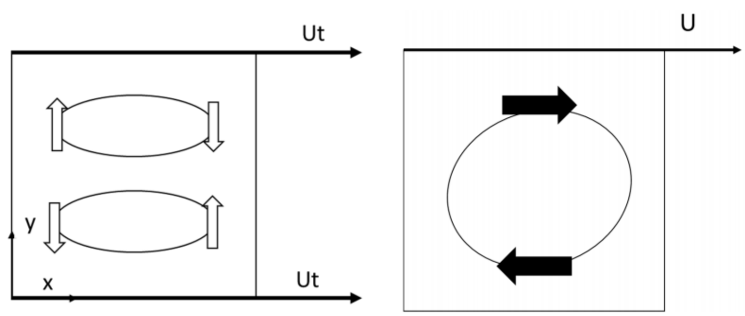
MEEN 644 - Homework 4

Logan Harbour
March 6, 2019

1 Problem statement

A viscous fluid (water at 20°C: $\rho = 998.3$ kg/m^3 and $\mu = 1.002 \times 10^{-3}$ N·s/m^2) is trapped in a square 2-D cavity of dimension 0.2 m by 0.2 m. Either top or bottom walls are pulled to the right at a uniform velocity on purpose.



Left: flow for problem (a) to verify symmetry; right: flow for problem (b) to compare with Roy et al. (2015).

Write a finite-volume base computer program to predict the 2-D steady laminar flow field for Re = 400. Solve the velocity and pressure fields by linking them through the SIMPLE algorithm in a staggered grid. Represent the solution to the one-dimensional convection-diffusion problem using the power law scheme.

(a) **(60 points)** In order to verify your code for symmetry, make calculations using 5 x 5 uniformly sized control volumes (CVs). Declare convergence when R_u and $R_v < 10^{-6}$ and $R_p < 10^{-5}$. Print your velocity and pressure fields up to 5 decimal places.

(b) With the top plate pulled right at a constant velocity at Re = 400, calculate velocity and pressure fields using 8 x 8, 16 x 16, 32 x 32, 64 x 64, and 128 x 128 CVs.

   i) **(20 points)** Plot the centerline u and v velocities for each case (for centerline u plot at $x = 0.1$ m, while for centerline v plot at $y = 0.1$ m).

   ii) **(20 points)** Compare your solutions of the 128 x 128 CV case with the benchmark solution of Roy et al (2015) on Tables 4 and 5.

2 Preliminaries

2.1 Two-dimensional diffusion-convection

With two-dimensional diffusion and convection with constant material properties, we have the PDE

$$\begin{cases} \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0\,, \\ \rho u\frac{\partial u}{\partial x} + \rho v\frac{\partial v}{\partial x} = -\frac{\partial P}{\partial x} + \mu\frac{\partial^2 u}{\partial x^2} + \mu\frac{\partial^2 u}{\partial y^2}\,, \\ \rho u\frac{\partial u}{\partial y} + \rho v\frac{\partial v}{\partial y} = -\frac{\partial P}{\partial y} + \mu\frac{\partial^2 v}{\partial x^2} + \mu\frac{\partial^2 v}{\partial y^2}\,. \end{cases} \tag{1}$$

where Dirichlet boundary conditions are applied on the boundary for u and v.

The boundary conditions for problem (a) are:

$$\begin{cases} u(x, L_y) = \frac{\mu \mathrm{Re}}{\rho L_x}\,, \\ u(L_x, y) = 0\,, \\ u(x, 0) = \frac{\mu \mathrm{Re}}{\rho L_x}\,, \\ u(0, y) = 0\,, \\ v(x, L_y) = 0\,, \\ v(L_x, y) = 0\,, \\ v(x, 0) = 0\,, \\ v(0, y) = 0\,, \end{cases} \quad , \tag{2}$$

and the boundary conditions for problem (b) are:

$$\begin{cases} u(x, L_y) = \frac{\mu \mathrm{Re}}{\rho L_x}\,, \\ u(L_x, y) = 0\,, \\ u(x, 0) = 0\,, \\ u(0, y) = 0\,, \\ v(x, L_y) = 0\,, \\ v(L_x, y) = 0\,, \\ v(x, 0) = 0\,, \\ v(0, y) = 0\,. \end{cases} \quad . \tag{3}$$

2.2 Solving methodology

Using the SIMPLE algorithm, the problem is solved in the following order:

1. Explicitly fill the boundary conditions into the u and v solution vector in order to enforce them in all of the integrations that follow.
2. Guess a pressure field, p^*.
3. Use the guessed (or previously iterated) pressure field to obtain the velocity guesses, u^* and v^*, as discussed in 2.4.
4. Solve the pressure correction, p', as discussed in 2.5.
5. Compute the velocity corrections, u' and v', and correct the velocity and pressure field, as discussed in 2.6.
6. Check for convergence. If not converged, return to 2.

2.3 Domain discretization

The domain of size $L_x \times L_y$ is discretized into $N_x \times N_y$ uniformly sized control volumes with $\Delta x = L_x/N_x$ and $\Delta y = L_y/N_y$. The numbering for all variables begins at the origin at $(i, j) = (0, 0)$. The maximum index for each variable, ϕ, is defined as (M_x^ϕ, M_y^ϕ).

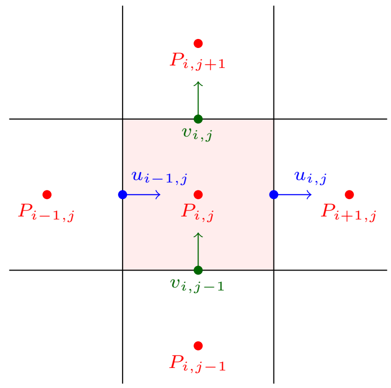


Figure 1: An internal pressure control volume.

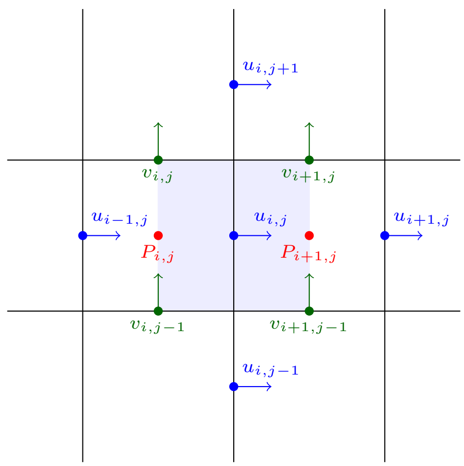


Figure 2: An internal u-velocity control volume.

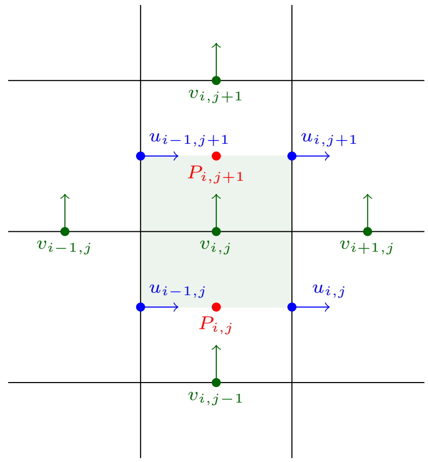


Figure 3: An internal v-velocity control volume.

