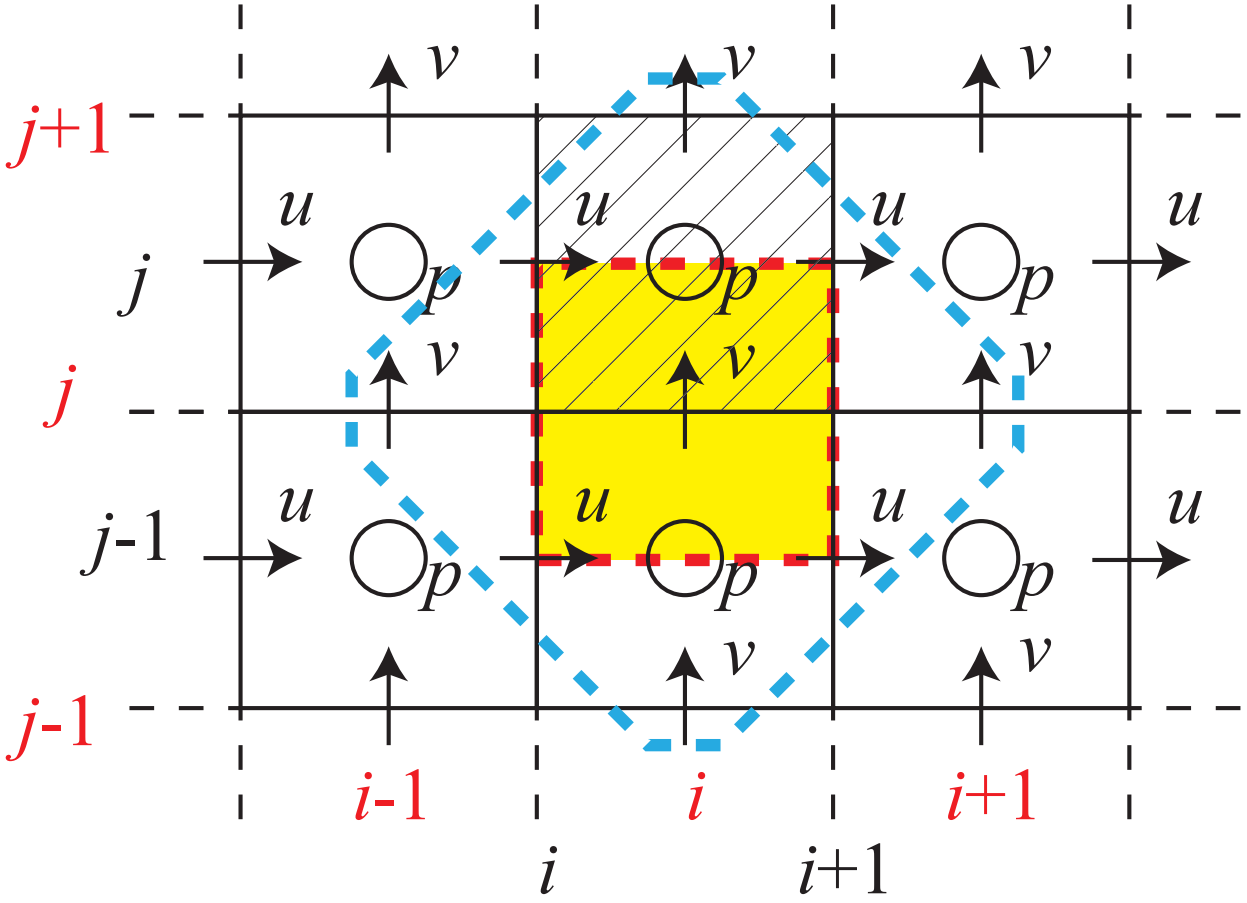

図 3.3 v の検査空間とステンシル（赤い破線で囲った黄色い四角が検査空間，青い破線で囲った範囲の変数がステンシル）．ハッチングしてあるセルは (i, j) 番目のセル

ています．微分を差分商で近似しているので，この近似方法を**差分近似**と呼びます．

この計算をするために，まず検査空間左右での u を求めます．図 3.4 にしたがって変数名を決めることにして，検査空間の左側での u を uw，右側での u を ue とします．これを最も単純に計算するには，検査面（検査空間の表面）を挟んで左右の値を平均すれば良いわけです．

```
double uw = (u[i-1][j]+u[i][j])/2.;
double ue = (u[i][j]+u[i][j+1])/2.;
```

ここで求めた uw と ue を用いて，対流項の最初の項を，

$$\frac{\partial(uu)}{\partial x} \approx \frac{(\mathtt{ue}*\mathtt{ue})-(\mathtt{uw}*\mathtt{uw})}{\Delta x}$$

のように計算できます．プログラムのコードでは下記のようになります．

```
double uux = ((ue*ue)-(uw*uw))/dx;
```

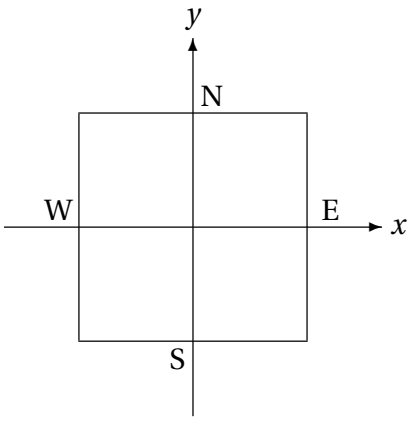

図 3.4　検査空間周囲の変数の命名方法（東西南北）

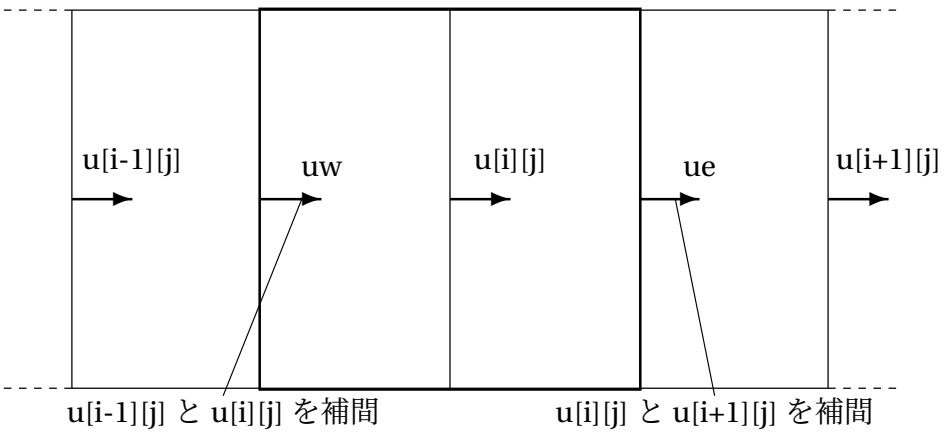

図 3.5　u の対流項の計算方法

✐ これは瑣末なことですが，コンピュータの内部では掛け算のほうが割り算よりも速く計算できます．ですので 2 で割るより 0.5 をかけるほうが少しだけよいです．

```
double uw = 0.5*(u[i-1][j]+u[i][j]);
double ue = 0.5*(u[i][j]+u[i][j+1]);
```

次に対流項の $\frac{\partial(uv)}{\partial y}$ を差分近似しましょう．この項は y に関する偏微分なので，検査空間の y 方向に対する uv の勾配を計算します．最も単純な計算方法は，

$$\frac{\partial(uv)}{\partial y} \approx \frac{(\text{検査空間の上面の } uv) - (\text{検査空間の下面の } uv)}{\Delta y}$$

と表されます．

境界条件のことを考えなければ，この計算は難しくありません．図 3.2 をじーっと見て，次のように計算します．

```
// 境界条件のことを考えていないコード
double us = 0.5*(u[i][j-1] + u[i][j]);
double vs = 0.5*(v[i-1][j] + v[i][j]);
double un = 0.5*(u[i][j] + u[i][j+1]);
double vn = 0.5*(v[i-1][j+1] + v[i][j+1]);
double uvy = (un*vn - us*vs)/dy;
```

ただし，ここでは境界条件のことを考える必要があります．もし j = 0 だったら，検査空間の下は固定壁ですから，us = vs = 0 です．またもし j = Ny − 1 だったら，検査空間の上は移動壁ですから，un = U，vn = 0 です．これをプログラムに反映させる必要があります．

```
// 境界条件のことを考えたコード
double us, vs, un, vn;
if (0 == j) {
  us = vs = 0.;
} else {
  us = 0.5*(u[i][j-1] + u[i][j]);
  vs = 0.5*(v[i-1][j] + v[i][j]);
}
if (Ny - 1 == j) {
  un = U;
  vn = 0.;
} else {
  un = 0.5*(u[i][j] + u[i][j+1]);
  vn = 0.5*(v[i-1][j+1] + v[i][j+1]);
}
double uvy = (un*vn - us*vs)/dy;
```

⚠ 配列の添字の間違いを起こしやすいので，図 3.2 をよく見てプログラムを作りましょう．図 3.2 では黒い添字が u のインデックスを，赤い添字が v のインデックスを表しています．

✎ 上記のコードの中で，“if (j == 0)” ではなく “if (0 == j)” と書いています．これも間違い

をなるべく避けるための書き方です．C++では，==と書くべきところを間違えて=と書いても，コンパイル時にはエラーにならない場合があります．“if (j=0)”と書くと，jに0が代入されて，計算が進みます．ただし本当はjがゼロか否かを調べたいのにゼロを代入してしまっているわけですから，計算結果はおかしくなります．
しかし“if (0=j)”と間違えた場合には，コンパイラがエラーを出してくれます．jに0を代入することはできても，定数にjを代入することはできないからです．だからこの書き方をするように心がけておけば，間違いを早期に発見して修正することができます．

3.2.2　圧力勾配項の計算

次は圧力勾配項を計算します．図3.2を見ると分かりますが，圧力勾配の計算は簡単です．検査空間の左右に，圧力の変数があるからです．

```
double dpdx = (p[i][j] - p[i-1][j])/dx;
```

⚠ ここでも配列の添字の間違いを起こしやすいので，図3.2をよく見てプログラムを作りましょう．

✎ このプログラムの中ではpは動圧力 p/ρ を表しているので，改めて流体密度で割る必要はありません．

✎ 配列変数を参照したり，代入したりする場合は，できるだけ配列に格納されている順番にアクセスするほうが，計算が速い可能性があります．これは教科書[1]にも記されています．
上のプログラムでは“(p[i][j]-p[i-1][j])/dx”と書きましたが，“(-p[i-1][j]+p[i][j])/dx”と書いたほうがメモリアクセスが速い可能性がある，ということです．付録Aで紹介しているプログラムでは，メモリアクセスが速くなるような書き方を一貫して採用します．

3.2.3　粘性項の計算

粘性項 $\nu\left(\frac{\partial^2 u}{\partial x \partial x}+\frac{\partial^2 v}{\partial y \partial y}\right)$ は2階微分なので，計算を2段階に分けて考えます．まず $\frac{\partial^2 u}{\partial x \partial x}$ から考えますが，はじめに検査空間の左右（図3.6中に∘で表した位置）でuの微分値を差分近似し，それをさらに差分近似します．

