Project 3 Solution: Driven-Cavity Flow

AERO 523 Fall 2021
Author: Erik Rutyna

December 12, 2021

Introduction

The goal of this project was to solve incompressible Navier-Stokes equations until steady state for a wall-driven cavity using the using the Finite Volume Method (FVM). However, this wasn't the standard FVM. Instead it was one that involved updating and correction the state known as the Projection-Correction Method. A mock representation of what the solution looks like for the wall-driven cavity can be seen in Figure 1 [2].

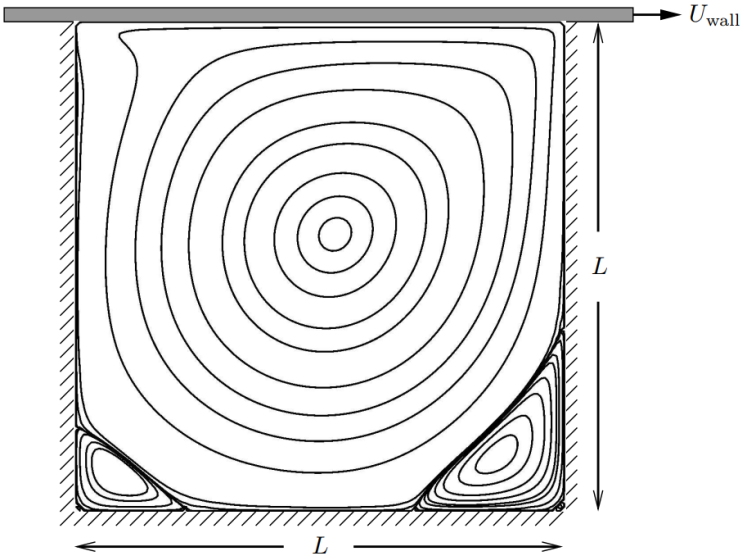


Figure 1: An example solution of the flow in the wall-driven cavity.

With this solution method a different-ish geometry is used. While fluxes are still computed between cells for a given mesh, instead of it being unstructured, it is instead standardized domain of squares. However, instead of storing everything as an average value in the center of a cell, like with first order FVM, some values are stored in-between cells in a method known as staggered storage. An example of what this looks like can be seen in Figure 2 [1].

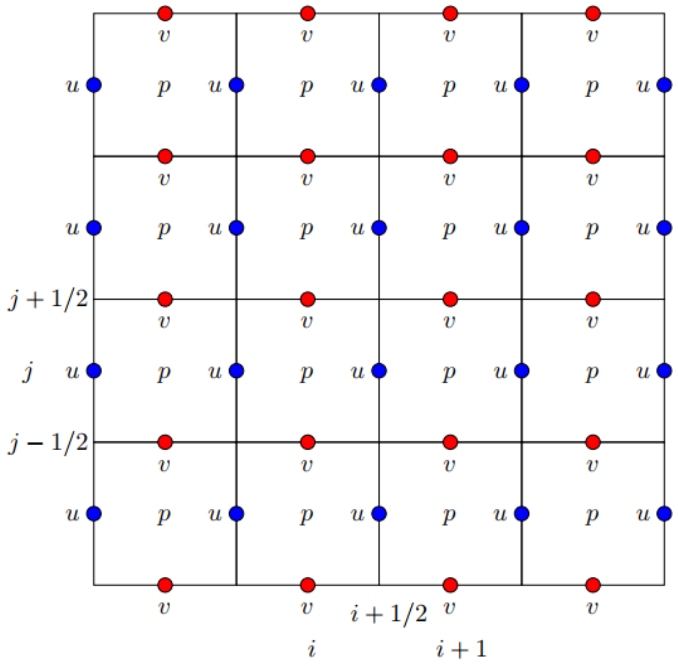


Figure 2: An example picture of how one might store values using staggered storage.

Using a mesh like the one shown, various fluxes for each cell can be calculated by iterating over the cells and using the state values stored on the boundaries and interiors of cells. Instead of doing a full update at each timestep, this method allowed me to march my velocity a "half-step" (projection) in time. Then, I was able to calculate a pressure field that made this velocity field divergence free. Using that new pressure field, I could go back and correct my velocity field from the "half-step" to the "full-step" (correction).

This projection-correction process is one iteration, and at the end of this iteration the residuals were calculated in order to see how close I was to the steady state solution. The smaller the residuals, the closer I was to the steady state solution. If my residual was too large - meaning I was not close close to the steady state solution - another iteration was performed. This continued until I had reached steady state defined as residuals being less than $1 \cdot 10^{-5}$.

Setup

Initial Conditions

The initial condition is relatively simple as the fluid starts at rest. Since the fluid starts at rest, and cannot flow through any boundaries are they are inviscid walls, the flow field was initialized to be zero everywhere.