2.4 Velocity guess

Define the Pechlet number on each boundary of a CV for variable ϕ centered at node $\phi_{i,j}$ as

$$P_{\mathrm{bd}}^{\phi_{i,j}} = \frac{F_{\mathrm{bd}}^{\phi_{i,j}}}{D_{\mathrm{bd}}^{\phi_{i,j}}}\,, \quad \text{where} \quad \mathrm{bd} = [n, e, s, w] \quad \text{and} \quad \phi = [u, v]\,. \tag{4}$$

The integration of the x and y-momentum equations (generalizing again with $\phi = [u, v]$) using the power-law scheme results in the equation (for $i = 1, \ldots, M_x^{\phi} - 1$, $j = 1, \ldots, M_y^{\phi} - 1$)

$$a_p^{\phi_{i,j}} \phi_{i,j}^* = a_n^{\phi_{i,j}} \phi_{i,j+1}^* + a_e^{\phi_{i,j}} \phi_{i+1,j}^* + a_s^{\phi_{i,j}} \phi_{i,j-1}^* + a_w^{\phi_{i,j}} \phi_{i-1,j}^* + a_b^{\phi_{i,j}}\,, \tag{5a}$$

$$a_n^{\phi_{i,j}} = D_n^{\phi_{i,j}} \max\left[0, (1 - 0.1|P_n^{\phi_{i,j}}|)^5\right] + \max\left[-F_n^{\phi_{i,j}}, 0\right]\,, \tag{5b}$$

$$a_e^{\phi_{i,j}} = D_e^{\phi_{i,j}} \max\left[0, (1 - 0.1|P_e^{\phi_{i,j}}|)^5\right] + \max\left[-F_e^{\phi_{i,j}}, 0\right]\,, \tag{5c}$$

$$a_s^{\phi_{i,j}} = D_s^{\phi_{i,j}} \max\left[0, (1 - 0.1|P_s^{\phi_{i,j}}|)^5\right] + \max\left[F_s^{\phi_{i,j}}, 0\right]\,, \tag{5d}$$

$$a_w^{\phi_{i,j}} = D_w^{\phi_{i,j}} \max\left[0, (1 - 0.1|P_w^{\phi_{i,j}}|)^5\right] + \max\left[F_w^{\phi_{i,j}}, 0\right]\,, \tag{5e}$$

$$a_p^{\phi_{i,j}} = a_n^{\phi_{i,j}} + a_e^{\phi_{i,j}} + a_s^{\phi_{i,j}} + a_w^{\phi_{i,j}}\,, \tag{5f}$$

$$a_b^{\phi_{i,j}} = \begin{cases} \Delta y(p_{i,j}^* - p_{i+1,j}^*)\,, & \phi = u \\ \Delta x(p_{i,j}^* - p_{i,j+1}^*)\,, & \phi = v \end{cases}\,. \tag{5g}$$

2.4.1 u-velocity guess update

In all discussion that follow, we are considering a u-CV defined by the central node $u_{i,j}$. For simplicity we will define the width of each u-CV as

$$\Delta x^{u_{i,j}} = \begin{cases} \Delta x\,, & 1 < i < M_x^u - 1 \\ \frac{3}{2}\Delta x\,, & \text{otherwise} \end{cases}\,, \tag{6}$$

and we will also define the y-distance from $u_{i,j}$ to the north and south pressure interfaces, respectively, as

$$\delta y_{p_n}^{u_{i,j}} = \begin{cases} \frac{1}{2}\Delta y\,, & j = M_y^u - 1\,, \\ \Delta y\,, & \text{otherwise} \end{cases}\,, \tag{7a}$$

$$\delta y_{p_s}^{u_{i,j}} = \begin{cases} \frac{1}{2}\Delta y\,, & j = 1\,, \\ \Delta y\,, & \text{otherwise} \end{cases}\,. \tag{7b}$$

The diffusion coefficients are then defined as

$$D_n^{u_{i,j}} = \frac{\mu \Delta x^{u_{i,j}}}{\delta y_{p_n}^{u_{i,j}}}\,, \tag{8a}$$

$$D_e^{u_{i,j}} = \frac{\mu \Delta y}{\Delta x}\,, \tag{8b}$$

$$D_s^{u_{i,j}} = \frac{\mu \Delta x^{u_{i,j}}}{\delta y_{p_s}^{u_{i,j}}}\,, \tag{8c}$$

$$D_w^{u_{i,j}} = \frac{\mu \Delta y}{\Delta x}\,. \tag{8d}$$

Lastly, the flow rates are defined as

$$F_n^{u_{i,j}} = \rho \Delta x^{u_{i,j}} \begin{cases} \frac{1}{6}\left(v_{0,j}^* + 3v_{1,j}^* + 2v_{2,j}^*\right)\,, & i = 1 \\ \frac{1}{6}\left(2v_{i,j}^* + 3v_{i+1,j}^* + v_{i+2,j}^*\right)\,, & i = M_x^u - 1 \\ \frac{1}{2}\left(v_{i,j}^* + v_{i+1,j}^*\right)\,, & \text{otherwise} \end{cases}\,, \tag{9a}$$

$$F_e^{u_{i,j}} = \rho \Delta y \begin{cases} u_{M_x^u,j}^*\,, & i = M_x^u - 1 \\ \frac{1}{2}\left(u_{i+1,j}^* + u_{i,j}^*\right)\,, & \text{otherwise} \end{cases}\,, \tag{9b}$$

$$F_s^{u_{i,j}} = \rho \Delta x^{u_{i,j}} \begin{cases} \frac{1}{6}\left(v_{0,j-1}^* + 3v_{1,j-1}^* + 2v_{2,j-1}^*\right)\,, & i = 1 \\ \frac{1}{6}\left(2v_{i,j-1}^* + 3v_{i+1,j-1}^* + v_{i+2,j-1}^*\right)\,, & i = M_x^u - 1 \\ \frac{1}{2}\left(v_{i,j-1}^* + v_{i+1,j-1}^*\right)\,, & \text{otherwise} \end{cases} \tag{9c}$$

$$F_w^{u_{i,j}} = \rho \Delta y \begin{cases} u_{0,j}^*\,, & i = 1 \\ \frac{1}{2}\left(u_{i-1,j}^* + u_{i,j}^*\right)\,, & \text{otherwise} \end{cases}\,. \tag{9d}$$

2.4.2 v-velocity guess update

Similarly, we will consider a v-CV defined by the central node $v_{i,j}$. The width of each v-CV is defined as

$$\Delta y^{v_{i,j}} = \begin{cases} \Delta y\,, & 1 < j < M_y^v - 1 \\ \frac{3}{2}\Delta y\,, & \text{otherwise} \end{cases}\,, \tag{10}$$

and we will also define the x-distance from $v_{i,j}$ to the east and west pressure interfaces, respectively, as

$$\delta x_{p_e}^{v_{i,j}} = \begin{cases} \frac{1}{2}\Delta x\,, & i = M_x^v - 1 \\ \Delta x\,, & \text{otherwise} \end{cases}\,, \tag{11a}$$

$$\delta x_{p_w}^{v_{i,j}} = \begin{cases} \frac{1}{2}\Delta x\,, & i = 1\,, \\ \Delta x\,, & \text{otherwise} \end{cases}\,. \tag{11b}$$

The diffusion coefficients are then defined as

$$D_n^{v_{i,j}} = \frac{\mu \Delta x}{\Delta y}\,, \tag{12a}$$

$$D_e^{v_{i,j}} = \frac{\mu \Delta y^{v_{i,j}}}{\delta x_{p_e}^{v_{i,j}}}\,, \tag{12b}$$

$$D_s^{v_{i,j}} = \frac{\mu \Delta x}{\Delta y}\,, \tag{12c}$$

$$D_w^{v_{i,j}} = \frac{\mu \Delta y^{v_{i,j}}}{\delta x_{p_w}^{v_{i,j}}}\,. \tag{12d}$$

