J. KSIAM Vol.19, No.2, 103–121, 2015 http://dx.doi.org/10.12941/jksiam.2015.19.103

[SPECIAL SECTION : **Software and Education**]

NUMERICAL IMPLEMENTATION OF THE TWO-DIMENSIONAL INCOMPRESSIBLE NAVIER–STOKES EQUATION

YONGHO CHOI, DARAE JEONG, SEUNGGYU LEE AND JUNSEOK KIM†

DEPARTMENT OF MATHEMATICS, KOREA UNIVERSITY, SEOUL 136-713, REPUBLIC OF KOREA

ABSTRACT. In this paper, we briefly review and describe a projection algorithm for numerically computing the two-dimensional time-dependent incompressible Navier–Stokes equation. The projection method, which was originally introduced by Alexandre Chorin [A.J. Chorin, Numerical solution of the Navier–Stokes equations, Math. Comput., 22 (1968), pp. 745–762], is an effective numerical method for solving time-dependent incompressible fluid flow problems. The key advantage of the projection method is that we do not compute the momentum and the continuity equations at the same time, which is computationally difficult and costly. In the projection method, we compute an intermediate velocity vector field that is then projected onto divergence-free fields to recover the divergence-free velocity. Numerical solutions for flows inside a driven cavity are presented. We also provide the source code for the programs so that interested readers can modify the programs and adapt them for their own purposes.

1. INTRODUCTION

The time-dependent Navier–Stokes (NS) equations, which describe the motion of viscous fluids, for the two-dimensional incompressible fluids are given as:

$$\rho\left(u_t + uu_x + vu_y\right) = -p_x + \eta\Delta u, \tag{1.1}$$

$$\rho\left(v_t + uv_x + vv_y\right) = -p_y + \eta\Delta v, \tag{1.2}$$

$$u_x + v_y = 0, \tag{1.3}$$

where ρ is the density, $(u(x,y,t), v(x,y,t))$ is the velocity field, $p(x,y,t)$ is the hydrodynamic pressure, and η is the dynamic viscosity. Equation (1.3) represents an incompressible fluid [19]. Equations (1.1) – (1.3) can be rewritten as Eqs. (1.4) and (1.5) after non-dimensionalization.

$$\mathbf{u}_t + \mathbf{u}\cdot\nabla\mathbf{u} = -\nabla p + \frac{1}{Re}\Delta\mathbf{u}, \tag{1.4}$$

$$\nabla\cdot\mathbf{u} = 0, \tag{1.5}$$

Received by the editors May 18 2015; Revised June 2 2015; Accepted in revised form June 2 2015; Published online June 10 2015.

2000 *Mathematics Subject Classification.* 93B05.

Key words and phrases. Navier–Stokes equation, projection method, multigrid method.

† Corresponding author. Tel.:+82 2 3290 3077, http://math.korea.ac.kr/∼cfdkim/, cfdkim@korea.ac.kr.

where $\mathbf{u} = (u, v)$, Re is the Reynolds number which is defined as $Re = \rho U_c L_c/\eta$ and U_c is a characteristic velocity, and L_c is a characteristic length.

The main purpose of the present paper is to review and describe a projection algorithm for numerically computing the incompressible NS Eqs. (1.1)–(1.3). The projection method, which was originally introduced by Alexandre Chorin [4], is a fast and efficient numerical method for solving the unsteady NS equations. To solve the momentum Eq. (1.4) and the continuity Eq. (1.5) as a system at the same time is computationally difficult and costly. However, in the projection method, first, we compute an intermediate velocity vector field, which is generally not divergence-free vector field. Second, we then project the intermediate velocity field onto divergence-free field to recover the divergence-free velocity.

The projection method can be classified into two broad categories, namely the pressure-correction method and the velocity-correction method. The pressure-correction method, sometimes called by fractional step method or Chorin's method, is time-marching techniques composed of two sub-steps for each time step: the pressure is first treated explicitly and then is corrected by projecting the intermediate velocity onto the divergence-free vector field. See [4, 18, 24, 25, 27] for more detail. The velocity-correction method is to change the role of the velocity and the pressure in the pressure-correction schemes: the viscous term is first explicitly and then the velocity is corrected. See [11, 12, 17, 21] for more detail.

After Chorin's projection method, many variants of the projection method have been proposed. Kim and Moin [18] addressed that the conventional use of velocity boundary conditions for the intermediate velocity field can lead to inconsistent numerical solutions. They derived appropriate boundary conditions for the intermediate velocity field. Instead of imposing a discrete form of the divergence-free constraint, we only approximately impose the constraint; that is, the velocity field is not exactly divergence-free vector field [1]. In [3], the authors developed a second-order projection method for the incompressible NS equations. In [29], the author proposed a fourth-order approximate projection method for numerically solving the incompressible NS equations using structured adaptive mesh refinement. See [10] for more detail review.

The contents of this paper are organized as follows. Section 2 describes the projection solution algorithm for the NS equations. We also present Helmholtz–Hodge decomposition and a linear multigrid algorithm. In Section 3, we perform numerical experiments for flows inside a driven cavity. Conclusion is given in Section 4. Finally, we also provide the source code for the programs so that interested readers can modify the programs and adapt them for their own purposes in Appendix.

2. NUMERICAL SOLUTION

The fundamental idea of the projection method is based on the Helmholtz–Hodge decomposition: A vector field can be uniquely decomposed into a divergence-free vector field and a curl-free vector field. The projection algorithm consists of two steps. In the first step, an intermediate velocity field that does not satisfy the divergence-free condition is solved. In the

second step, the intermediate velocity is decomposed into the divergence-free next time velocity and the pressure field.

Let a computational domain be partitioned in Cartesian geometry into a uniform mesh with mesh spacing h. The center of each cell, Ω_{ij}, is located at $(x_i, y_j) = ((i-0.5)h, (j-0.5)h)$ for $i = 1, \cdots, N_x$ and $j = 1, \cdots, N_y$. N_x and N_y are the numbers of cells in the x and y directions, respectively. The cell vertices are located at $(x_{i+\frac{1}{2}}, y_{j+\frac{1}{2}}) = (ih, jh)$. A staggered marker-and-cell (MAC) mesh of Harlow and Welch [16] is used in which pressure and phase fields are stored at cell centers and velocities at cell interfaces. Figure 1 shows the staggered grid.

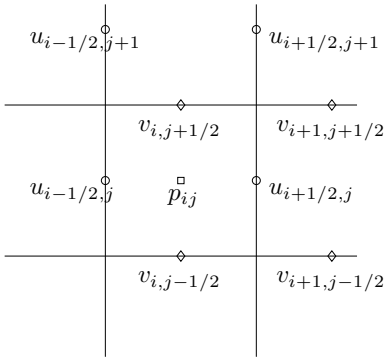


FIGURE 1. Velocities are defined at cell boundaries while the pressure field is defined at the cell centers.

At the beginning of each time step, given $\mathbf{u}^n$, we want to find $\mathbf{u}^{n+1}$ and p^{n+1} which solve the following temporal discretization of the dimensionless form of Eqs. (1.4) and (1.5) of motion:

$$\begin{aligned} \frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} &= -(\mathbf{u} \cdot \nabla_d \mathbf{u})^n - \nabla_d p^{n+1} + \frac{1}{Re} \Delta_d \mathbf{u}^n, \\ \nabla_d \cdot \mathbf{u}^{n+1} &= 0. \end{aligned}$$

The outline of the main procedures in one time step follows:

Step 1. Initialize $\mathbf{u}^0$ to be the divergence-free velocity field.