The pressure field is a bit different. Since the pressure field was constrained only with Neumann boundary conditions, the initial pressure at cell in the domain didn't matter as long as was uniform across the domain. This effectively means that the pressure is initialized to a constant across the entire field. For simplicity, I chose an initial constant of zero.

Boundary Conditions

The boundary conditions for the fluid are the same on all four walls - the walls are invsicid and have no-slip on them. That means that flow cannot pass through any walls so the normal

component to the wall at the wall is zero, $(\vec{v} \cdot \vec{n})|_{wall} = 0$, and that the tangential component must match the speed of the wall, $(\vec{v} \cdot \vec{s})|_{wall} = \vec{v}|_{wall}$.

These conditions make it simple to evaluate the boundary conditions as the vertical velocity is zero everywhere once the boundary conditions are applied. However the top wall has a moving velocity in the positive x-direction and so this needs to be properly applied. But with staggered storage, the nearest x-velocities are not the top wall, so they those points don't have any value. This meant that the entire flow field is initialized to zero velocity.

Ghost Cells

In order to evaluate the velocities to calculate fluxes, an additional layer of non-real, "ghost" cells are used. These ghost cells have a special boundary conditions. For their internal pressures, the ghost cells mirror their adjacent, in-domain neighboring cell. The x and y velocities on the vertical and horizontal edges of the ghost cells depend on what wall the ghost cell is bordering.

The x-velocities on vertical walls copy the value of the neighboring x-velocity two cells into the domain. For the horizontal walls, they follow the formula of $U_{ghost} = 2 \cdot U_{wall} - U_{interior}$.

For the vertical velocities, this phenomena is flipped. As this concept might be hard to grasp without a picture, one is provided in Figure 3 that illustrates the values of ghost cells by using a 4×4 sized domain.

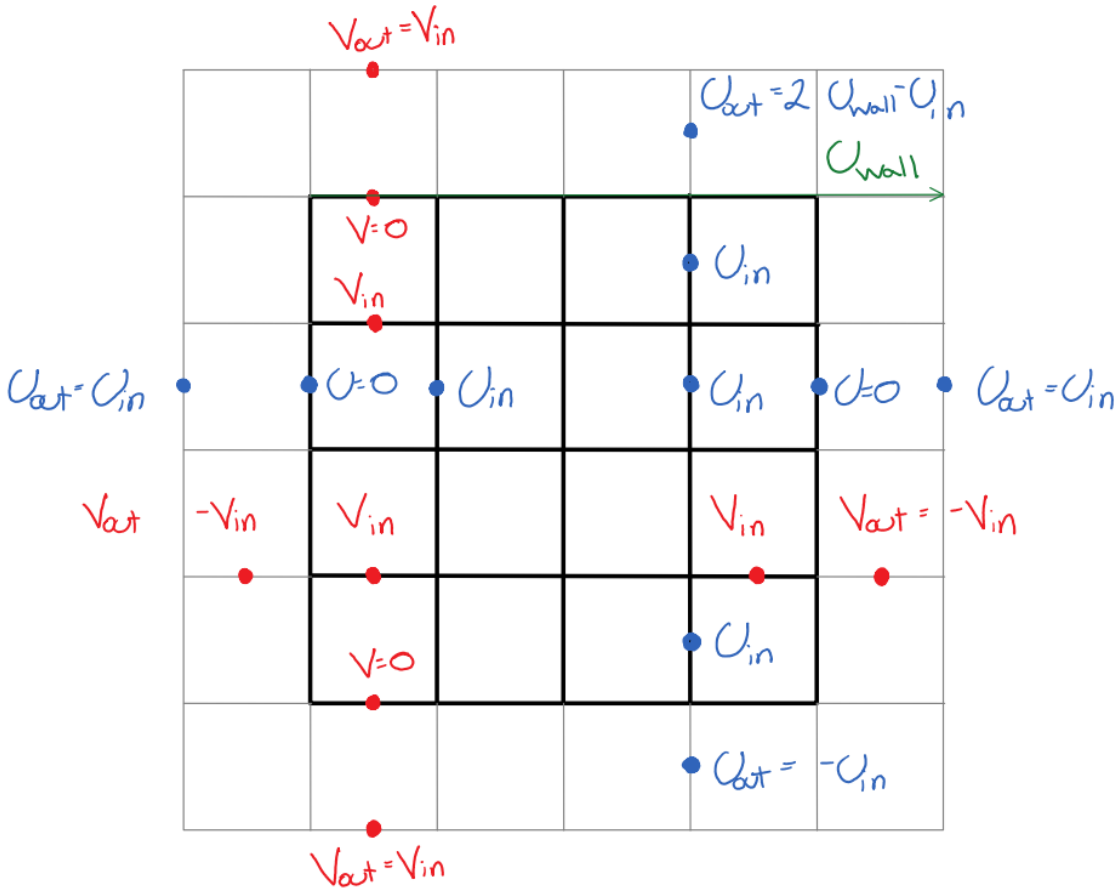


Figure 3: A grid with 4 cells in each direction with labels to show how the ghost cells are updated.