Lastly, the flow rates are defined as

$$F_n^{v_{i,j}} = \rho \Delta x \begin{cases} v^*_{i,M_y^v}\,, & j = M_y^v - 1 \\ \frac{1}{2}\left(v^*_{i,j+1} + v^*_{i,j}\right)\,, & \text{otherwise} \end{cases}\,, \tag{13a}$$

$$F_e^{v_{i,j}} = \rho \Delta y^{v_{i,j}} \begin{cases} \frac{1}{6}\left(u^*_{i,0} + 3u^*_{i,1} + 2u^*_{i,2}\right)\,, & j = 1 \\ \frac{1}{6}\left(2u^*_{i,j} + 3u^*_{i,j+1} + 2u^*_{i,j+2}\right)\,, & j = M_y^v - 1 \\ \frac{1}{2}\left(u^*_{i,j} + u^*_{i,j+1}\right)\,, & \text{otherwise} \end{cases}\,, \tag{13b}$$

$$F_s^{v_{i,j}} = \rho \Delta x \begin{cases} v^*_{i,0}\,, & j = 1 \\ \frac{1}{2}\left(v^*_{i,j+1} + v^*_{i,j}\right)\,, & \text{otherwise} \end{cases}\,, \tag{13c}$$

$$F_w^{v_{i,j}} = \rho \Delta y^{v_{i,j}} \begin{cases} \frac{1}{6}\left(u^*_{i-1,0} + 3u^*_{i-1,1} + 2u^*_{i-1,2}\right)\,, & j = 1 \\ \frac{1}{6}\left(2u^*_{i-1,j} + 3u^*_{i-1,j+1} + 2u^*_{i-1,j+2}\right)\,, & j = M_y^v - 1 \\ \frac{1}{2}\left(u^*_{i-1,j} + u^*_{i-1,j+1}\right)\,, & \text{otherwise} \end{cases}\,. \tag{13d}$$

2.5 Pressure correction solve

At convergence, $u' = v' = p' = 0$, therefore it is irrelevant as to how we find them. Subtracting (exact - guessed) forms of Equation (5) one obtains

$$a_p^{u_{i,j}} u'_{i,j} = a_n^{u_{i,j}} u'_{i,j+1} + a_e^{u_{i,j}} u'_{i+1,j} + a_s^{u_{i,j}} u'_{i,j-1} + a_w^{u_{i,j}} u'_{i-1,j} + \Delta y (p'_{i,j} - p'_{i+1,j})\,, \tag{14a}$$

$$a_p^{v_{i,j}} v'_{i,j} = a_n^{v_{i,j}} v'_{i,j+1} + a_e^{v_{i,j}} v'_{i+1,j} + a_s^{v_{i,j}} v'_{i,j-1} + a_w^{v_{i,j}} v'_{i-1,j} + \Delta x (p'_{i,j} - p'_{i,j+1})\,. \tag{14b}$$

Drop the neighbor terms in the equations above (implying that velocity corrections are local) to obtain

$$u'_{i,j} = \frac{\Delta y}{a_p^{u_{i,j}}}(p'_{i,j} - p'_{i+1,j})\,, \tag{15a}$$

$$v'_{i,j} = \frac{\Delta x}{a_p^{v_{i,j}}}(p'_{i,j} - p'_{i,j+1})\,. \tag{15b}$$

Integrate the continuity equation over a p-CV defined by the central node $p_{i,j}$ and substitute $u = u^* + u'$, $v = v^* + v'$, and the above Equations to obtain

$$
\begin{aligned}
a_p^{p'_{i,j}} p'_{i,j} &= a_n^{p'_{i,j}} p'_{i,j+1} + a_e^{p'_{i,j}} p'_{i+1,j} + a_s^{p'_{i,j}} p'_{i,j-1} + a_w^{p'_{i,j}} p'_{i-1,j} + a_b^{p'_{i,j}} , && \text{(16a)}\\
a_n^{p'_{i,j}} &= \begin{cases} \rho \Delta x^2 / a_p^{v_{i,j}} , & j < M_y^p - 1 \\ 0 , & \text{otherwise} \end{cases} , && \text{(16b)}\\
a_e^{p'_{i,j}} &= \begin{cases} \rho \Delta y^2 / a_p^{u_{i,j}} , & i < M_x^p - 1 \\ 0 , & \text{otherwise} \end{cases} , && \text{(16c)}\\
a_s^{p'_{i,j}} &= \begin{cases} \rho \Delta x^2 / a_p^{v_{i,j-1}} , & j > 1 \\ 0 , & \text{otherwise} \end{cases} , && \text{(16d)}\\
a_w^{p'_{i,j}} &= \begin{cases} \rho \Delta y^2 / a_p^{u_{i-1,j}} , & i > 1 \\ 0 , & \text{otherwise} \end{cases} , && \text{(16e)}\\
a_p^{p'_{i,j}} &= a_n^{p'_{i,j}} + a_e^{p'_{i,j}} + a_s^{p'_{i,j}} + a_w^{p'_{i,j}} , && \text{(16f)}\\
a_b^{p'_{i,j}} &= \rho \left(\Delta y (u^*_{i-1,j} - u^*_{i,j}) + \Delta x (v^*_{i,j-1} - v^*_{i,j}) \right) . && \text{(16g)}
\end{aligned}
$$

2.6 Velocity and pressure correction

Lastly, the velocities are then updated using Equation (15) with

$$
\begin{aligned}
u_{i,j} &= u_{i,j} + \frac{\Delta y}{a_p^{u_{i,j}}} (p'_{i,j} - p'_{i+1,j}) , \quad i = 1, \ldots, M_x^u - 1 , \quad j = 1, \ldots, M_y^u - 1 , && \text{(17a)}\\
v_{i,j} &= v_{i,j} + \frac{\Delta x}{a_p^{v_{i,j}}} (p'_{i,j} - p'_{i,j+1}) , \quad i = 1, \ldots, M_x^v - 1 , \quad j = 1, \ldots, M_y^v - 1 , && \text{(17b)}
\end{aligned}
$$

and the pressures (take note of the relaxation parameter α_p) with

$$
p_{i,j} = p_{i,j} + \alpha_p p'_{i,j} , \quad i = 1, \ldots, M_x^p - 1 , \quad j = 1, \ldots, M_y^p - 1 . \tag{18}
$$

2.7 System solver

The systems in Equations (5) and (16) are solved using the line-by-line method with TDMA as the matrix solver. In this method, a tri-diagonal system is formed as the terms from one of the dimensions are lagged. Consider the simple system

$$
a_p^{i,j} \phi_{i,j} = a_n^{i,j} \phi_{i,j+1} + a_e^{i,j} \phi_{i+1,j} + a_s^{i,j} \phi_{i,j-1} + a_w^{i,j} \phi_{i-1,j} + a_b^{i,j} , \quad i = 1, \ldots, N_x , \quad j = 1, \ldots, N_y . \tag{19}
$$

Now, consider ϕ^* to be a *lagged* value of ϕ, i.e., it is known and is moved to the right hand side of each equation. In solving a single physical column i using the line-by-line method, the following system is solved:

$$
a_p^{i,j} \phi_{i,j} = a_n^{i,j} \phi_{i,j+1} + a_e^{i,j} \phi^*_{i+1,j} + a_s^{i,j} \phi_{i,j-1} + a_w^{i,j} \phi^*_{i-1,j} + a_b^{i,j} , \qquad j = 1, \ldots, N_y . \tag{20}
$$

In solving a single physical row j using the line-by-line method, the following system is solved:

$$
a_p^{i,j} \phi_{i,j} = a_n^{i,j} \phi^*_{i,j+1} + a_e^{i,j} \phi_{i+1,j} + a_s^{i,j} \phi^*_{i,j-1} + a_w^{i,j} \phi_{i-1,j} + a_b^{i,j} , \qquad i = 1, \ldots, N_x . \tag{21}
$$

3 Results

3.1 Problem a: Lid driven cavity, symmetry check

Symmetric solutions were identified for problem (a), as presented in Tables 1, 2, and 3 below.

Table 1: The u-velocity solution with 5x5 pressure CVs and symmetric BCs.

	1	2	3	4	5	6
1	2.00741E−3	2.00741E−3	2.00741E−3	2.00741E−3	2.00741E−3	2.00741E−3
2	0.00000E0	1.69059E−4	2.96268E−4	3.47982E−4	2.74870E−4	0.00000E0
3	0.00000E0	−6.05705E−5	−1.24904E−4	−1.54163E−4	−1.09384E−4	0.00000E0
4	0.00000E0	−2.16978E−4	−3.42727E−4	−3.87638E−4	−3.30974E−4	0.00000E0
5	0.00000E0	−6.05705E−5	−1.24904E−4	−1.54163E−4	−1.09384E−4	0.00000E0
6	0.00000E0	1.69059E−4	2.96268E−4	3.47982E−4	2.74870E−4	0.00000E0
7	2.00741E−3	2.00741E−3	2.00741E−3	2.00741E−3	2.00741E−3	2.00741E−3

Table 2: The v-velocity solution with 5x5 pressure CVs and symmetric BCs.