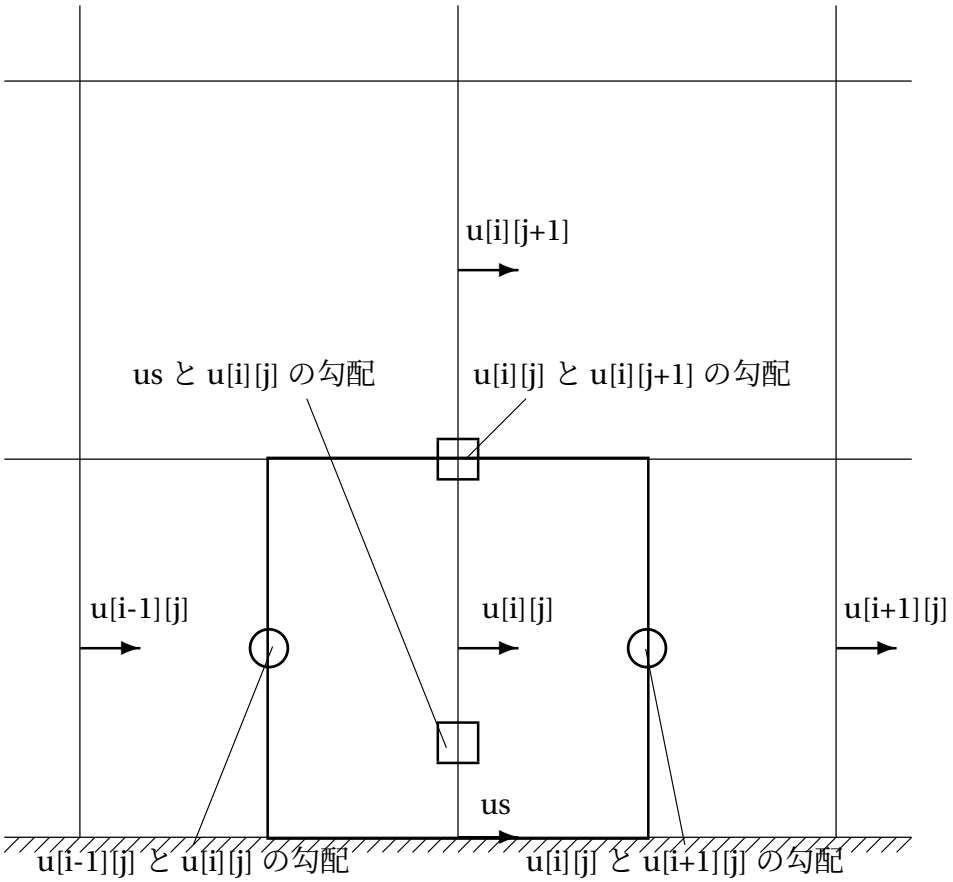

図 3.6 u の粘性項の計算方法（下面が境界で速度が与えられている場合）

$$
\begin{aligned}
\frac{\partial^2 u}{\partial x \partial x} &= \frac{\partial}{\partial x}\frac{\partial u}{\partial x} \\
&\approx \frac{-\left(\text{検査空間の左での } \frac{\partial u}{\partial x}\right) + \left(\text{検査空間の右での } \frac{\partial u}{\partial x}\right)}{\Delta x} \\
&\approx \frac{1}{\Delta x}\left\{-\frac{-u[i-1][j]+u[i][j]}{\Delta x} + \frac{-u[i][j]+u[i+1][j]}{\Delta x}\right\} \\
&= \frac{u[i-1][j]-2u[i][j]+u[i+1][j]}{\Delta x^2}
\end{aligned}
$$

次に $\frac{\partial^2 u}{\partial y \partial y}$ ですが，こちらは境界条件のことを考える必要があります．

境界に接していない検査空間の場合は，$\frac{\partial^2 u}{\partial x \partial x}$ のときと同じように考えることができます．y での 2 階偏微分なので，検査空間の上下で u の微分値を差分近似し，それをさらに差分近似します．

$$
\begin{aligned}
\frac{\partial^2 u}{\partial y \partial y} &= \frac{\partial}{\partial y}\frac{\partial u}{\partial y} \\
&\approx \frac{-\left(\text{検査空間の下での } \frac{\partial u}{\partial y}\right) + \left(\text{検査空間の上での } \frac{\partial u}{\partial y}\right)}{\Delta y} \\
&\approx \frac{1}{\Delta y}\left\{-\frac{-u[i][j-1]+u[i][j]}{\Delta y} + \frac{-u[i][j]+u[i][j+1]}{\Delta y}\right\} \\
&= \frac{u[i][j-1]-2u[i][j]+u[i][j+1]}{\Delta y^2}
\end{aligned}
$$

ただし u の検査空間の場合は，上下のいずれかの面が境界になっている可能性があります．今回の問題では境界の速度が与えられているので，その条件下で偏微分の差分近似をします．

検査空間の下面が境界だった場合，つまり j = 0 の場合は，図 3.6 中に□で表したように，us と u[i][j] との間で勾配を計算し，それと検査空間上面での u の勾配を使って，2 階偏微分を差分近似します．したがって差分近似式は以下のようになります．

$$
\begin{aligned}
\frac{\partial^2 u}{\partial y \partial y} &= \frac{\partial}{\partial y}\frac{\partial u}{\partial y} \\
&\approx \frac{-\left(\text{検査空間の下から } \Delta y/4 \text{ の位置での } \frac{\partial u}{\partial y}\right) + \left(\text{検査空間の上での } \frac{\partial u}{\partial y}\right)}{\frac{3}{4}\Delta y} \\
&\approx \frac{1}{\frac{3}{4}\Delta y}\left\{-\frac{-u_s+u[i][j]}{\frac{1}{2}\Delta y} + \frac{-u[i][j]+u[i][j+1]}{\Delta y}\right\} \\
&= \frac{8u_s-12u[i][j]+4u[i][j+1]}{3\Delta y^2}
\end{aligned}
$$

検査空間の上面が境界の場合には，同様に計算して $\frac{4u[i][j-1]-12u[i][j]+8u_n}{3\Delta y^2}$ という式が得られます．u_n が境界での速度です．

✐ 3 点での u が与えられていると考えて，u を放物線近似して，その近似式を 2 階微分する方法でも $\frac{\partial^2 u}{\partial y \partial y}$ を求めることができます．結果は上記の式と一致します．

以上をまとめると，u の空間 2 階微分を計算するプログラムは下記のようになります．

ソースコード 3.2　u の空間 2 階微分の計算コード

```
double uxx = (u[i-1][j] - 2*u[i][j] + u[i+1][j])/(dx*dx);
double uyy;
if (0 == j) {
  double us = 0.;
  uyy = (8./3.*us - (8./3.+4./3.)*u[i][j] + 4./3.*u[i][j+1])/(dy*dy);
```

```
} else if (Ny - 1 == j) {
  double un = U;
  uyy = (4./3.*u[i][j-1] - (4./3.+8./3.)*u[i][j] + 8./3.*un)/(dy*dy);
} else
  uyy = (u[i][j-1] - 2*u[i][j] + u[i][j+1])/(dy*dy);
```

✎ u[i][j] の係数は −12/3 = −4 になるのですが，−4 と書かずに −(8./3. + 4./3.) と書いています．こうしているのは，係数の合計をぴったり 0 にするためです．微分の近似式では，係数の合計は 0 になるはずですが，係数をプログラム中にそれぞれ 8/3，−4，4/3 と書いた場合には，数値誤差からその合計が 0 からわずかにずれた値となる可能性があります．そうすると，粘性項に非物理的な増加や減少が入ってきます．これを避けるために，絶対値の小さい 2 つの係数を 8./3.，4./3. と書いて，絶対値のいちばん大きい u[i][j] の係数を −(8./3. + 4./3.) と書いています．これは教科書 [1] でも説明されています．

3.2.4 u の予測値の計算

これまで計算してきた u の対流項，圧力勾配項，粘性項を合わせて，u の予測値 up を計算します．ここでは 1 次 Euler 陽解法を使っています．

```
up[i][j] = u[i][j] + dt*(-(uux+uvy) -dpdx + nu*(uxx+uyy));
```

3.3 v の対流項，圧力勾配項，粘性項

v の対流項，圧力勾配項，粘性項の計算も，考え方は u の場合と同じです．ただし検査空間とステンシルは，u とは異なるので注意が必要です．

ここではプログラムのみ示します．

ソースコード 3.3 v の予測値 vp の計算コード

```
// vの予測値vpの計算
for (int i = 0; i <= Nx - 1; ++i)
  for (int j = 1; j <= Ny - 1; ++j) {
    // 対流項 x
    double uw, vw, ue, ve;
    if (0 == i) {
      uw = vw = 0;
```

```
} else {
  uw = 0.5*(u[i][j-1] + u[i][j]);
  vw = 0.5*(v[i-1][j] + v[i][j]);
}

if (Nx - 1 == i) {
  ue = ve = 0;
} else {
  ue = 0.5*(u[i+1][j-1] + u[i+1][j]);
  ve = 0.5*(v[i][j] + v[i+1][j]);
}
double vux = (-(vw*uw)+(ve*ue))/dx;

// 対流項 y
double vs = 0.5*(v[i][j-1] + v[i][j]);
double vn = 0.5*(v[i][j] + v[i][j+1]);

double vvy = (-vs*vs + vn*vn)/dy;

// 圧力勾配項 y
double dpdy = (-p[i][j-1] + p[i][j])/dy;

// 粘性項 y
double vxx;
if (0 == i) {
  double vw = 0.;
  vxx = (8./3.*vw - (8./3.+4./3)*v[i][j] + 4./3.*v[i+1][j])/(dx*
      dx);
} else if (Nx - 1 == i) {
  double ve = 0;
  vxx = (4./3.*v[i-1][j] - (8./3.+4./3)*v[i][j] + 8./3.*ve)/(dx*
      dx);
} else
  vxx = (v[i-1][j] - 2*v[i][j] + v[i+1][j])/(dx*dx);
```

```
        double vyy = (v[i][j-1] - 2*v[i][j] + v[i][j+1])/(dy*dy);

        vp[i][j] = v[i][j] + dt*(-(vux+vvy) -dpdy + nu*(vxx+vyy));
    }
```

ここまでで SMAC 法の予測段階のプログラムが出来ました．次は圧力の修正量 ϕ の計算をする Poisson 方程式の解法を考えますが，ここで章を改めます．

✎ ここまで u，v，p を使って up，vp を計算する方法を説明してきました．しかし u，v，p の計算はまだで，この後の章で説明することになります．では計算がはじまった最初の段階では，u，v，p にはどんな値が入っているのでしょうか?
C++ では，double 変数は 0 で初期化されることになっています．なので特に指定しなければ，u，v，p には 0 が入っています．このため今回説明したプログラムには，u，v，p を初期化する（最初の値を入れる）コードが入っていません．
しかし，0 に初期化するのが常に正しいとは限りません．この点は後に説明します．

第 4 章

Poisson 方程式の構築とソルバー

この章では，SMAC 法で圧力の修正量 ϕ を求めるための Poisson 方程式を組み立て，それを解くためのプログラムを説明していきます．

4.1 Poisson 方程式の係数行列を作る

Poisson 方程式を「組み立てる」とはどういうことでしょうか? 今回解こうとしている **Poisson 方程式**は，下記のような式で表されます．（教科書 [1] の式）

$$\nabla^2 \phi = \frac{1}{\Delta t} \nabla \cdot \boldsymbol{u}^P$$

右辺の計算方法は後で述べます．ここでは左辺に着目します．直交格子を使う場合は，$\nabla^2 \phi$ は次のように計算されます．

$$\begin{aligned} \nabla^2 \phi &= \frac{\partial^2}{\partial x^2} \phi + \frac{\partial^2}{\partial y^2} \phi \\ &= \frac{\partial}{\partial x} \left(\frac{\partial}{\partial x} \phi \right) + \frac{\partial}{\partial y} \left(\frac{\partial}{\partial y} \phi \right) \end{aligned}$$

u や v の粘性項と同じような式が出てきました．これをこれから差分近似します．圧力修正量は圧力と同じくセル中心に配されているので，検査空間とステンシルの取り方が u や v とは違うことには気をつけなければいけませんが，基本的な考え方は粘性項と同じです．