Indexing

I did not like the indexing provided in the project description. Personally, I found it difficult to index effectively for calculating various values. Instead of using something I found difficult, I created my own indexing method by including a full layer of ghost cells and indexing everything from the bottom left corner. That corner was defined to be point $(0, 0)$ for all indexing purposes. A picture is provided in Figure 4 that demonstrates this type of indexing with all state and flux quantities labelled alongside their respective indices. Note the difference in the two border colors, the thicker, black bordered cells are the actual domain, while the thinner, gray cells are the ghost cells.

This may be inefficient from a memory standpoint, but I was not concerned about efficiency as my grid sizes were small. Instead I wanted to have a system that worked for me. By having this standardized grid, I could easily reference which cell I was at, and index the needed state quantities for calculating a value or updating the state itself relatively quickly.

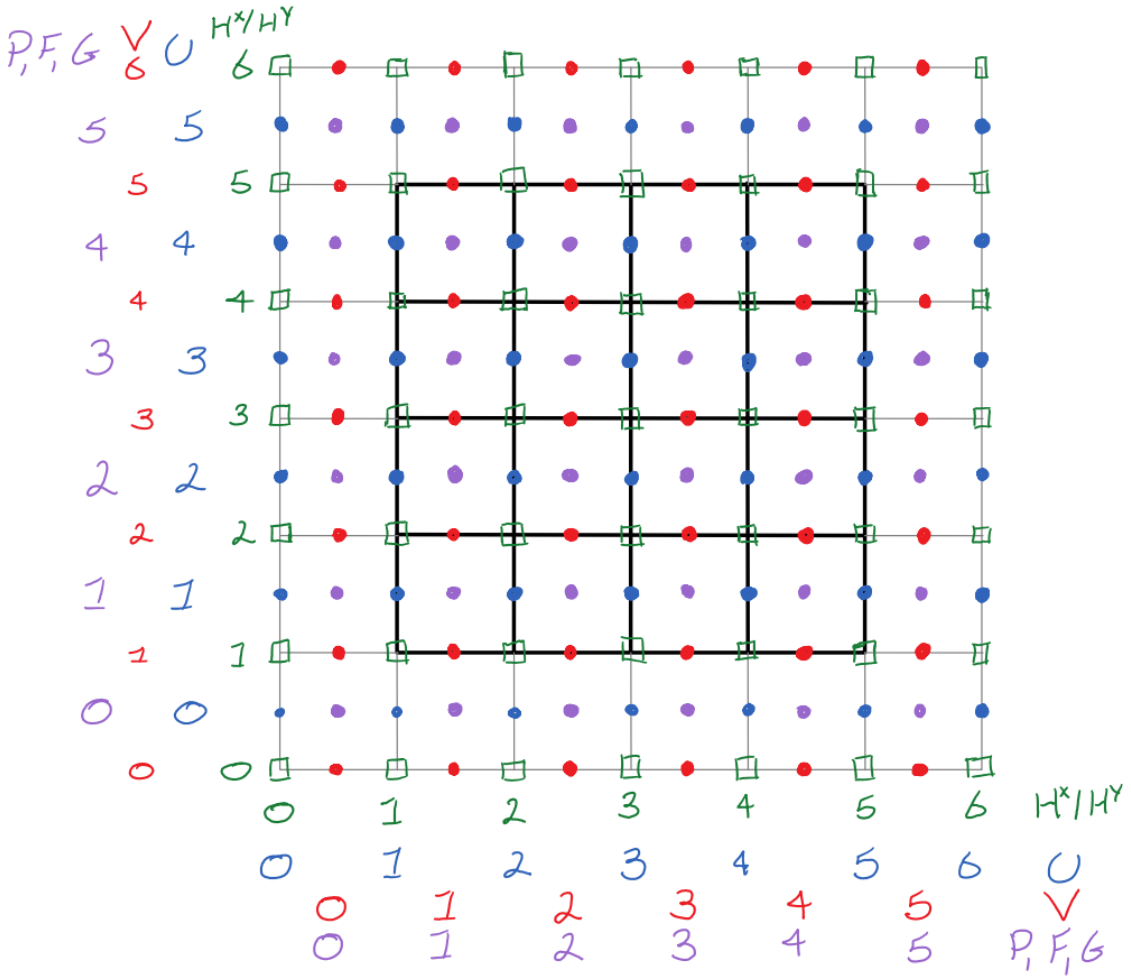


Figure 4: A 4×4 sized mesh with all of the quantities labelled with how they are indexed according to my scheme.

