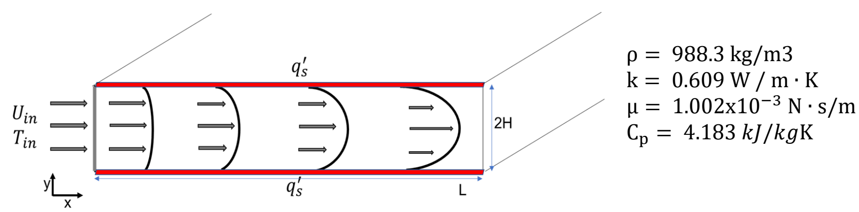
# MEEN 644 - Homework 4

Logan Harbour
April 11, 2019

## 1 Problem statement




Consider an incompressible laminar flow between two infinite parallel plates. Flow enters at constant velocity $u_{\text{in}}$ and aconstant heat flux, $q' = 500$ W/m$^2$ is applied to each wall. Height of channel $2H$ and length of channel $L$ are 0.02 m and 2.0 m, respectively. Constant velocity $u_{\text{in}}$ and temperature $T_{\text{in}} = 25°$C is set as the inlet condition. Reynolds number is defined as $Re = 2\rho u_{\text{in}} H/\mu$.

1. **(10 points)** Specify your boundary condition for $u, v, P$, and $T$.
2. **(50 points)** Write a finite volume method code to predict velocity, pressure, and temperature profiles for Re = 100. Employ the Power Law scheme to represent a solution to a 1-D convection-diffusion equation. Use the SIMPLE algorithm to link velocity and pressure fields. Use 10 uniformly sized CVs in the $x$-direction and 5 uniformly sized CVs in the $y$-direction. Declare convergence at $R_u, R_v, R_P$ and $R_T < 10^{-6}$.
3. **(10 points)** Run your program with the 10 × 5 grid and tabulate $u, (P - P_0)$, and $T$ to check for symmetry. $P_0$ is the pressure at the outlet.
4. **(30 points)** Use a 180 × 54 grid to solve for $u, v, P$, and $T$.
   (a) Plot $u/u_{\text{in}}$ along the centerline of the channel (as a function of $x$).
   (b) Plot $u, v$, and $T$ at $x = 0.8$m m (as a function of $y$).
   (c) Plot the Nusselt number as a function of stream-wise distance $x$ from the channel entrance.

## 2 Preliminaries

### 2.1 Two-dimensional diffusion-convection

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

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

with the boundary conditions

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

## 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. Correct for the $u$-velocity outflow boundary condition.
7. Check for convergence. If not converged, return to 2.
8. Solve the auxiliary temperature until convergence.

## 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, $\phi$, is defined as $(M_x^\phi, M_y^\phi)$.

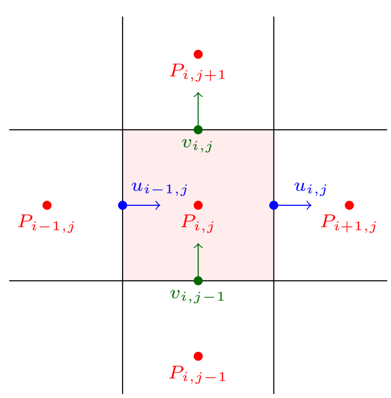



Figure 1: An internal pressure control volume.

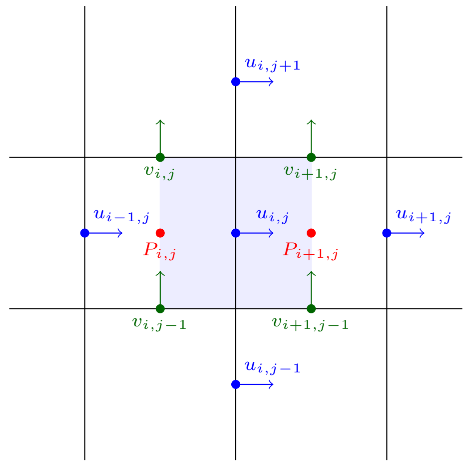



Figure 2: An internal $u$-velocity control volume.

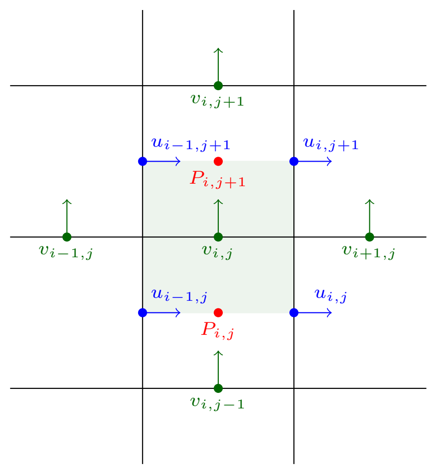



Figure 3: An internal $v$-velocity control volume.

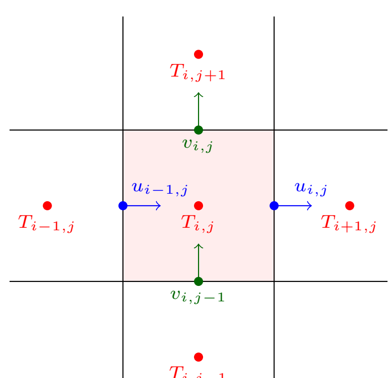



Figure 4: An internal temperature control volume.

