APC 523 Final Project

Developing an Efficient Navier-Stokes Solver Using Parallelized Iterative Schemes

Kevin Andrade and Nathan Spilker

3 May 2022

Introduction

This project is centered around the development of a Navier Stokes solver for modeling the fluid mechanics around various geometries in two-dimensional flows. Perhaps one of the greatest challenges in the field of fluid mechanics, is the high computational cost and time needed to run simulations in CFD (Computationl Fluid Dynamics) software. This is the main motivation behind this study as we aim to leverage the paralellization methods developed in class to improve the efficiency of iterative methods for solving the two-dimensional pressure Poisson equation within the Navier Stokes solver.

Theory

In order to analyze the fluid mechanic properties, we can refer to the governing equations for this problem which includes Navier Stokes (1) as well as the continuity equation (2) which is supported by the incompressible flow assumption:

$$\frac{\partial U}{\partial t} + U \cdot \nabla U = -\frac{1}{\rho}\nabla p + \nu \nabla^2 U \tag{1}$$

$$\nabla \cdot U = 0 \tag{2}$$

Note that in this equation, ρ is density, U is the velocity vector, p is the pressure, and ν is the kinematic viscosity. In order to further analyze this problem numerically, it is crucial to discretize the Navier Stokes equation which can be done by rearranging the equation and replacing the time dependent term of the material derivative with an alternative definition of the derivative:

$$\frac{U^{n+1} - U^n}{\delta t} = -U^n \cdot \nabla U^n - \frac{1}{\rho}\nabla p^{n+1} + \nu \nabla^2 U^n \tag{3}$$

As mentioned previously, the problem is then divided into the intermediate and corrective velocity steps. This is largely due to the fact that we are attempting to solve for both the pressure and velocity simultaneously. This process is simplified by first considering the intermediate velocity where we do not account for the pressure term of the discretized Navier Stokes and replace the U^{n+1} term with a U_i term representing the intermediate velocity:

$$\frac{U_i - U^n}{\delta t} = -U^n \cdot \nabla U^n + \nu \nabla^2 U^n \tag{4}$$

This is followed by the corrective step of the velocity where we can then consider the intermediate velocity as the U^n and also account for the pressure term:

$$\frac{U^{n+1} - U^*}{\delta t} = -\frac{1}{\rho}\nabla p^{n+1} \tag{5}$$

With both of these equations in place, we now only need an equation for pressure in order to accurately calculate the pressures and velocities acting on each element. This can be extracted from the continuity equation and the corrective velocity step. By taking the gradient on both sides of Equation 5, we get:

$$\nabla\left[\frac{U^{n+1} - U^*}{\delta t}\right] = \nabla\left[-\frac{1}{\rho}\nabla p^{n+1}\right] \tag{6}$$

$$\frac{1}{\delta t}(\nabla \cdot U^{n+1} - \nabla \cdot U^*) = -\frac{1}{\rho}\nabla^2 p^{n+1} \tag{7}$$

Using the continuity relationship ($\nabla \cdot U = 0$) we get the following:

$$\frac{1}{\delta t}\nabla \cdot U^* = \frac{1}{\rho}\nabla^2 p^{n+1} \tag{8}$$

Rearranging the terms in this equation is the final step needed to derive the Pressure Poisson Equation:

$$\nabla^2 p^{n+1} = \frac{\rho}{\delta t}\nabla \cdot U^* \tag{9}$$

Solver Workflow

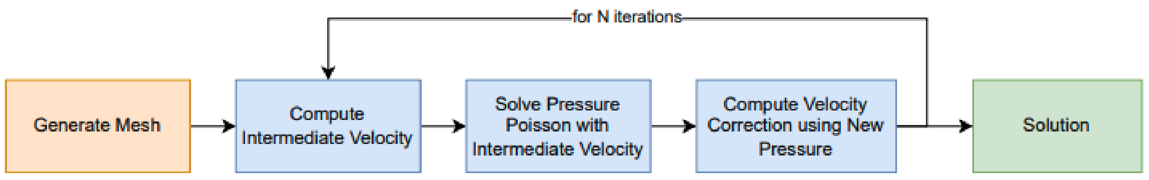
for N iterations
Generate Mesh
Compute Intermediate Velocity
Solve Pressure Poisson with Intermediate Velocity
Compute Velocity Correction using New Pressure
Solution

Figure 1: Navier Stokes Solver Algorithm

Mesh Generation

The first step in the process of solving for the pressures acting on the geometry via NS is to develop a mesh or grid for the geometry. This is done using an input file which specifies an X length (L_x), a Y length L_y, a resolution in each direction (n_x, n_y), and one of three cases: "none", "cylinder", or "airfoil. In all cases, the mesh itself is represented as an [n_x x n_y] array. For the "none" case, we simply return a blank grid of zeros. However in the rest of the cases, we return a representation of the geometry across the mesh. For these cases we a mesh generation algorithm comprised of an outline step followed by a fill step.
In the first part, we generate the X and Y coordinates that make up the outside outline of each geometry. This is done by calculating the dx and dy using the length in either direction:

$$\text{dx} = \frac{L_x}{n_x}$$
$$\text{dy} = \frac{L_y}{n_y}$$

We then approximate each of the array indices as having length dx or dy depending on the row or column. Given this representation, we then mapped each of the coordinates to one of the array indices based on proximity to the length. This is depicted in Figure 2 where there are examples of points located within the bounds of the grid locations.