Step 2. Solve an intermediate velocity field, $\tilde{\mathbf{u}}$, which generally does not satisfy the incompressible condition, without the pressure gradient term,

$$\frac{\tilde{\mathbf{u}} - \mathbf{u}^n}{\Delta t} = -\mathbf{u}^n \cdot \nabla_d \mathbf{u}^n + \frac{1}{Re} \Delta_d \mathbf{u}^n.$$

The resulting finite difference equations are written out explicitly. They take the form

$$\begin{aligned} \tilde{u}_{i+\frac{1}{2},j} &= u^n_{i+\frac{1}{2},j} - \Delta t (u u_x + v u_y)^n_{i+\frac{1}{2},j} \\ &\quad + \frac{\Delta t}{h^2 Re} \left(u^n_{i+\frac{3}{2},j} + u^n_{i-\frac{1}{2},j} - 4u^n_{i+\frac{1}{2},j} + u^n_{i+\frac{1}{2},j+1} + u^n_{i+\frac{1}{2},j-1} \right), \end{aligned} \tag{2.1}$$

$$\tilde{v}_{i,j+\frac{1}{2}} = v^n_{i,j+\frac{1}{2}} - \Delta t(uv_x + vv_y)^n_{i,j+\frac{1}{2}} + \frac{\Delta t}{h^2 Re}\left(v^n_{i+1,j+\frac{1}{2}} + v^n_{i-1,j+\frac{1}{2}} - 4v^n_{i,j+\frac{1}{2}} + v^n_{i,j+\frac{3}{2}} + v^n_{i,j-\frac{1}{2}}\right), \tag{2.2}$$

where the advection terms, $(uu_x + vu_y)^n_{i+\frac{1}{2},j}$ and $(uv_x + vv_y)^n_{i,j+\frac{1}{2}}$, are defined by

$$(uu_x + vu_y)^n_{i+\frac{1}{2},j} = u^n_{i+\frac{1}{2},j}\bar{u}^n_{x_{i+\frac{1}{2},j}} + \frac{v^n_{i,j-\frac{1}{2}} + v^n_{i+1,j-\frac{1}{2}} + v^n_{i,j+\frac{1}{2}} + v^n_{i+1,j+\frac{1}{2}}}{4}\bar{u}^n_{y_{i+\frac{1}{2},j}},$$

$$(uv_x + vv_y)^n_{i,j+\frac{1}{2}} = \frac{u^n_{i-\frac{1}{2},j} + u^n_{i-\frac{1}{2},j+1} + u^n_{i+\frac{1}{2},j} + u^n_{i+\frac{1}{2},j+1}}{4}\bar{v}^n_{x_{i,j+\frac{1}{2}}} + v^n_{i,j+\frac{1}{2}}\bar{v}^n_{y_{i,j+\frac{1}{2}}}.$$

The values $\bar{u}^n_{x_{i+\frac{1}{2},j}}$ and $\bar{u}^n_{y_{i+\frac{1}{2},j}}$ are computed using the upwind procedure. The procedure is

$$\bar{u}^n_{x_{i+\frac{1}{2},j}} = \begin{cases} \frac{u^n_{i+\frac{1}{2},j} - u^n_{i-\frac{1}{2},j}}{h} & \text{if } u^n_{i+\frac{1}{2},j} > 0 \\ \frac{u^n_{i+\frac{3}{2},j} - u^n_{i+\frac{1}{2},j}}{h} & \text{otherwise} \end{cases}$$

and

$$\bar{u}^n_{y_{i+\frac{1}{2},j}} = \begin{cases} \frac{u^n_{i+\frac{1}{2},j} - u^n_{i+\frac{1}{2},j-1}}{h} & \text{if } v^n_{i,j-\frac{1}{2}} + v^n_{i+1,j-\frac{1}{2}} + v^n_{i,j+\frac{1}{2}} + v^n_{i+1,j+\frac{1}{2}} > 0 \\ \frac{u^n_{i+\frac{1}{2},j+1} - u^n_{i+\frac{1}{2},j}}{h} & \text{otherwise.} \end{cases}$$

The quantities $\bar{v}^n_{x_{i,j+\frac{1}{2}}}$ and $\bar{v}^n_{y_{i,j+\frac{1}{2}}}$ are computed in a similar manner.

In the projection method, we use the Helmholtz–Hodge decomposition: A vector field $\mathbf{w}$ on Ω can be uniquely decomposed in the form

$$\mathbf{w} = \mathbf{u} + \nabla p, \tag{2.3}$$

where $\mathbf{u}$ has zero divergence and $\mathbf{u}\cdot\mathbf{n} = 0$ on $\partial\Omega$. The proof of this theorem will be provided in the next section. In our context, we apply the theorem to the intermediate velocity field $\mathbf{w} = \tilde{\mathbf{u}}$, i.e.,

$$\tilde{\mathbf{u}} = \mathbf{u}^{n+1} + \nabla_d(\Delta t p^{n+1}). \tag{2.4}$$

Then, we solve the following equations for the advanced pressure field at the $(n+1)$ time step.

$$\frac{\mathbf{u}^{n+1} - \tilde{\mathbf{u}}}{\Delta t} = -\nabla_d p^{n+1}, \tag{2.5}$$

$$\nabla_d \cdot \mathbf{u}^{n+1} = 0. \tag{2.6}$$

With application of the divergence operator to Eq. (2.5), we find that the Poisson equation for the pressure at the advanced time $(n+1)$ is

$$\Delta_d p^{n+1} = \frac{1}{\Delta t}\nabla_d \cdot \tilde{\mathbf{u}}, \tag{2.7}$$

where we have made use of Eq. (2.6) and the terms are defined as follows:

$$\Delta_d p^{n+1} = \frac{p^{n+1}_{i+1,j} + p^{n+1}_{i-1,j} - 4p^{n+1}_{ij} + p^{n+1}_{i,j+1} + p^{n+1}_{i,j-1}}{h^2}$$
$$\nabla_d \cdot \tilde{\mathbf{u}}_{ij} = \frac{\tilde{u}_{i+\frac{1}{2},j} - \tilde{u}_{i-\frac{1}{2},j}}{h} + \frac{\tilde{v}_{i,j+\frac{1}{2}} - \tilde{v}_{i,j-\frac{1}{2}}}{h}.$$

The boundary condition for the pressure is

$$\mathbf{n} \cdot \nabla_d p^{n+1} = \mathbf{n} \cdot \left(-\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} - (\mathbf{u} \cdot \nabla_d \mathbf{u})^n + \frac{1}{Re} \Delta_d \mathbf{u}^n \right),$$

where $\mathbf{n}$ is the unit normal vector to the domain boundary. We use no-slip and linearity boundary conditions, ($\mathbf{n} \cdot \Delta_d \mathbf{u}^n = 0$), to the domain boundaries. Therefore,

$$\mathbf{n} \cdot \nabla_d p^{n+1} = 0. \tag{2.8}$$

The Poisson equation (2.7) with the homogeneous Neumann boundary condition in Eq. (2.8) dose not have a unique solution. Instead, its solution is unique up to a constant. There are two approaches [6, 7] to make the solution unique as follows: (i) imposing the Dirichlet condition at a single point and (ii) forcing its summation to be zero. Here, we use the following correction

$$p^{n+1}_{ij} = p^{n+1}_{ij} - \frac{1}{N_x N_y} \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} p^{n+1}_{ij}. \tag{2.9}$$

The resulting linear system of Eq. (2.7) with the boundary condition in Eq. (2.8) is solved using a multigrid method [26]. It is well known that classical iterative methods, such as Gauss–Seidel, Jacobi, SOR, or CG, converge very slowly for solving large linear systems [14]. On the other hand, the multigrid method is to damp strongly the oscillatory error components [13] and its solution is obtained in $O(N)$ time, where N is the total number of grid points [8].