Algorithm Walkthrough

The algorithm for this project was relatively straightforward. It primarily consisted of looping over each individual cell in the mesh and calculating a certain quantity, then updating another,

and repeating this until convergence. A more high-level approach can be seen in the algorithm outlined in Algorithm 1. Additionally, once the simulation finished there was an additional post-processing file that was used to generate the required figures for the tasks.

Algorithm 1 Projection-Correction Algorithm

1: **while** not converged **do**
2: Update ghost cells based upon ghost cell boundary conditions
3: Calculate fluxes F, G, H^x, H^y for all cells
4: Update velocity to $n + \frac{1}{2}$ via momentum equation
5: Solve the Pressure-Poisson Equation (PPE)
6: Update the velocity field based on solved PPE
7: Calculate residuals and check for convergence
8: **end while**
9: Save converged velocity fields

Going more in depth for each step, step 2 is the simplest. In this step a quick pass was done over the velocity and pressure fields and it updated the ghost cells using the ghost cell conditions previously mentioned in the section on ghost cells.

Step 3 is where it gets a bit more involved. Here I ran a loop over all the cells and calculated the 4 fluxes based on the equation for them,

$$F = q\phi - \nu u_x$$

$$G = q\phi - nuv_y$$

$$H^x = q\phi - nuu_y$$

$$H^y = q\phi - nuv_x,$$

where q and ϕ are the transported quantity and transported quantity velocity for the specific flux. The breakdown for these can be seen in Figure 5. The reason that they are done in this format is that q can be done using the local information across the cell where the flux is being evaluated, but ϕ requires a method known as the Simple Monotone Advection for Realistic Transport (SMART) limiter. This limiter ensures that the proper value of ϕ is calculated using only the right upwind data.

$$\begin{array}{lcll} F & : & uu & = \text{ transport of } u \text{ in the } x\text{-direction} \\ G & : & vv & = \text{ transport of } v \text{ in the } y\text{-direction} \\ H^x & : & vu & = \text{ transport of } u \text{ in the } y\text{-direction} \\ H^y & : & uv & = \text{ transport of } v \text{ in the } x\text{-direction} \end{array}$$

Figure 5: The fluxes and their respective $q \cdot \phi$ quantities [1].

Once all these fluxes are computed, we can move on to step 4 where the projected velocity field can be computed using

$$u_{i,j}^{n+1/2} = u_{i,j} - \Delta t(F_x + H_y^x)$$

$$v_{i,j}^{n+1/2} = v_{i,j} - \Delta t(G_y + H_x^y),$$

where the derivatives of the fluxes are local to the values across that velocity in the mesh.

At step 5 the pressure field in the Pressure-Poisson Equation (PPE) is solved for. The PPE looks like

$$\nabla^2 p^{n+1} = \frac{\nabla \cdot \vec{v}^{n+1/2}}{\Delta t}$$

and this is the solved for pressure field, p^{n+1}. This updated pressure field makes our updated velocity field at $n + 1/2$ divergence free. This allows us to finish the velocity update and go from $\vec{v}^{n+1/2}$ to $\vec{v}^{n+1}$. Instead of setting up a linear system of equations and solving $\boldsymbol{A}\bar{x} = \bar{b}$, I chose to use an iterative smoother instead. The smoother I chose was just a standard Gauss-Siedel smoother. I ran this smoother across my entire pressure field and called this a single Gauss-Siedel iteration. I then ran 100 of these iterations before considering the pressure field to be updated and moving on to the next step.

After solving for this updated pressure field, we can move on to step 6 where I corrected the velocities to $n + 1$ using the divergence-free pressure field. This pressure field ensures continuity in our equations and so it is a vital step to take.

$$u_{i,j}^{n+1} = u_{i,j}^{n+1/2} - \Delta t(p_x^{n+1})$$

$$v_{i,j}^{n+1} = v_{i,j}^{n+1/2} - \Delta t(p_y^{n+1}),$$

Again this is in the format of the the derivatives are taken using the values across the velocity node at (i, j) in the domain.

At this point the residuals are calculated using the formula provided by Professor Fidkowski. Assuming that the simulation has not meat the residual tolerance for convergence, process loops back to step 2.

Helper Functions

Due to the complexity of the code, I wrote a large series of smaller helper functions that focused on doing a single calculation when given information about the current state and something like an indexing location. The following list contains the more important functions and a brief description of them.