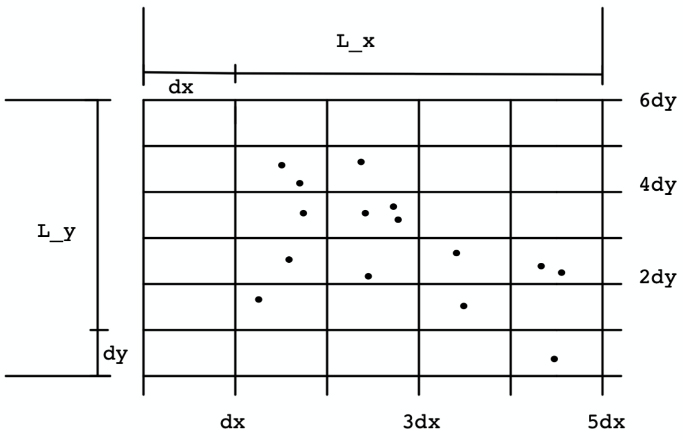

Figure 2: Mesh Generation Grid

For the cylinder, we generate the points using the equation for a circle with a number of points that scales as a function of the grid sizes n_x and n_y. We do this differently for the airfoil, where we extract the points from a data file. Additionally for the airfoil, we interpolated points between each of the provided airfoil points (from a NACA database) in order to increase the density of the points. For both cases, we scale the default coordinates which for the cylinder is a radius of 1 and a chord of 2 for the airfoil. The optional center argument also allows the user to shift the location of the cylinder from the default location which is the center of the grid. For the airfoil case the center is the location of the leading edge. As a result, the default center case creates an airfoil off-center to the left. The two mesh types are shown in Figures 3 and 4.

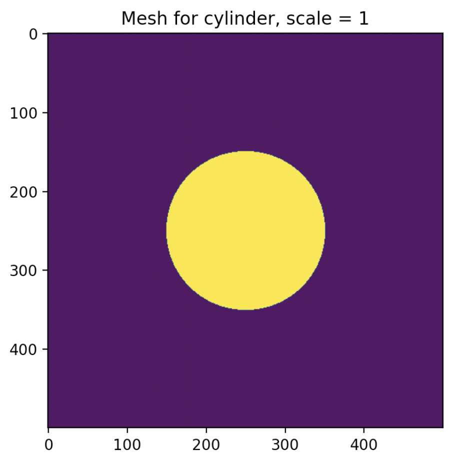

Figure 3: Cylinder Mesh for $N_x = 500$, $N_y = 500$, $L_x = 5$, $L_y = 5$, scale$= 1$, and center$= [0, 0]$

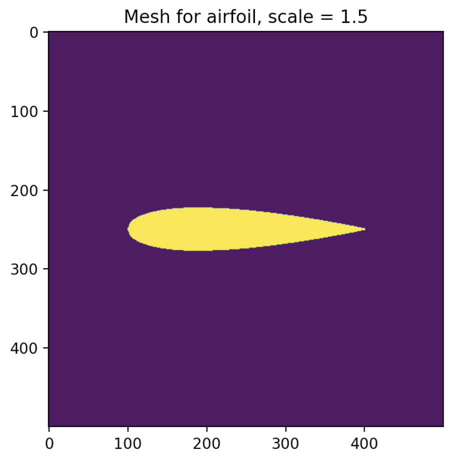

Figure 4: Airfoil Mesh for $N_x = 500$, $N_y = 500$, $L_x = 5$, $L_y = 5$, scale$= 1.5$, and center$= [0, -1.5]$

Intermediate Velocity

The next step in the solver involves calculating the velocities and pressures acting on each mesh element of the geometry. This is first done with intermediate velocities in both the U and V direction given by:

$$U^* = U^n + \Delta t[\nu(\frac{\partial^2 U^n}{\partial x^2} + \frac{\partial^2 U^n}{\partial y^2}) - (U^n \frac{\partial U^n}{\partial x} + V_n \frac{\partial U^n}{\partial y})]$$

$$V_i = V^n + \Delta t[\nu(\frac{\partial^2 V_n}{\partial x^2} + \frac{\partial^2 V_n}{\partial y^2}) - (U^n \frac{\partial V_n}{\partial x} + V_n \frac{\partial U^n}{\partial y})]$$

With each partial derivative being calculated using the following discretization scheme for the U direction:

$$\begin{aligned}
\frac{\partial^2 U}{\partial x^2} &= \frac{U(i-1,j) - U(i,j) + U(i+1,j)}{\Delta x^2} \\
\frac{\partial^2 U}{\partial y^2} &= \frac{U(i,j-1) - 2U(i,j) + U(i,j+1)}{\Delta y^2} \\
U\frac{\partial U}{\partial x} &= U(i,j)\frac{U(i+1,j) - U(i-1,j)}{2\Delta x} \\
V\frac{\partial U}{\partial y} &= \frac{1}{4}(V(i-1,j) + V(i,j) + V(i-1,j+1) + V(i,j+1))\frac{U(i+1,j) - U(i-1,j)}{2\Delta y}
\end{aligned}$$

And analogously for the V direction:

$$\begin{aligned}
\frac{\partial^2 V}{\partial x^2} &= \frac{V(i-1,j) - 2V(i,j) + V(i+1,j)}{\Delta x^2} \\
\frac{\partial^2 V}{\partial y^2} &= \frac{V(i,j-1) - 2V(i,j) + V(i,j+1)}{\Delta y^2} \\
U\frac{\partial V}{\partial x} &= \frac{1}{4}(U(i,j-1) + U(i,j) + U(i+1,j-1) + U(i+1,j))\frac{V(i+1,j) - V(i-1,j)}{2\Delta x} \\
V\frac{\partial V}{\partial y} &= V(i,j)\frac{V(i,j+1) - V(i,j-1)}{2\Delta y}
\end{aligned}$$

Pressure Poisson Equation

The next step in the workflow is to use the intermediate velocity to solve the pressure Poisson equation given by:

$$\nabla^2 p^{n+1} = -\frac{\rho}{\Delta t}\nabla \cdot U^*$$

For this step we use a similar discretization process for the RHS:

$$-\frac{\rho}{\Delta t}\nabla \cdot U^* = -\frac{\rho}{\Delta t}[\frac{U^*(i+1,j) - U^*(i,j)}{\Delta x} + \frac{V_i(i,j+1) - V_i(i,j)}{\Delta y}]$$

Since this is the most computationally expensive step in the solver workflow, we opted for two parallel algorithms that would allow us to increase the efficiency of the solver: the Parallel Red-Black Gauss-Seidel and the Parallel Jacobi Algorithm. We elaborate on the structure of these schemes below.