## 2.4 Velocity guess

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

$$P_{\mathrm{bd}}^{\phi_{i,j}} = \frac{F_{\mathrm{bd}}^{\phi_{i,j}}}{D_{\mathrm{bd}}^{\phi_{i,j}}}\,, \quad \text{where} \quad \mathrm{bd} = [n, e, s, w] \quad \text{and} \quad \phi = [u, v]\,. \tag{3}$$

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 $i = 1, \ldots, M_x^\phi - 1$ , $j = 1, \ldots, M_y^\phi - 1$)

$$a_p^{\phi_{i,j}} \phi_{i,j}^* = a_n^{\phi_{i,j}} \phi_{i,j+1}^* + a_e^{\phi_{i,j}} \phi_{i+1,j}^* + a_s^{\phi_{i,j}} \phi_{i,j-1}^* + a_w^{\phi_{i,j}} \phi_{i-1,j}^* + a_b^{\phi_{i,j}}\,, \tag{4a}$$

$$a_n^{\phi_{i,j}} = D_n^{\phi_{i,j}} \max\left[0, (1 - 0.1|P_n^{\phi_{i,j}}|)^5\right] + \max\left[-F_n^{\phi_{i,j}}, 0\right]\,, \tag{4b}$$

$$a_e^{\phi_{i,j}} = D_e^{\phi_{i,j}} \max\left[0, (1 - 0.1|P_e^{\phi_{i,j}}|)^5\right] + \max\left[-F_e^{\phi_{i,j}}, 0\right]\,, \tag{4c}$$

$$a_s^{\phi_{i,j}} = D_s^{\phi_{i,j}} \max\left[0, (1 - 0.1|P_s^{\phi_{i,j}}|)^5\right] + \max\left[F_s^{\phi_{i,j}}, 0\right]\,, \tag{4d}$$

$$a_w^{\phi_{i,j}} = D_w^{\phi_{i,j}} \max\left[0, (1 - 0.1|P_w^{\phi_{i,j}}|)^5\right] + \max\left[F_w^{\phi_{i,j}}, 0\right]\,, \tag{4e}$$

$$a_p^{\phi_{i,j}} = a_n^{\phi_{i,j}} + a_e^{\phi_{i,j}} + a_s^{\phi_{i,j}} + a_w^{\phi_{i,j}}\,, \tag{4f}$$

$$a_b^{\phi_{i,j}} = \begin{cases} \Delta y(p_{i,j}^* - p_{i+1,j}^*)\,, & \phi = u \\ \Delta x(p_{i,j}^* - p_{i,j+1}^*)\,, & \phi = v \end{cases}\,. \tag{4g}$$

### 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

$$\Delta x^{u_{i,j}} = \begin{cases} \Delta x\,, & 1 < i < M_x^u - 1 \\ \frac{3}{2}\Delta x\,, & \text{otherwise} \end{cases}\,, \tag{5}$$

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

$$\delta y_{p_n}^{u_{i,j}} = \begin{cases} \frac{1}{2}\Delta y\,, & j = M_y^u - 1\,, \\ \Delta y\,, & \text{otherwise} \end{cases}\,, \tag{6a}$$

$$\delta y_{p_s}^{u_{i,j}} = \begin{cases} \frac{1}{2}\Delta y\,, & j = 1\,, \\ \Delta y\,, & \text{otherwise} \end{cases}\,. \tag{6b}$$

The diffusion coefficients are then defined as

$$D_n^{u_{i,j}} = \frac{\mu \Delta x^{u_{i,j}}}{\delta y_{p_n}^{u_{i,j}}}\,, \tag{7a}$$

$$D_e^{u_{i,j}} = \frac{\mu \Delta y}{\Delta x}\,, \tag{7b}$$

$$D_s^{u_{i,j}} = \frac{\mu \Delta x^{u_{i,j}}}{\delta y_{p_s}^{u_{i,j}}}\,, \tag{7c}$$

$$D_w^{u_{i,j}} = \frac{\mu \Delta y}{\Delta x}\,. \tag{7d}$$

Lastly, the flow rates are defined as

$$F_n^{u_{i,j}} = \rho \Delta x^{u_{i,j}} \begin{cases} \frac{1}{6}\left(v^*_{0,j} + 3v^*_{1,j} + 2v^*_{2,j}\right) \,, & i = 1 \\ \frac{1}{6}\left(2v^*_{i,j} + 3v^*_{i+1,j} + v^*_{i+2,j}\right) \,, & i = M_x^u - 1 \\ \frac{1}{2}\left(v^*_{i,j} + v^*_{i+1,j}\right) \,, & \text{otherwise} \end{cases} \,, \tag{8a}$$

$$F_e^{u_{i,j}} = \rho \Delta y \begin{cases} u^*_{M_x^u,j} \,, & i = M_x^u - 1 \\ \frac{1}{2}\left(u^*_{i+1,j} + u^*_{i,j}\right) \,, & \text{otherwise} \end{cases} \,, \tag{8b}$$

$$F_s^{u_{i,j}} = \rho \Delta x^{u_{i,j}} \begin{cases} \frac{1}{6}\left(v^*_{0,j-1} + 3v^*_{1,j-1} + 2v^*_{2,j-1}\right) \,, & i = 1 \\ \frac{1}{6}\left(2v^*_{i,j-1} + 3v^*_{i+1,j-1} + v^*_{i+2,j-1}\right) \,, & i = M_x^u - 1 \\ \frac{1}{2}\left(v^*_{i,j-1} + v^*_{i+1,j-1}\right) \,, & \text{otherwise} \end{cases} \tag{8c}$$

$$F_w^{u_{i,j}} = \rho \Delta y \begin{cases} u^*_{0,j} \,, & i = 1 \\ \frac{1}{2}\left(u^*_{i-1,j} + u^*_{i,j}\right) \,, & \text{otherwise} \end{cases} \,. \tag{8d}$$

### 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

$$\Delta y^{v_{i,j}} = \begin{cases} \Delta y \,, & 1 < j < M_y^v - 1 \\ \frac{3}{2}\Delta y \,, & \text{otherwise} \end{cases} \,, \tag{9}$$

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

$$\delta x_{p_e}^{v_{i,j}} = \begin{cases} \frac{1}{2}\Delta x \,, & i = M_x^v - 1 \\ \Delta x \,, & \text{otherwise} \end{cases} \,, \tag{10a}$$

$$\delta x_{p_w}^{v_{i,j}} = \begin{cases} \frac{1}{2}\Delta x \,, & i = 1 \,, \\ \Delta x \,, & \text{otherwise} \end{cases} \,. \tag{10b}$$

The diffusion coefficients are then defined as

$$D_n^{v_{i,j}} = \frac{\mu \Delta x}{\Delta y} \,, \tag{11a}$$

$$D_e^{v_{i,j}} = \frac{\mu \Delta y^{v_{i,j}}}{\delta x_{p_e}^{v_{i,j}}} \,, \tag{11b}$$

$$D_s^{v_{i,j}} = \frac{\mu \Delta x}{\Delta y} \,, \tag{11c}$$

$$D_w^{v_{i,j}} = \frac{\mu \Delta y^{v_{i,j}}}{\delta x_{p_w}^{v_{i,j}}} \,. \tag{11d}$$