図 4.1 をよく見ると，次のように差分近似できます．

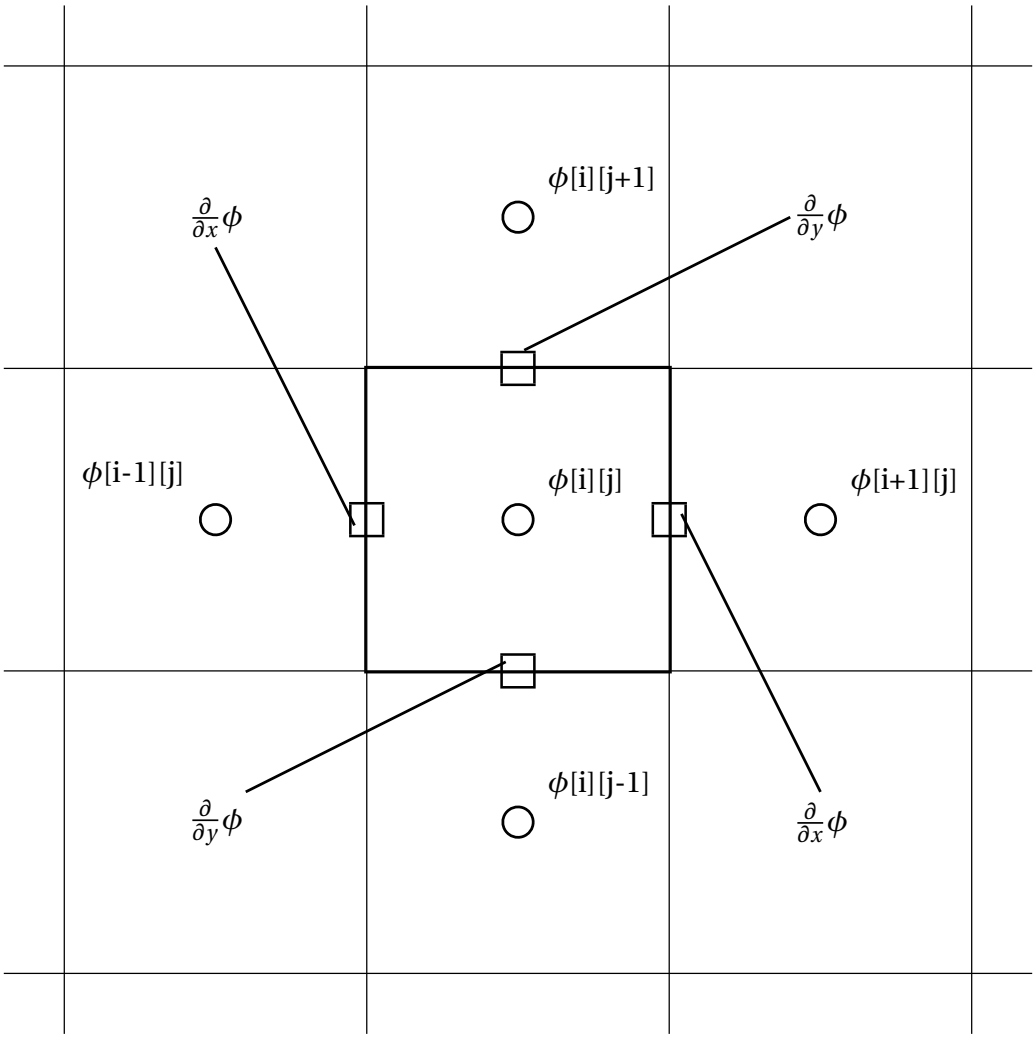

図 4.1　圧力修正量 ϕ の検査空間とステンシル（係数行列計算用）

$$\nabla^2\phi = \frac{\partial}{\partial x}\left(\frac{\partial}{\partial x}\phi\right) + \frac{\partial}{\partial y}\left(\frac{\partial}{\partial y}\phi\right)$$

$$\approx \frac{-\frac{-\mathtt{phi}[i-1][j]+\mathtt{phi}[i][j]}{\Delta x} + \frac{-\mathtt{phi}[i][j]+\mathtt{phi}[i+1][j]}{\Delta x}}{\Delta x} + \frac{-\frac{-\mathtt{phi}[i][j-1]+\mathtt{phi}[i][j]}{\Delta y} + \frac{-\mathtt{phi}[i][j]+\mathtt{phi}[i][j+1]}{\Delta y}}{\Delta y} \tag{4.1}$$

$$= \frac{1}{\Delta y^2}\mathtt{phi}[i][j-1] + \frac{1}{\Delta x^2}\mathtt{phi}[i-1][j] - \left(\frac{2}{\Delta x^2} + \frac{2}{\Delta y^2}\right)\mathtt{phi}[i][j]$$
$$+ \frac{1}{\Delta x^2}\mathtt{phi}[i+1][j] + \frac{1}{\Delta y^2}\mathtt{phi}[i][j+1] \tag{4.2}$$

式 (4.2) は phi に関する方程式の左辺になっています．つまり phi[i][j] と，phi[i][j-1]，phi[i-1][j]，phi[i+1][j]，phi[i][j+1] の 5 つの変数が未知数で，$\frac{1}{\Delta y^2}$ などがその係数です．

各セルごとに上記のような方程式が 1 本ずつ作られますので，合計で Nx × Ny 本の連立 1 次

方程式が出来上がります.

これを行列方程式の形で書くと，係数行列を M として,

$$M\boldsymbol{\phi} = \boldsymbol{b}$$

の形に書くことができます．Poisson 方程式を組み立てると書いたのは，つまりこの係数行列 M を作ることです.

この係数行列は Nx × Ny の行列 ではない ことに気をつけてください．連立方程式の本数が Nx × Ny なので，係数行列は (Nx·Ny) × (Nx·Ny) の正方行列になります.

これから係数行列 M を作っていきますが，その前に境界条件の扱いを決めておきましょう．今回の問題では壁面境界を想定していますが，壁面に垂直な方向の力は釣り合っているはずです．ですので壁面では圧力の勾配がないと仮定します．つまり例えば下面（南の境界）では,

$$\frac{\partial}{\partial y} p = 0$$

ということです.

初期状態で上記の式が成り立っているとして，計算の途中でも上記の式が成り立ち続ける（つまり圧力勾配がずっとゼロのままである）ためには，圧力の修正量 ϕ の勾配もゼロである必要があります．したがって同じく下面では,

$$\frac{\partial}{\partial y} \phi = 0$$

ということになりますので，これを ϕ の境界条件として採用します.

そうすると，式 (4.1) で第 2 項の分子の第 1 項が 0 になるということです.

$$\nabla^2\phi \approx \frac{-\frac{-\mathtt{phi}[i-1][j]+\mathtt{phi}[i][j]}{\Delta x} + \frac{-\mathtt{phi}[i][j]+\mathtt{phi}[i+1][j]}{\Delta x}}{\Delta x} + \frac{-0 + \frac{-\mathtt{phi}[i][j]+\mathtt{phi}[i][j+1]}{\Delta y}}{\Delta y} \tag{4.3}$$

$$= \frac{1}{\Delta y^2}\mathtt{phi}[i][j-1] - \left(\frac{2}{\Delta x^2} + \frac{1}{\Delta y^2}\right)\mathtt{phi}[i][j] + \frac{1}{\Delta x^2}\mathtt{phi}[i+1][j] + \frac{1}{\Delta y^2}\mathtt{phi}[i][j+1] \tag{4.4}$$

この式 (4.4) と元の式 (4.2) を比べると，元の式は phi[i][j-1] の項があってその係数が $\frac{1}{\Delta y^2}$ であり，また当該セルの圧力変化量 phi[i][j] の項の係数も，係数の絶対値で見ると $\frac{1}{\Delta y^2}$ だけ減っています．これを逆に考えると，検査空間の下面が境界 ではない ときは,

- phi[i][j-1] の項の係数を $\frac{1}{\Delta y^2}$ にする.
- phi[i][j] の項の係数から $\frac{1}{\Delta y^2}$ を引く.

という操作をすれば，下面に対応する係数が準備できることになります.

4.2　変数の番号付けと，係数行列の構築のプログラミング

上記の操作をプログラムに書きますが，線型方程式は $M\boldsymbol{\phi} = \boldsymbol{b}$ の形をしていて，圧力の変化量 phi（$\boldsymbol{\phi}$）と右辺ベクトル（$\boldsymbol{b}$）は 1 次元のベクトル量で表されます．元の phi は（今のところは）2 次元配列になっているので，これを 1 次元のベクトルとしても扱えるように，順番を決めておきます．

✎「だったら最初から全部 1 次元配列にすればいいじゃないか!?」と思いましたか? その通りです! 後の章でプログラムを改造します．

計算空間内で，x 方向（横方向）に変数を並べていくように番号をつけることにしましょう．つまり下記のような順番です．

phi[0][0], phi[1][0], phi[2][0], ..., phi[Nx-1][0], phi[0][1], phi[0][2], ...

これを式で書くと，

$$\texttt{phi}[i][j] = \phi(i + j \cdot \texttt{Nx})$$

となります．この番号付けを使って，これから係数行列を作っていきます．

各セルごとに，まずは自分自身のセル番号（P と書きます）と，その上下左右のセル番号（S, W, E, N）を計算します．

ソースコード 4.1　Poisson 方程式の係数行列を作るループ

```
for (int i = 0; i <= Nx - 1; ++i)
  for (int j = 0; j <= Ny - 1; ++j) {
    int P =  i    + j    *Nx;
    int S =  i    +(j-1)*Nx;
    int W = (i-1)+ j    *Nx;
    int E = (i+1)+ j    *Nx;
    int N = i     +(j+1)*Nx;
```

境界に隣接している面でなければ，隣のセルに対応する項の係数を $\frac{1}{\Delta x^2}$ または $\frac{1}{\Delta y^2}$ にして，自分のの項の係数から同じ値を引きます．

ソースコード 4.2　Poisson 方程式の係数計算

```
    if (j >= 1) {
      M[P][S] = 1./(dy*dy);
      M[P][P] -= 1./(dy*dy);
    }
```

```
    if (i >= 1) {
      M[P][W] = 1./(dx*dx);
      M[P][P] -= 1./(dx*dx);
    }
    if (i <= Nx - 2) {
      M[P][E] = 1./(dx*dx);
      M[P][P] -= 1./(dx*dx);
    }
    if (j <= Ny - 2) {
      M[P][N] = 1./(dy*dy);
      M[P][P] -= 1./(dy*dy);
    }
  }
```

✐ C++ で a -= b; というのは a = a - b; と同じ意味です.

長々と説明してきましたが，プログラムは上記のように簡単です.

4.3 線型方程式の解法・LU 分解

線型方程式の解法にはいろいろありますが，ここでは数値計算の教科書によく出てくる，LU **分解**による解法を使うことにします.

⚠ 第 2 章でも述べましたが，ここで説明する LU 分解は，計算に時間がかかりすぎるので，実用には適しません! 線型方程式の解法の基本なのでここで取り上げています．このことを理解した上で使ってください．あとの章でより速い方法と置き換えます.
なお LU 分解を用いた線型方程式の解法は，それでも直接解法と分類される解法の中では最速です.

LU 分解の解説は省略します．プログラムは下記のようになります.

ソースコード 4.3 LU 分解

```
// 線型方程式を解くための関数
// Aを LU分解して，そのままAに入れる．ピボット選択はしていない.
void LU_decomp(double A[][Ncells])
{
  for (int k = 0; k < Ncells - 1; ++k) {
```

```
    double w = 1./A[k][k];
    for (int i = k+1; i < Ncells; ++i) {
      A[i][k] *= w;
      for (int j = k+1; j < Ncells; ++j)
        A[i][j] -= A[i][k]*A[k][j];
    }
  }
}
```

ソースコード 4.4　LU 分解された係数行列を使って，線型方程式を解く

```
// 線型方程式LU*x=bを解いて，根をbに格納する.
void LU_solve(double const LU[][Ncells], double b[])
{
  // LUはLU分解済みの係数行列
  for (int k = 1; k < Ncells; ++k)
    for (int i = 0; i < k; ++i)
      b[k] -= LU[k][i]*b[i];
  for (int k = Ncells-1; k >= 0; --k) {
    for (int i = k+1; i < Ncells; ++i)
      b[k] -= LU[k][i]*b[i];
    b[k] /= LU[k][k];
  }
}
```

係数行列を LU 分解する関数と，LU 分解済みの係数行列を使って方程式を解く関数とに分けてあります．今回の問題では計算の途中で格子の構造が変わらないので，係数行列も変わりません．だから計算のはじめにいちど係数行列を構築し，LU 分解しておけば，それを計算中ずっと使いまわすことができます．

4.4　右辺ベクトルの計算

次に Poisson 方程式の右辺ベクトルを見ていきます．今回解こうとしている方程式は $\nabla^2\phi = \frac{1}{\Delta t}\nabla\cdot\boldsymbol{u}^P$ なので，右辺は $\frac{1}{\Delta t}\nabla\cdot\boldsymbol{u}^P$ です．圧力の修正量 `phi` と同じ検査空間で，$\frac{1}{\Delta t}\nabla\cdot\boldsymbol{u}^P$ を評価（離散化）します．