Parallel Jacobi Algorithm

The Parallel Jacobi Algorithm can be derived from the non-parallel Jacobi scheme given by:

$$u_i^{n+1} = \frac{1}{4}(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - h^2 f_{i,j})$$

Where h is the step size dx and f is the linear RHS. In order to paralellize this algorithm, we aim to distribute the computational workload across multiple cores. For this grid-based problem, our approach was to subdivide sections of the 2D grid and assign them to different cores with communication happening between cores of connecting layers. We also added "ghost cells" to the top, bottom, left, and right edge sections, allowing us to enforce the boundary conditions. An example of this subdivision for 3 cores is depicted in Figures 5 and 6. We note that since the first core is the fastest (due to its proximity hardware-wise), we typically give it the most amount of rows. Throughout the algorithm, the yellow cells serve as a place for the adjacent rows to communicate their values. This is done throughout the grid, while also communicating the error throughout the grid.

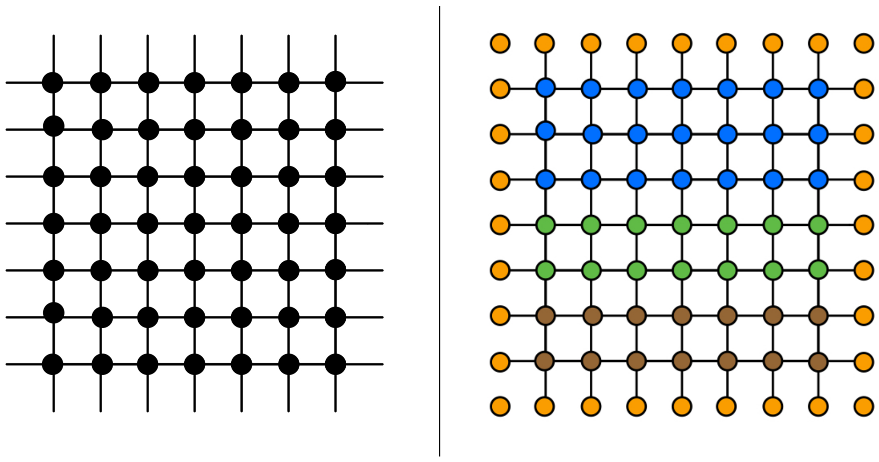
Figure 5: Division of Grid into 3 Cores for Jacobi Parallelization

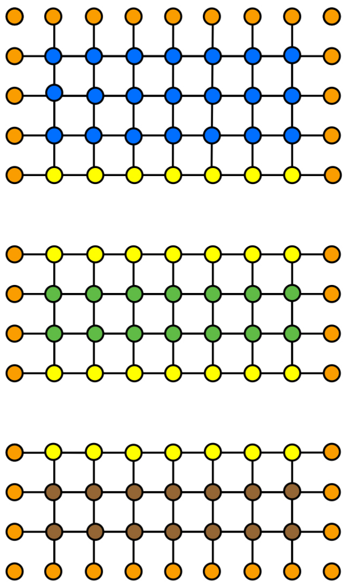
Figure 6: Division of Grid into 3 Cores for Jacobi Parallelization with Ghost Cells

Parallel Red-Black Gauss-Seidel Algorithm

The Parallel Red-Black Gauss-Seidel Algorithm can likewise be derived from the non-parallel Gauss-Seidel given by:

$$u_i^{n+1} = \frac{1}{4}(u_{i+1,j}^n + u_{i-1,j}^{n+1} + u_{i,j+1}^n + u_{i,j-1}^{n+1} - h^2)$$

From here, we notice that as opposed to the Jacobi scheme, the Gauss-Seidel scheme requires information from the new iteration which makes it extremely difficult to parallelize. As a result, we leverage what is known as the Red-Black multigrid discretization where we essentially separate the grid into points that are independent of each other. This leads to the grid in Figure 7 where all the red cells can be used independent of black cells and vice-versa. In order to appropriately incorporate the next time step, there are two phases to the algorihtm. In the first phase, we calculate the next iteration values using the Red Cell Grid. In the second phase, we use these values to calculate the next iteration for the Black Cell Grid. The way that we introduce parallelization into this method is by individually parallelizaing the two calculations of the Red and Black Cell Grids individually in the same way we parallelized a single run of Jacobi.