Lastly, the flow rates are defined as

$$F_n^{v_{i,j}} = \rho \Delta x \begin{cases} v^*_{i,M^v_y} , & j = M^v_y - 1 \\ \frac{1}{2}\left(v^*_{i,j+1} + v^*_{i,j}\right) , & \text{otherwise} \end{cases} , \tag{12a}$$

$$F_e^{v_{i,j}} = \rho \Delta y^{v_{i,j}} \begin{cases} \frac{1}{6}\left(u^*_{i,0} + 3u^*_{i,1} + 2u^*_{i,2}\right) , & j = 1 \\ \frac{1}{6}\left(2u^*_{i,j} + 3u^*_{i,j+1} + 2u^*_{i,j+2}\right) , & j = M^v_y - 1 \\ \frac{1}{2}\left(u^*_{i,j} + u^*_{i,j+1}\right) , & \text{otherwise} \end{cases} , \tag{12b}$$

$$F_s^{v_{i,j}} = \rho \Delta x \begin{cases} v^*_{i,0} , & j = 1 \\ \frac{1}{2}\left(v^*_{i,j+1} + v^*_{i,j}\right) , & \text{otherwise} \end{cases} , \tag{12c}$$

$$F_w^{v_{i,j}} = \rho \Delta y^{v_{i,j}} \begin{cases} \frac{1}{6}\left(u^*_{i-1,0} + 3u^*_{i-1,1} + 2u^*_{i-1,2}\right) , & j = 1 \\ \frac{1}{6}\left(2u^*_{i-1,j} + 3u^*_{i-1,j+1} + 2u^*_{i-1,j+2}\right) , & j = M^v_y - 1 \\ \frac{1}{2}\left(u^*_{i-1,j} + u^*_{i-1,j+1}\right) , & \text{otherwise} \end{cases} . \tag{12d}$$

## 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 (4) one obtains

$$a_p^{u_{i,j}} u'_{i,j} = a_n^{u_{i,j}} u'_{i,j+1} + a_e^{u_{i,j}} u'_{i+1,j} + a_s^{u_{i,j}} u'_{i,j-1} + a_w^{u_{i,j}} u'_{i-1,j} + \Delta y (p'_{i,j} - p'_{i+1,j}) , \tag{13a}$$

$$a_p^{v_{i,j}} v'_{i,j} = a_n^{v_{i,j}} v'_{i,j+1} + a_e^{v_{i,j}} v'_{i+1,j} + a_s^{v_{i,j}} v'_{i,j-1} + a_w^{v_{i,j}} v'_{i-1,j} + \Delta x (p'_{i,j} - p'_{i,j+1}) . \tag{13b}$$

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

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

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

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

$$a_p^{p'_{i,j}} p'_{i,j} = a_n^{p'_{i,j}} p'_{i,j+1} + a_e^{p'_{i,j}} p'_{i+1,j} + a_s^{p'_{i,j}} p'_{i,j-1} + a_w^{p'_{i,j}} p'_{i-1,j} + a_b^{p'_{i,j}} , \tag{15a}$$

$$a_n^{p'_{i,j}} = \begin{cases} \rho \Delta x^2 / a_p^{v_{i,j}} , & j < M^p_y - 1 \\ 0 , & \text{otherwise} \end{cases} , \tag{15b}$$

$$a_e^{p'_{i,j}} = \begin{cases} \rho \Delta y^2 / a_p^{u_{i,j}} , & i < M^p_x - 1 \\ 0 , & \text{otherwise} \end{cases} , \tag{15c}$$

$$a_s^{p'_{i,j}} = \begin{cases} \rho \Delta x^2 / a_p^{v_{i,j-1}} , & j > 1 \\ 0 , & \text{otherwise} \end{cases} , \tag{15d}$$

$$a_w^{p'_{i,j}} = \begin{cases} \rho \Delta y^2 / a_p^{u_{i-1,j}} , & i > 1 \\ 0 , & \text{otherwise} \end{cases} , \tag{15e}$$

$$a_p^{p'_{i,j}} = a_n^{p'_{i,j}} + a_e^{p'_{i,j}} + a_s^{p'_{i,j}} + a_w^{p'_{i,j}} , \tag{15f}$$

$$a_b^{p'_{i,j}} = \rho \left( \Delta y (u^*_{i-1,j} - u^*_{i,j}) + \Delta x (v^*_{i,j-1} - v^*_{i,j}) \right) . \tag{15g}$$

## 2.6 Velocity and pressure correction

Lastly, the velocities are then updated using Equation (14) with

$$u_{i,j} = u_{i,j} + \frac{\Delta y}{a_p^{u_{i,j}}}(p'_{i,j} - p'_{i+1,j}) \,, \quad i = 1, \ldots, M_x^u - 1 \,, \quad j = 1, \ldots, M_y^u - 1 \,, \tag{16a}$$

$$v_{i,j} = v_{i,j} + \frac{\Delta x}{a_p^{v_{i,j}}}(p'_{i,j} - p'_{i,j+1}) \,, \quad i = 1, \ldots, M_x^v - 1 \,, \quad j = 1, \ldots, M_y^v - 1 \,, \tag{16b}$$

and the pressures (take note of the relaxation parameter $\alpha_p$) with

$$p_{i,j} = p_{i,j} + \alpha_p p'_{i,j} \,, \quad i = 1, \ldots, M_x^p - 1 \,, \quad j = 1, \ldots, M_y^p - 1 \,. \tag{17}$$

## 2.7 System solver

The systems in Equations (4) and (15) 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

$$a_p^{i,j}\phi_{i,j} = a_n^{i,j}\phi_{i,j+1} + a_e^{i,j}\phi_{i+1,j} + a_s^{i,j}\phi_{i,j-1} + a_w^{i,j}\phi_{i-1,j} + a_b^{i,j} \,, \quad i = 1, \ldots, N_x \,, \quad j = 1, \ldots, N_y \,. \tag{18}$$

Now, consider $\phi^*$ to be a *lagged* value of $\phi$, 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:

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

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

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

# 3 Results

## 3.1 Problem 3: 10 × 5 grid

The results requested for problem 3 follow in Tables 1, 2, and 3.

