2D Lid-Driven Cavity Flow: Navier-Stokes Equations

1 Overview

This document goes over the work done for the 2D Lid-Driven Cavity flow problem for VINCI. The requirements are as follows:

1. Python prototype
2. Production Implementation for parallel CPU and GPU architecture
3. Production readiness

Unfortunately, due to various time constraints, I was only able to complete the first requirement and port that to C++ in preparation for Open-MPI parallelization. Both the Python and C++ implementations have unit testing to ensure correctness, however the python implementaion is more comprehensive in its test coverage.

2 Governing Equations

The incompressible Navier-Stokes equations in 2D are given by:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \frac{1}{Re}\nabla^2 \mathbf{u} \tag{1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2}$$

where $\mathbf{u} = (u, v)$ is the velocity field, p is the pressure, and Re is the Reynolds number. In component form:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{\partial p}{\partial x} + \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{3}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{\partial p}{\partial y} + \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \tag{4}$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{5}$$

3 Boundary Conditions

The domain is $\Omega = [0,1] \times [0,1]$ with boundaries:

- Top boundary ($y = 1$): $\mathbf{u} = (1, 0)$ (lid moving with unit velocity)
- Bottom boundary ($y = 0$): $\mathbf{u} = (0, 0)$ (no-slip)
- Left boundary ($x = 0$): $\mathbf{u} = (0, 0)$ (no-slip)
- Right boundary ($x = 1$): $\mathbf{u} = (0, 0)$ (no-slip)

Mathematically:

$$\mathbf{u}(x, 1, t) = (1, 0) \quad \forall x \in [0, 1], \quad t > 0 \tag{6}$$

$$\mathbf{u}(x, 0, t) = \mathbf{u}(0, y, t) = \mathbf{u}(1, y, t) = (0, 0) \quad \forall x, y \in [0, 1], \quad t > 0 \tag{7}$$

Initial condition:

$$\mathbf{u}(x, y, 0) = (0, 0) \quad \forall (x, y) \in \Omega \tag{8}$$

4 Numerical methods

The finite volume method was chosen to solve this problem. In particular, I implemented a PISO algorithm for the pressure-velocity coupling. The spatial discretization is second-order central difference for the diffusion terms and a first-order upwind scheme for the convection terms. The time integration is done using a first order backward Euler method. Had I had more time, explicit forward Euler time integration and second-order upwinding with blending of central differencing schemes and upwinding would have been implemented as well.

The computational domain was chosen to be a uniform cartesian grid. The grid formation and numbering scheme is shown in Figure 1, along with a sample control volume of where computation takes place.

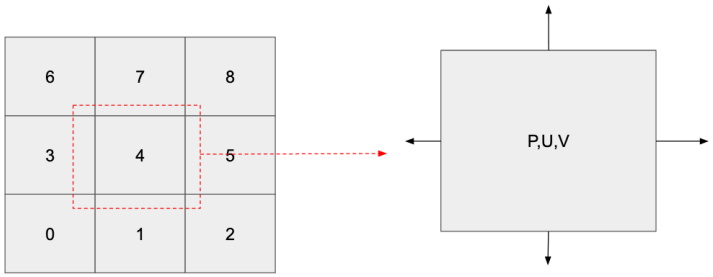


Figure 1: Finite volume cell schematic showing the control volume and face locations.

For this project, a cell-centered approach was taken, where the velocity (u, v) and pressure (p) values are stored at the center of each control volume. Other approaches, such as

staggered grids, could have worked here as well, however for unstructured grids these tend to be unwieldy in terms of implementation.

Face areas naturally fall out of the uniform grid structure, and the cell volumes are simply $\Delta x \Delta y$.

The face fluxes at each cell face are computed through linear interpolation, and a Rhie-Chow correction is used to correct the mass (or more correctly, volume) fluxes and saved off at each face entity. The volume fluxes at each face are computed as:

$$\dot{m}_f = \left(\mathbf{u_f} + \Delta t(\overline{\nabla P} - \nabla P_f)\right) \cdot \mathbf{A_f} \tag{9}$$

where $\mathbf{u_f}$ is the interpolated velocity at the face, $\overline{\nabla P}$ is the interpolated cell-centered pressure gradient, ∇P_f is the face-centered pressure gradient, and $\mathbf{A_f}$ is the face area vector. Eq. (9) also forms the pressure Poisson equation in order enforce the incompressibility constraint:

$$\Delta t \nabla \delta P_f \cdot \mathbf{A_f} = \left(\mathbf{u}_f + \Delta t(\overline{\nabla P} - \nabla P_f)\right) \cdot \mathbf{A_f} \tag{10}$$