1. *UIndicesGen*: Returns a list of all x-velocity indices that are inside the domain, not on a boundary condition, and that need to be marched forward in time.
2. *VIndicesGen*: Returns a list of all y-velocity indices that are inside the domain, not on a boundary condition, and that need to be marched forward in time.
3. *FluxIndicesGen*: Returns a list of all cell-based flux indices that are inside the domain that need to be calculated.
4. *HFluxIndicesGen*: Returns a list of all node-based flux indices that are inside the domain that need to be calculated.
5. *SMART*: Completes the SMART process as outlined in the notes in section 7.3.10 [1].
6. *FCalc*: Calculates the momentum flux F.
7. *GCalc*: Calculates the momentum flux G.

8. *HXCalc*: Calculates the momentum flux H^x.
9. *HYCalc*: Calculates the momentum flux H^y.
10. *UHalf*: Steps the local x-velocity at index (i, j) from n to $n + 1/2$.
11. *VHalf*: Steps the local y-velocity at index (i, j) from n to $n + 1/2$.
12. *PUpdate*: Updates the local pressure at cell (i, j) via the discretized PPE for the Gauss-Siedel smoother.
13. *LocalUUpdate*: Steps the local x-velocity at index (i, j) from $n + 1/2$ to $n + 1$.
14. *LocalVUpdate*: Steps the local y-velocity at index (i, j) from $n + 1/2$ to $n + 1$.
15. *ErrorNorm*: Compute the residual at the current time using the formula in section 7.3.6 [1].

Tasks

Convergence Plots

I was successful in completing the base cases for the project. Due to difficulties with implementing the algorithm and various sub-routines, I was unable to complete any additional cases in time for the extra credit. Figures 6 and 7 show the convergence of my residuals for the $N = 32$ and $Re = 100,\ 400$ test cases. The $Re = 100$ case didn't take long, so I let it converge to an order of magnitude smaller residuals, but the $Re = 400$ case took almost an hour to converge so I only went to the required value of $R = 10^{-5}$.

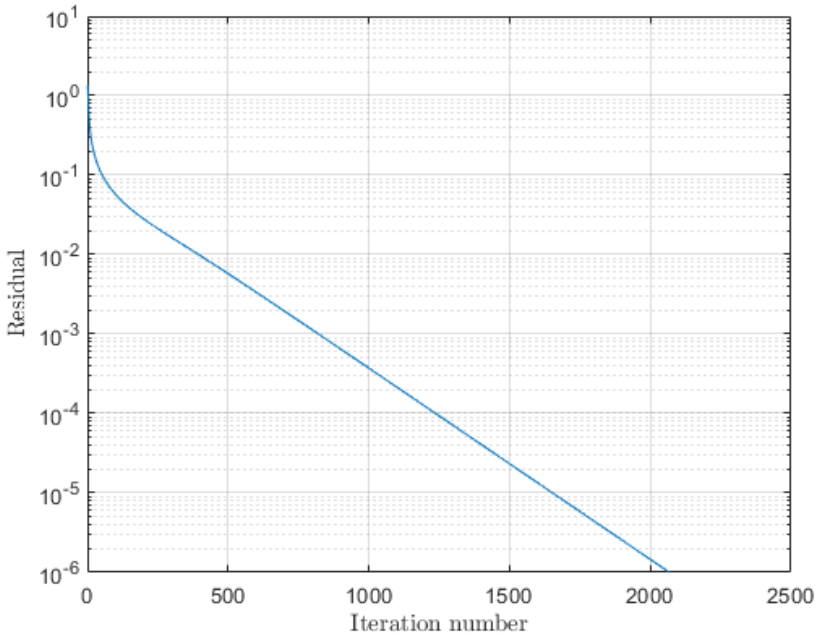


Figure 6: The residuals for my $Re = 100$ case converged to 10^{-6}, an order of magnitude higher than required.