Table 1: The $u$-velocity solution with the 10 × 5 grid. A row corresponds to a $x$-position and a column corresponds to an $y$-position.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 5.06931E−3 | 5.06931E−3 | 5.06931E−3 | 5.06931E−3 | 5.06931E−3 | 5.06931E−3 | 5.06931E−3 |
| 2 | 0.00000E0 | 3.02156E−3 | 6.10805E−3 | 7.08733E−3 | 6.10805E−3 | 3.02156E−3 | 0.00000E0 |
| 3 | 0.00000E0 | 2.84164E−3 | 6.18930E−3 | 7.28469E−3 | 6.18930E−3 | 2.84164E−3 | 0.00000E0 |
| 4 | 0.00000E0 | 2.81967E−3 | 6.19551E−3 | 7.31618E−3 | 6.19551E−3 | 2.81967E−3 | 0.00000E0 |
| 5 | 0.00000E0 | 2.81677E−3 | 6.19585E−3 | 7.32132E−3 | 6.19585E−3 | 2.81677E−3 | 0.00000E0 |
| 6 | 0.00000E0 | 2.81636E−3 | 6.19583E−3 | 7.32217E−3 | 6.19583E−3 | 2.81636E−3 | 0.00000E0 |
| 7 | 0.00000E0 | 2.81630E−3 | 6.19583E−3 | 7.32231E−3 | 6.19583E−3 | 2.81630E−3 | 0.00000E0 |
| 8 | 0.00000E0 | 2.81629E−3 | 6.19582E−3 | 7.32233E−3 | 6.19582E−3 | 2.81629E−3 | 0.00000E0 |
| 9 | 0.00000E0 | 2.81628E−3 | 6.19582E−3 | 7.32234E−3 | 6.19582E−3 | 2.81628E−3 | 0.00000E0 |
| 10 | 0.00000E0 | 2.81628E−3 | 6.19582E−3 | 7.32234E−3 | 6.19582E−3 | 2.81628E−3 | 0.00000E0 |
| 11 | 0.00000E0 | 2.81628E−3 | 6.19582E−3 | 7.32234E−3 | 6.19582E−3 | 2.81628E−3 | 0.00000E0 |

Table 2: The $(P - P_0)$ solution with the $10 \times 5$ grid. A row corresponds to a $x$-position and a column corresponds to an $y$-position.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 2.88374E−1 | 2.88374E−1 | 2.88353E−1 | 2.88346E−1 | 2.88353E−1 | 2.88374E−1 | 2.88374E−1 |
| 2 | 2.88374E−1 | 2.88374E−1 | 2.88353E−1 | 2.88346E−1 | 2.88353E−1 | 2.88374E−1 | 2.88374E−1 |
| 3 | 2.40648E−1 | 2.40648E−1 | 2.40651E−1 | 2.40653E−1 | 2.40651E−1 | 2.40648E−1 | 2.40648E−1 |
| 4 | 2.11749E−1 | 2.11749E−1 | 2.11750E−1 | 2.11750E−1 | 2.11750E−1 | 2.11749E−1 | 2.11749E−1 |
| 5 | 1.83440E−1 | 1.83440E−1 | 1.83440E−1 | 1.83440E−1 | 1.83440E−1 | 1.83440E−1 | 1.83440E−1 |
| 6 | 1.55208E−1 | 1.55208E−1 | 1.55208E−1 | 1.55208E−1 | 1.55208E−1 | 1.55208E−1 | 1.55208E−1 |
| 7 | 1.26987E−1 | 1.26987E−1 | 1.26987E−1 | 1.26987E−1 | 1.26987E−1 | 1.26987E−1 | 1.26987E−1 |
| 8 | 9.87671E−2 | 9.87671E−2 | 9.87671E−2 | 9.87671E−2 | 9.87671E−2 | 9.87671E−2 | 9.87671E−2 |
| 9 | 7.05479E−2 | 7.05479E−2 | 7.05479E−2 | 7.05479E−2 | 7.05479E−2 | 7.05479E−2 | 7.05479E−2 |
| 10 | 4.23287E−2 | 4.23287E−2 | 4.23287E−2 | 4.23287E−2 | 4.23287E−2 | 4.23287E−2 | 4.23287E−2 |
| 11 | 0.00000E0 | 0.00000E0 | 0.00000E0 | 0.00000E0 | 0.00000E0 | 0.00000E0 | 0.00000E0 |
| 12 | 0.00000E0 | 0.00000E0 | 0.00000E0 | 0.00000E0 | 0.00000E0 | 0.00000E0 | 0.00000E0 |

Table 3: The $T$ solution with the $10 \times 5$ grid. A row corresponds to a $x$-position and a column corresponds to an $y$-position.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 26.64204 | 25.00000 | 25.00000 | 25.00000 | 25.00000 | 25.00000 | 26.64204 |
| 2 | 27.69621 | 26.05418 | 25.36548 | 25.17769 | 25.36548 | 26.05418 | 27.69621 |
| 3 | 28.84253 | 27.20050 | 25.72681 | 25.36877 | 25.72681 | 27.20050 | 28.84253 |
| 4 | 29.72586 | 28.08382 | 26.15140 | 25.63232 | 26.15140 | 28.08382 | 29.72586 |
| 5 | 30.44158 | 28.79955 | 26.60972 | 25.95974 | 26.60972 | 28.79955 | 30.44158 |
| 6 | 31.06105 | 29.41902 | 27.08232 | 26.33547 | 27.08232 | 29.41902 | 31.06105 |
| 7 | 31.62445 | 29.98241 | 27.56009 | 26.74529 | 27.56009 | 29.98241 | 31.62445 |
| 8 | 32.15448 | 30.51244 | 28.03926 | 27.17831 | 28.03926 | 30.51244 | 32.15448 |
| 9 | 32.66429 | 31.02226 | 28.51841 | 27.62677 | 28.51841 | 31.02226 | 32.66429 |
| 10 | 33.16165 | 31.51962 | 28.99712 | 28.08535 | 28.99712 | 31.51962 | 33.16165 |
| 11 | 33.65119 | 32.00915 | 29.47532 | 28.55050 | 29.47532 | 32.00915 | 33.65119 |
| 12 | 34.14073 | 32.49869 | 29.95352 | 29.01564 | 29.95352 | 32.49869 | 34.14073 |

## 3.2 Problem 4: 180 × 54 grid

The requirements for problem (b) part i and ii were combined into Figure **??** 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.