	1	2	3	4	5	6	7
1	0.00000E0	0.00000E0	0.00000E0	0.00000E0	0.00000E0	0.00000E0	0.00000E0
2	0.00000E0	−1.69059E−4	−1.27208E−4	−5.17144E−5	7.31116E−5	2.74870E−4	0.00000E0
3	0.00000E0	−1.08489E−4	−6.28747E−5	−2.24555E−5	2.83322E−5	1.65487E−4	0.00000E0
4	0.00000E0	1.08489E−4	6.28746E−5	2.24555E−5	−2.83322E−5	−1.65487E−4	0.00000E0
5	0.00000E0	1.69059E−4	1.27208E−4	5.17144E−5	−7.31116E−5	−2.74870E−4	0.00000E0
6	0.00000E0	0.00000E0	0.00000E0	0.00000E0	0.00000E0	0.00000E0	0.00000E0

Table 3: The p solution with 5x5 pressure CVs and symmetric BCs.

	1	2	3	4	5	6	7
1	−9.94865E−5	−9.94865E−5	−9.47033E−6	7.93315E−6	6.43241E−5	2.10288E−4	2.10288E−4
2	−9.94865E−5	−9.94865E−5	−9.47033E−6	7.93315E−6	6.43241E−5	2.10288E−4	2.10288E−4
3	−2.08208E−6	−2.08208E−6	−2.50303E−5	−3.92207E−5	−2.33249E−5	7.17292E−5	7.17292E−5
4	1.51012E−5	1.51012E−5	−1.26152E−5	−2.46901E−5	8.03299E−6	7.91904E−5	7.91904E−5
5	−2.08206E−6	−2.08206E−6	−2.50303E−5	−3.92207E−5	−2.33249E−5	7.17293E−5	7.17293E−5
6	−9.94865E−5	−9.94865E−5	−9.47034E−6	7.93315E−6	6.43241E−5	2.10288E−4	2.10288E−4
7	−9.94865E−5	−9.94865E−5	−9.47034E−6	7.93315E−6	6.43241E−5	2.10288E−4	2.10288E−4

3.2 Problem b: Lid driven cavity, top right BC

The requirements for problem (b) part i and ii were combined into Figure 4 as seen below. With increasing grid refinement, both centerline velocity profiles approached towards the reference solution obtained from Roy et. al. In addition, a once-more-refined run is compared with 256x256 CVs with good agreement.

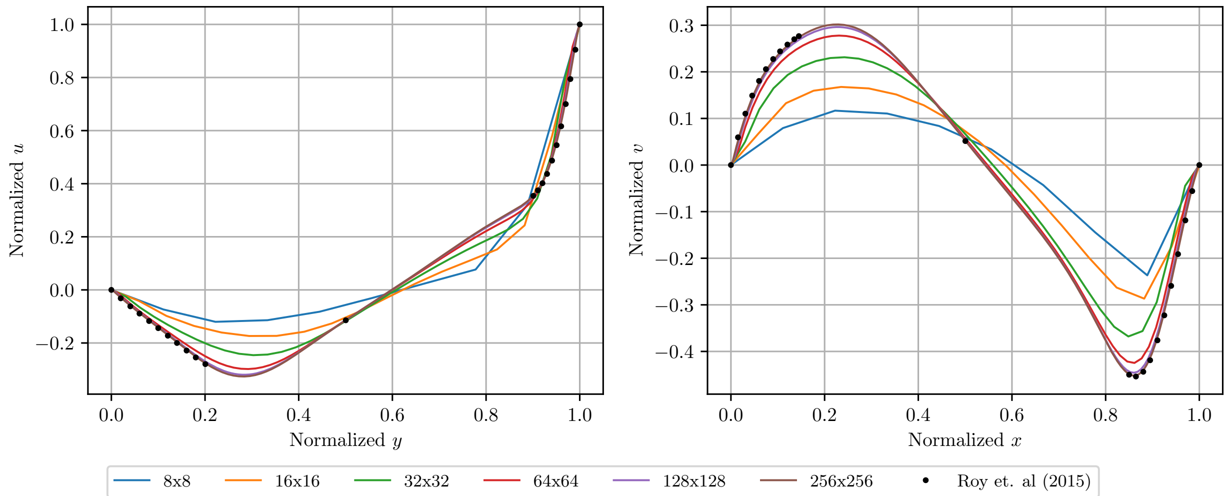


Figure 4: Centerline u and v-velocity fields with the top plate pulled right at a constant velocity.

Code listing

For the implementation, we have the following files:

- `Makefile` – Allows for compiling the c++ project with `make`.
- `hwk4.cpp` – Contains the `main()` function that is required by C that runs the cases requested in this problem set.
- `Problem.h` – Contains the header for the `Problem` class which is the main driver for a `Flow2D::Problem`.
- `Variable.h` – Contains the `Flow2D::Variable` class, which is a storage container for a single variable (i.e., u).
- `Problem.cpp` – Contains the `run()` functions that executes a `Problem`.
- `Problem_coefficients.cpp` – Contains the functions for solving coefficients in a `Problem`.
- `Problem_corrections.cpp` – Contains the functions for correcting solutions in a `Problem`.
- `Problem_residuals.cpp` – Contains the functions for computing residuals in a `Problem`.
- `Problem_solvers.cpp` – Contains the functions for sweeping and solving in a `Problem`.
- `Matrix.h` – Contains the `Matrix` class which provides storage for a matrix with various standard matrix operations.
- `TriDiagonal.h` – Contains the `TriDiagonal` class which provides storage for a tri-diagonal matrix including the TDMA solver found in the member function `solveTDMA()`.
- `Vector.h` – Contains the `Vector` class for one-dimensional vector storage.
- `plots.py` - Produces the plots in this report.

Makefile

```
src = $(wildcard *.cpp)
obj = $(src:.cpp=.o)
CXXFLAGS = -std=c++14
CCFLAGS = $(CXXFLAGS)

hwk-opt: $(obj)
        clang++ -o $@ $^

.PHONY: clean
clean:
        rm -f $(obj) hwk-opt
```

hwk4.cpp

```
#include "Problem.h"

using namespace Flow2D;

int
main()
{
  // Problem wide constants
  double L = 0.2;
  double Re = 400;
  double rho = 998.3;
  double mu = 1.002e-3;
  double bc_val = Re * mu / (rho * L);

  // Standard inputs
  InputArguments input;
  input.Lx = L;
  input.Ly = L;
  input.mu = mu;
  input.rho = rho;
  input.u_ref = bc_val;
  input.L_ref = L;

  // Problem 1: check symmetry
  input.u_bc = BoundaryCondition(bc_val, 0, bc_val, 0);
  input.v_bc = BoundaryCondition(0, 0, 0, 0);
  std::cout << "Problem 1, check symmetry" << std::endl;
  {
    Problem problem(5, 5, input);
    problem.run();
    problem.print(Variables::u, "u =");
    problem.print(Variables::v, "v =");
    problem.print(Variables::p, "p =", true);
  }

  // Problem 2: change to top plate BC
  input.u_bc = BoundaryCondition(bc_val, 0, 0, 0);
  input.v_bc = BoundaryCondition(0, 0, 0, 0);
  std::cout << "Problem 2, top plate BC" << std::endl;
  for (unsigned int N : {8, 16, 32, 64, 128, 256})
  {
    std::cout << "N = " << N << "x" << N << " - ";
    Problem problem(N, N, input);
    problem.run();
    problem.save(Variables::u, "results/" + to_string(N) + "_u.csv");
    problem.save(Variables::v, "results/" + to_string(N) + "_v.csv");
  }
}
```