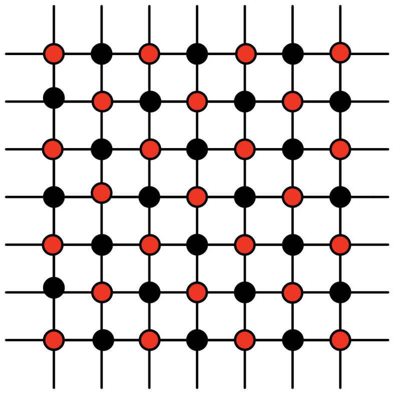

Figure 7: Red Black Gauss Seidel Grid

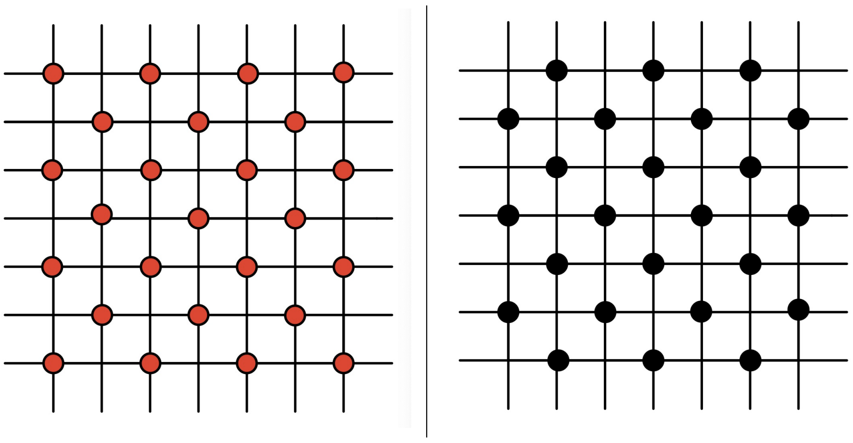

Figure 8: Red Black Gauss Seidel Grid Separation

Corrective Velocity

After solving for the pressure via the Poisson equation, we can apply a corrective step to our intermediate velocity given by:

$$U^{n+1} = -\frac{\Delta t}{\rho}\nabla p^{n+1} + U^*$$

Where we can once again represent the partial derivatives of the pressure using the following discretization scheme:

$$\frac{\partial p}{\partial x} = \frac{p(i,j) - p(i-1,j)}{\Delta x}$$
$$\frac{\partial p}{\partial y} = \frac{p(i,j) - p(i,j-1)}{\Delta y}$$

We then iterate over this algorithm for the entirety of the input period given a dt value determined using the stability of the algorithms. It is almost important to note that the boundary conditions are applied to the velocity and pressure grids throughout the workflow. Once this is completed, the solver returns to the intermediate velocity calculation and is run iteratively for N iterations.

Solver Architecture

The Navier Stokes solver has the structure shown in Figure 9. Note that the solver requires MPI for python which requires the `pip install mpi4py` on Adroit.

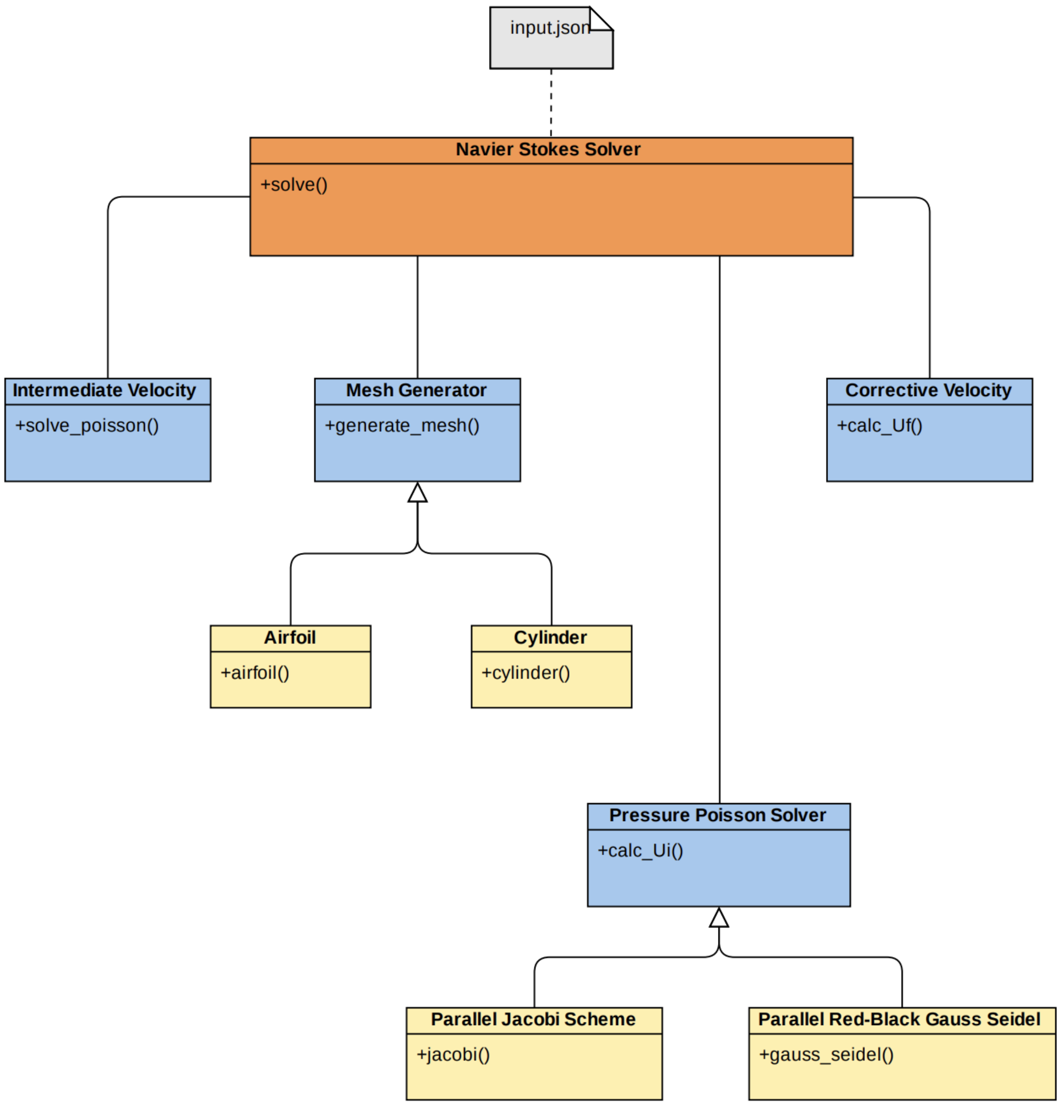

Figure 9: Class Diagram for the Navier-Stokes Solver

The solver also takes in an "input.json" file which requires the arguments in Figures 1, 2, and 3. The input file contains three sections of which each contains a different category of input parameters.