圧力の修正量の場合，検査空間の上下左右の流速値が変数として用意されています．ですの

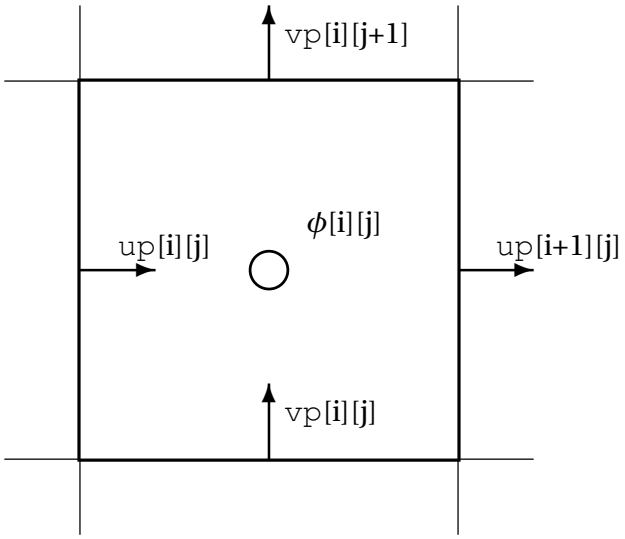

図 4.2 圧力修正量 ϕ の検査空間とステンシル（右辺ベクトル計算用）

で補間は必要なく，右辺の値を計算できます．

$$\begin{aligned}\frac{1}{\Delta t}\nabla\cdot\boldsymbol{u}^p &= \frac{1}{\Delta t}\left(\frac{\partial}{\partial x}u^P+\frac{\partial}{\partial y}v^P\right)\\ &\approx \frac{1}{\Delta t}\left(\frac{-\mathrm{up}[i][j]+\mathrm{up}[i+1][j]}{\Delta x}+\frac{-\mathrm{vp}[i][j]+\mathrm{vp}[i][j+1]}{\Delta y}\right)\end{aligned}$$

ソースコード 4.5 右辺ベクトルの計算

```
// 右辺ベクトルの構築
for (int i = 0; i <= Nx - 1; ++i)
  for (int j = 0; j <= Ny - 1; ++j)
    b[i+j*Nx] = ((-up[i][j]+up[i+1][j])/dx + (-vp[i][j]+vp[i][j+1])/
        dy)/dt;
```

✐ このように単純に書くことができるのは，境界の流速値も変数として持っているからです．境界の値を変数として持たない場合は，それを考慮したプログラミングが必要です．

右辺ベクトルが構築できたら，それと LU 分解された係数行列とを使って，方程式を解きます．

```
// 線型方程式を解く.
LU_solve(LU, b);
```

計算結果が b に格納されるので，これを phi に代入します．

⚠ 言うまでもなく，無駄な計算です!! b をそのまま使えば，コピーする必要はありませんし，そもそも変数 phi が必要無くなります．ここは後に改善します．

ソースコード 4.6　計算結果を phi に格納

```
  for (int i = 0; i <= Nx - 1; ++i)
    for (int j = 0; j <= Ny - 1; ++j)
      phi[i][j] = b[i+j*Nx];
```

ここまでで，流速の予測値と圧力の修正量の計算が終わりました．これを使って圧力と流速を修正すれば，このステップでの計算は終わりです．ここで章を改めます．

第 5 章

圧力と流速の修正と計算結果の出力

圧力と流速は，次のように修正します．

$$P^{n+1} = P^n + \phi$$
$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^P - \Delta t \nabla \phi$$

P（プログラム中の変数名では p）は，単に ϕ（phi）を足すだけです．

$\boldsymbol{u}^{n+1}$（プログラム中では u，v）は，圧力修正量の勾配を計算して，それに Δt をかけたものを引きます．圧力修正量の勾配の計算は，圧力勾配の計算と同じで，例えば u だったら dpdx を計算したときと同じ形の式になります．（p が phi になるだけ）

以上をまとめたものが，下記のコードです．

ソースコード 5.1　圧力と流速の修正

```
  for (int i = 0; i <= Nx - 1; ++i)
    for (int j = 0; j <= Ny - 1; ++j)
      p[i][j] += phi[i][j];

  for (int i = 1; i <= Nx - 1; ++i)
    for (int j = 0; j <= Ny - 1; ++j)
      u[i][j] = up[i][j] - dt*(-phi[i-1][j] + phi[i][j])/dx;

  for (int i = 0; i <= Nx - 1; ++i)
    for (int j = 1; j <= Ny - 1; ++j)
      v[i][j] = vp[i][j] - dt*(-phi[i][j-1] + phi[i][j])/dy;
```

これでこのステップの計算自体は終わりです．次のステップの計算に進む前に，現在の計算結果をファイルなどに出力します．今回は圧力と流速の値を羅列した出力にします．

ソースコード 5.2　計算結果の出力

```
// 計算結果の出力
// P
std::cout << "P" << std::endl;
for (int i = 0; i <= Nx - 1; ++i)
  for (int j = 0; j <= Ny - 1; ++j)
    std::cout << p[i][j] << std::endl;

// U
std::cout << "U" << std::endl;
for (int i = 1; i <= Nx - 1; ++i)
  for (int j = 0; j <= Ny - 1; ++j)
    std::cout << u[i][j] << std::endl;

// V
std::cout << "V" << std::endl;
for (int i = 0; i <= Nx - 1; ++i)
  for (int j = 1; j <= Ny - 1; ++j)
    std::cout << v[i][j] << std::endl;
```

ここまでのプログラムをまとめたものを付録 A に載せます．これで，2 次元キャビティの計算を行うことができます．

第 6 章

コンパイルと計算実行

これまで作ってきたプログラムのソースコードが cavity1.cpp というファイルに入っているとします．これを GNU C++ でコンパイルするには，そのソースコードがあるディレクトリで，次のコマンドを実行します．

```
% g++ cavity1.cpp -o cavity1↵
%
```

% はコマンドプロンプトです．コンパイルエラーがなければ，何も言わずにコマンドが終了するはずです．

✎ ここで説明したコンパイル方法は最も基本的なものです．より計算の速い実行プログラムを作る方法や，デバッグしやすくする方法などを後に説明します．

コンパイルして作ったプログラムを実行するには，下記のようにします．

```
% ./cavity1↵
```

画面上に，計算結果が大量に表示されるはずです．これだと計算結果を読むことができないので，計算結果をファイルに保存するには，**リダイレクト**します．

```
% ./cavity1 > output.dat↵
```

>がリダイレクトの記号で，プログラムの**標準出力**を指定したファイルに切り替えます．リダイレクト，標準出力などは UNIX や Linux のシェルを使うときの基本語彙ですので，知らない場合は勉強してください．

第 7 章

計算結果のチェック

この章では，前章までに作ってきたプログラムが正しく動作しているかどうかを確かめていきます．

流れ場を計算するためのプログラムなので，流れ場を目で見て，それが妥当かどうかを確かめるべきですが，それは次の章以降の課題にとっておきましょう．今回は，例えば線型方程式の根が正しいか（元の方程式を満たしているか）の確認と，計算した結果が連続の式を満たしているかの確認をします．

7.1 デバッグ用のコードを挿入する

プログラムの中にある間違いのことを**バグ**（虫）と呼び，プログラムのバグを見つけて修正することを**デバグ**または**デバッグ**と言います．

デバッグは，プログラムの動作を確かめるときにはぜひ行うべきですが，プログラムが正しく動作していることが分かったあとは必要がなくなります．なので，プログラムをデバッグ用にコンパイルしたときと，そうでないときで，実行するコードを切り替えることができると便利です．

これを実現するためには，まずソースコードの中に，デバッグの時だけ実行したいコードを，下記のように`#ifdef DEBUG`〜`#endif`でくくって書いておきます．

ソースコード 7.1　デバッグ用コードの例

```
#ifdef DEBUG
  // デバッグ時にのみ有効となるコード
#endif  // DEBUG
```

そして，コンパイル時に`-DDEBUG`オプションをつけてコンパイルします．

```
% g++ -DDEBUG=1 cavity1.cpp -o cavity1↵
%
```

-DDEBUG オプションをつけたときだけ，#ifdef DEBUG〜#endif の部分が有効になります．-DDEBUG オプションを付けないと，#ifdef DEBUG〜#endif の部分は書いていないのと同じようにコンパイルされます．

この機能は DEBUG だけでなく，プログラムの機能を様々に切り替えるのに使うことができます．後の章で利用しています．

7.2　線型方程式の根の確認

このプログラムの中では、Poisson 方程式 $\nabla^2\phi = \frac{1}{\Delta t}\nabla\cdot\boldsymbol{u}^P$ を離散化して線型方程式 $M\boldsymbol{x} = \boldsymbol{b}$ の形にし，それを解いています．ここが正しく計算されているかどうかを確かめます．

線型方程式 $M\boldsymbol{x} = \boldsymbol{b}$ を解くというのは，M と $\boldsymbol{b}$ が与えられて，そこから $\boldsymbol{x}$ を求めている，ということです．ですからこれが正しく計算できていれば，**行列–ベクトル積** $M\boldsymbol{x}$ を計算すれば，それが $\boldsymbol{b}$ になるはずです．

行列–ベクトル積は Matvec などと略されます．これを計算する関数をまず作りましょう．

ソースコード 7.2　行列–ベクトル積（Matvec）

```
#ifdef DEBUG
// 行列・ベクトル積 b = A*x
void Matvec(double const A[][Ncells], double const x[], double b[])
{
  for (int i = 0; i < Ncells; ++i) {
    b[i] = 0.;
    for (int j = 0; j < Ncells; ++j)
      b[i] += A[i][j]*x[j];
  }
}
#endif  // DEBUG
```

さて，我々がこれまで作ってきたプログラムは LU 分解を使って線型方程式を解いています．線型方程式の根が正しいかどうかを確かめるためには，LU 分解する前の係数行列を保存しておく必要があります．

ソースコード 7.3　係数行列の保存

```
#ifdef DEBUG
  double OrigM[Ncells][Ncells];
  for (int i = 0; i <= Nx*Ny - 1; ++i)
    for (int j = 0; j <= Nx*Ny - 1; ++j)
      OrigM[i][j] = M[i][j];
#endif  // DEBUG
```

また，各タイムステップで右辺ベクトルも保存しておく必要があります．関数 LU_solve は右辺ベクトルの配列を受け取り，その配列に根を入れて返すようになっているからです．

ソースコード 7.4　右辺ベクトルの保存

```
#ifdef DEBUG
    double RHS[N];
    for (int i = 0; i < N; ++i)
      RHS[i] = b[i];
#endif  // DEBUG
```

そして，各タイムステップで線型方程式を解いた直後に，下記のコードを挿入して，係数行列と求めた根ベクトルとの積を計算します．そして，積の各要素と，右辺ベクトルの要素との差の RMS（root mean square，2 乗の平均の平方根）を計算します．

ソースコード 7.5　線型方程式の根の妥当性確認

```
#ifdef DEBUG
    // 根のテスト
    double prod[Ncells];
    Matvec(OrigM, b, prod);
    double sum_diff = 0.;
    for (int i = 0; i < Ncells; ++i)
      sum_diff += (RHS[i]-prod[i])*(RHS[i]-prod[i]);
    std::cerr << "RMS␣of␣Diff␣==␣" << sqrt(sum_diff / Ncells) << std::
        endl;
#endif  // DEBUG
```

計算結果を std::cerr に出力しています．これは C++ で**標準エラー出力**に出力するための方法です．UNIX 系の OS では標準入力，標準出力，標準エラー出力があり，標準出力と標準エラー出力は，何も指定しなければ画面に出力されます．今回，標準エラー出力を使ったのは，

このプログラムが標準出力をファイルにリダイレクトして使うことを想定しているからです．プログラムの出力に関してはすでに第 5 章で説明しています．

また，ここでは平方根を求める sqrt() という関数を使っていますが，これを使うためには数学関数用のヘッダをインクルード（C++ のソースコードに取り込むこと）します．プログラムの最初の方に，下記のように書いておきます．

ソースコード 7.6　数学関数用ヘッダファイルをインクルード

```
#ifdef DEBUG
#include <cmath>                    // for debug
#endif                              // DEBUG
```

✎ コンパイルするときに，数学関数ライブラリを指定する必要があるかもしれません．次のように指定します．

```
% g++ -DDEBUG=1 cavity1.cpp -lm -o cavity1↵
%
```

`-lm` というオプションが，数学関数ライブラリ libm を使うことを指定しています．コンパイル時に「sqrt がない」という内容のエラーが出た場合には，このオプションを指定してコンパイルしてください．

今回対象としている Ubuntu 14.04 と GNU C++ 4.8 の場合は，数学関数ライブラリを指定しなくとも sqrt を使うことができました．

7.3　連続の式を満たすかの確認

今回作ってきた SMAC 法のコードは非圧縮性流れを前提としています．ですので，各セルに入ってくる流体の体積と，出て行く流体の体積は釣り合っているはずです．つまり，入ってくる量を正，出て行く量を負として計算すると，合計はゼロになっているはずだ，ということです．これを確認します．