Problem.h

```cpp
#ifndef PROBLEM_H
#define PROBLEM_H

#include <cmath>
#include <iomanip>
#include <iostream>
#include <map>

#include "Variable.h"

namespace Flow2D
{

using namespace std;

struct InputArguments
{
  double Lx, Ly;
  BoundaryCondition u_bc, v_bc;
  double L_ref, u_ref;
  double rho, mu;
  bool debug = false;
  double alpha_p = 0.7;
  double alpha_uv = 0.5;
  unsigned int max_its = 100000;
  double tol = 1.0e-6;
};

class Problem
{
public:
  Problem(const unsigned int Nx, const unsigned int Ny, const InputArguments & input);

  void run();

  // Public access to printing and saving variable results
  void print(const Variables var,
             const string prefix = "",
             const bool newline = false,
             const unsigned int pr = 5) const
  {
    variables.at(var).print(prefix, newline, pr);
  }
  void save(const Variables var, const string filename) const { variables.at(var).save(filename); }

private:
  // Problem_corrections.cpp
  void correct();
  void pCorrect();
  void uCorrect();
  void vCorrect();

  // Problem_coefficients.cpp
  void fillCoefficients(Variable & var);
  void pcCoefficients();
  void uCoefficients();
  void vCoefficients();
  void velocityCoefficients(Coefficients & a,
                            const Coefficients & D,
                            const Coefficients & F,
                            const double & b);

  // Problem_residuals.cpp
  void computeResiduals();
  double pResidual() const;
  double velocityResidual(const Variable & var) const;

  // Problem_solvers.cpp
  void solve();
  void solve(Variable & var);
  void sweepColumns(Variable & var, const bool west_east = true);
  void sweepRows(Variable & var, const bool south_north = true);
  void sweepColumn(const unsigned int i, Variable & var);
  void sweepRow(const unsigned int j, Variable & var);
  void solveVelocities();

protected:
```

```cpp
  // Number of pressure CVs
  const unsigned int Nx, Ny;

  // Geometry [m]
  const double Lx, Ly, dx, dy;
  // Material properties
  const double rho, mu;
  // Residual references
  const double L_ref, u_ref;

  // Enable debug mode (printing extra output)
  const bool debug;

  // Maximum iterations
  const unsigned int max_its;
  // Iteration tolerance
  const double tol;
  // Pressure relaxation
  const double alpha_p;
  // Number of iterations completed
  unsigned int iterations = 0;

  // Variables
  Variable u, v, pc, p;
  // Variable map
  map<const Variables, const Variable &> variables;

  // Whether or not we converged
  bool converged = false;
};

} // namespace Flow2D
#endif /* PROBLEM_H */
```

Variable.h

```
#ifndef VARIABLE_H
#define VARIABLE_H

#include "Matrix.h"
#include "TriDiagonal.h"
#include "Vector.h"

namespace Flow2D
{

using namespace std;

// Storage for boundary conditions
struct BoundaryCondition
{
  BoundaryCondition() {}
  BoundaryCondition(const double top, const double right, const double bottom, const double left)
    : top(top), right(right), bottom(bottom), left(left)
  {
  }
  bool nonzero() const { return top != 0 || right != 0 || bottom != 0 || left != 0; }
  double top = 0, right = 0, bottom = 0, left = 0;
};

// Storage for coefficients for a single CV
struct Coefficients
{
  double p = 0, n = 0, e = 0, s = 0, w = 0, b = 0;
  void print(const unsigned int pr = 5)
  {
    cout << setprecision(pr) << scientific << "n = " << n << ", e = " << e << ", s = " << s
         << ", w = " << w << ", p = " << p << ", b = " << b << endl;
  }
};

// Enum for variable types
enum Variables
{
  u,
  v,
  pc,
  p
};

// Conversion from variable type to its string
static string
VariableString(Variables var)
{
  switch (var)
  {
    case Variables::u:
      return "u";
    case Variables::v:
      return "v";
    case Variables::pc:
      return "pc";
    case Variables::p:
      return "p";
  }
}

// General storage structure for primary and auxilary variables
struct Variable
{
  // Constructor for a primary variable
  Variable(const Variables name,
           const unsigned int Nx,
           const unsigned int Ny,
           const double alpha,
           const BoundaryCondition bc = BoundaryCondition())
    : name(name),
      string(VariableString(name)),
      Nx(Nx),
      Ny(Ny),
      Mx(Nx - 1),
      My(Ny - 1),
      w(1 / alpha),
```

```cpp
      bc(bc),
      a(Nx, Ny),
      phi(Nx, Ny),
      Ax(Nx - 2),
      Ay(Ny - 2),
      bx(Nx - 2),
      by(Ny - 2)
  {
    // Apply initial boundary conditions
    if (bc.left != 0)
      phi.setColumn(0, bc.left);
    if (bc.right != 0)
      phi.setColumn(Mx, bc.right);
    if (bc.bottom != 0)
      phi.setRow(0, bc.bottom);
    if (bc.top != 0)
      phi.setRow(My, bc.top);
  }

  // Constructor for an auxilary variable (no solver storage)
  Variable(const Variables name, const unsigned int Nx, const unsigned int Ny)
    : name(name), string(VariableString(name)), Nx(Nx), Ny(Ny), Mx(Nx - 1), My(Ny - 1), phi(Nx, Ny)
  {
  }

  // Solution matrix operations
  const double & operator()(const unsigned int i, const unsigned int j) const { return phi(i, j); }
  double & operator()(const unsigned int i, const unsigned int j) { return phi(i, j); }
  void print(const string prefix = "", const bool newline = false, const unsigned int pr = 5) const
  {
    phi.print(prefix, newline, pr);
  }
  void save(const string filename) const { phi.save(filename); }
  void reset() { phi = 0; }

  // Coefficient debug
  void
  printCoefficients(const string prefix = "", const bool newline = false, const unsigned int pr = 5)
  {
    for (unsigned int i = 1; i < Nx - 1; ++i)
      for (unsigned int j = 1; j < Ny - 1; ++j)
      {
        cout << prefix << "(" << i << ", " << j << "): ";
        a(i, j).print(pr);
      }
    if (newline)
      cout << endl;
  }

  // Variable enum name
  const Variables name;
  // Variable string
  const string string;
  // Variable size
  const unsigned int Nx, Ny;
  // Maximum variable index that is being solved
  const unsigned int Mx, My;
  // Relaxation coefficient used in solving linear systems
  const double w = 0;
  // Boundary conditions
  const BoundaryCondition bc = BoundaryCondition();
  // Matrix coefficients
  Matrix<Coefficients> a;
  // Variable solution
  Matrix<double> phi;
  // Linear system LHS for both sweep directions
  TriDiagonal<double> Ax, Ay;
  // Linear system RHS for both sweep directions
  Vector<double> bx, by;
};

} // namespace Flow2D
#endif /* VARIABLE_H */
```

Problem.cpp

```cpp
#include "Problem.h"

namespace Flow2D
{

Problem::Problem(const unsigned int Nx, const unsigned int Ny, const InputArguments & input)
  : // Number of pressure CVs
    Nx(Nx),
    Ny(Ny),
    // Domain sizes
    Lx(input.Lx),
    Ly(input.Ly),
    dx(Lx / Nx),
    dy(Ly / Ny),
    // Residual references
    L_ref(input.L_ref),
    u_ref(input.u_ref),
    // Material properties
    rho(input.rho),
    mu(input.mu),
    // Enable debug
    debug(input.debug),
    // Solver properties
    max_its(input.max_its),
    tol(input.tol),
    alpha_p(input.alpha_p),
    // Initialize variables for u, v, pc (solved variables)
    u(Variables::u, Nx + 1, Ny + 2, input.alpha_uv, input.u_bc),
    v(Variables::v, Nx + 2, Ny + 1, input.alpha_uv, input.v_bc),
    pc(Variables::pc, Nx + 2, Ny + 2, 1),
    // Initialize aux variables
    p(Variables::p, Nx + 2, Ny + 2)
{
  // Add into variable map for access outside of class
  variables.emplace(Variables::u, u);
  variables.emplace(Variables::v, v);
  variables.emplace(Variables::pc, pc);
  variables.emplace(Variables::p, p);
}

void
Problem::run()
{
  for (unsigned int l = 0; l < max_its; ++l)
  {
    ++iterations;
    if (debug)
      cout << "Iteration " << setw(3) << left << l << ": " << endl;

    // Solve for all variables
    solve();

    // Apply corrections
    correct();

    // Compute residuals and exit if converged
    computeResiduals();
    if (converged)
      break;
  }

  // Oops. Didn't converge
  if (!converged)
    cout << "Did not converge after " << max_its << " iterations!" << endl;
}

}
```