Parameter	Definition	Options
geometry	Shape	'airfoil', 'cylinder', 'none'
scale	Scaling of the geometry	
center	Center point of the geometry	

Table 1: Geometric parameters for the geometry_params section of the input file

Parameter	Definition	Options
input_solver	Method for solving Pressure Poisson	'jacobi', 'gauss_seidel'
length_x	Length of the grid in the x-direction	
length_y	Length of the grid in the y-direction	
n_x	Number of grid points in the x-direction	
n_y	Number of grid points in the y-direction	
nu	Kinematic viscosity	
rho	Density	
iters	Number of iterations for the NS solver	
dt	Timestep	
eps	Minimum error allowed for solver convergence	
maxitr	Maximum number of iterations allowed for convergence per timestep	

Table 2: Solver parameters for the solver_params section of the input file

Parameter	Definition	Options
u_init	Initial u-velocity for boundary conditions	
v_init	Initial v-velocity for the boundary	
u_flow	Initial flow velocity for the U grid	
v_flow	Initial flow velocity for the V grid	
cond_type	Type of flow condition	'lid_driven_cavity', 'geometry_flow'

Table 3: Initial parameters for the init_conditions section of the input file

The 'geometry_flow' must be used for any case where the airfoil or cylinder mesh is used.

Results

Performance Analysis of Algorithms

Our parallelized Jacobi and Red Black Gauss-Seidel iteration algorithms were tested on Adroit for their performance. The algorithms iteratively solve the problem $\nabla^2 x = b$, where b is a 1000 x 1000 flattened grid, initialized entirely with zeros except for a one at $b(n/2, n/2)$. The algorithms run until the maximum component-wise error reaches 1e-10. The results are shown in Figures 10 and 11.

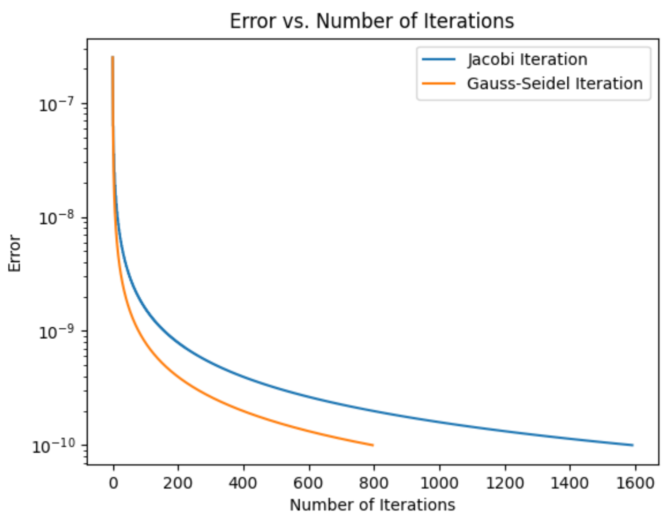

Figure 10: Error Diagram

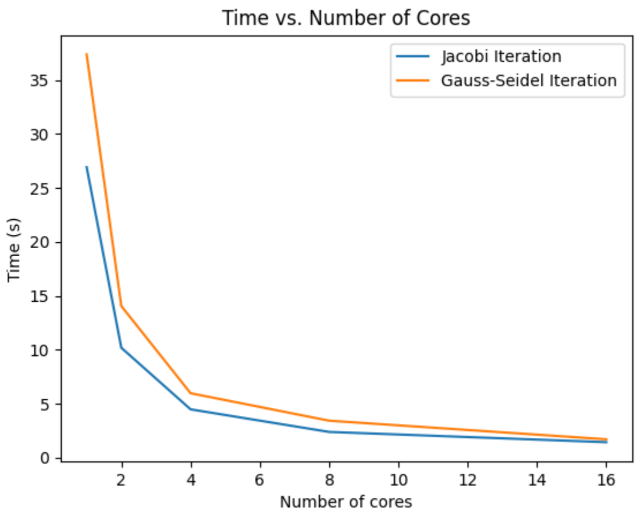

Figure 11: Timing Diagram

We find that the Red Black Gauss Seidel algorithm takes approximately half as many iterations to reach the desired error. The two algorithms perform similarly with respect to computational time, with Jacobi iterations slightly outperforming Red Black Gauss Seidel iterations for lower number of cores.

Lid-Driven Cavity Flow

We now move on to testing our Navier Stokes solver. The lid-driven cavity flow is a useful first-test for a CFD solver. The setup is a 1x1 box, with internal initial u-velocity and v-velocity zero. The velocity boundary conditions are: $(u, v) = (0, 0)$ for the left, right, and bottom sides, while $(u, v) = (1, 0)$ for the top side. Pressure boundary conditions are Neumann, or zero derivative. The results outputted by our solver are then compared to the results put forward by Ghia et al.
We ran our solver for the lid-driven cavity problem with three different Reynolds numbers: 100, 1000, and 3200. The x- and y-lengths are both 1. The boundary condition u_{bound} was set to be unitary, with the dynamic viscosity scaled to create various Reynolds number flows. Density was also set to be unitary. The mesh was selected was a 64x64 grid. From the u-velocity and v-velocity contour plots for all Reynolds numbers, we can see a clockwise vortex forming. This is confirmed by our streamline plots in Figure 14.