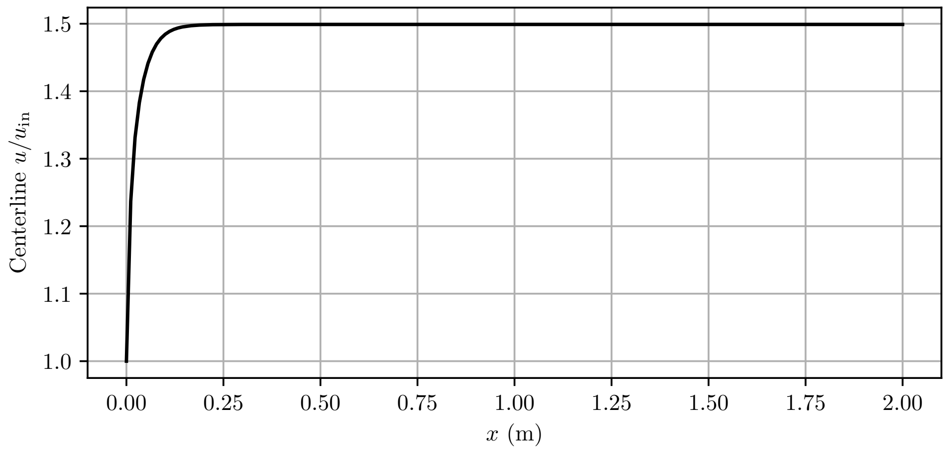



Figure 5: $u/u_{\mathrm{in}}$ plotted along the centerline of the channel for the 180 $\times$ 54 grid.

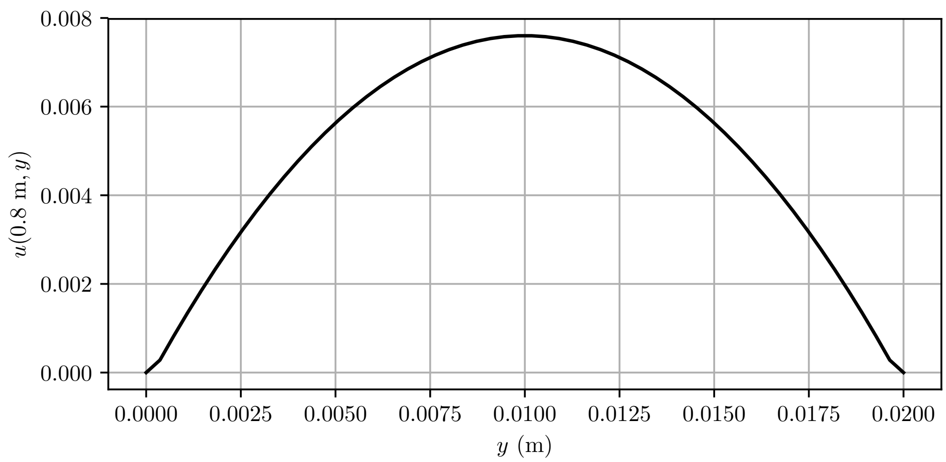



Figure 6: $u$ plotted at $x = 0.8$ m for the 180 $\times$ 54 grid.

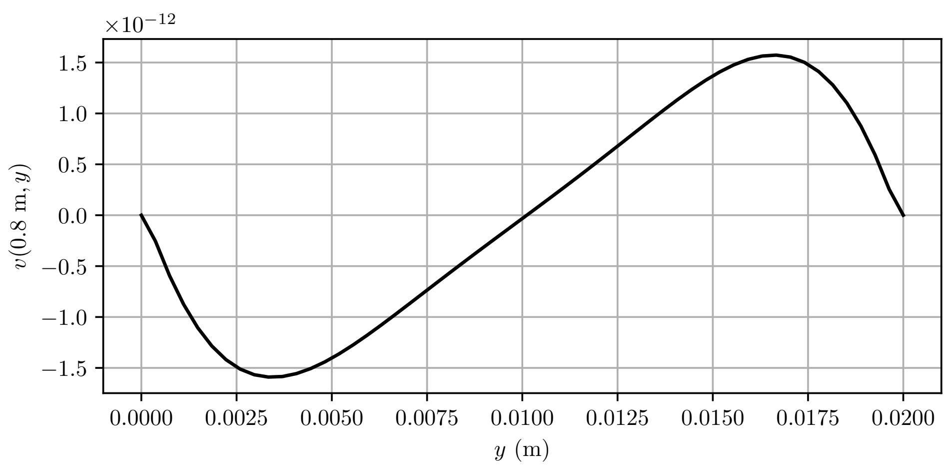



Figure 7: $v$ plotted at $x = 0.8$ m for the 180 $\times$ 54 grid.

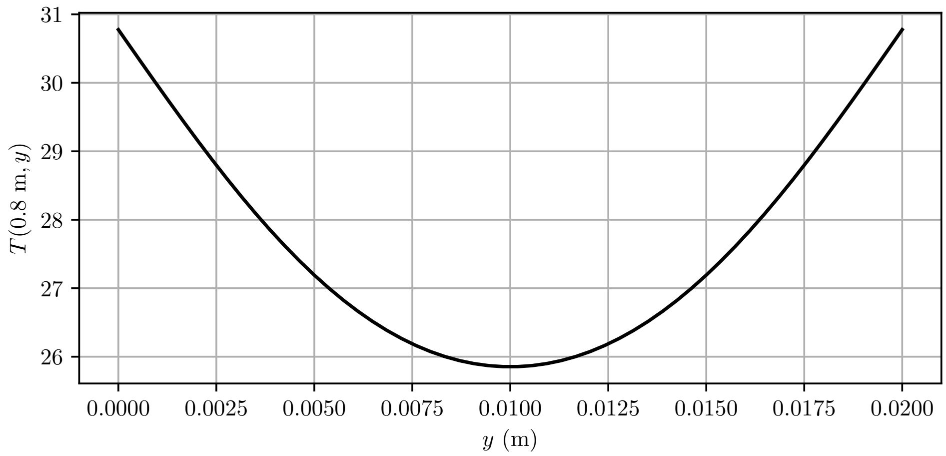



Figure 8: $T$ plotted at $x = 0.8$ m for the $180 \times 54$ grid.