Problem_coefficients.cpp

```cpp
#include "Problem.h"

namespace Flow2D
{

void
Problem::fillCoefficients(Variable & var)
{
  switch (var.name)
  {
    case Variables::u:
      uCoefficients();
      break;
    case Variables::v:
      vCoefficients();
      break;
    case Variables::pc:
      pcCoefficients();
      break;
    case Variables::p:
      break;
  }

  if (debug)
    var.printCoefficients(var.string, true);
}

void
Problem::pcCoefficients()
{
  for (unsigned int i = 1; i < pc.Mx; ++i)
    for (unsigned int j = 1; j < pc.My; ++j)
    {
      Coefficients & a = pc.a(i, j);

      if (i != 1)
        a.w = rho * dy * dy / u.a(i - 1, j).p;
      if (i != pc.Mx - 1)
        a.e = rho * dy * dy / u.a(i, j).p;
      if (j != 1)
        a.s = rho * dx * dx / v.a(i, j - 1).p;
      if (j != pc.My - 1)
        a.n = rho * dx * dx / v.a(i, j).p;
      a.p = a.n + a.e + a.s + a.w;
      a.b = rho * (dy * (u(i - 1, j) - u(i, j)) + dx * (v(i, j - 1) - v(i, j)));
    }
}

void
Problem::uCoefficients()
{
  Coefficients D, F;
  double W, dy_pn, dy_ps, b;

  for (unsigned int i = 1; i < u.Mx; ++i)
    for (unsigned int j = 1; j < u.My; ++j)
    {
      // Width of the cell
      W = (i == 1 || i == u.Mx - 1 ? 3 * dx / 2 : dx);
      // North/south distances to pressure nodes
      dy_pn = (j == u.My - 1 ? dy / 2 : dy);
      dy_ps = (j == 1 ? dy / 2 : dy);

      // Diffusion coefficients
      D.n = mu * W / dy_pn;
      D.e = mu * dy / dx;
      D.s = mu * W / dy_ps;
      D.w = mu * dy / dx;

      // East and west flows
      F.e = (i == u.Mx - 1 ? rho * dy * u(u.Mx, j) : rho * dy * (u(i + 1, j) + u(i, j)) / 2);
      F.w = (i == 1 ? rho * dy * u(0, j) : rho * dy * (u(i - 1, j) + u(i, j)) / 2);
      // North and south flows
      if (i == 1) // Left boundary
      {
        F.n = rho * W * (v(0, j) + 3 * v(1, j) + 2 * v(2, j)) / 6;
        F.s = rho * W * (v(0, j - 1) + 3 * v(1, j - 1) + 2 * v(2, j - 1)) / 6;
```

```
      }
      else if (i == u.Mx - 1) // Right boundary
      {
        F.n = rho * W * (2 * v(i, j) + 3 * v(i + 1, j) + v(i + 2, j)) / 6;
        F.s = rho * W * (2 * v(i, j - 1) + 3 * v(i + 1, j - 1) + v(i + 2, j - 1)) / 6;
      }
      else // Interior (not left or right boundary)
      {
        F.n = rho * W * (v(i, j) + v(i + 1, j)) / 2;
        F.s = rho * W * (v(i, j - 1) + v(i + 1, j - 1)) / 2;
      }

      // Pressure RHS
      b = dy * (p(i, j) - p(i + 1, j));

      // Compute and store power law coefficients
      velocityCoefficients(u.a(i, j), D, F, b);
    }
}

void
Problem::vCoefficients()
{
  Coefficients D, F;
  double H, dx_pe, dx_pw, b;

  for (unsigned int i = 1; i < v.Mx; ++i)
    for (unsigned int j = 1; j < v.My; ++j)
    {
      // Height of the cell
      H = (j == 1 || j == v.My - 1 ? 3 * dy / 2 : dy);
      // East/west distances to pressure nodes
      dx_pe = (i == v.Mx - 1 ? dx / 2 : dx);
      dx_pw = (i == 1 ? dx / 2 : dx);

      // Diffusion coefficient
      D.n = mu * dx / dy;
      D.e = mu * H / dx_pe;
      D.s = mu * dx / dy;
      D.w = mu * H / dx_pw;

      // North and east flows
      F.n = (j == v.My - 1 ? rho * dx * v(i, v.My) : rho * dx * (v(i, j + 1) + v(i, j)) / 2);
      F.s = (j == 1 ? rho * dx * v(i, 0) : rho * dx * (v(i, j - 1) + v(i, j)) / 2);
      // East and west flows
      if (j == 1) // Bottom boundary
      {
        F.e = rho * H * (u(i, 0) + 3 * u(i, 1) + 2 * u(i, 2)) / 6;
        F.w = rho * H * (u(i - 1, 0) + 3 * u(i - 1, 1) + 2 * u(i - 1, 2)) / 6;
      }
      else if (j == v.My - 1) // Top boundary
      {
        F.e = rho * H * (2 * u(i, j) + 3 * u(i, j + 1) + u(i, j + 2)) / 6;
        F.w = rho * H * (2 * u(i - 1, j) + 3 * u(i - 1, j + 1) + u(i - 1, j + 2)) / 6;
      }
      else // Interior (not top or bottom boundary)
      {
        F.e = rho * H * (u(i, j) + u(i, j + 1)) / 2;
        F.w = rho * H * (u(i - 1, j) + u(i - 1, j + 1)) / 2;
      }

      // Pressure RHS
      b = dx * (p(i, j) - p(i, j + 1));

      // Compute and store power law coefficients
      velocityCoefficients(v.a(i, j), D, F, b);
    }
}

void
Problem::velocityCoefficients(Coefficients & a,
                              const Coefficients & D,
                              const Coefficients & F,
                              const double & b)
{
  a.n = D.n * fmax(0, pow(1 - 0.1 * fabs(F.n / D.n), 5)) + fmax(-F.n, 0);
  a.e = D.e * fmax(0, pow(1 - 0.1 * fabs(F.e / D.e), 5)) + fmax(-F.e, 0);
  a.s = D.s * fmax(0, pow(1 - 0.1 * fabs(F.s / D.s), 5)) + fmax(F.s, 0);
  a.w = D.w * fmax(0, pow(1 - 0.1 * fabs(F.w / D.w), 5)) + fmax(F.w, 0);
  a.p = a.n + a.e + a.s + a.w;
  a.b = b;
```

```
}

} // namespace Flow2D
```

Problem_corrections.cpp

```
#include "Problem.h"

namespace Flow2D
{

void
Problem::correct()
{
  uCorrect();
  vCorrect();
  pCorrect();
}

void
Problem::pCorrect()
{
  for (unsigned int i = 1; i < pc.Mx; ++i)
    for (unsigned int j = 1; j < pc.My; ++j)
      p(i, j) += alpha_p * pc(i, j);

  // Set pressure correction back to zero
  pc.reset();

  // Apply the edge values as velocity is set
  for (unsigned int i = 0; i <= pc.Mx; ++i)
  {
    p(i, 0) = p(i, 1);
    p(i, pc.My) = p(i, pc.My - 1);
  }
  for (unsigned int j = 0; j <= pc.My; ++j)
  {
    p(0, j) = p(1, j);
    p(pc.Mx, j) = p(pc.Mx - 1, j);
  }

  if (debug)
    p.print("p corrected = ", true);
}

void
Problem::uCorrect()
{
  for (unsigned int i = 1; i < u.Mx; ++i)
    for (unsigned int j = 1; j < u.My; ++j)
      u(i, j) += dy * (pc(i, j) - pc(i + 1, j)) / u.a(i, j).p;

  if (debug)
    u.print("u corrected = ", true);
}

void
Problem::vCorrect()
{
  for (unsigned int i = 1; i < v.Mx; ++i)
    for (unsigned int j = 1; j < v.My; ++j)
      v(i, j) += dx * (pc(i, j) - pc(i, j + 1)) / v.a(i, j).p;

  if (debug)
    v.print("v corrected = ", true);
}

}
```