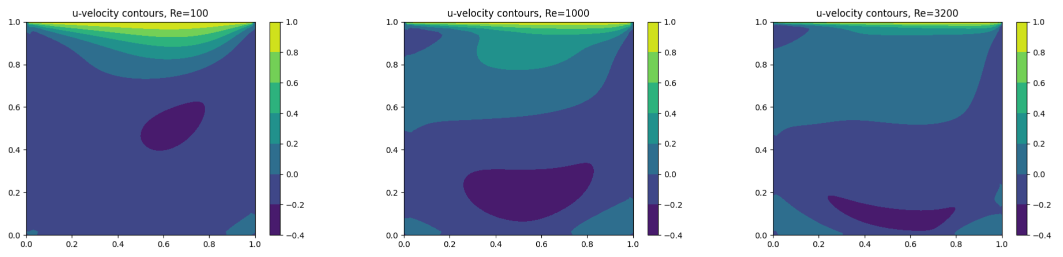

Figure 12: Horizontal velocity contours, various Reynolds numbers, lid-driven cavity flow, 64x64 mesh.

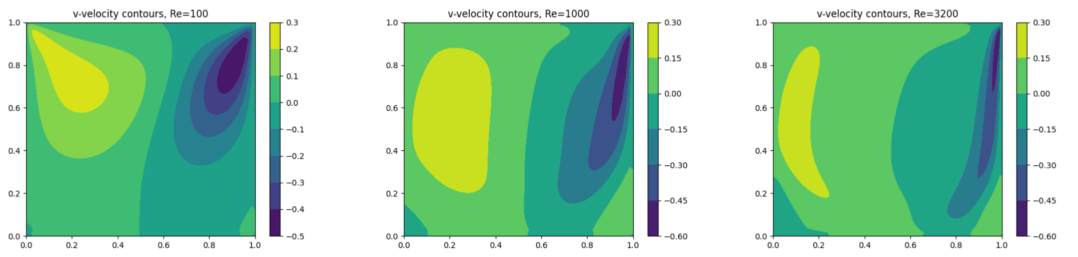

Figure 13: Vertical velocity contours, various Reynolds numbers, lid-driven cavity flow, 64x64 mesh.

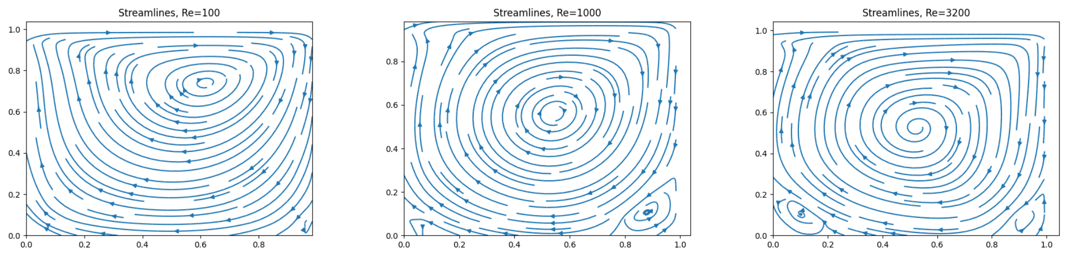

Figure 14: Streamlines at various Reynolds numbers, lid-driven cavity flow, 64x64 mesh.

From the streamline plots in Figure 14, we see one major vortex forming in all three cases. For the case of Re=1000, we see a small counterclockwise vortex forming in the bottom right corner. In the case of Re=3200, we see the counterclockwise vortex in the bottom right as well as another counterclockwise vortex in the bottom left corner. In order to test the accuracy of our solver, we compare to the solver developed by Ghia et al. in [8]. They document the centerline velocity profile ($x = 0.5$) for various Reynolds number flows. Figures 15, 16, and 16 compare these results with our solver for the cases of Re$= 100, 1000$, and 3200.