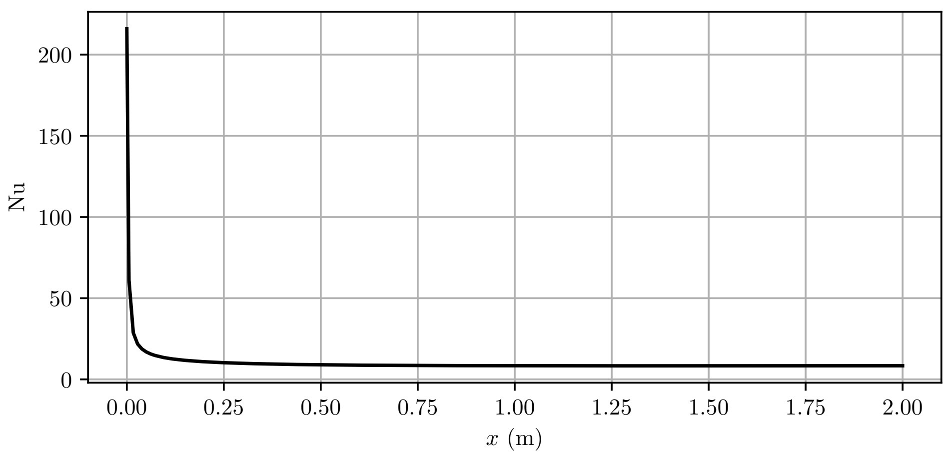



Figure 9: The Nusselt number plotted as a function of stream-wise distance for the $180 \times 54$ grid.

## 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.
- `postprocess.py` - Produces the plots and tables 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
```

## hwk5.cpp

```
#include "Problem.h"

using namespace Flow2D;

int
main()
{
  // Problem wide constants
  double Lx = 2;
  double Ly = 0.02;
  double cp = 4183;
  double k = 0.609;
  double rho = 988.3;
  double mu = 0.001002;
  double Re = 100;
  double u_bc_val = Re * mu / (rho * Ly);
  double T_bc_val = 25;
  double q_bc_val = 500;

  // Standard inputs
  InputArguments input;
  input.Lx = Lx;
  input.Ly = Ly;
  input.cp = cp;
  input.k = k;
  input.mu = mu;
  input.rho = rho;
  input.u_ref = u_bc_val;
  input.L_ref = Lx;
  input.T_left = T_bc_val;
  input.u_bc = BoundaryCondition(0, 0, 0, u_bc_val);
  input.v_bc = BoundaryCondition(0, 0, 0, 0);
  input.q_top_bot = q_bc_val;

  // Problem 3: check symmetry
  std::cout << "Problem 3: check symmetry" << std::endl;
  Problem problem3(10, 5, input);
  problem3.run();
  problem3.save(Variables::u, "results/coarse_u.csv");
  problem3.save(Variables::p, "results/coarse_p.csv");
```

```
  problem3.save(Variables::T, "results/coarse_T.csv");

  // Problem 4: 180x54 grid
  std::cout << "Problem 4: 180 x 54 grid" << std::endl;
  Problem problem4(180, 54, input);
  problem4.run();
  problem4.save(Variables::u, "results/fine_u.csv");
  problem4.save(Variables::v, "results/fine_v.csv");
  problem4.save(Variables::T, "results/fine_T.csv");
}
```

## Problem.h

```
#ifndef PROBLEM_H
#define PROBLEM_H

#include <cmath>
#include <ctime>
#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 T_left, q_top_bot;
  double L_ref, u_ref;
  double cp, k, mu, rho;
  bool debug = false;
  double alpha_p = 0.7;
  double alpha_uv = 0.5;
  unsigned int max_main_its = 50000;
  unsigned int max_aux_its = 20000;
  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 correctMain();
  void correctAux();
  void pCorrect();
  void pBCCorrect();
  void TBCCorrect();
  void uCorrect();
  void uBCCorrect();
  void vCorrect();

  // Problem_coefficients.cpp
  void fillCoefficients(const Variable & var);
  void pcCoefficients();
  void TCoefficients();
  void uCoefficients();
  void vCoefficients();
  void fillPowerLaw(Coefficients & a,
                    const Coefficients & D,
                    const Coefficients & F,
                    const double & b = 0);

  // Problem_residuals.cpp
  void computeMainResiduals();
  void computeAuxResiduals();
  double pResidual() const;
  double TResidual() const;
  double velocityResidual(const Variable & var) const;
```

```
  // Problem_solvers.cpp
  void solveMain();
  void solveAux();
  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();

  // Quicker v^5 for velocityCoefficients() (yes, it's actually much faster...)
  static const double pow5(const double & v) { return v * v * v * v * v; }

protected:
  // Number of pressure CVs
  const unsigned int Nx, Ny;

  // Geometry [m]
  const double Lx, Ly, dx, dy;
  // Material properties
  const double cp, k, mu, rho;
  // Residual references
  const double L_ref, u_ref;
  // Other boundary conditions
  const double q_top_bot;
  // Mass inflow
  const double m_in;

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

  // Maximum iterations
  const unsigned int max_main_its, max_aux_its;
  // Iteration tolerance
  const double tol;
  // Pressure relaxation
  const double alpha_p;
  // Number of iterations completed
  unsigned int main_iterations = 0;
  unsigned int aux_iterations = 0;

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

  // Whether or not we converged
  bool main_converged = false;
  bool converged = false;
  // Run start time
  clock_t start;
};