「体積」と書きましたが，2 次元の計算をしているので，例えばセルの左側（西側）から入ってくる流体の体積は u[i][j]*dy と計算されます．右側（東側）から出て行く流体の体積は u[i+1][j]*dy です．ここで「出て行く」と書きましたが，もし u[i+1][j] が負だったら，実際には流体は入ってきていることになります．

下記のコードを，圧力と流速の修正が終わった後，または出力の後に挿入します．

ソースコード 7.7　連続の式を満たすかの確認

```
#ifdef DEBUG
    // 連続の式に対する誤差のチェック
    double sum_voldiff = 0;
    for (int i = 0; i <= Nx - 1; ++i)
      for (int j = 0; j <= Ny - 1; ++j) {
        double voldiff = (u[i][j] - u[i+1][j])*dy + (v[i][j] - v[i][j
            +1])*dx;
        sum_voldiff += voldiff*voldiff;
      }
    std::cerr << "RMS␣error␣of␣Continuity:␣" << sqrt(sum_voldiff / Ncells
        ) << std::endl;
#endif  // DEBUG
```

線型方程式の根の確認と同じく，誤差の RMS を計算しています．

7.4 デバッグ用にコンパイルしたプログラムの実行

実際にプログラムを動かしてみましょう．

```
% g++ -DDEBUG=1 cavity1.cpp -o cavity1↵
% ./cavity1 > output.dat↵
Time:  0.0001
RMS of Diff == 8.26887e-12
RMS error of Continuity:  3.1511e-18
Time:  0.0002
RMS of Diff == 4.94043e-13
RMS error of Continuity:  2.17171e-19
Time:  0.0003
RMS of Diff == 4.90036e-13
RMS error of Continuity:  2.17343e-19
   :
   :
```

上記と同じ数値が出るとは限りませんが，線型方程式の誤差の RMS と，連続の式の誤差の RMS のいずれもぴったり 0 にはならないでしょう．小さな数値が表示されるはずです．これ

はプログラムのエラー（バグ）ではなく，数値誤差のせいです．

SMAC 法の場合は，連続の式の誤差が拡大したりせずに計算開始時と同じオーダーで推移していれば，解析は妥当だろうと推測されます．それ以外の時は，以下のように考えるとよいでしょう．

- 線型方程式の誤差が最初から大きい—線型方程式のソルバーにバグがあることを疑います．
- 線型方程式の誤差は小さいが，連続の式の誤差が大きい—圧力と流速の修正にバグがあることを疑います．
- 連続の式の誤差は最初小さいが，計算するに連れて大きくなる—これはプログラムそのもののバグよりも，時間刻み `dt`（Δt）が大きすぎることや，メッシュが粗いこと（`Nx` や `Ny` が小さすぎること）を疑います．

上に書いたことはあくまで最初の着眼点だと思ってください．必ずしも上記の場所にバグがあるとは限りません．

SMAC 法のプログラムの場合，線型方程式のソルバーと圧力と流速の修正の部分は，妥当性を確かめやすいためバグを見つけやすいのですが，流速の予測値を計算する部分は，コードが複雑な上に妥当性の確認が難しいです．

なお，標準エラー出力をファイルにリダイレクトすることもできます．Linux で標準的な bash の場合は，以下のようにします．（標準出力を output.dat に，標準エラー出力を error.dat にリダイレクトする例）

```
% ./cavity1 > output.dat 2> error.dat↵
```

標準エラー出力を標準出力と同じファイルにリダイレクトするには，以下のようにします．

```
% ./cavity1 > output.dat 2>&1↵
```

第 8 章

計算結果の可視化

8.1 ParaView と Python

計算結果を目で見てわかりやすい形にしましょう．目に見えない・見えにくいデータや現象を目で見てわかるようにする作業のことを**可視化**と呼びます．ここでは ParaView という可視化ソフトウェアを使うことにします．

☞ ParaView

ParaView はオープンソースのデータ解析・可視化ソフトウェアです．Windows 版，Linux 版，Mac OS X 版などがあります．

今回作ったプログラムが出力したファイルは，圧力と流速をただ書き連ねただけのものなので，そのままでは ParaView で読むことができません．そこで，これを ParaView が読める Legacy VTK ファイルフォーマットに変換します．

変換プログラムは，C++ ではなく Python で書いています．

☞ Python（Wikipedia）

Python はインタプリタ言語で，ソースコードをコンパイルする必要がなく，また高度な機能を比較的簡単に実装することができます．ファイルを読んで操作するのは，C++ よりも得意です．ですので筆者はデータ処理のプログラムは，特に理由がなければ C++ は使わず，なるべく Python で書くようにしていています．また，Python インタプリタをほかのプログラムに組み込むのも簡単にできるので，筆者が書いたプログラムではパラメタファイルの読み込みにも使われています．

✐ だったら Python で CFD プログラムを書けばいいじゃないかと思うかもしれませんが，Python で CFD プログラムを書くと解析速度が非常に遅いはずです．試したことはあり

ませんが，計算規模が大きくなった時には使い物にならないと推測されます．

✐ 筆者は以前は AWK や Perl も使っていました．いずれもファイルの処理が得意なプログラミング言語です．AWK ファイルの中でデータがスペースやカンマで区切られているものを扱うのが得意です．Perl はファイル 1 行 1 行に対して，AWK よりも複雑な処理をしたいときに便利です．

なお，C++ のプログラムで最初から ParaView が読めるようなファイルを書くことも考えられますが，筆者は以下の理由から，CFD のプログラムと，その出力を可視化用に加工するプログラムを分けて作っています．

- なるべく加工前の生のデータを残しておくべきです．生データとは，CFD の場合はプログラムの中で計算された圧力，流速などのことです．
  生データを加工するのは難しくありませんが，加工したデータを元に戻すのはほとんど不可能です．だから CFD のプログラムでは生データをそのまま出力するようにしたい．（現実には，ファイルが大きくなりすぎるなどの理由から，いつでも生データを出力できるわけではありませんが．）
- もしファイル出力にエラーがあったら，流れの解析からやり直すことになり，時間の無駄です．また，CFD のプログラムでは，ファイル出力はなるべくシンプルにしておきたい．
- Python でプログラムを書いている場合，修正や機能追加が容易です．

データ変換プログラムは付録 B に掲載しています．プログラムの中身の解説は省略し，ここでは使い方だけ説明します．データ変換プログラムのファイルを data2vtk.py という名前だとすると，下記のようなコマンドを実行します．

```
% mkdir vtk↵
% python data2vtk.py output.dat↵
```

はじめのコマンドで，vtk という名前のディレクトリを作ります．これは 1 回だけ実行すればよいです．

次のコマンドで，output.dat というファイルを読んで，data*XXXX*.vtk というファイルを，vtk ディレクトリの中に書き出しています．（*XXXX* は数値で，0001, 0002, ...）

vtk ディレクトリの中には，data*XXXX*.vtk というファイルが多数生成されます．ParaView はこれをまとめて読んで，時系列につながった 1 つのデータとして扱ってくれます．

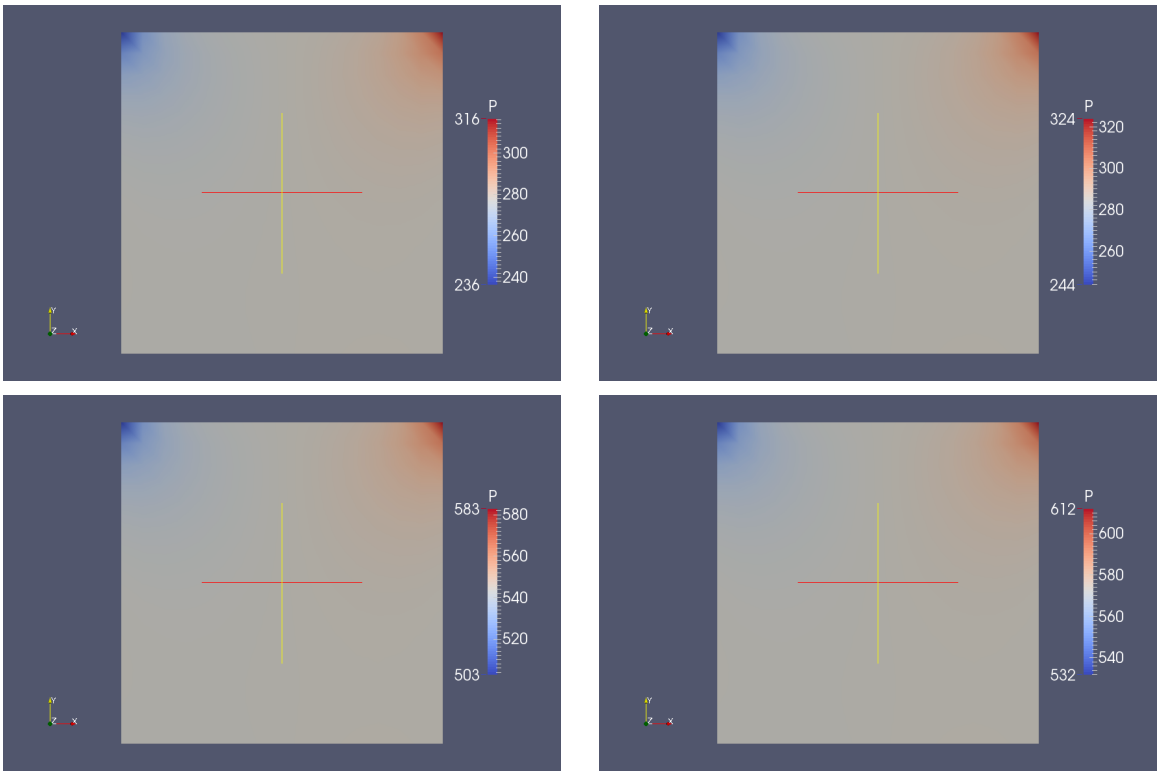

図 8.1　異なるタイムステップでの圧力分布

8.2　圧力の可視化結果・圧力が安定しない理由は?

圧力分布を可視化したのが図 8.1 です．（ここでは「圧力」と表現していますが，教科書 [1] での P のことです．）

見てすぐわかるのは，どれも同じような分布になっているということです．右上の角の圧力が高く，左上の角の圧力が低いです．キャビティ上部では流れは左から右に向かっているので，妥当な結果です．

ただし圧力の絶対値は，一定していないこともわかります．それぞれの画像の右側に色と圧力の関係を表すカラーバーが描かれていますが，上限と下限との差が 80 あることは共通しているものの，絶対値は全て異なっています．なぜでしょうか? 計算がおかしいのでしょうか?

圧力の絶対値が一定しない理由は，今回作ったプログラムが圧力の絶対値に依存しない形の計算をしているからです．圧力の修正量は，Poisson 方程式

$$\nabla^2 \phi = \frac{1}{\Delta t} \nabla \cdot \boldsymbol{u}^P$$

を解いて求めていますが，この方程式は圧力の修正量の勾配に関する方程式です．だからこの方程式からは圧力修正量の絶対値は決まらないので，圧力の絶対値も決まりません．

例えばあるセルの圧力修正量が 0，隣のセルの圧力修正量が 20 という条件が，Poisson 方程式を満たしていたとします．この場合，あるセルの圧力修正量が 100，隣のセルの圧力修正量が 120 だったとしても，Poisson 方程式を満たすはずです．圧力修正量の勾配は同じだからです．

このような場合，根の存在（係数行列の正規性）を厳密に調べるソルバーを使うと，エラーになります．今回採用したソルバー（第 4 章で説明したもの）は手抜きのソルバーだったので，エラーは出ませんでしたが，計算結果の絶対値は一定しない，というわけです．

✐ 数学用語では「根が不定」と言います．根が一意に定まらず，いろいろな答えがありうる，ということです．

圧力の絶対値を安定させるためには，どこかの点での圧力を固定する方法があります．例えば図 8.1 の左下の点で圧力を 0 にし，それに合わせてほかの圧力を決める，などです．今回のプログラムでは，圧力の絶対値が安定していなくても計算が破綻せずに継続するので，放置することにします．

流速の可視化は，後の賞で Ghia らの解析結果 [2] と比較するときに実施します．

第 9 章

計算時間の測定と最適化コンパイル

9.1 プログラムの実行時間の測定

今回作成したプログラムは，流れ場を解析するのにどのぐらいの時間を要しているでしょうか．これを計測してみます．UNIX 系の OS ではこれを計測する time コマンドが用意されています．使い方は簡単で，コマンドの前に `time␣` を入れるだけです．

```
% g++ cavity1.cpp -o cavity1↵
% time ./cavity1 > output.dat↵

real 2m37.612s
user 1m21.885s
sys 1m13.867s
%
```