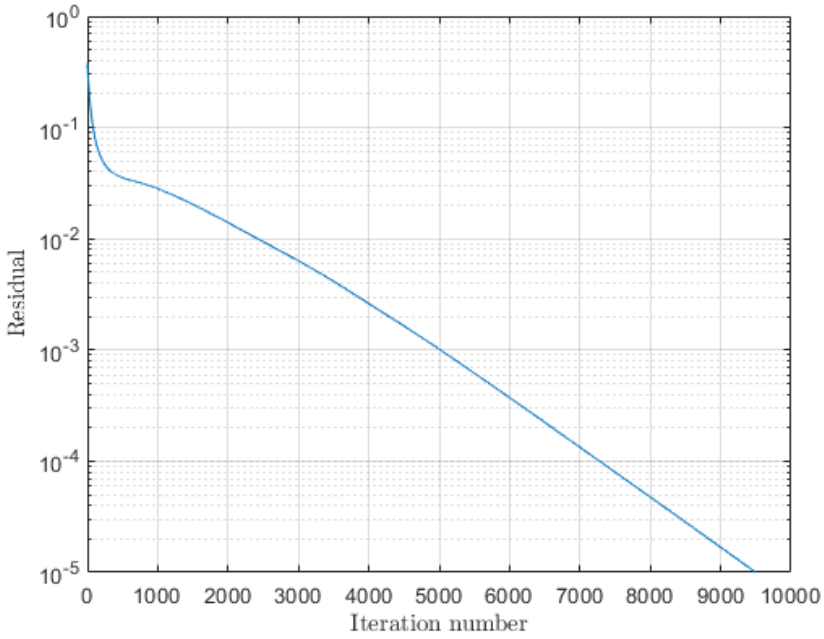
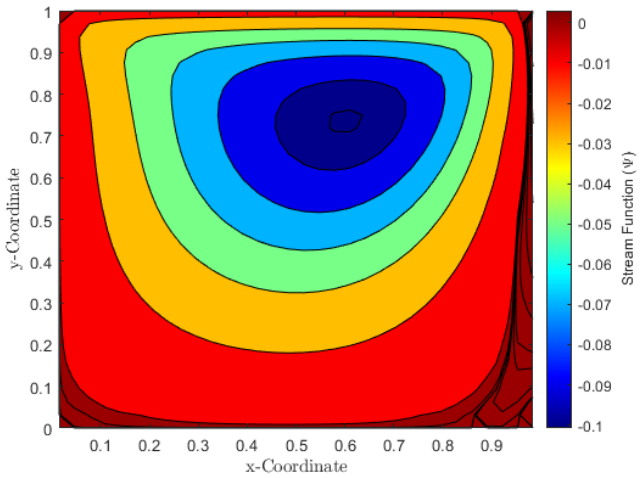


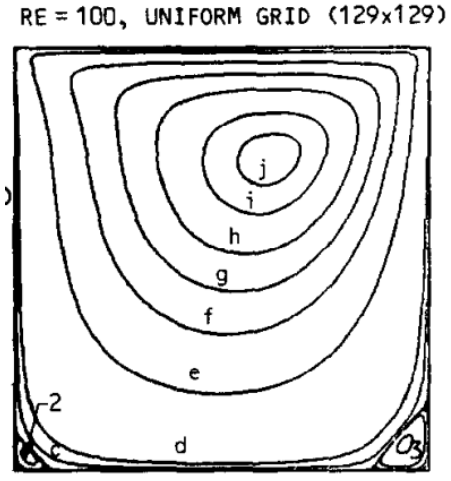
Figure 7: The residuals for my $Re = 400$ case converged to 10^{-5}.

Stream Function Contours

Since I was able to complete both base cases, I was able to plot the stream function contours. Both of my contour plots match those from the Ghia, Ghia, and Shin paper, and you can see a direct side-by-side comparison in Figures 8 and 9. The only notable difference is my $Re = 400$ contour map has one less contour on the outer-most level (where Ghai-Shin has contours "e" and "d"), and I think this has to due with the fact that my mesh was much more coarse than theirs. I had roughly 1/16th the amount of total cells as Ghia, Ghia, and Shin. Outside of this, the two sets of contour maps are almost indistinguishable.

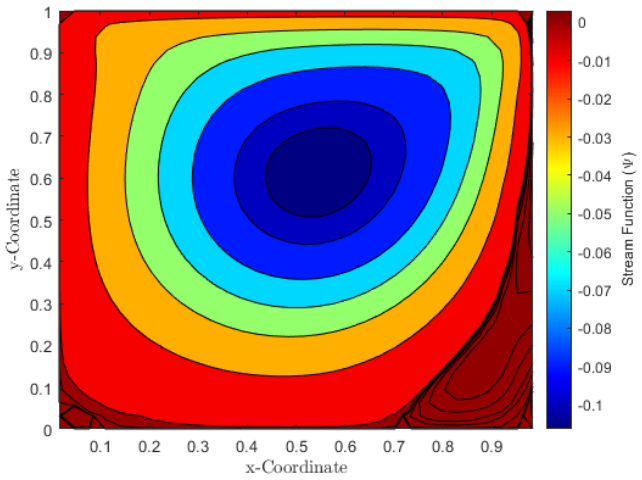


(a) My contour plot of the stream function, Ψ, for $Re = 100$.

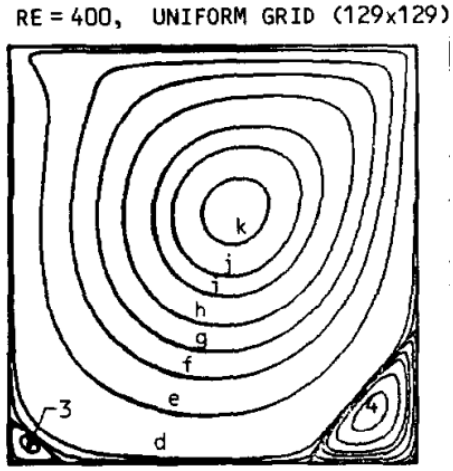


(b) The Ghia-Shin contour plot of the stream function, Ψ, for $Re = 100$ [3].