Then the divergence-free normal velocities u^{n+1} and v^{n+1} are defined by

$$\mathbf{u}^{n+1} = \tilde{\mathbf{u}} - \Delta t \nabla_d p^{n+1}, \text{ i.e.,}$$
$$u^{n+1}_{i+\frac{1}{2},j} = \tilde{u}_{i+\frac{1}{2},j} - \frac{\Delta t}{h}(p^{n+1}_{i+1,j} - p^{n+1}_{ij}), \quad v^{n+1}_{i,j+\frac{1}{2}} = \tilde{v}_{i,j+\frac{1}{2}} - \frac{\Delta t}{h}(p^{n+1}_{i,j+1} - p^{n+1}_{ij}).$$

These complete the one time step of the NS projection scheme.

2.1. **Helmholtz–Hodge decomposition.** A vector field $\mathbf{w}$ on Ω can be uniquely decomposed in the form

$$\mathbf{w} = \mathbf{u} + \nabla p, \tag{2.10}$$

where $\mathbf{u}$ has zero divergence and $\mathbf{u} \cdot \mathbf{n} = 0$ on $\partial\Omega$. The proof of Eq. (2.10) is as follows [5]: Let p be defined by a solution to the Neumann problem

$$\Delta p = \nabla \cdot \mathbf{w} \text{ in } \Omega, \text{ with } \nabla p \cdot \mathbf{n} = \mathbf{w} \cdot \mathbf{n} \text{ on } \partial\Omega, \tag{2.11}$$

where $\mathbf{n}$ is normal vector (see [2] for the existence and uniqueness up to an additive constant of the solution). Then

$$\mathbf{u} = \mathbf{w} - \nabla p,\ \nabla \cdot \mathbf{u} = \nabla \cdot \mathbf{w} - \Delta p = 0,\ \text{and}\ \mathbf{u} \cdot \mathbf{n} = \mathbf{w} \cdot \mathbf{n} - \nabla p \cdot \mathbf{n} = 0. \tag{2.12}$$

These imply the existence. Next, we need to show the uniqueness. To show the uniqueness of the decomposition, suppose $\mathbf{w} = \mathbf{u}_1 + \nabla p_1 = \mathbf{u}_2 + \nabla p_2$. Then, we have

$$\mathbf{u}_1 - \mathbf{u}_2 + \nabla p_1 - \nabla p_2 = 0. \tag{2.13}$$

Next, taking the inner product with $\mathbf{u}_1 - \mathbf{u}_2$ to Eq. (2.13) and integrating, we have

$$\int_\Omega |\mathbf{u}_1 - \mathbf{u}_2|^2 \, dV + \int_\Omega (\mathbf{u}_1 - \mathbf{u}_2) \cdot (\nabla p_1 - \nabla p_2) \, dV = 0. \tag{2.14}$$

$$\int_\Omega \mathbf{u} \cdot \nabla p \, dV = \int_\Omega \left[\nabla \cdot (p\mathbf{u}) - p \nabla \cdot \mathbf{u}\right] \, dV = \int_{\partial\Omega} p\mathbf{u} \cdot \mathbf{n} dS = 0, \tag{2.15}$$

where have used $\nabla \cdot \mathbf{u} = 0$, the divergence theorem, and the boundary condition for $\mathbf{u}$. Therefore, Eq. (2.14) becomes

$$\int_\Omega |\mathbf{u}_1 - \mathbf{u}_2|^2 \, dV = 0. \tag{2.16}$$

Consequently we obtain $\mathbf{u}_1 = \mathbf{u}_2$ and $\nabla p_1 = \nabla p_2$.

2.2. **Linear multigrid V-cycle algorithm.** In this section we describe the algorithm of the linear multigrid method for solving the discrete system in Eq. (2.7). In order to explain clearly the steps taken during a single V-cycle, we focus on a numerical solution on a 8×8 mesh. We define discrete domains, $\Omega_3,\ \Omega_2,\ \Omega_1$, and Ω_0, where

$$\Omega_k = \{(x_{k,i} = (i - 0.5)h_k, y_{k,j} = (j - 0.5)h_k) | 1 \le i, j \le 2^{k+1} \text{ and } h_k = 2^{3-k}h\}.$$

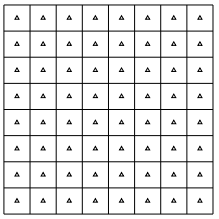
Ω_{k-1} is coarser than Ω_k by a factor of 2. The multigrid solution of the discrete Eq. (2.7) makes use of a hierarchy of meshes ($\Omega_3,\ \Omega_2,\ \Omega_1$, and Ω_0) created by successively coarsening the original mesh, Ω_3 as shown in Fig. 2. A pointwise Gauss–Seidel relaxation scheme is used as the smoother in the multigrid method. The algorithm of the multigrid method for solving Eq. (2.7) is as follows. We rewrite the Eq. (2.7) by

$$L_3(p_{3,ij}^{n+1}) = f_{3,ij} \text{ on } \Omega_3, \tag{2.17}$$

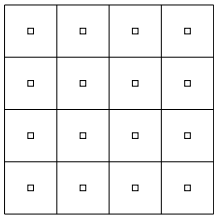
where

$$L_3(p_{3,ij}^{n+1}) = \Delta_d p_{3,ij}^{n+1} \text{ and } f_{3,ij} = \frac{1}{\Delta t} \nabla_d \cdot \tilde{\mathbf{u}}_{3,ij}^n.$$

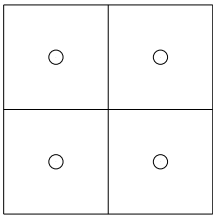
Given the numbers, ν_1 and ν_2, of pre- and post- smoothing relaxation sweeps, an iteration step for the multigrid method using the V-cycle is formally written as follows [26]. That is, starting an initial condition p_3^0, we want to find p_3^n for $n = 1, 2, \cdots$. Given p_3^n, we want to find the p_3^{n+1} solution that satisfies Eq. (2.7). At the very beginning of the multigrid cycle the solution from the previous time step is used to provide an initial guess for the multigrid procedure. First, let $p_3^{n+1,0} = p_3^n$.



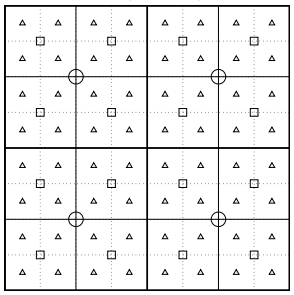
(a) Ω_2 (8×8) h



(b) Ω_1 (4×4) $2h$



(c) Ω_0 (2×2) $4h$



(d)

FIGURE 2. (a), (b), and (c) are a sequence of coarse grids starting with $h = L/N_x$. (d) is a composition of grids, $\Omega_2,\ \Omega_1$, and Ω_0.

Multigrid cycle

$$p_k^{n+1,m+1} = MGcycle(k, p_k^{n+1,m}, L_k, f_k, \nu_1, \nu_2).$$

That is, $p_k^{n+1,m}$ and $p_k^{n+1,m+1}$ are the approximations of p_k^{n+1} before and after an MGcycle. Now, define the MGcycle.