Problem_residuals.cpp

```cpp
#include "Problem.h"

namespace Flow2D
{

void
Problem::computeResiduals()
{
  double Ru = velocityResidual(u);
  double Rv = velocityResidual(v);
  double Rp = pResidual();

  // Check for convergence
  if (Ru < tol && Rv < tol && Rp < tol)
    converged = true;

  // Print residuals
  if (converged)
    cout << "Converged in " << iterations << " iterations: ";
  if (converged || debug) {
    cout << noshowpos << setprecision(2) << scientific;
    cout << "u = " << Ru;
    cout << ", v = " << Rv;
    cout << ", p = " << Rp << endl;
  }
}

double
Problem::pResidual() const
{
  double numer = 0;
  for (unsigned int i = 1; i < pc.Mx; ++i)
    for (unsigned int j = 1; j < pc.My; ++j)
      numer += abs(dy * (u(i - 1, j) - u(i, j)) + dx * (v(i, j - 1) - v(i, j)));
  return numer / (u_ref * L_ref);
}

double
Problem::velocityResidual(const Variable & var) const
{
  double numer, numer_temp, denom = 0;
  for (unsigned int i = 1; i < var.Mx; ++i)
    for (unsigned int j = 1; j < var.My; ++j)
    {
      const Coefficients & a = var.a(i, j);
      numer_temp = a.p * var(i, j);
      denom += abs(numer_temp);
      numer_temp -= a.n * var(i, j + 1);
      numer_temp -= a.e * var(i + 1, j);
      numer_temp -= a.s * var(i, j - 1);
      numer_temp -= a.w * var(i - 1, j);
      numer_temp -= a.b;
      numer += abs(numer_temp);
    }
  return numer / denom;
}

}
```

Problem_solvers.cpp

```cpp
#include "Problem.h"

namespace Flow2D
{

void
Problem::solve()
{
  solve(u);
  solve(v);
  solve(pc);
}

void
Problem::solve(Variable & var)
{
  // Fill the coefficients
  fillCoefficients(var);

  // BC is in the x-direction, sweep left to right
  if (u.bc.nonzero()) {
    sweepColumns(var);
    sweepRows(var);
  }
  // BC is in the y-direction, sweep south to north
  else {
    sweepRows(var);
    sweepColumns(var);
  }

  if (debug)
    var.print(var.string + " sweep solution = ", true);
}

void
Problem::sweepRows(Variable & var, const bool south_north)
{
  // Sweep south to north
  if (south_north)
    for (int j = 1; j < var.My; ++j)
      sweepRow(j, var);
  // Sweep north to south
  else
    for (int j = var.My - 1; j > 0; --j)
      sweepRow(j, var);
}

void
Problem::sweepColumns(Variable & var, const bool west_east)
{
  // Sweep west to east
  if (west_east)
    for (int i = 1; i < var.Mx; ++i)
      sweepColumn(i, var);
  // Sweep east to west
  else
    for (int i = var.Mx - 1; i > 0; --i)
      sweepColumn(i, var);
}

void
Problem::sweepColumn(const unsigned int i, Variable & var)
{
  if (debug)
    cout << "Solving " << var.string << " column " << i << endl;

  auto & A = var.Ay;
  auto & b = var.by;

  // Fill for each cell
  for (unsigned int j = 1; j < var.My; ++j)
  {
    const Coefficients & a = var.a(i, j);
    b[j - 1] = a.b + a.w * var(i - 1, j) + a.e * var(i + 1, j) + a.p * var(i, j) * (var.w - 1);
    if (j == 1)
    {
      A.setTopRow(a.p * var.w, -a.n);
```

```
      if (var.name != Variables::pc)
        b[j - 1] += a.s * var(i, j - 1);
    }
    else if (j == var.My - 1)
    {
      A.setBottomRow(-a.s, a.p * var.w);
      if (var.name != Variables::pc)
        b[j - 1] += a.n * var(i, j + 1);
    }
    else
      A.setMiddleRow(j - 1, -a.s, a.p * var.w, -a.n);
  }

  if (debug)
  {
    A.print("A =");
    b.print("b =");
  }

  // Solve
  A.solveTDMA(b);

  if (debug)
    b.print("sol =", true);

  // Store solution
  for (unsigned int j = 1; j < var.My; ++j)
    var(i, j) = b[j - 1];
}

void
Problem::sweepRow(const unsigned int j, Variable & var)
{
  if (debug)
    cout << "Solving " << var.string << " row " << j << endl;

  auto & A = var.Ax;
  auto & b = var.bx;

  // Fill for each cell
  for (unsigned int i = 1; i < var.Mx; ++i)
  {
    const Coefficients & a = var.a(i, j);
    b[i - 1] = a.b + a.s * var(i, j - 1) + a.n * var(i, j + 1) + a.p * var(i, j) * (var.w - 1);
    if (i == 1)
    {
      A.setTopRow(a.p * var.w, -a.e);
      if (var.name != Variables::pc)
        b[i - 1] += a.w * var(i - 1, j);
    }
    else if (i == var.Mx - 1)
    {
      A.setBottomRow(-a.w, a.p * var.w);
      if (var.name != Variables::pc)
        b[i - 1] += a.e * var(i + 1, j);
    }
    else
      A.setMiddleRow(i - 1, -a.w, a.p * var.w, -a.e);
  }

  if (debug)
  {
    A.print("A =");
    b.print("b =");
  }

  // Solve
  A.solveTDMA(b);

  if (debug)
    b.print("sol =", true);

  // Store solution
  for (unsigned int i = 1; i < var.Mx; ++i)
    var(i, j) = b[i - 1];
}

} // namespace Flow2D
```

Matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H

// #define NDEBUG
#include <cassert>
#include <fstream>
#include <vector>

using namespace std;

/**
 * Class that holds a N x M matrix with common matrix operations.
 */
template <typename T>
class Matrix
{
public:
  Matrix() {}
  Matrix(const unsigned int N, const unsigned int M) : N(N), M(M), A(N, vector<T>(M)) {}

  // Const operator for getting the (i, j) element
  const T & operator()(const unsigned int i, const unsigned int j) const
  {
    assert(i < N && j < M);
    return A[i][j];
  }
  // Operator for getting the (i, j) element
  T & operator()(const unsigned int i, const unsigned int j)
  {
    assert(i < N && j < M);
    return A[i][j];
  }
  // Operator for setting the entire matrix to a value
  void operator=(const T v)
  {
    for (unsigned int j = 0; j < M; ++j)
      setRow(j, v);
  }

  // Prints the matrix
  void print(const string prefix = "", const bool newline = false, const unsigned int pr = 5) const
  {
    if (prefix.length() != 0)
      cout << prefix << endl;
    for (unsigned int j = 0; j < M; ++j)
    {
      for (unsigned int i = 0; i < N; ++i)
        cout << showpos << scientific << setprecision(pr) << A[i][j] << " ";
      cout << endl;
    }
    if (newline)
      cout << endl;
  }
  // Saves the matrix in csv format
  void save(const string filename, const unsigned int pr = 12) const
  {
    ofstream f;
    f.open(filename);
    for (unsigned int j = 0; j < M; ++j)
    {
      for (unsigned int i = 0; i < N; ++i)
      {
        if (i > 0)
          f << ",";
        f << setprecision(pr) << A[i][j];
      }
      f << endl;
    }
    f.close();
  }

  // Set the j-th row to v
  void setRow(const unsigned int j, const T v)
  {
    assert(j < M);
    for (unsigned int i = 0; i < N; ++i)
      A[i][j] = v;
```