‘real’ がプログラムの実行時間で，今回は約 2 分 38 秒かかっています．‘user’ がプログラム内部で要した時間で，今回は約 1 分 22 秒です．‘sys’ はプログラムが OS の機能を使った時間で，今回のプログラムの場合は計算結果をファイルに出力するのに要した時間です．

✐ Linux 上で bash を使っている場合（Linux の標準的な設定）は，time コマンドは bash に組み込まれているものが使われます．上記のような出力の場合は bash の組み込みコマンドです．Linux の time コマンドを起動したい場合は，`/usr/bin/time` のようにコマンドをフルパスで指定します．Linux の time コマンドのほうが表示される情報量が多いですが，今回は bash の組み込みコマンドの情報量で十分なので，そちらを使います．

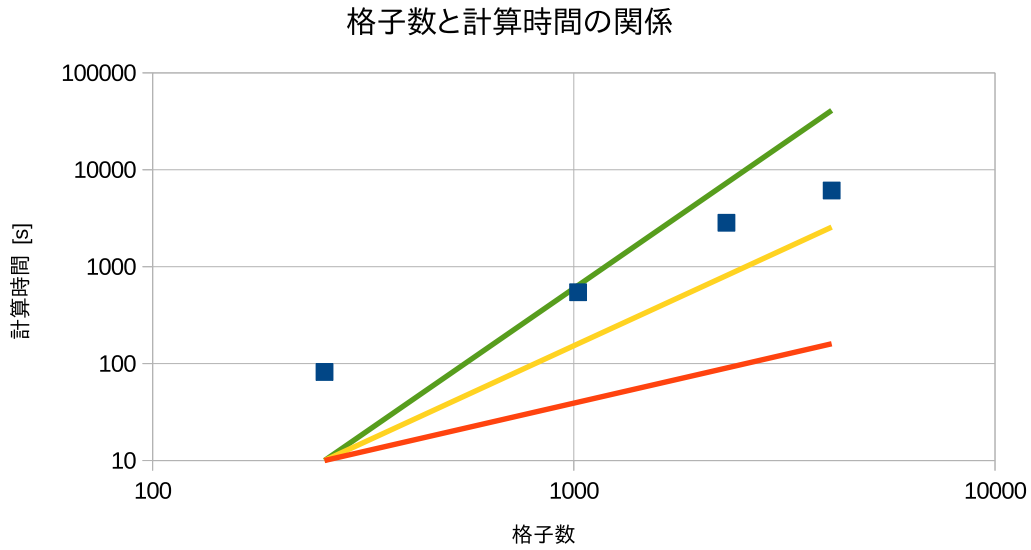

図 9.1　格子数と計算時間の関係．図中の直線は，下から $y \propto x$，$y \propto x^2$，$y \propto x^3$ の傾きの線

9.2　格子数と計算時間との関係

今回作成したプログラムは，`Nx` = 16，`Ny` = 16 と指定していたので，格子数は 16 × 16 = 256 です．Ghia らの解析 [2] は 128 × 128 あるいは 256 × 256 の格子を用いた解析をしているので，われわれもこれと同じ規模の解析を目指します．

✎ Ghia らの論文 [2] 中では，格子の節点数 129 × 129 または 257 × 257 と記されています．われわれのプログラムでは計算空間の分割数で表示しているので，それぞれ 128 × 128，256 × 256 に対応します．

格子数 16 × 16，32 × 32，48 × 48，64 × 64 の場合の格子数と計算時間の関係を図 9.1 に示しています．時間刻みは全て等しく $\Delta t = 0.0001$ としています．図中の四角いシンボルが time コマンドで測定したプログラムの計算時間（‘user’ 時間）を表し，3 本の直線は下から順に，傾きが格子数の 1 乗，2 乗，3 乗に比例する線です．

✎ ここで示している計算時間は，筆者が所有する PC で実施したときのものです．CPU: Intel Xeon E3-1270V2 (3.50 GHz)，メモリ 32 GB DDR3 1600.
✎ 今回はプログラムを 1 回だけ実行してその時間で比較しています．

計算結果の傾きを，直選の傾きと比べてください．（直線は傾きを見るために引いたものなので，絶対値には意味がありません．）計算時間が，格子数の 2 乗より少し少ない程度の時間

で増えているように見えます．悪いほうに見積もって，格子数の 2 乗程度に比例すると考えると，格子数 256×256 の計算は，64×64 の計算の $\{(256 \times 256) \div (64 \times 64)\}^2 = 256$ 倍の計算時間で終わると予想されます．

64×64 の格子での計算時間が，106 分ほどです．その 256 倍は 27000 分程度，約 19 日間です．これはちょっときついですね…．2 次元の CFD で 256×256 程度の格子でこんなに時間がかかるようでは，実用になりません．

9.3 最適化コンパイル

計算時間を短縮するために，次章以降でアルゴリズムの検討などを行っていきますが，その前にコンパイラの機能を使ってプログラムの実行を速くすることを試みます．今回使っている GNU C++ コンパイラを含め，現在使われているほとんどすべてのコンパイラは**最適化**（optimization）をすることができます．コンパイラがプログラムの中身を調べて，計算時間が短縮できるようにプログラムの実行順序や実行方法を調整してくれる，という意味です．

✎ 厳密に言うとプログラムのサイズを小さくする最適化もありますが，ここでは立ち入りません．これ以降では最適化という言葉を，プログラムの実行時間を短くするための方法，という意味で使っていきます．

最適化はコンパイルオプションで指定できます．GNU C++ の場合，`-O` または`-O1`，`-O2`，`-O3` のようなオプションを指定し，数字が大きいほど最適化のレベルが上がります．ここでは`-O3` を指定してみます．

```
% g++ -O3 cavity1.cpp -o cavity1↵
% time ./cavity1 > output.dat↵

real 2m19.187s
user 1m4.318s
sys 1m14.625s
%
```

‘user’ 時間が少し短くなりました．約 82 秒→約 64 秒ですから，2 割程度の削減です．一方，‘sys’ 時間はほとんど変わりません．これはプログラムの内部で費やしている時間ではないので，プログラムを最適化してもあまり影響を受けないからです．

最適化した場合の，格子数と計算時間の関係を図 9.2 に示しています．図中の●印が最適化コンパイルしたプログラムの結果，■は最適化していない場合（図 9.1 と同じデータ）です．

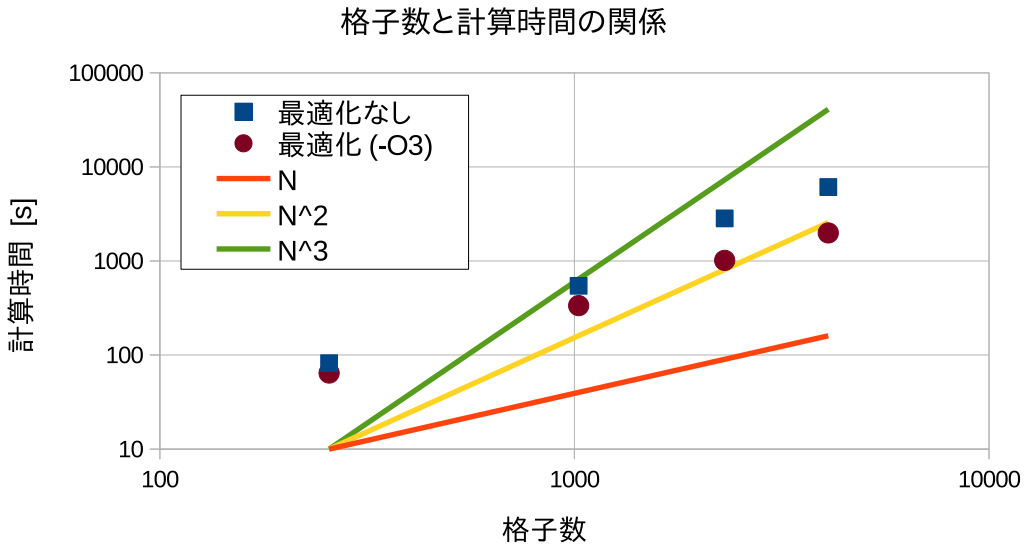

図 9.2　格子数と計算時間の関係，最適化したプログラムの結果入り

最適化したことで計算時間が短縮されています．また最適化した場合は，計算時間が格子数のおよそ 1 乗に比例して増えていることが分かります．

第 10 章

プログラムの性能解析（プロファイリング）

前章でプログラムの実行時間を計測しましたが，プログラムの中のどの部分が時間を費やしているのかは，全体の実行時間だけからは分かりません．この章ではプログラムの内部まで踏み込んだ性能解析（**プロファイリング**）をしてみます．

GNU C++ では gprof を用いたプロファイリングができますので，今回はこれを使います．

☞ 性能解析（Wikipedia）

gprof を使ってプロファイリングするには，コンパイルするときに`-pg` オプションをつけます．

```
% g++ -pg cavity1.cpp -o cavity1↵
% ./cavity1 > output.dat↵
%
```

プログラムが終了したときに，gmon.out というファイルが生成されるはずです．これを gprof コマンドで解析します．解析結果は標準出力に出力されるので，必要に応じてファイルにリダイレクトします．

```
% gprof ./cavity1 gmon.out > gprof.out↵
%
```

gprof の出力は Flat profile の部分，Call graph の部分，関数のインデックスの部分に分けられます．今回は出力の先頭にある Flat profile の部分を見ます．筆者の環境では，格子数

16 × 16 の計算で，以下のように出力されました．（適宜改行しています．）

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 89.55     30.23    30.23   100000     0.30     0.30  LU_solve(dou
ble const (*) [256], double*)
 10.44     33.76     3.52                             main
  0.09     33.79     0.03        1    30.03    30.03  LU_decomp(do
uble (*) [256])
  0.00     33.79     0.00        1     0.00     0.00  _GLOBAL__sub
_I_u
  0.00     33.79     0.00        1     0.00     0.00  __static_ini
tialization_and_destruction_0(int, int)
```

関数 LU_solve() が計算時間の約 90% を費やしていることが分かります．関数が呼び出された回数が 100000 回で，1 回呼ばれる毎に費やす時間は約 0.3 ms です．次に時間を費やしているのは main 関数で約 10%．関数 LU_decomp() にはほとんど時間を取られていません．

10.1　最適化とプロファイリング

最適化とプロファイリングを同時に行うこともできます．はじめに -O2 で最適化してみます．

```
% g++ -O2 -pg cavity1.cpp -o cavity1↵
% ./cavity1 > output.dat↵
%
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.78      9.39     9.39                             LU_solve(dou
ble const (*) [256], double*)
  0.11      9.40     0.01                             LU_decomp(do
uble (*) [256])
  0.00      9.40     0.00        1     0.00     0.00  _GLOBAL__sub
_I_u
```

LU_solve() が 100%（以上!?）になって，main が消えてしまいました．また，関数が何回呼ばれたかがわからなくなりました．-O3 で最適化するとどうなるでしょうか．

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.90      9.10     9.10                             LU_solve(dou
ble const (*) [256], double*)
  0.00      9.10     0.00        1     0.00     0.00  _GLOBAL__sub
_I_u
```

今度は LU_decomp() も見えなくなってしまいました．

最適化はコンパイラの中で複雑な解析の結果として行われているので，筆者の知識の範囲ではこの理由を明確には説明できません．推測すると，まず最適化レベル 2 の場合に main() 関数がなくなるのは，main() という関数を作ってそれを呼び出すのではなく，そこからプログラムをはじめるようにしたからでしょう．レベル 3 で LU_decomp() が見えなくなったのは，LU_decomp() はプログラム中で 1 回しか呼ばれないので，やはりわざわざ関数を作ってそれを呼び出すのではなく，LU_decomp() を呼び出す部分にそのコードを直接埋め込むような最適化をしたのでしょう．

ただしどの場合でも，LU_solve() が計算時間の大部分を占めていることに変わりはありません．SMAC 法の計算では，Poisson 方程式を解くところが最も計算時間を費やします．ここを高速化することが，SMAC 法の計算を速くするための鍵となります．それ以外の部分は，頑張って高速化しても計算時間をほとんど短縮できないということも分かります．