Step 1) Presmoothing

$$\bar{p}_k^{n+1,m} = SMOOTH^{\nu_1}(p_k^{n+1,m}, L_k, f_k),$$

means performing ν_1 smoothing steps with the initial approximation $p_k^{n+1,m}$, source terms f_k, and a $SMOOTH$ relaxation operator to get the approximation $\bar{p}_k^{n+1,m}$. Here, we derive the smoothing operator in two dimensions.

Now we derive a Gauss–Seidel relaxation operator. First, we rewrite Eq. (2.17) as

$$p_{k,ij}^{n+1} = \left[-f_{k,ij} + \frac{p_{i+1,j}^{n+1} + p_{i-1,j}^{n+1} + p_{i,j-1}^{n+1} + p_{i,j-1}^{n+1}}{h^2} \right] \Bigg/ \left(\frac{4}{h^2} \right). \tag{2.18}$$

Next, we replace $p_{k,\alpha\beta}^{n+1}$ in Eq. (2.18) with $\bar{p}_{k,\alpha\beta}^{n+1,m}$ if $(\alpha < i)$ or $(\alpha = i$ and $\beta \leq j)$, otherwise with $p_{k,\alpha\beta}^{n+1,m}$, i.e.,

$$\bar{p}_{k,ij}^{n+1,m} = \left[-f_{k,ij} + \frac{p_{i+1,j}^{n+1,m} + \bar{p}_{i-1,j}^{n+1,m} + p_{i,j+1}^{n+1,m} + \bar{p}_{i,j-1}^{n+1,m}}{h^2} \right] \Bigg/ \left(\frac{4}{h^2} \right). \quad (2.19)$$

Therefore, in a multigrid cycle, one smooth relaxation operator step consists of solving Eq. (2.19) given above for $1 \le i \le 2^{k-3}N_x$ and $1 \le j \le 2^{k-3}N_y$.

Step 2) Coarse grid correction

• Compute the defect: $\bar{d}_k^m = f_k - L_k(\bar{p}_k^{n+1,m})$.

• Restrict the defect and $\bar{p}_k^m$: $\bar{d}_{k-1}^m = I_k^{k-1}\bar{d}_k^m$

The restriction operator I_k^{k-1} maps k-level functions to $(k-1)$-level functions.

$$d_{k-1}(x_i, y_j) = I_k^{k-1} d_k(x_i, y_j) = \frac{1}{4}[d_k(x_{i-\frac{1}{2}}, y_{j-\frac{1}{2}}) + d_k(x_{i-\frac{1}{2}}, y_{j+\frac{1}{2}}) + d_k(x_{i+\frac{1}{2}}, y_{j-\frac{1}{2}}) + d_k(x_{i+\frac{1}{2}}, y_{j+\frac{1}{2}})].$$

• Compute an approximate solution $\hat{p}_{k-1}^{n+1,m}$ of the coarse grid equation on Ω_{k-1}, i.e.,

$$L_{k-1}(p_{k-1}^{n+1,m}) = \bar{d}_{k-1}^m. \quad (2.20)$$

If $k = 1$, we use a direct or fast iteration solver for Eq. (2.20). If $k > 1$, we solve Eq. (2.20) approximately by performing k-grid cycles using the zero grid function as an initial approximation:

$$\hat{v}_{k-1}^{n+1,m} = MGcycle(k-1, 0, L_{k-1}, \bar{d}_{k-1}^m, \nu_1, \nu_2).$$

• Interpolate the correction: $\hat{q}_k^{n+1,m} = I_{k-1}^k \hat{q}_{k-1}^{n+1,m}$. Here, the coarse values are simply transferred to the four nearby fine grid points, i.e., $q_k(x_i, y_j) = I_{k-1}^k q_{k-1}(x_i, y_j) = q_{k-1}(x_{i+\frac{1}{2}}, y_{j+\frac{1}{2}})$ for the i and j odd-numbered integers.

• Compute the corrected approximation on Ω_k

$$p_k^{m,\text{ after } CGC} = \bar{p}_k^{n+1,m} + \hat{q}_k^{n+1,m}.$$

Step 3) Postsmoothing: $p_k^{n+1,m+1} = SMOOTH^{\nu_2}(p_k^{m,\text{ after } CGC}, L_k, f_k)$.

This completes the description of a MGcycle. Then, for unique solution, we redefine the pressure using Eq. (2.9) as follows:

$$p_{ij}^{n+1,m+1} = p_{ij}^{n+1,m+1} - \frac{1}{N_x N_y} \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} p_{ij}^{n+1,m+1}. \quad (2.21)$$

One MGcycle step stops if the consequence error $\|p^{n+1,m+1} - p^{n+1,m}\|_\infty$ is smaller than a given tolerance, where

$$\|p\|_\infty = \max_{1 \le i \le N_x, 1 \le j \le N_y} |p_{ij}|.$$

2.3. **Stability condition.** For the stability and accuracy of the numerical solution, Welch *et al.* [28] suggested three stability condition. The first one is

$$\Delta t < \frac{h^2}{4} Re. \tag{2.22}$$

The two conditions are the famous Courant–Friedrichs–Lewy (CFL) conditions as

$$\Delta t < \frac{h}{|u|_{\max}}, \qquad \Delta t < \frac{h}{|v|_{\max}}. \tag{2.23}$$

Here, $|u|_{\max}$ and $|v|_{\max}$ are the maximum absolute values of u and v velocities. These conditions mean that no fluid particle may cross more than mesh spacing h in a given time interval Δt [20, 22, 23]. By three conditions Eqs. (2.22) and (2.23), we can choose the time step size Δt as follows.

$$\Delta t = C \min\left(\frac{h^2}{4} Re, \frac{h}{|u|_{\max}}, \frac{h}{|v|_{\max}}\right), \tag{2.24}$$

where C is constant value in $(0, 1)$ as a safety factor.

3. NUMERICAL EXPERIMENTS

In this section, we consider a lid-driven cavity flow in a two-dimensional domain. Figure 3 shows the computational domain and the boundary conditions for the flow in a driven cavity. The initial velocity is zero inside the domain. Boundary conditions are zero at three walls except the top lid, where $(u, v) = (1, 0)$. Therefore, flow is driven by the upper wall [9].

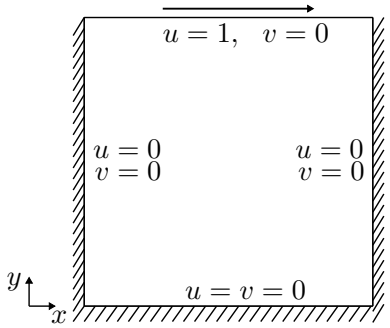


FIGURE 3. Schematic illustration of the lid-driven cavity flow

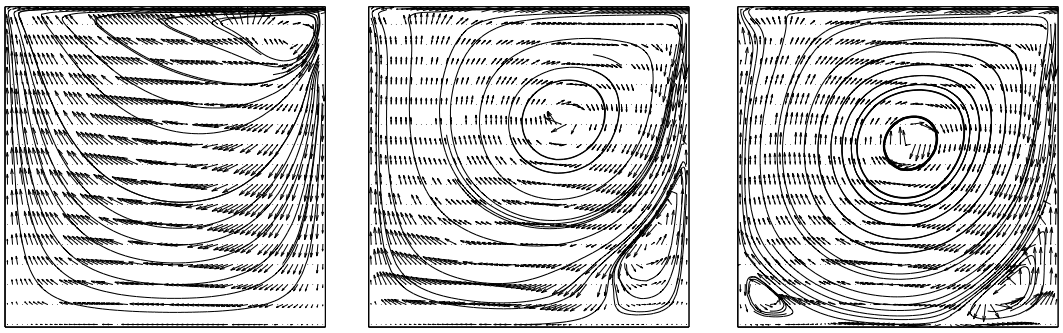
3.1. **Convergence test.** We demonstrate the convergence of the velocities u and v on $\Omega = (0, 1)\times(0, 1)$. To calculate the convergence rate of the numerical scheme, we perform a number of simulations on a set of increasingly finer grids. The numerical velocities are computed on the grid sizes, $h = 1/2^{n+1}$ for $n = 5, 6, 7, 8$, and 9. For each case, the calculation is run up to time $T = 0.2$ with the time step size $\Delta t = 0.032h$ and $Re = 100$. We define the error of a grid solution as the discrete l_2-norm of the difference between that grid and the average of the next