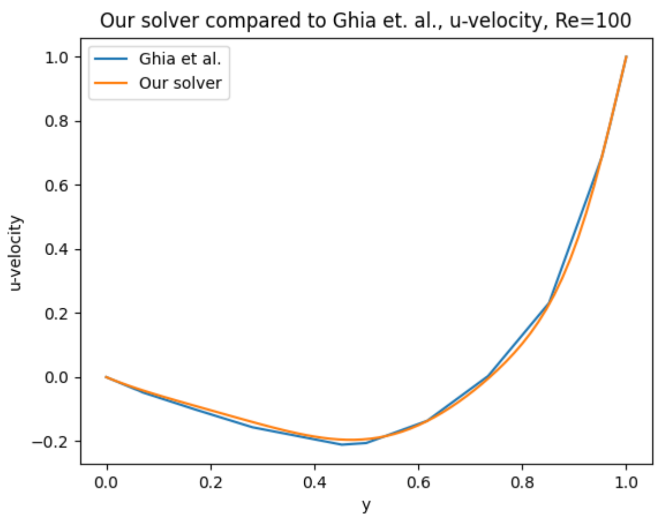

Figure 15: Centerline velocity, Our solver compared to Ghia et al. Re= 100

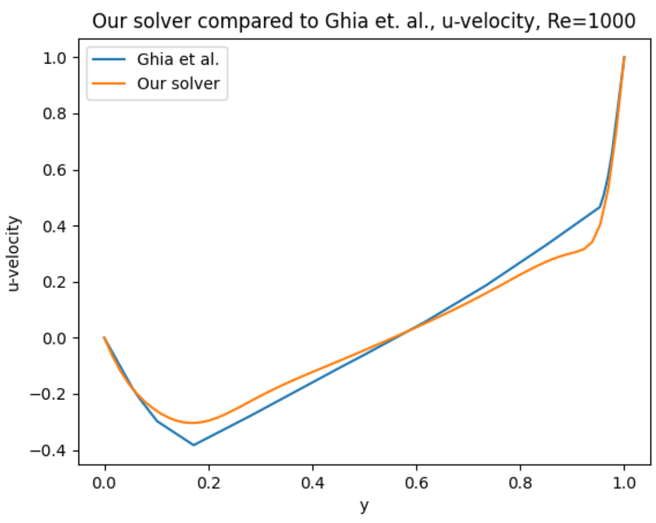

Figure 16: Centerline velocity, Our solver compared to Ghia et al. Re= 1000

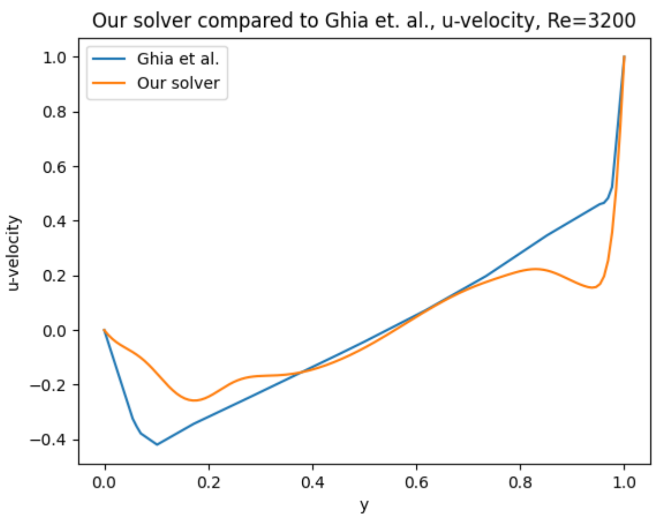

Figure 17: Centerline velocity, Our solver compared to Ghia et al. Re= 3200

We can see that our solver compares well to the results shown in Ghia et al. Our solver follows almost exactly the results in Ghia et al. in the case of Re= 100, with some differences with Re= 1000 and Re= 3200.

Lid Driven Cavity Flow for Parallelized Algorithm

In order to benchmark the computation time improvement our parallelized algorithms give, we ran our parallelized solver on Adroit for various mesh sizes, solving the lid driven cavity problem. For all computations Re= 100 was used and dt was chosen based on stability. We performed this analysis on the Jacobi iterative scheme.
For the 64 x 64 and 128 x 128 grid sizes, we ran 1000 timesteps on various number of cores. For the 256 x 256 grid, we ran 500 timesteps. The resulting plots are shown in Figures 18, 19, and 20.

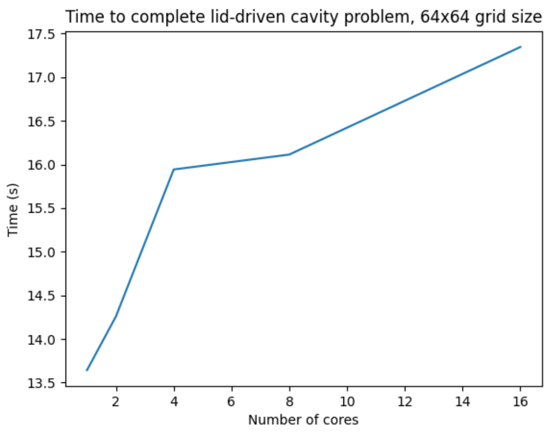

Figure 18: Time to complete 1000 timesteps of the lid driven cavity problem, 64 x 64 grid, Jacobi solver

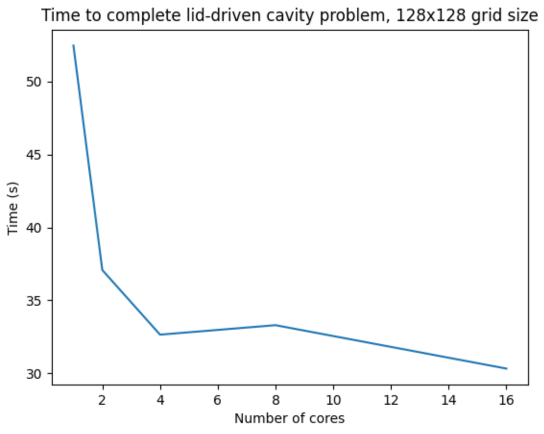

Figure 19: Time to complete 1000 timesteps of the lid driven cavity problem, 128 x 128 grid, Jacobi solver

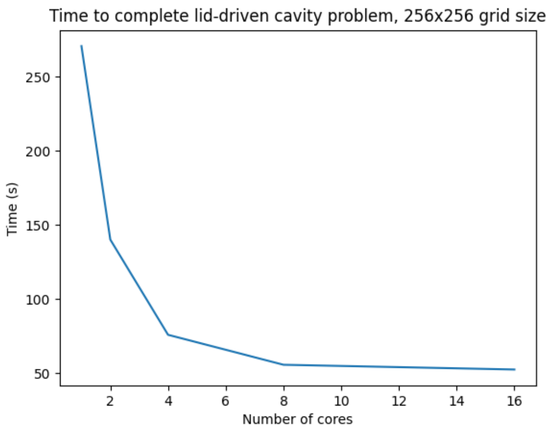

Figure 20: Time to complete 1000 timesteps of the lid driven cavity problem, 256 x 256 grid, Jacobi, solver

For the 64 x 64 grid, we see an increase in computational time as number of cores increases. This is due to the overhead that comes with parallelization; for this small of a grid size, we would not want to use parallelization. In the cases of 128x128 and 256x256 grids, though, we see major improvement in computational time as number of cores increases. For running larger meshes, parallelization gives significant improvement.