第 11 章

プログラムの改善・2 次元配列を 1 次元配列に

計算時間は長そうだけど，でも解析結果を Ghia らの論文 [2] と比較したいから，しょうがないか…．ということで，格子数 128 × 128 の計算を試みます．Nx = 128，Ny = 128 に変更してコンパイルすると，

```
% g++ cavity1.cpp -o cavity1↵
cavity1.cpp:(.text+0x1633):  再配置がオーバーフローしないように切り詰められました:  R_X86_64_32S (シンボル `b' に対して、 .bss セクション、/tmp/ccHuNwnA.o 内)
cavity1.cpp:(.text+0x169f):  再配置がオーバーフローしないように切り詰められました:  R_X86_64_32S (シンボル `b' に対して、 .bss セクション、/tmp/ccHuNwnA.o 内)
collect2:  error:  ld returned 1 exit status
%
```

…あれ，なにかエラーが出てコンパイルできません．

これは配列や行列など変数が多すぎて，bss セクション（静的に確保された変数を保持するメモリ）のサイズを超えてしまった場合に出るエラーです．

☞ .bss（Wikipedia）

変数を格納するためのメモリの確保方法には，静的と動的の 2 つの方法があります．メモリが静的に確保されたという意味は，そのメモリがプログラムの実行開始時にすでに確保されていて，その後ずっとそのメモリを使い続けることができる，という意味です．簡便ですが上記の

ようにたくさんのメモリを確保するのには向きません．また今回のプログラムで行っているように，必要なメモリ数がプログラムをコンパイルするときにわかっている必要があります．

我々が作ってきたプログラムは，これまで変数の配列をプログラムの中に明示していました．つまりメモリをすべて静的に確保していたことになります．これだと計算規模に限界があるので，メモリを動的に確保することにします．動的に確保するとは，プログラム開始時にはメモリが確保されておらず，プログラムの中でメモリを確保する命令やコマンドを使って，必要なサイズのメモリを確保する，ということです．

同時に，これまで 2 次元配列として確保していた変数を，1 次元配列に置き換えます．4.2 節で述べたように，2 次元配列にしているせいで計算が複雑になる場合もあるからです．

11.1　配列を動的に確保する

メモリを動的に確保する方法はいろいろありますが，ここでは C++ の標準の機能である new を使って配列を確保します．

⚠ 筆者は new で配列を確保するよりも STL を使うほうが良いと考えています．後の章で説明しますが，ここでは C++ の基本の機能を使います．

はじめに，変数の定義を変更します．配列として扱いたい変数をポインタとして宣言します．

✎「ポインタ」は C，C++ の言語の中でもわかりにくい概念のようです．ですがそのために，ポインタの概念を説明する文献やウェブサイトもたくさん見つかります．ここでは説明を放棄しますので，適切な文献をあたってください．

ソースコード 11.1　配列として扱う変数の定義

```
// 流れの物理量
double *u;
double *v;
double *p;

// SMAC法の予測値
double *up;
double *vp;

// 圧力の修正量
double *phi;
```

```
// 係数行列
double *M;
```

main() 関数の中で，Poisson 方程式の係数行列を作るよりも前のところに，配列のメモリを確保する手続きを追加します．

ソースコード 11.2 配列のメモリ確保

```
// メモリを動的に確保する
u = new double[(Nx+1)*Ny];
v = new double[Nx*(Ny+1)];
p = new double[Nx*Ny];

up = new double[(Nx+1)*Ny];
vp = new double[Nx*(Ny+1)];
phi = new double[Nx*Ny];

M = new double[Ncells*Ncells];
```

あとは例えばこれまで u[i][j] と書いていたところを u[i+j*(Nx-1)] などと書き換えていけば，これまでと同様の計算をすることができます．v[i][j] であれば v[i+j*Nx] と書き換えます．

u と v とで書き方が違うので（M もまた違います），書き間違いを起こしやすい欠点があります．これをいくらかでも改善するために，下記のようなマクロを導入します．

ソースコード 11.3 配列を扱いやすくするためのマクロ

```
#define u_(i,j)  u[(i)+(j)*(Nx+1)]
#define v_(i,j)  v[(i)+(j)*Nx]
#define p_(i,j)  p[(i)+(j)*Nx]

#define up_(i,j)  up[(i)+(j)*(Nx+1)]
#define vp_(i,j)  vp[(i)+(j)*Nx]
#define phi_(i,j)  phi[(i)+(j)*Nx]

#define M_(i,j)  M[(i)*Ncells+(j)]
```

このマクロを使うことで，これまで u[i][j] と書いていたところを，u_(i,j) と書き換えればよいようになります．例えば 3.2.1 節で説明した対流項の計算は，下記のように書きなおされ

ます.

ソースコード 11.4　対流項の計算コードの一部（1 次元配列・マクロ利用版）

```
double uw = 0.5*(u_(i-1,j)+u_(i,j));
double ue = 0.5*(u_(i,j)+u_(i,j+1));
```

LU 分解のコードにも，同様のマクロを導入して書き直します.

ソースコード 11.5　LU 分解のコード（1 次元配列・マクロ利用版）

```
// 線型方程式を解くための関数
// AをLU分解して，そのままAに入れる．ピボット選択はしていない．
void LU_decomp(double *A)
{
#define A_(i,j)  A[(i)*Ncells+(j)]
  // #define A_(i,j)  A[(i)+(j)*Ncells]
  std::cerr << __FILE__ << '(' << __LINE__ << "): Enter " <<
      __PRETTY_FUNCTION__ << '.' << std::endl;
  for (int k = 0; k < Ncells - 1; ++k) {
    double w = 1./A_(k,k);
    for (int i = k+1; i < Ncells; ++i) {
      A_(i,k) *= w;
      for (int j = k+1; j < Ncells; ++j)
        A_(i,j) -= A_(i,k)*A_(k,j);
    }
  }
  std::cerr << __FILE__ << '(' << __LINE__ << "): Leave " <<
      __PRETTY_FUNCTION__ << '.' << std::endl;
}

// 線型方程式LU*x=bを解いて，根をbに格納する．
void LU_solve(double const *LU, double *b)
{
#define LU_(i,j)  LU[(i)*Ncells+(j)]
  // #define LU_(i,j)  LU[(i)+(j)*Ncells]
  // LUはLU分解済みの係数行列
  for (int k = 1; k < Ncells; ++k)
```

```
    for (int i = 0; i < k; ++i)
      b[k] -= LU_(k,i)*b[i];
  for (int k = Ncells-1; k >= 0; --k) {
    for (int i = k+1; i < Ncells; ++i)
      b[k] -= LU_(k,i)*b[i];
    b[k] /= LU_(k,k);
  }
}
```

データ構造は変わっていますが，プログラムの見かけ上は，4.3 節で示した LU 分解のコードとほとんど変わりません．

このように変更すれば，格子数 128×128 のプログラムもコンパイルすることができます．配列を動的に確保する場合は，特に制限がかけられていなければ，使っている計算機の利用可能なメモリをすべて利用することができます．

11.2 ループの工夫でメモリアクセス順序を改善

C++ では，2 次元配列は例えば u[0][0]，u[0][1]，u[0][2]，...，u[0][Ny-1]，u[1][0]，u[1][1]，u[1][2]，...の順でメモリに格納されていました．メモリはなるべく順番にアクセスしたほうが速いので，この順序にアクセスするようにしていました．3.1 節に記しています．

今回 1 次元配列を使うに当たり，これまで u[i][j] と書いていたところを u[i+j*(Nx-1)] などと書き換えました．これは FORTRAN で使われている配列の順序なのですが，この式で変換すると u[0][0]，u[1][0]，u[2][0]，...，u[Nx][0]，u[0][1]，u[1][1]，u[2][1]，...の順でメモリに格納されることになります．したがってループもこの順序に計算するループに書き換えます．例えば 3.1 節で示したループは，下記のように書き換えます．

ソースコード 11.6　予測計算のループ

```
  for (int j = 0; j <= Ny - 1; ++j)
    for (int i = 1; i <= Nx - 1; ++i) {
      // upの計算
    }

  for (int j = 1; j <= Ny - 1; ++j)
    for (int i = 0; i <= Nx - 1; ++i) {
      // vpの計算
    }
```

これで配列へのアクセスを速くすることができます.

第 12 章

線型ソルバーの変更・定常反復法

第 13 章

プログラムの改善・疎行列の格納方法の変更

第 14 章

プログラムの改善・STL の利用

第 15 章

線型ソルバーの変更・非定常反復法

参考文献

[1] 梶島岳夫, 乱流の数値シミュレーション 改訂版, 養賢堂 (2014). ISBN978–4–8425–0526–8

[2] Ghia, UKNG and Ghia, Kirti N and Shin, CT, High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method, Journal of Computational Physics Vol. 48, No. 3, pp. 387–411 (1982).