finer grid cells covering it, for example, $e_{h/\frac{h}{2}ij} := u_{hij} - (u_{\frac{h}{2}2i,2j} + u_{\frac{h}{2}2i,2j-1})/2$. The rate of convergence is defined as the ratio of successive errors as $\log_2(\|e_{h/\frac{h}{2}}\|_2/\|e_{\frac{h}{2}/\frac{h}{4}}\|_2)$. Table 1 shows the errors and rates of convergence. The numerical results show that the convergence rates of u and v are first-order in space and time as we expect from the first-order upwind scheme.

TABLE 1. Errors and rates of convergence for velocities u and v.

Case	32-64	Rate	64-128	Rate	128-256	Rate	256-512
u	5.81490E-3	1.3264	2.31871E-3	1.1460	1.04779E-3	1.0976	4.89617E-4
v	3.77710E-3	1.1070	1.75359E-3	1.1034	8.16143E-4	1.1006	3.80588E-4

3.2. **Effect of domain size.** We first consider a lid-driven cavity flow in $\Omega = (0,1) \times (0,1)$ [9]. The result is shown in Fig. 4 with $N_x = N_y = 64$, i.e., $h = 1/64$, $Re = 10000$, and $\Delta t = 0.01h^2Re$. We can observe that the eye of principal vortex moves into the core of the cavity and the lower left/right corner-eddies as time evolves.

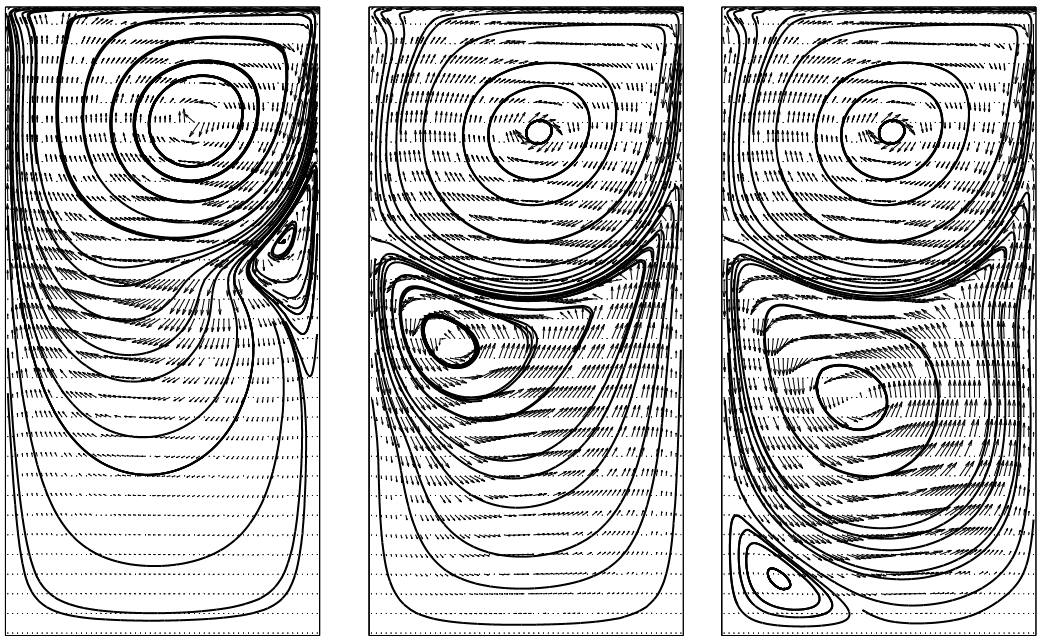


$t = 200\Delta t$ $t = 2000\Delta t$ $t = 100000\Delta t$

FIGURE 4. The evolution of cavity flow on the square domain $\Omega = (0,1) \times (0,1)$. The dimensionless times are shown below each figure.

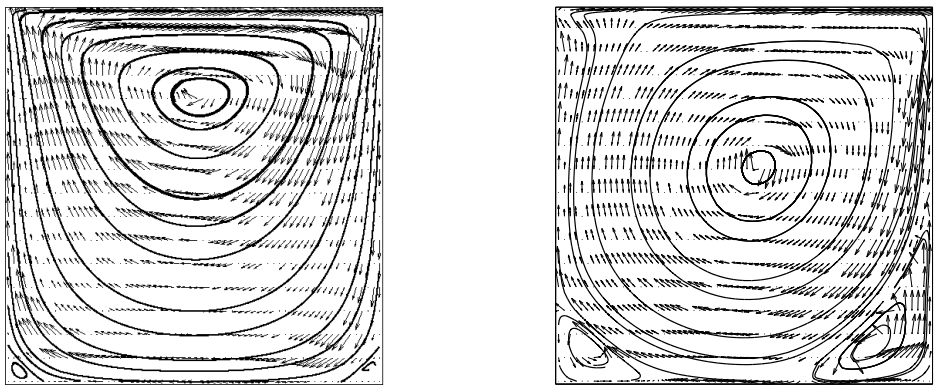
To see the effect of domain size, we perform a numerical simulation on a non-square domain, i.e., $\Omega = (0,1) \times (0,2)$ with mesh size (64×128) [15]. We use $h = 1/64$, $Re = 10000$, and $\Delta t = 0.01h^2Re$. Figure 5 shows the numerical results of the lid-driven cavity flow at each time on the rectangle domain. Unlike the results in Fig. 4, we observe that two principal vortices with an opposite directional rotation at the lower corner.

3.3. **Effect of Re number.** Finally, we investigate the effect of Re number. On the computational domain $\Omega = (0,1) \times (0,1)$, we use the parameters $N_x = 64$, $N_y = 64$, i.e., $h = 1/64$, $\Delta t = 0.01h^2Re$, and total simulation time $T = 100000\Delta t$. Figure 6 shows velocity fields of lid-driven cavity flows with different Re numbers at the final time. Here, (a) and (b) are the results with $Re = 10$ and 10000, respectively. The center of vortex moves downward when the Reynolds number is getting larger.



$t = 2000\Delta t$ $t = 5000\Delta t$ $t = 100000\Delta t$

FIGURE 5. The evolution of cavity flow on the rectangle domain $\Omega = (0, 1) \times (0, 2)$. Each simulation time shows on the bottom of figures. Moving the flows is represented by arrow and the overall flow of the fluid are displayed by streamline.



(a) $Re = 10$ (b) $Re = 10000$

FIGURE 6. Velocity field of a lid-driven cavity flow at steady state with different Re numbers. Here, (a) and (b) are the results with $Re = 10$ and 10000, respectively.