```
  }
  // Set the i-th column to v
  void setColumn(const unsigned int i, const T v)
  {
    assert(i < N);
    for (unsigned int j = 0; j < M; ++j)
      A[i][j] = v;
  }

private:
  // The size of this matrix
  const unsigned int N = 0, M = 0;

  // Matrix storage
  vector<vector<T>> A;
};

#endif /* MATRIX_H */
```

TriDiagonal.h

```
#ifndef TRIDIAGONAL_H
#define TRIDIAGONAL_H

#define NDEBUG
#include <cassert>
#include <fstream>
#include "Vector.h"

using namespace std;

/**
 * Class that holds a tri-diagonal matrix and is able to perform TDMA in place
 * with a given RHS.
 */
template <typename T>
class TriDiagonal
{
public:
  TriDiagonal() {}
  TriDiagonal(const unsigned int N, const T v = 0) : N(N), A(N, v), B(N, v), C(N - 1, v) {}

  // Setters for the top, middle, and bottom rows
  void setTopRow(const T b, const T c)
  {
    B[0] = b;
    C[0] = c;
  }
  void setMiddleRow(const unsigned int i, const T a, const T b, const T c)
  {
    assert(i < N - 1 && i != 0);
    A[i] = a;
    B[i] = b;
    C[i] = c;
  }
  void setBottomRow(const T a, const T b)
  {
    A[N - 1] = a;
    B[N - 1] = b;
  }

  // Prints the matrix
  void print(const string prefix = "", const bool newline = false, const unsigned int pr = 6) const
  {
    if (prefix.length() != 0)
      cout << prefix << endl;
    for (unsigned int i = 0; i < N; ++i)
      cout << showpos << scientific << setprecision(pr) << (i > 0 ? A[i] : 0) << " " << B[i] << " "
           << (i < N - 1 ? C[i] : 0) << endl;
    if (newline)
      cout << endl;
  }
  // Saves the matrix in csv format
  void save(const string filename, const unsigned int pr = 12) const
  {
    ofstream f;
    f.open(filename);
    for (unsigned int i = 0; i < N; ++i)
    {
      if (i > 0)
        f << setprecision(pr) << A[i] << ",";
      else
        f << "0"
          << ",";
      f << setprecision(pr) << B[i] << ",";
      if (i != N - 1)
        f << setprecision(pr) << C[i] << endl;
      else
        f << 0 << endl;
    }
    f.close();
  }

  // Solves the system Ax = d in place where d eventually stores the solution
  void solveTDMA(Vector<T> & d)
  {
    // Forward sweep
    T tmp = 0;
```

```
    for (unsigned int i = 1; i < N; ++i)
    {
      tmp = A[i] / B[i - 1];
      B[i] -= tmp * C[i - 1];
      d[i] -= tmp * d[i - 1];
    }

    // Backward sweep
    d[N - 1] /= B[N - 1];
    for (unsigned int i = N - 2; i != numeric_limits<unsigned int>::max(); --i)
    {
      d[i] -= C[i] * d[i + 1];
      d[i] /= B[i];
    }
  }

protected:
  // Matrix size (N x N)
  unsigned int N = 0;

  // Left/main/right diagonal storage
  vector<T> A, B, C;
};

#endif /* TRIDIAGONAL_H */
```

Vector.h

```
#ifndef VECTOR_H
#define VECTOR_H

#define NDEBUG
#include <cassert>
#include <fstream>
#include <vector>

using namespace std;

template <typename T>
class Vector
{
public:
  Vector(const unsigned int N) : v(N), N(N) {}
  Vector() {}

  const T & operator()(const unsigned int i) const
  {
    assert(i < N);
    return v[i];
  }
  T & operator()(const unsigned int i)
  {
    assert(i < N);
    return v[i];
  }
  const T & operator[](const unsigned int i) const
  {
    assert(i < N);
    return v[i];
  }
  T & operator[](const unsigned int i)
  {
    assert(i < N);
    return v[i];
  }

  // Prints the vector
  void print(const string prefix = "", const bool newline = false, const unsigned int pr = 6) const
  {
    if (prefix.length() != 0)
      cout << prefix << endl;
    for (unsigned int i = 0; i < v.size(); ++i)
      cout << showpos << scientific << setprecision(pr) << v[i] << " ";
    cout << endl;
    if (newline)
      cout << endl;
  }

  // Saves the vector
  void save(const string filename, const unsigned int pr = 12) const
  {
    ofstream f;
    f.open(filename);
    for (unsigned int i = 0; i < v.size(); ++i)
      f << scientific << v[i] << endl;
    f.close();
  }

private:
  vector<T> v;
  const unsigned int N = 0;
};

#endif /* VECTOR_H */
```

plots.py

```python
import numpy as np
import matplotlib.pyplot as plt

plt.rc('text', usetex=True)
plt.rc('font', family='serif')

# Load problem 2 results
L = 0.2
Re = 400
rho = 998.3
mu = 1.002e-3
u_ref = Re * mu / (rho * L)
Ns = [8, 16, 32, 64, 128, 256]
x, y, u, v = {}, {}, {}, {}
for N in Ns:
    valu = np.loadtxt('results/{}_u.csv'.format(N), delimiter=',')
    u[N] = valu[:, int(N / 2)] / u_ref
    y[N] = np.linspace(0, 1, num=len(u[N]))
    valv = np.loadtxt('results/{}_v.csv'.format(N), delimiter=',')
    v[N] = valv[int(N / 2), :] / u_ref
    x[N] = np.linspace(0, 1, num=len(v[N]))

# Problem 2 reference
ref_u = [0, -0.03177, -0.06189, -0.08923, -0.11732, -0.14410, -0.17257, -0.19996,
         -0.22849, -0.25458, -0.27956, -0.11446, 0.35427, 0.37502, 0.40187,
         0.43679, 0.48695, 0.54512, 0.61626, 0.70001, 0.79419, 0.90472, 1.0]
ref_y = [0, 0.02, 0.0405, 0.0601, 0.0806, 0.1001, 0.1206, 0.1401, 0.1606, 0.1802,
         0.2007, 0.5005, 0.9009, 0.9106, 0.9204, 0.9302, 0.9409,  0.9507, 0.9604,
         0.9702, 0.9800, 0.9907, 1]
ref_v = [0, 0.05951, 0.11028, 0.14906, 0.18047, 0.20578, 0.22746, 0.24397,
         0.25854, 0.27001, 0.27667, 0.05146, -0.44994, -0.45381, -0.44362,
         -0.41888, -0.37613, -0.32251, -0.25931, -0.19118, -0.11873, -0.05590, 0]
ref_x = [0, 0.0151, 0.0308, 0.0454, 0.0600, 0.0747, 0.0902, 0.1049, 0.1206,
         0.1352, 0.1450, 0.5005, 0.8501, 0.8647, 0.8804, 0.8950, 0.9106, 0.9253,
         0.9399, 0.9546, 0.9702, 0.9849, 1]

# Problem 2 plot
fig, ax = plt.subplots(1, 2)
fig.set_figwidth(9)
fig.set_figheight(3.5)
for N in Ns:
    ax[0].plot(y[N], u[N], label='{}x{}'.format(N, N), linewidth=1)
    ax[1].plot(x[N], v[N], linewidth=1)
ax[0].plot(ref_y, ref_u, '.k', label='Roy et. al (2015)', markersize=4)
ax[1].plot(ref_x, ref_v, '.k', markersize=4)
ax[0].set_xlabel(r'Normalized $y$')
ax[0].set_ylabel(r'Normalized $u$')
ax[1].set_xlabel(r'Normalized $x$')
ax[1].set_ylabel(r'Normalized $v$')
ax[0].grid()
ax[1].grid()
handles, labels = ax[0].get_legend_handles_labels()
lgd = ax[0].legend(handles, labels, loc='lower center', bbox_to_anchor=(1.0, -0.29),
                   ncol=7, fontsize=9)
fig.tight_layout()
fig.savefig('results/p2.pdf', bbox_inches='tight', bbox_extra_artists=(lgd,))
```