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

# Variable.h

```cpp
#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)
  {
  }
  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) const
  {
    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,
  T
};

// 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";
    case Variables::T:
      return "T";
  }
}

// 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),
```

```cpp
      My(Ny - 1),
      w(1 / alpha),
      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
  void operator=(const double v) { phi = v; }
  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) const
  {
    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),
    // Other boundary conditions
    q_top_bot(input.q_top_bot),
    // Mass inflow
    m_in(input.u_bc.left * input.rho * Ly),
    // Material properties
    cp(input.cp),
    k(input.k),
    mu(input.mu),
    rho(input.rho),
    // Enable debug
    debug(input.debug),
    // Solver properties
    max_main_its(input.max_main_its),
    max_aux_its(input.max_aux_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),
    T(Variables::T, 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);
  variables.emplace(Variables::T, T);

  // T initial condition
  T = input.T_left;
}

void
Problem::run()
{
  // Store start time
  start = clock();

  // Solve main variables
  for (unsigned int l = 0; l < max_main_its; ++l)
  {
    solveMain();
    correctMain();
    computeMainResiduals();

    // Break out if we've converged
    if (main_converged)
      break;
  }

  // Ensure main variables converged
  if (!main_converged)
    cout << "Main variables did not converge after " << max_main_its << " iterations!" << endl;

  // Solve aux variables
  for (unsigned int l = 0; l < max_aux_its; ++l)
  {
```

```
    solveAux();
    correctAux();
    computeAuxResiduals();

    // Exit if everything is converged
    if (converged)
      return;
  }

  // Oops. Didn't converge
  cout << "Aux variables did not converge after " << max_aux_its << " iterations!" << endl;
}
} // namespace Flow2D
```

## Problem_coefficients.cpp

```cpp
#include "Problem.h"

namespace Flow2D
{

void
Problem::fillCoefficients(const Variable & var)
{
  if (var.name == Variables::pc)
    pcCoefficients();
  else if (var.name == Variables::T)
    TCoefficients();
  else if (var.name == Variables::u)
    uCoefficients();
  else if (var.name == Variables::v)
    vCoefficients();

  if (debug)
  {
    cout << var.string << " coefficients: " << endl;
    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::TCoefficients()
{
  Coefficients D, F;

  for (unsigned int i = 1; i < T.Mx; ++i)
    for (unsigned int j = 1; j < T.My; ++j)
    {
      // Diffusion coefficient
      D.n = (j == T.My - 1 ? 2 * dx * k / dy : dx * k / dy);
      D.e = (i == T.Mx - 1 ? 2 * dy * k / dx : dy * k / dx);
      D.s = (j == 1 ? 2 * dx * k / dy : dx * k / dy);
      D.w = (i == 1 ? 2 * dy * k / dx : dy * k / dx);

      // Heat flows
      F.n = dx * cp * rho * v(i, j);
      F.e = dy * cp * rho * u(i, j);
      F.s = dx * cp * rho * v(i, j - 1);
      F.w = dy * cp * rho * u(i - 1, j);

      // Compute and store power law coefficients
      fillPowerLaw(T.a(i, j), D, F);
    }
}

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
      fillPowerLaw(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
      fillPowerLaw(v.a(i, j), D, F, b);
    }
}

void
Problem::fillPowerLaw(Coefficients & a,
                      const Coefficients & D,
                      const Coefficients & F,
                      const double & b)
{
  a.n = D.n * fmax(0, pow5(1 - 0.1 * fabs(F.n / D.n))) + fmax(-F.n, 0);
  a.e = D.e * fmax(0, pow5(1 - 0.1 * fabs(F.e / D.e))) + fmax(-F.e, 0);
  a.s = D.s * fmax(0, pow5(1 - 0.1 * fabs(F.s / D.s))) + fmax(F.s, 0);
  a.w = D.w * fmax(0, pow5(1 - 0.1 * fabs(F.w / D.w))) + fmax(F.w, 0);
  a.p = a.n + a.e + a.s + a.w;
  a.b = b;
}

} // namespace Flow2D
```

## Problem_corrections.cpp

```cpp
#include "Problem.h"

namespace Flow2D
{

void
Problem::correctMain()
{
  uCorrect();
  vCorrect();
  pCorrect();
  pBCCorrect();
  uBCCorrect();
}

void
Problem::correctAux()
{
  TBCCorrect();
}

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();

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

void
Problem::pBCCorrect()
{
  // 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 boundary condition corrected = ", true);
}
void
Problem::TBCCorrect()
{
  for (unsigned int j = 0; j <= T.My; ++j)
    T(T.Mx, j) = 2 * T(T.Mx - 1, j) - T(T.Mx - 2, j);
  for (unsigned int i = 0; i <= T.Mx; ++i)
  {
    T(i, 0) = T(i, 1) + q_top_bot * dy / (2 * k);
    T(i, T.My) = T(i, T.My - 1) + q_top_bot * dy / (2 * k);
  }
}

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::uBCCorrect()
{
  double m_out = 0;
  for (unsigned int j = 0; j <= u.My; ++j)
    m_out += rho * dy * u(u.Mx - 1, j);
  for (unsigned int j = 0; j <= u.My; ++j)
    u(u.Mx, j) = m_in * u(u.Mx - 1, j) / m_out;

  if (debug)
    u.print("u boundary condition 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);
}

} // namespace Flow2D
```

## Problem_residuals.cpp

```
#include "Problem.h"

namespace Flow2D
{

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

  if (Ru < tol && Rv < tol && Rp < tol)
  {
    if (debug)
      cout << "Main variables converged in " << main_iterations << " iterations" << endl;
    main_converged = true;
  }
}

void
Problem::computeAuxResiduals()
{
  double RT = TResidual();

  // Still not converged
  if (RT > tol)
    return;

  // Converged, finish up
  converged = true;

  // Print the result
  const double Rp = pResidual();
  const double Ru = velocityResidual(u);
  const double Rv = velocityResidual(v);
  cout << "Converged in " << noshowpos << fixed << setprecision(3)
       << 1.0 * (clock() - start) / CLOCKS_PER_SEC << " sec in " << main_iterations << " main and "
       << aux_iterations << " aux iterations: ";
  cout << noshowpos << setprecision(1) << scientific;
  cout << "p = " << Rp;
  cout << ", v = " << Rv;
  cout << ", p = " << Rp;
  cout << ", T = " << RT << 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::TResidual() const
{
  double numer, numer_temp, denom = 0;
  for (unsigned int i = 1; i < T.Mx; ++i)
    for (unsigned int j = 1; j < T.My; ++j)
    {
      const Coefficients & a = T.a(i, j);
      numer_temp = a.p * T(i, j);
      denom += abs(numer_temp);
      numer_temp -= a.n * T(i, j + 1) + a.e * T(i + 1, j);
      numer_temp -= a.s * T(i, j - 1) + a.w * T(i - 1, j) + a.b;
      numer += abs(numer_temp);
    }
  return numer / denom;
}

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) + a.e * var(i + 1, j);
      numer_temp -= a.s * var(i, j - 1) + a.w * var(i - 1, j) + a.b;
      numer += abs(numer_temp);
    }
  return numer / denom;
}

} // namespace Flow2D
```