4. Conclusion

In this article, we briefly reviewed and described the projection algorithm for numerically computing the two-dimensional time-dependent incompressible NS equation. The projection method, which was originally introduced by Alexandre Chorin [4], is an efficient numerical method for solving the unsteady incompressible fluid flow problems and has been widely used by many researchers. Solving the momentum and the continuity equations at the same time is computationally difficult and costly. In the original projection method, we compute an intermediate velocity vector field without pressure gradient, and then we project the temporary velocity onto divergence-free fields to recover the divergence-free velocity. As a standard test problem, we considered a driven cavity flow. We also provided the source code for the programs so that interested readers can modify the programs and adapt them for their own purposes. As a future work, three-dimensional extension would be useful because most of real world problems are three-dimensional space.

Acknowledgments

The corresponding author (J.S. Kim) was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIP) (NRF-2014R1A2A2A01003683).

References

[1] A.S. Almgren, J.B. Bell, and W.G. Szymczak, *A numerical method for the incompressible Navier–Stokes equations based on an approximate projection*, SIAM J. Sci. Comput., **17**(2) (1996), 358–369.
[2] V. Barbu, *Partial Differential Equations and Boundary Value Problems*, Kluwer, Dordrecht, 1998.
[3] J.B. Bell, P. Colella, and H.M. Glaz, *A second-order projection method for the incompressible Navier-Stokes equations*, J. Comput. Phys., **85**(2) (1989), 257–283.
[4] A.J. Chorin, *Numerical solution of the Navier–Stokes equations*, Math. Comput., **22** (1968), 745–762.
[5] A.J. Chorin, J.E. Marsden, and J.E. Marsden, *A Mathematical Introduction to Fluid Mechanics*, Springer, New York, 1990.
[6] M.O. Deville, P.F. Fischer, and E.H. Mund, *High-order Methods for Incompressible Fluid Flow*, **9** Cambridge University Press, 2002.
[7] J.A. Escobar-Vargas, P.J. Diamessis, and C.F. Van Loan, *The numerical solution of the pressure Poisson equation for the incompressible Navier–Stokes equations using a quadrilateral spectral multidomain penalty method*, J. Comp. Phys.(submitted) (2011).
[8] S.R. Fulton, P.E. Ciesielski, and W.H. Schubert, *Multigrid methods for elliptic problems: A review*, Mon. Weather Rev., **114**(5) (1986), 943–959.
[9] M. Griebel, T. Dornseifer, and T. Neunhoeffer, *Numerical Simulation in Fluid Dynamics: a Practical Introduction*, SIAM, Philadelphia, 1997.
[10] J.L. Guermond, P. Minev, and J. Shen, *An overview of projection methods for incompressible flows*, Comput. Methods Appl. Mech. Engrg., **195**(44) (2006), 6011–6045.
[11] J.L. Guermond and J. Shen, *Quelques résultats nouveaux sur les méthodes de projection*, C. R. Acad. Sci. Paris, **333**(12) (2001), 1111–1116.
[12] J.L. Guermond and J. Shen, *Velocity-correction projection methods for incompressible flows*, SIAM J. Numer. Anal., **41**(1) (2003), 112–134.
[13] T. Guillet and R. Teyssier, *A simple multigrid scheme for solving the Poisson equation with arbitrary domain boundaries*, J. Comput. Phys., **230**(12) (2011), 4756–4771.

[14] M.M. Gupta and J. Zhang, *High accuracy multigrid solution of the 3D convection-diffusion equation*, Appl. Math. Comput., **113**(2) (2000), 249–274.
[15] K. Gustafson and K. Halasi, *Cavity flow dynamics at higher Reynolds number and higher aspect ratio*, J. Comput. Phys., **70**(2) (1987), 271–283.
[16] F.H. Harlow and J.E. Welch, *Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface*, Phys. Fluids, **8**(12) (1965), 2182–2189.
[17] G.E. Karniadakis, M. Israeli, and S.A. Orszag, *High-order splitting methods for the incompressible Navier–Stokes equations*, J. Comput. Phys., **97**(2) (1991), 414–443.
[18] J. Kim and P. Moin, *Application of a fractional-step method to incompressible Navier–Stokes equations*, J. Comput. Phys., **59**(2) (1985), 308–323.
[19] L. Landau and E.M. Lifshitz, *Fluid mechanics, Course of Theoretical Physics* **6** (2nd revised ed.), Pergamon Press (1987), 552.
[20] G. Markham and M.V. Proctor, *Modifications to the two-dimensional incompressible fluid flow code ZUNI to provide enhanced performance*, CEGB Report TPRD/L/0063/M82, 1983.
[21] S.A. Orszag, M. Israeli, and M. Deville, *Boundary conditions for incompressible flows*, J. Sci. Comput., **1**(1) (1986), 75–111.
[22] R. Peyret and T.D. Taylor, *Computational Methods for Fluid Flow*, Springer-Verlag, New York, 1985.
[23] P.J. Roache, *Computational Fluid Dynamics*, Hermosa publishers, Albuquerque, 1972.
[24] R. Temam, *Sur l'approximation de la solution des équations de Navier–Stokes par la méthode des pas fractionnaires (II)*, Arch. Rational Mech. Anal., **33**(5) (1969), 377–385.
[25] L.J.P. Timmermans, P.D. Minev, and F.N. Van De Vosse, *An approximate projection scheme for incompressible flow using spectral elements*, Int. J. Numer. Meth. Fluids, **22**(7) (1996), 673–688.
[26] U. Trottenberg, C.W. Oosterlee, and A. Schüller, *Multigrid*, Academic press, London, 2000.
[27] J.J.I.M. Van Kan, *A second-order accurate pressure-correction scheme for viscous incompressible flow*, SIAM J. Sci. Comput., **7**(3) (1986), 870–891.
[28] J.E. Welch, F.H. Harlow, J.P. Shannon, and B.J. Daly, *The MAC (Marker-and-Cell) Method - A computing technique for solving viscous, incompressible, transient fluid-flow problems involving free surfaces, Los Alamos Scientific Laboratory Report LA-3425*, University of California, Los Alamos, 1966.
[29] Q. Zhang, *A fourth-order approximate projection method for the incompressible Navier–Stokes equations on locally-refined periodic domains*, Appl. Numer. Math., **77** (2014), 16–30.

APPENDIX

Following code[1] is for the two-dimensional cavity flow of Fig. 4 and parameters are enumerated in the Table 2.

```
/* Two-dimensional Navier-Stokes equation */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#define gnx 32
#define gny 32
#define iloop for(i=1;i<=gnx;i++)
#define i0loop for(i=0;i<=gnx;i++)
#define jloop for(j=1;j<=gny;j++)
#define j0loop for(j=0;j<=gny;j++)
```

[1] `http://www.ksiam.org/archive/supplement/jksiam-2015v19p103.zip` contains the codes and related files in this section.