Figure 8: The $Re = 100$ side-by-side comparison of contour plots.



(a) My contour plot of the stream function, Ψ, for $Re = 400$.



(b) The Ghia-Shin contour plot of the stream function, Ψ, for $Re = 400$ [3].

Figure 9: The $Re = 400$ side-by-side comparison of contour plots.

Velocity Plots

Since I was able to complete both base cases, I was able to plot the center-line velocities as a function of the other direction. This means that I was able to plot the x-velocity in the middle of the domain as a function of height, and the y-velocity in the center of the domain as a function of distance along the length of the domain. Then I overlayed the data from the Tables 1 and 2 from pages 398 and 399 from the Ghia-Shin paper to verify the validity and accuracy of my numerical results. The results of this process can be seen in Figures 10 and 13 [3].

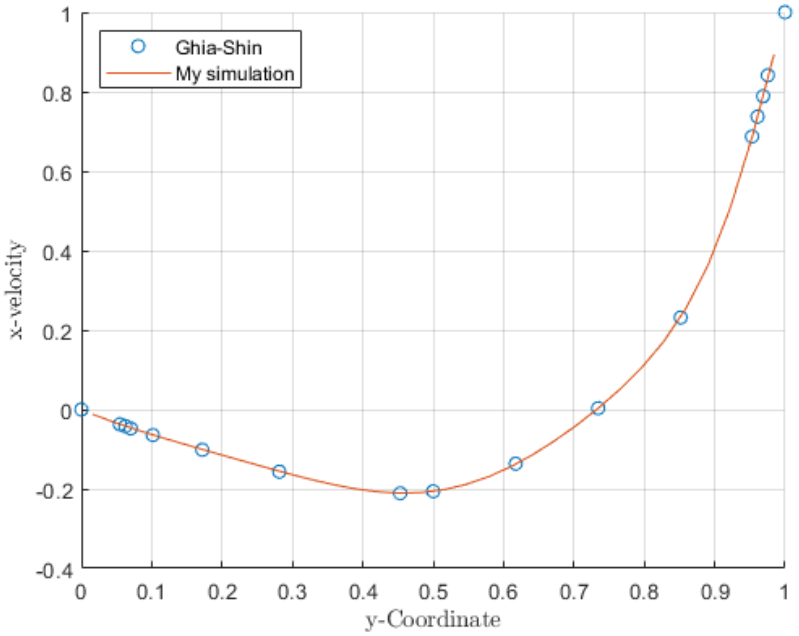


Figure 10: The x-velocity lines up nicely with the Ghia-Shin data at $Re = 100$.

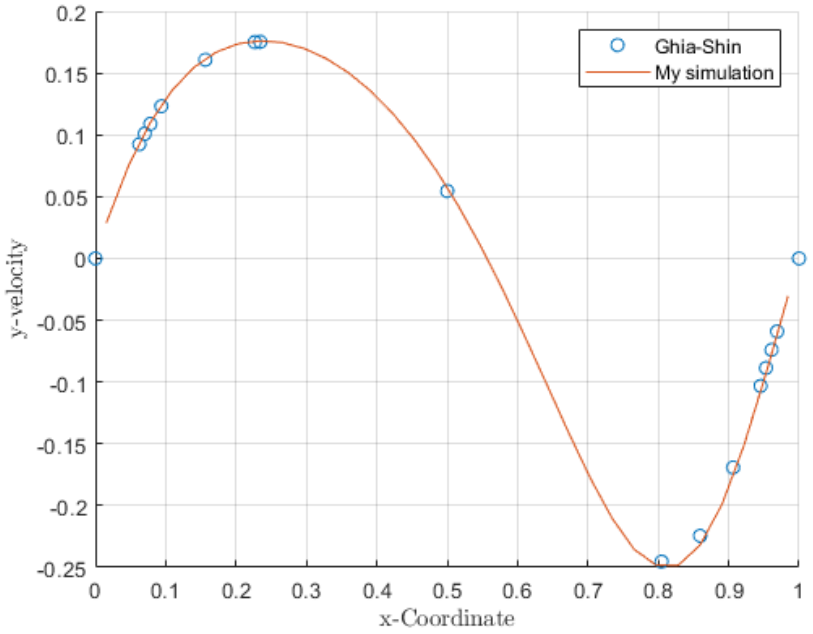


Figure 11: The y-velocity lines up nicely with the Ghia-Shin data at $Re = 100$.