The cell pressure gradients (and any cell gradient quantities) are computed through the Green-Gauss Theorem:

$$\nabla P = \frac{1}{V_c} \sum_f P_f \mathbf{A_f} \tag{11}$$

where V_c is the cell volume and P_f is the face-centered pressure, which is computed through a linear interpolation from the cells. Once the pressure correction is solved for, the velocity field is updated through a pressure correction step:

$$\mathbf{u}^{n+1} = \mathbf{u}^* - \Delta t \nabla \delta P \tag{12}$$

The rough algorithmic steps for each time step are as follows:

1. Solve for the intermediate velocity field $\mathbf{u}^*$ through Eq. (3) with a guess of the pressure P^*.
2. Compute the face flux via Eq. (9) with intermediate velocity and guessed pressure fields.
3. Form and solve the pressure Poisson equation in Eq. (10) to get the pressure correction δP.
4. Update the velocity field using the pressure correction from Eq. (12).
5. Update the pressure field: $P^{n+1} = P^n + \delta P$.
6. Compute the face flux with updated quantities for pressure and velocity via Eq. (9).
7. Repeat until convergence.

5 Usage

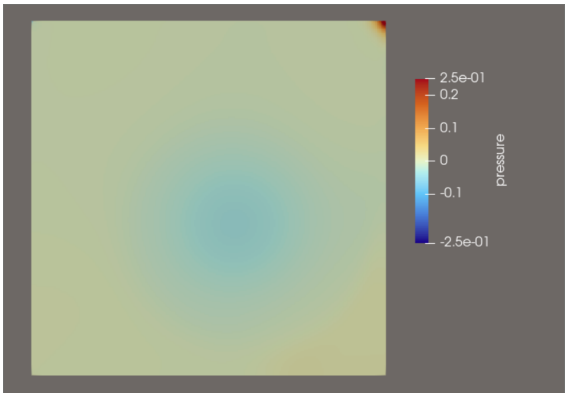
Again, since time was a constraint, all of the asked for implementation details and demonstrations were not completed. The python and C++ implementation both were run in a unit testing framework, but has support for reading in a YAML input file to set up a simulation. The python implementation is run with:

```
python -m unittest -k test_nxn
```

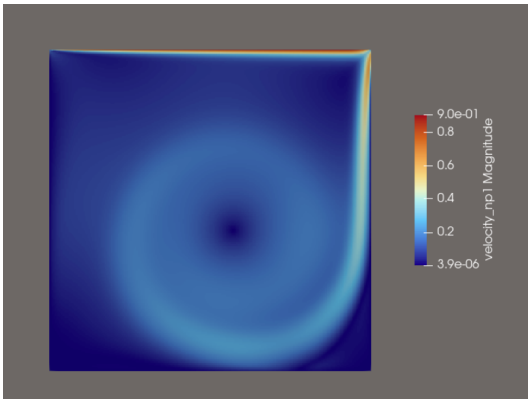
Visualization was saved off to a png file just to visualize velocity contours. The C++ implementation is run with:

```
./test_pseudo_sim_setup
```

which reads in a YAML file from a separate directory. Results were output to a vtk file format for visualization in Paraview. Figure 2 presents sample outputs from the solver. The velocity magnitude and pressure field distributions are shown in Figure 2a and 2b, respectively, for a 250-point grid.



(a) Pressure field on 250-point grid



(b) Velocity field on 250-point grid

Figure 2: Sample computational results showing (A) pressure field distribution and (B) velocity field distribution on a 250-point grid.

6 Considerations

At my current work, we are limited in our usage of AI tools and had only used them in my spare time. This project was a nice opportunity to explore the use of AI in a targeted code development setting outside of just python.

Things that went surprisingly well include a clean conversion from my python implementaiton to C++, although I had found that the testing was not ported over properly and required extra time to accurately reproduce those results.

Things that did not go well were trying to port the C++ implementation all at once to a parallel implementation with Open-MPI. Had I had another chance (and will probably

pursue this further on my own time), I would have taken a more targeted approach to parallelization, starting with verification of mesh decomposition first, and then working my way through the assembly algorithms, linear solve and then the actual PISO updates.