Parameters	Description
nx, ny	maximum number of grid points in the x-, y-direction
n_level	number of multigrid level
p_relax	number of times being relax
dt	Δt
xleft, yleft	minimum value on the x-, y-axis
xright, yright	maximum value on the x-, y-axis
ns	number of print out data
max_it	maximum number of iteration
max_it_mg	maximum number of multigrid iteration
tol_mg	tolerance for multigrid
h	space step size
h2	h^2
Re	Reynolds number

TABLE 2. Parameters used for the 2D NS equation.

```
#define ijloop iloop jloop
#define i0jloop i0loop jloop
#define ij0loop iloop j0loop
#define iloopt for(i=1;i<=nxt;i++)
#define i0loopt for(i=0;i<=nxt;i++)
#define jloopt for(j=1;j<=nyt;j++)
#define j0loopt for(j=0;j<=nyt;j++)
#define ijloopt iloopt jloopt
#define i0jloopt i0loopt jloopt
#define ij0loopt iloopt j0loopt
int nx,ny,n_level,p_relax,nt;
double **sor,h,h2,**tu,**tv,**workp,**worku,**workv,**adv_u,
        **adv_v,dt,xleft,xright,yleft,yright,Re;
char bufferu[20],bufferv[20],bufferp[20];
void initialization(double **p,double **u,double **v){
    int i,j;
    ijloop p[i][j] = 0.0; ij0loop v[i][j] = 0.0;
    i0jloop u[i][j] = 0.0;
}
void augmenuv(double **u,double **v,int nx,int ny){
    int i,j;
    double bdvel=1.0;
    iloop {u[i][0] = -u[i][1]; u[i][ny+1] = 2.0*bdvel-u[i][ny];}
    jloop {v[0][j] = -v[1][j]; v[nx+1][j] = -v[nx][j];}
}
void advection_uv(double **u,double **v,double **adv_u,double **adv_v){
    int i,j;
    augmenuv(u,v,nx,ny);
    for (i=1; i<nx; i++) { jloop {
    if (u[i][j]>0.0) {adv_u[i][j] = u[i][j]*(u[i][j]-u[i-1][j])/h;}
    else {adv_u[i][j] = u[i][j]*(u[i+1][j]-u[i][j])/h;}
```

```
    if (v[i][j-1]+v[i+1][j-1]+v[i][j]+v[i+1][j]>0.0)
    adv_u[i][j] += 0.25*(v[i][j-1]+v[i+1][j-1]+v[i][j]+v[i+1][j])
                *(u[i][j]-u[i][j-1])/h;
    else adv_u[i][j] += 0.25*(v[i][j-1]+v[i+1][j-1]+v[i][j]+v[i+1][j])
                *(u[i][j+1]-u[i][j])/h;}}
    iloop { for (j=1; j<ny; j++) {
    if (u[i-1][j]+u[i][j]+u[i-1][j+1]+u[i][j+1]>0.0)
    adv_v[i][j] = 0.25*(u[i-1][j]+u[i][j]+u[i-1][j+1]+u[i][j+1])
                *(v[i][j]-v[i-1][j])/h;
    else adv_v[i][j] = 0.25*(u[i-1][j]+u[i][j]+u[i-1][j+1]+u[i][j+1])
                *(v[i+1][j]-v[i][j])/h;
    if (v[i][j]>0.0) {adv_v[i][j] += v[i][j]*(v[i][j]-v[i][j-1])/h;}
    else {adv_v[i][j] += v[i][j]*(v[i][j+1] - v[i][j])/h;}}}
}
void temp_uv(double **tu,double **tv,double **u,double **v,
        double **adv_u,double **adv_v){
            int i,j;
    for (i=1; i<nx; i++) {jloop {tu[i][j] = u[i][j]+dt*( (u[i+1][j]
    +u[i-1][j]-4.0*u[i][j]+u[i][j+1]+u[i][j-1])/(Re*h2)-adv_u[i][j]);}}
    iloop {for (j=1; j<ny; j++) {tv[i][j] = v[i][j]+dt*( (v[i+1][j]
    +v[i-1][j]-4.0*v[i][j]+v[i][j+1]+v[i][j-1])/(Re*h2)-adv_v[i][j]);}}
}
void mat_copy(double **a,double **b,int xl,int xr,int yl,int yr){
    int i,j;
    for (i=xl;i<=xr;i++) {for (j=yl;j<=yr;j++) {a[i][j]=b[i][j];}}
}
void relax_p(double **p,double **f,int nxt,int nyt){
    int i,j,iter;
    double ht,ht2,coef,src;
    ht2 = pow((xright-xleft) / (double) nxt,2);
    for (iter=1; iter<=p_relax; iter++) {ijloopt {src = f[i][j];
            if (i==1) {src -= p[2][j]/ht2; coef = -1.0/ht2;}
            else if (i==nxt) {src -= p[nxt-1][j]/ht2; coef = -1.0/ht2;}
            else {src -= (p[i+1][j] + p[i-1][j])/ht2; coef = -2.0/ht2;}
            if (j==1) {src -= p[i][2]/ht2; coef += -1.0/ht2;}
            else if (j==nyt) {src -= p[i][nyt-1]/ht2; coef += -1.0/ht2;}
            else {src -= (p[i][j+1] + p[i][j-1])/ht2; coef += -2.0/ht2;}
            p[i][j] = src / coef;}}
}
double **dmatrix(long nrl,long nrh,long ncl,long nch){
    double **m;
    long i,nrow=nrh-nrl+1+1,ncol=nch-ncl+1+1;
    m=(double **) malloc((nrow)*sizeof(double*));
    m+=1; m-=nrl;
    m[nrl]=(double *) malloc((nrow*ncol)*sizeof(double));
    m[nrl]+=1; m[nrl]-=ncl;
    for (i=nrl+1; i<=nrh; i++) m[i]=m[i-1]+ncol;
    return m;
}
void free_dmatrix(double **m,long nrl,long nrh,long ncl,long nch){
    free(m[nrl]+ncl-1); free(m+nrl-1);
}
```

```
void grad_p(double **p,double **dpdx,double **dpdy,int nxt,int nyt){
    int i,j;
    double ht;
    ht = xright / (double) nxt;
    i0jloopt {if (i==0) {dpdx[0][j] = 0.0;}
        else if (i==nxt) {dpdx[nxt][j] = 0.0;}
        else {dpdx[i][j] = (p[i+1][j] - p[i][j])/ht;}}
    ij0loopt {if (j==0) {dpdy[i][0] = 0.0;}
        else if (j==nyt) {dpdy[i][nyt] = 0.0;}
        else {dpdy[i][j] = (p[i][j+1] - p[i][j])/ht;}}
}
void div_uv(double **tu,double **tv,double **divuv,int nxt,int nyt){
    int i,j;
    double ht;
    ht = xright / (double) nxt;
ijloopt {divuv[i][j]=(tu[i][j]-tu[i-1][j]+tv[i][j]-tv[i][j-1])/ht;}
}
void laplace_p(double **p,double **lap_p,int nxt,int nyt){
    double **dpdx,**dpdy;
    dpdx = dmatrix(0,nxt,1,nyt);dpdy = dmatrix(1,nxt,0,nyt);
    grad_p(p,dpdx,dpdy,nxt,nyt);div_uv(dpdx,dpdy,lap_p,nxt,nyt);
    free_dmatrix(dpdx,0,nxt,1,nyt);free_dmatrix(dpdy,1,nxt,0,nyt);
}
void mat_sub(double **a,double **b,double **c,int xl,int xr,int yl,int yr){
    int i,j;
    for (i=xl;i<=xr;i++) {for (j=yl;j<=yr;j++) {a[i][j]=b[i][j]-c[i][j];}}
}
void residual_p(double **r,double **u,double **f,int nxt,int nyt){
    laplace_p(u,r,nxt,nyt); mat_sub(r,f,r,1,nxt,1,nyt);
}