付録 A

2 次元キャビティの計算プログラム

ソースコード A.1　2 次元キャビティの計算プログラム

```
1  #include <iostream>
2  #ifdef DEBUG
3  #include <cmath>                    // for debug
4  #endif                              // DEBUG
5
6  // 計算空間のサイズと天井の速度
7  double const Lx = 1.;
8  double const Ly = 1.;
9  double const U = 1.;
10
11 // Reynolds 数
12 double const Re = 1.;
13
14 // 計算空間の分割数
15 int const Nx = 16;
16 int const Ny = 16;
17 int const Ncells = Nx*Ny;          // number of cells
18
19 // 流れの物理量
20 double u[Nx+1][Ny];
21 double v[Nx][Ny+1];
22 double p[Nx][Ny];
```

```
23
24 // SMAC法の予測値
25 double up[Nx+1][Ny];
26 double vp[Nx][Ny+1];
27
28 // 圧力の修正量
29 double phi[Nx][Ny];
30
31 // 解析時間と時間刻み
32 double t_end = 10.;
33 double dt = 0.0001;
34
35 // 係数行列
36 double M[Ncells][Ncells];
37
38 // 右辺ベクトルを格納する配列．線型方程式を解き終わった時には，根が格
39 // 納される．
40 double b[Ncells];
41
42 // 線型方程式を解くための関数
43 // AをLU分解して，そのままAに入れる．ピボット選択はしていない．
44 void LU_decomp(double A[][Ncells])
45 {
46   for (int k = 0; k < Ncells - 1; ++k) {
47     double w = 1./A[k][k];
48     for (int i = k+1; i < Ncells; ++i) {
49       A[i][k] *= w;
50       for (int j = k+1; j < Ncells; ++j)
51         A[i][j] -= A[i][k]*A[k][j];
52     }
53   }
54 }
55
56 // 線型方程式LU*x=bを解いて，根をbに格納する．
```

```
57 void LU_solve(double const LU[][Ncells], double b[])
58 {
59   // LUはLU分解済みの係数行列
60   for (int k = 1; k < Ncells; ++k)
61     for (int i = 0; i < k; ++i)
62       b[k] -= LU[k][i]*b[i];
63   for (int k = Ncells-1; k >= 0; --k) {
64     for (int i = k+1; i < Ncells; ++i)
65       b[k] -= LU[k][i]*b[i];
66     b[k] /= LU[k][k];
67   }
68 }
69
70 #ifdef DEBUG
71 // 行列・ベクトル積 b = A*x
72 void Matvec(double const A[][Ncells], double const x[], double b[])
73 {
74   for (int i = 0; i < Ncells; ++i) {
75     b[i] = 0.;
76     for (int j = 0; j < Ncells; ++j)
77       b[i] += A[i][j]*x[j];
78   }
79 }
80 #endif  // DEBUG
81 int main()
82 {
83   // 各セルの大きさ
84   double dx = Lx / Nx;
85   double dy = Ly / Ny;
86
87   // 動粘性係数をReから計算
88   double nu = U*Lx/Re;
89
90   double t = dt;
```

```
91
92    // Poisson方程式の係数行列を作る
93    // Mは0で初期化されている前提
94    for (int i = 0; i <= Nx - 1; ++i)
95      for (int j = 0; j <= Ny - 1; ++j) {
96        int P =  i   + j   *Nx;
97        int S =  i   +(j-1)*Nx;
98        int W = (i-1)+ j   *Nx;
99        int E = (i+1)+ j   *Nx;
100       int N = i    +(j+1)*Nx;
101       if (j >= 1) {
102         M[P][S] = 1./(dy*dy);
103         M[P][P] -= 1./(dy*dy);
104       }
105       if (i >= 1) {
106         M[P][W] = 1./(dx*dx);
107         M[P][P] -= 1./(dx*dx);
108       }
109       if (i <= Nx - 2) {
110         M[P][E] = 1./(dx*dx);
111         M[P][P] -= 1./(dx*dx);
112       }
113       if (j <= Ny - 2) {
114         M[P][N] = 1./(dy*dy);
115         M[P][P] -= 1./(dy*dy);
116       }
117     }
118 #ifdef DEBUG
119   double OrigM[Ncells][Ncells];
120   for (int i = 0; i <= Nx*Ny - 1; ++i)
121     for (int j = 0; j <= Nx*Ny - 1; ++j)
122       OrigM[i][j] = M[i][j];
123 #endif  // DEBUG
124   // Mを予めLU分解しておく.
```

```
125   LU_decomp(M);
126
127   // 時間進行
128   while (t <= t_end) {
129     std::cout << "Time:␣" << t << std::endl;
130     std::cerr << "Time:␣" << t << std::endl;
131
132     // ステップ1: 流速の予測値の計算
133     // uの予測値upの計算
134     for (int i = 1; i <= Nx - 1; ++i)
135       for (int j = 0; j <= Ny - 1; ++j) {
136         // 対流項 x
137         double uw = 0.5*(u[i-1][j]+u[i][j]);
138         double ue = 0.5*(u[i][j]+u[i+1][j]);
139
140         double uux = ((ue*ue)-(uw*uw))/dx;
141
142         // 対流項 y
143         // 境界条件のことを考えたコード
144         double us, vs, un, vn;
145         if (0 == j) {
146           us = vs = 0.;
147         } else {
148           us = 0.5*(u[i][j-1] + u[i][j]);
149           vs = 0.5*(v[i-1][j] + v[i][j]);
150         }
151         if (Ny - 1 == j) {
152           un = U;
153           vn = 0.;
154         } else {
155           un = 0.5*(u[i][j] + u[i][j+1]);
156           vn = 0.5*(v[i-1][j+1] + v[i][j+1]);
157         }
158         double uvy = (un*vn - us*vs)/dy;
```

```
159
160         // 圧力勾配項 x
161         double dpdx = (p[i][j] - p[i-1][j])/dx;
162
163         // 粘性項 x
164         double uxx = (u[i-1][j] - 2*u[i][j] + u[i+1][j])/(dx*dx);
165         double uyy;
166         if (0 == j) {
167           double us = 0.;
168           uyy = (8./3.*us - (8./3.+4./3.)*u[i][j] + 4./3.*u[i][j+1])/(dy*
                dy);
169         } else if (Ny - 1 == j) {
170           double un = U;
171           uyy = (4./3.*u[i][j-1] - (4./3.+8./3.)*u[i][j] + 8./3.*un)/(dy*
                dy);
172         } else
173           uyy = (u[i][j-1] - 2*u[i][j] + u[i][j+1])/(dy*dy);
174
175         up[i][j] = u[i][j] + dt*(-(uux+uvy) -dpdx + nu*(uxx+uyy));
176       }
177
178     // vの予測値vpの計算
179     for (int i = 0; i <= Nx - 1; ++i)
180       for (int j = 1; j <= Ny - 1; ++j) {
181         // 対流項 x
182         double uw, vw, ue, ve;
183         if (0 == i) {
184           uw = vw = 0;
185         } else {
186           uw = 0.5*(u[i][j-1] + u[i][j]);
187           vw = 0.5*(v[i-1][j] + v[i][j]);
188         }
189
190         if (Nx - 1 == i) {
```

```
191             ue = ve = 0;
192           } else {
193             ue = 0.5*(u[i+1][j-1] + u[i+1][j]);
194             ve = 0.5*(v[i][j] + v[i+1][j]);
195           }
196           double vux = (-(vw*uw)+(ve*ue))/dx;
197
198           // 対流項 y
199           double vs = 0.5*(v[i][j-1] + v[i][j]);
200           double vn = 0.5*(v[i][j] + v[i][j+1]);
201
202           double vvy = (-vs*vs + vn*vn)/dy;
203
204           // 圧力勾配項 y
205           double dpdy = (-p[i][j-1] + p[i][j])/dy;
206
207           // 粘性項 y
208           double vxx;
209           if (0 == i) {
210             double vw = 0.;
211             vxx = (8./3.*vw - (8./3.+4./3)*v[i][j] + 4./3.*v[i+1][j])/(dx*
                    dx);
212           } else if (Nx - 1 == i) {
213             double ve = 0;
214             vxx = (4./3.*v[i-1][j] - (8./3.+4./3)*v[i][j] + 8./3.*ve)/(dx*
                    dx);
215           } else
216             vxx = (v[i-1][j] - 2*v[i][j] + v[i+1][j])/(dx*dx);
217           double vyy = (v[i][j-1] - 2*v[i][j] + v[i][j+1])/(dy*dy);
218
219           vp[i][j] = v[i][j] + dt*(-(vux+vvy) -dpdy + nu*(vxx+vyy));
220         }
221
222       // ステップ2: 圧力の修正量を計算
```

```
223     // 右辺ベクトルの構築
224     for (int i = 0; i <= Nx - 1; ++i)
225       for (int j = 0; j <= Ny - 1; ++j)
226         b[i+j*Nx] = ((-up[i][j]+up[i+1][j])/dx + (-vp[i][j]+vp[i][j+1])/
                dy)/dt;
227
228 #ifdef DEBUG
229     double RHS[Ncells];
230     for (int i = 0; i < Ncells; ++i)
231       RHS[i] = b[i];
232 #endif  // DEBUG
233
234     // 線型方程式を解く.
235     LU_solve(M, b);
236
237 #ifdef DEBUG
238     // 根のテスト
239     double prod[Ncells];
240     Matvec(OrigM, b, prod);
241     double sum_diff = 0.;
242     for (int i = 0; i < Ncells; ++i)
243       sum_diff += (RHS[i]-prod[i])*(RHS[i]-prod[i]);
244     std::cerr << "RMS␣of␣Diff␣==␣" << sqrt(sum_diff / Ncells) << std::
            endl;
245 #endif  // DEBUG
246
247     // b --> phiにデータをコピー（無駄!!）
248     for (int i = 0; i <= Nx - 1; ++i)
249       for (int j = 0; j <= Ny - 1; ++j)
250         phi[i][j] = b[i+j*Nx];
251
252     // ステップ3: 圧力と流速の修正
253     for (int i = 0; i <= Nx - 1; ++i)
254       for (int j = 0; j <= Ny - 1; ++j)
```

```
255         p[i][j] += phi[i][j];
256
257     for (int i = 1; i <= Nx - 1; ++i)
258       for (int j = 0; j <= Ny - 1; ++j)
259         u[i][j] = up[i][j] - dt*(-phi[i-1][j] + phi[i][j])/dx;
260
261     for (int i = 0; i <= Nx - 1; ++i)
262       for (int j = 1; j <= Ny - 1; ++j)
263         v[i][j] = vp[i][j] - dt*(-phi[i][j-1] + phi[i][j])/dy;
264
265     // 計算結果の出力
266     // P
267     std::cout << "P" << std::endl;
268     for (int i = 0; i <= Nx - 1; ++i)
269       for (int j = 0; j <= Ny - 1; ++j)
270         std::cout << p[i][j] << std::endl;
271
272     // U
273     std::cout << "U" << std::endl;
274     for (int i = 1; i <= Nx - 1; ++i)
275       for (int j = 0; j <= Ny - 1; ++j)
276         std::cout << u[i][j] << std::endl;
277
278     // V
279     std::cout << "V" << std::endl;
280     for (int i = 0; i <= Nx - 1; ++i)
281       for (int j = 1; j <= Ny - 1; ++j)
282         std::cout << v[i][j] << std::endl;
283
284 #ifdef DEBUG
285     // 連続の式に対する誤差のチェック
286     double sum_voldiff = 0;
287     for (int i = 0; i <= Nx - 1; ++i)
288       for (int j = 0; j <= Ny - 1; ++j) {
```

```
289             double voldiff = (u[i][j] - u[i+1][j])*dy + (v[i][j] - v[i][j
                    +1])*dx;
290             sum_voldiff += voldiff*voldiff;
291           }
292         std::cerr << "RMS error of Continuity: " << sqrt(sum_voldiff / Ncells
                ) << std::endl;
293 #endif  // DEBUG
294
295         t = t + dt;
296       }
297
298       return 0;
299 }
```

付録 B

データを VTK 形式に変換する Python プログラム

ソースコード B.1　データを VTK 形式に変換するプログラム

```
1  #!/usr/bin/env python
2  # data2vtk.py - convert kartesian data to VTK files
3
4  import sys, re
5
6  Nx = 16
7  Ny = 16
8
9  dx = 1./Nx
10 dy = 1./Ny
11
12 def write_flow_field(dstname, srcname, t, p, u, v):
13     "Write data in VTK format to a file"
14     f = open(dstname, 'w')
15     f.write("# vtk DataFile Version 2.0\n")
16     f.write("Flow field generated from %s, time: %g\n" % (srcname, t))
17     f.write("ASCII\n")
18     f.write("DATASET STRUCTURED_GRID\n")
19     f.write("DIMENSIONS %d %d 1\n" % (Nx, Ny))
20     f.write("POINTS %d float\n" % (Nx*Ny))
```

```
21
22      # Coordinates
23      for i in range(Nx):
24          for j in range(Ny):
25              xc = (i+0.5) * dx
26              yc = (j+0.5) * dy
27              f.write("%g %g 0\n" % (xc, yc))
28
29      f.write("POINT_DATA %d \n" % (Nx*Ny))
30      f.write("SCALARS P float 1\n")
31      f.write("LOOKUP_TABLE default\n")
32      for i in range(Nx):
33          for j in range(Ny):
34              pc = p[i*Ny+j]
35              f.write("%g\n" % pc)
36
37      f.write("VECTORS U float\n")
38      for i in range(Nx):
39          for j in range(Ny):
40              if i == 0:
41                  uw = 0
42              else:
43                  uw = u[(i-1)*Ny+j]
44              if i == Nx - 1:
45                  ue = 0
46              else:
47                  ue = u[i*Ny+j]
48              if j == 0:
49                  vs = 0
50              else:
51                  vs = v[i*(Ny-1)+(j-1)]
52              if j == Ny - 1:
53                  vn = 0
54              else:
```

```
55                  vn = v[i*(Ny-1)+j]
56              f.write("%g %g 0\n" % (0.5*(uw+ue), 0.5*(vs+vn)))
57      f.close()
58
59  argvs = sys.argv
60  argc = len(argvs)
61
62  if argc != 2:
63      sys.exit('Usage: python %s datafile' % argvs[0])
64
65  ##############################################################
66  # Read flow data
67  ##############################################################
68  f = open(argvs[1])
69  lines = f.readlines()
70  f.close()
71
72  mode = 'T'
73  p = []
74  u = []
75  v = []
76  curtime = 0.
77  count = 0
78  write_results = False
79  for line in lines:
80      if re.match('^\#', line):
81          continue
82      if mode == 'T':
83          if re.match('^Time:', line):
84              curtime = float(line[6:])
85              if re.match('^\d+(\.\d)?$', line[6:]):
86                  write_results = True
87                  print "Time:", curtime
88              # else:
```

```
89              #      print "\tTime:", curtime
90          else:
91              sys.exit('Time␣is␣expected␣but␣read␣\"%s\"' % line)
92          mode = 'PL'
93      elif mode == 'PL':
94          if not re.match('^P\s*$', line):
95              sys.exit('P␣label␣is␣expected␣but␣read␣\"%s\"' % line)
96          mode = 'P'
97      elif mode == 'P':
98          p.append(float(line))
99          count += 1
100         if count == Nx*Ny:
101             count = 0
102             mode = 'UL'
103     elif mode == 'UL':
104         if not re.match('^U\s*$', line):
105             sys.exit('U␣label␣is␣expected␣but␣read␣\"%s\"' % line)
106         mode = 'U'
107     elif mode == 'U':
108         u.append(float(line))
109         count += 1
110         if count == (Nx-1)*Ny:
111             count = 0
112             mode = 'VL'
113     elif mode == 'VL':
114         if not re.match('^V\s*$', line):
115             sys.exit('V␣label␣is␣expected␣but␣read␣\"%s\"' % line)
116         mode = 'V'
117     elif mode == 'V':
118         v.append(float(line))
119         count += 1
120         if count == Nx*(Ny-1):
121             if write_results == True:
122                 dstfname = 'vtk/data%04d.vtk' % (curtime * 10)
```

```
123                 write_flow_field(dstfname, argvs[1], curtime, p, u, v)
124                 write_results = False
125             count = 0
126             mode = 'T'
127             p = []
128             u = []
129             v = []
130     else:
131         sys.exit('Unsupported␣line␣\"%s\"' % line)
132 #eof
```

索引