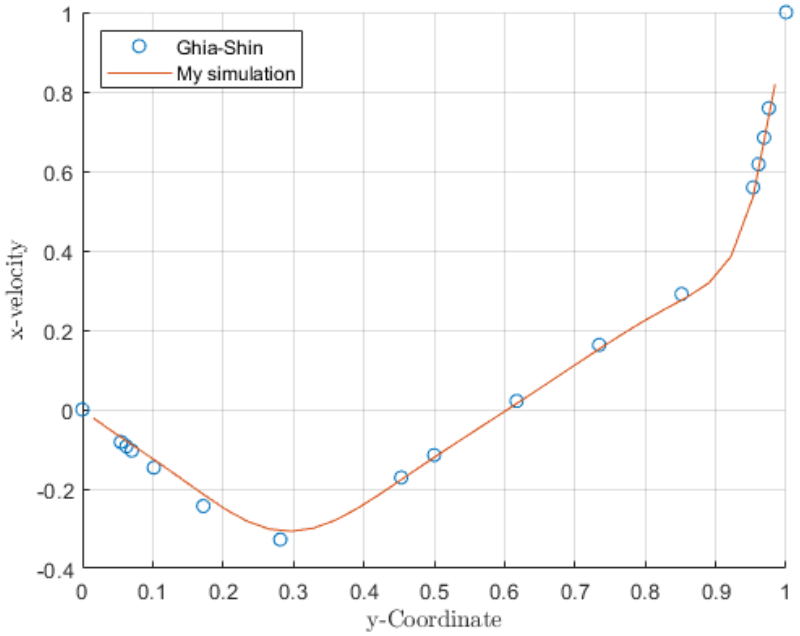


Figure 12: The x-velocity still lines up nicely with the Ghia-Shin data at $Re = 400$.

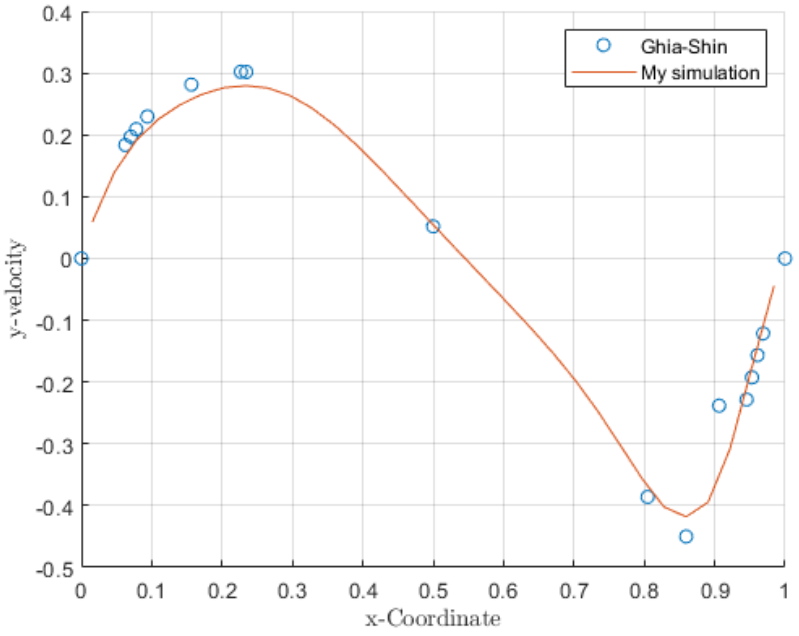


Figure 13: The y-velocity still lines up nicely with the Ghia-Shin data at $Re = 400$.

The data values from Ghia-Shin overlap quite well with my numerical results, which helps to verify the accuracy of my results. The data from Ghia-Shin begins to shift away from my numerical results, but again I believe that has to do with the fact that their mesh for $Re = 400$ is 129×129 and mine is only 32×32, which about 1/16th the size. I believe if I had ran the same Reynolds number at a more refined mesh like 128×128 I would have gotten better matching results. However such a task is beyond the scope of this project and would have taken too long on my code.

I would argue that with these matching velocity results, along with my matching contour plots, that I was still successful in my original goal of simulating the cavity-driven flow, and proven that my code works.

Conclusion

The goal of this project was to simulate a wall-driven cavity using a projection-correction based FVM solver. I was able to implement this projection-correction based solver using the algorithm outlined above in 1. From there I was able to produce results for the two mandatory cases of $Re = 100$ and $Re = 400$. I then compared my results to Ghia-Shin and was able to verify the accuracy of my results, proving that I was successful in simulating the wall-driven cavity.

REFERENCES

[1] Krzysztof Fidkowski. *Computational Fluid Dynamics*. University of Michigan, Aug. 2021, p. 196.

[2] Krzysztof Fidkowski. *Project 3: Driven-Cavity Flow*. Dec. 2021.

[3] UKNG Ghia, Kirti N Ghia, and CT Shin. "High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method". In: *Journal of computational physics* 48.3 (1982), pp. 387–411.