void restrict(double **u_fine,double **u_coarse,int nxt,int nyt){
    int i,j;
    ijloopt {u_coarse[i][j]=0.25*(u_fine[2*i-1][2*j-1]
    +u_fine[2*i-1][2*j]+u_fine[2*i][2*j-1]+u_fine[2*i][2*j]);}
}
void zero_matrix(double **a,int xl,int xr,int yl,int yr){
    int i,j;
    for (i=xl;i<=xr;i++) {for (j=yl;j<=yr;j++) {a[i][j]=0.0;}}
}
void prolong(double **u_coarse,double **u_fine,int nxt,int nyt){
    int i,j;
    ijloopt {u_fine[2*i-1][2*j-1]=u_fine[2*i-1][2*j]=
    u_fine[2*i][2*j-1]=u_fine[2*i][2*j]=u_coarse[i][j];}
}
void mat_add(double **a,double **b,double **c,int xl,int xr,int yl,int yr){
    int i,j;
    for (i=xl;i<=xr;i++) {for (j=yl;j<=yr;j++) {a[i][j]=b[i][j]+c[i][j];}}
}
void vcycle_uv(double **uf,double **ff,int nxf,int nyf,int ilevel){
    relax_p(uf,ff,nxf,nyf);
    if (ilevel < n_level) {
        int nxc,nyc;
```

```
        double **rf,**uc,**fc;
        nxc=nxf / 2; nyc=nyf / 2; rf=dmatrix(1,nxf,1,nyf);
        uc=dmatrix(1,nxc,1,nyc); fc=dmatrix(1,nxc,1,nyc);
        residual_p(rf,uf,ff,nxf,nyf); restrict(rf,fc,nxc,nyc);
        zero_matrix(uc,1,nxc,1,nyc);
        vcycle_uv(uc,fc,nxc,nyc,ilevel + 1);
        prolong(uc,rf,nxc,nyc); mat_add(uf,uf,rf,1,nxf,1,nyf);
        relax_p(uf,ff,nxf,nyf); free_dmatrix(rf,1,nxf,1,nyf);
        free_dmatrix(uc,1,nxc,1,nyc); free_dmatrix(fc,1,nxc,1,nyc);}
}
void pressure_update(double **a){
    int i,j;
    double ave = 0.0;
    ijloop {ave = ave + a[i][j];} ave /= (nx+0.0)*(ny+0.0);
    ijloop {a[i][j] -= ave;}
}
double mat_max(double **a,int nrl,int nrh,int ncl,int nch){
    int i,j;
    double x = 0.0;
    for (i=nrl;i<=nrh;i++) {for (j=ncl;j<=nch;j++)
    {if (fabs(a[i][j]) > x) {x = fabs(a[i][j]);}}}
    return x;
}
void MG_Poisson(double **p,double **f){
    int i,j,max_it = 2000,it_mg = 1;
    double tol = 1.0e-5,resid = 1.0;
    mat_copy(workv,p,1,nx,1,ny);
    while (it_mg <= max_it && resid >= tol) {it_mg++;
        vcycle_uv(p,f,nx,ny,1); pressure_update(p);
        ijloop {sor[i][j] = workv[i][j] - p[i][j];}
 resid=mat_max(sor,1,nx,1,ny);mat_copy(workv,p,1,nx,1,ny);}
    printf("Mac iteration = %d  residual = %16.15f \n",it_mg,resid);
}
void source_uv(double **tu,double **tv,double **divuv,int nxt,int nyt){
    int i,j;
    div_uv(tu,tv,divuv,nxt,nyt); ijloopt {divuv[i][j] /= dt;}
}
void Poisson(double **tu,double **tv,double **p){
    source_uv(tu,tv,workp,nx,ny); MG_Poisson(p,workp);
}
void full_step(double **u,double **v,double **nu,double **nv,double **p){
    int i,j;
    advection_uv(u,v,adv_u,adv_v);temp_uv(tu,tv,u,v,adv_u,adv_v);
    Poisson(tu,tv,p); grad_p(p,worku,workv,nx,ny);
    for (i=1;i<nx;i++){jloop {nu[i][j]=tu[i][j]-dt*worku[i][j];}}
    iloop {for (j=1;j<ny;j++){nv[i][j]=tv[i][j]-dt*workv[i][j];}}
}
void print_data1(double **u,double **v,double **p){
    int i,j;
    FILE *fu,*fv,*fp;
fu=fopen(bufferu,"a");fv=fopen(bufferv,"a");fp=fopen(bufferp,"a");
iloop {jloop {fprintf(fu,"  %16.14f",0.5*(u[i][j]+u[i-1][j]));
```

```
fprintf(fv,"  %16.14f",0.5*(v[i][j]+v[i][j-1]));
fprintf(fp,"  %16.14f",p[i][j]);} fprintf(fu,"\n");fprintf(fv,"\n");
fprintf(fp,"\n");}fclose(fu); fclose(fv); fclose(fp);
}
int main(){
    int it,max_it,ns,count = 1;
    double **u,**v,**nu,**nv,**p;
    FILE *fu,*fv,*fp;
    p_relax=5;nx=gnx;ny=gny;n_level=(int)(log(ny)/log(2)-0.9);
    xleft=0.0;xright=1.0;yleft=0.0;yright=1.0*gny/gnx*xright;
    h = (xright-xleft)/ (double)nx; h2 = pow(h,2);
    max_it=1000;ns=(int)(max_it/10+0.001);Re=100.0;dt=0.1*h*h*Re;
    p = dmatrix(1,nx,1,ny); sor = dmatrix(1,nx,1,ny);
    workp=dmatrix(0,nx+1,0,ny+1);worku=dmatrix(0,nx+1,0,ny+1);
    workv=dmatrix(0,nx+1,0,ny+1);u=dmatrix(-1,nx+1,0,ny+1);
    v = dmatrix(0,nx+1,-1,ny+1);nu = dmatrix(-1,nx+1,0,ny+1);
    nv = dmatrix(0,nx+1,-1,ny+1);tu = dmatrix(0,nx,1,ny);
    tv = dmatrix(1,nx,0,ny);adv_u = dmatrix(0,nx,1,ny);
    adv_v = dmatrix(1,nx,0,ny);zero_matrix(tu,0,nx,1,ny);
    zero_matrix(tv,1,nx,0,ny);sprintf(bufferu,"u.m");
    sprintf(bufferv,"v.m"); sprintf(bufferp,"p.m");
    fu = fopen(bufferu,"w"); fv = fopen(bufferv,"w");
    fp = fopen(bufferp,"w");fclose(fu);fclose(fv);fclose(fp);
    initialization(p,u,v);  print_data1(u,v,p);
    mat_copy(nu,u,0,nx,1,ny); mat_copy(nv,v,1,nx,0,ny);
    for (it=1; it<=max_it; it++) {printf("iteration = %d\n",it);
        full_step(u,v,nu,nv,p); mat_copy(u,nu,0,nx,1,ny);
        mat_copy(v,nv,1,nx,0,ny);
        if (it % ns==0) {print_data1(nu,nv,p);
      printf("print out counts %d \n",count);count++;}}
    return 0;
}
```

The velocity field in Fig. 4 is obtained by running following MATLAB code:

```
clear; clc; clf; close all;
ss=sprintf('u.m'); uu=load(ss); ss=sprintf('v.m'); vv=load(ss);
nx=32; ny=nx; yright=1; xright=1; h=xright/nx;
x=linspace(0.5*h,xright-0.5*h,nx); y=linspace(0.5*h,yright-0.5*h,ny);
[xx,yy]=meshgrid(x,y); N=size(uu,1)/nx; s=0.1;
for kk=1:11
    figure;
    u=uu(1+(kk-1)*nx:kk*nx,:); v=vv(1+(kk-1)*nx:kk*nx,:);
    us=u'; vs=v';
    for i=1:nx
        for j=1:ny
            if rand() < 0.2
                us(i,j)=0;
                vs(i,j)=0;
            end
        end
    end
    quiver(xx(1:2:end,:),yy(1:2:end,:),s*us(1:2:end,:),s*vs(1:2:end,:),0,'k')
    axis image
    set(gca,'xtick',[]); set(gca,'ytick',[]);
end
```