# Problem_solvers.cpp

```cpp
#include "Problem.h"

namespace Flow2D
{

void
Problem::solveMain()
{
  ++main_iterations;
  if (debug)
    cout << endl << "Main iteration " << main_iterations << endl << endl;

    solve(u);
    solve(v);
    solve(pc);
}

void
Problem::solveAux()
{
  ++aux_iterations;
  if (debug)
    cout << endl << "Aux iteration " << aux_iterations << endl << endl;

  solve(T);
}

void
Problem::solve(Variable & var)
{
  if (debug)
    cout << "Solving variable " << var.string << endl << endl;

  // Fill the coefficients
  fillCoefficients(var);

  // Solve west to east
  sweepColumns(var);
  // Solve south to north
  sweepRows(var);
  // Solve east to west
  sweepColumns(var, false);

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

void
Problem::sweepRows(Variable & var, const bool south_north)
{
  if (debug)
    cout << "Sweeping " << var.string << (south_north ? " south to north" : " north to south")
         << endl;

  // 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)
{
  if (debug)
    cout << "Sweeping " << var.string << (west_east ? " east to west" : " west to east") << endl;

  // 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);
    if (var.w != 1)
      b[j - 1] += 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);
    if (var.w != 1)
      b[i - 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

```cpp
#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;

/**
 * Class that stores a 1D vector and enables printing and saving.
 */
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 */
```

## postprocess.py

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

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

Lx = 2
Ly = 0.02
cp = 4183
rho = 998.3
q = 500
k = 0.609

################################################################################
# Problem 3 prints

# Load coarse results
u = np.loadtxt('results/coarse_u.csv', delimiter=',').T
p = np.loadtxt('results/coarse_p.csv', delimiter=',').T
T = np.loadtxt('results/coarse_T.csv', delimiter=',').T
print(p.shape)

# Normalize P
p -= p[p.shape[0] - 1, :]

for var in [u, p, T]:
    for i in range(var.shape[0]):
        line = '{} & '.format(i + 1)
        for j in range(var.shape[1]):
            if var is T:
                line += '{:.5f}'.format(var[i,j])
            else:
                line += '{:.5e}'.format(var[i,j])
            if j == var.shape[1] - 1:
                line += " \\\\"
            else:
                line += " & "
        print(line)
    print()

################################################################################
# Load refined results

u = np.loadtxt('results/fine_u.csv', delimiter=',').T
v = np.loadtxt('results/fine_v.csv', delimiter=',').T
T = np.loadtxt('results/fine_T.csv', delimiter=',').T

dx = Lx / (T.shape[0] - 2)
dy = Ly / (T.shape[1] - 2)

################################################################################
# Plot problem 4a

x_center = np.linspace(0, Lx, num = u.shape[0])
u_center = np.copy(u)[:, int(u.shape[1] / 2)]
u_center /= u_center[0]
fig, ax = plt.subplots(1)
fig.set_figwidth(6)
fig.set_figheight(3)
ax.plot(x_center, u_center, 'k')
ax.set_xlabel(r'$x$ (m)')
ax.set_ylabel('Centerline $u / u_{{\\mathrm{{in}}}}$')
ax.grid()
fig.tight_layout()
fig.savefig('results/u_centerline.pdf', bbox_inches='tight')

################################################################################
# Plot problem 4b

for var, name in [(u, 'u'), (v, 'v'), (T, 'T')]:
    if name == 'T':
        y_8 = np.hstack((0, (np.linspace(dy / 2, Ly - dy / 2, num = var.shape[1] - 2)), Ly))
    else:
        y_8 = np.linspace(0, Ly, num = var.shape[1])
    var_8 = var[int(0.8 * var.shape[0] / Lx), :]

    fig, ax = plt.subplots(1)
```

```
    fig.set_figwidth(6)
    fig.set_figheight(3)
    if name == 'y':
        ax.semilogy(y_8, var_8, 'k')
    else:
        ax.plot(y_8, var_8, 'k')
    ax.set_xlabel(r'$y$ (m)')
    ax.set_ylabel('${}(0.8$ m$, y)$'.format(name))
    ax.grid()
    fig.tight_layout()
    fig.savefig('results/{}_0p8.pdf'.format(name), bbox_inches='tight')

################################################################################
# Plot problem 4b

# Sample u at temperature nodes
uT = np.zeros(T.shape)
for i in range(1, T.shape[0] - 1):
    for j in range(u.shape[1]):
        uT[0, j] = u[0, j]
        uT[u.shape[0], j] = u[u.shape[0] - 1, j]
        uT[i, j] = (u[i - 1, j] + u[i, j]) / 2

# Compute Nusselt numbers
Nu = np.zeros(T.shape[0])
u_left = u[0, 0]
for i in range(T.shape[0]):
    Tw = T[i, 0]
    Tb = 0
    for j in range(1, T.shape[1] - 1):
        Tb += uT[i, j] * T[i, j]
    Tb *= rho * cp * dy / (u_left * cp * Ly * rho)
    Nu[i] = 2 * Ly * q / (k * (Tw - Tb))

# Plot
x = np.hstack((0, (np.linspace(dx / 2, Lx - dx / 2, num = len(Nu) - 2)), Lx))
fig, ax = plt.subplots(1)
fig.set_figwidth(6)
fig.set_figheight(3)
ax.plot(x, Nu, 'k')
ax.set_xlabel(r'$x$ (m)')
ax.set_ylabel('Nu')
ax.grid()
fig.tight_layout()
fig.savefig('results/Nu.pdf', bbox_inches='tight')
```