Duct Flow

To model duct flow, we again restrict our boundary to x- and y-lengths of 1. The velocity boundary conditions are zero on the top and bottom sides, and zero derivative on the left and right sides. Pressure boundary conditions are derivative on the top and bottom sides. On the left, we force a pressure boundary condition of 5e-7, and on the right a pressure boundary condition of 1e-7. All internal velocities are all initialized to 0. These conditions were simulated for $t = 1s$, $t = 10s$, and $t = 100s$. The results are shown in Figures 21, 22, and 23.

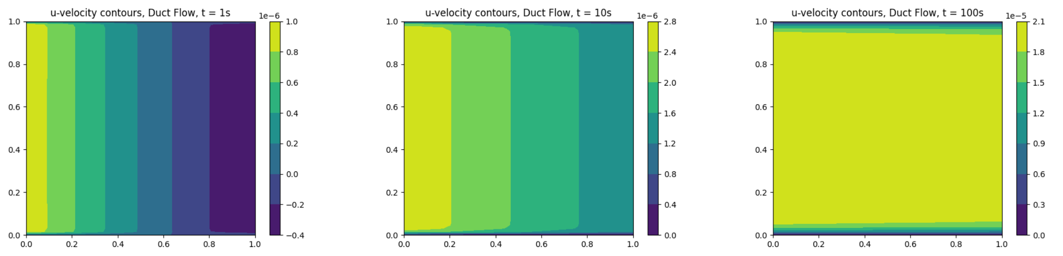

Figure 21: Horizontal velocity contours for duct flow.

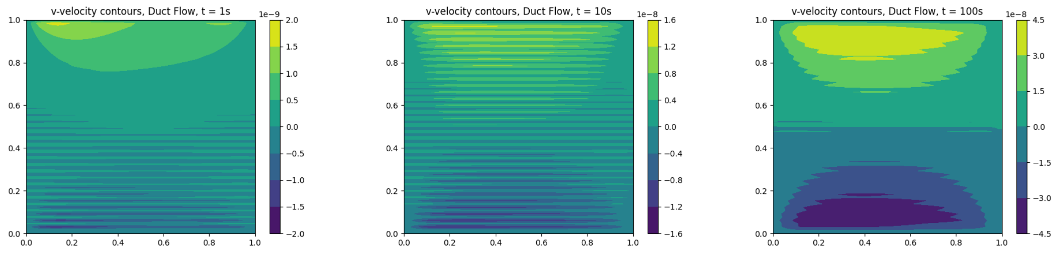

Figure 22: Vertical velocity contours for duct flow.

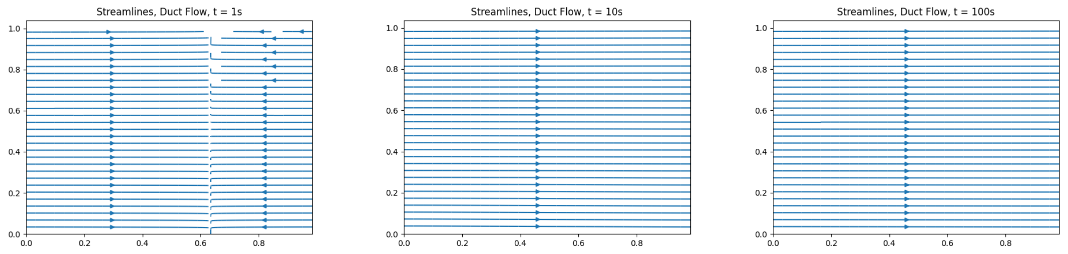

Figure 23: Streamlines for duct flow.

In the case of $t = 1s$, we can see that the flow is entering from both sides, and creating a stagnation point near $x = 0.6$. For the case of $t = 10s$, we can see that the flow is directed to the right. In the case of $t = 100s$, the flow looks much more uniform compared to $t = 10s$.

Duct Flow, Equal Pressure

This test case was the same as duct flow in the previous section, except we apply an equal pressure of 5e-7 to both the left and right sides. We can see that the flow meets at a stagnation point of $x = 0.5$, in Figure 24.

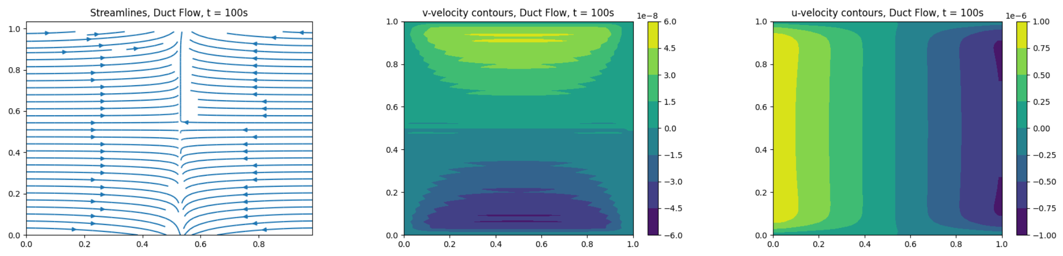

Figure 24: Duct flow plots, equal pressure applied to both sides.

L-Duct

We model an L-duct by holding applying a pressure boundary condition of 1e-7 to the left half of the upper boundary and 5e-7 to the bottom half of the right boundary. Zero derivative pressure boundary conditions and zero velocity boundary conditions were enforced on the "L" boundary. The results are shown in Figure 25. As expected, we observe a flow from the bottom right to the top left of our square, with the flow turning to move through the L-duct.

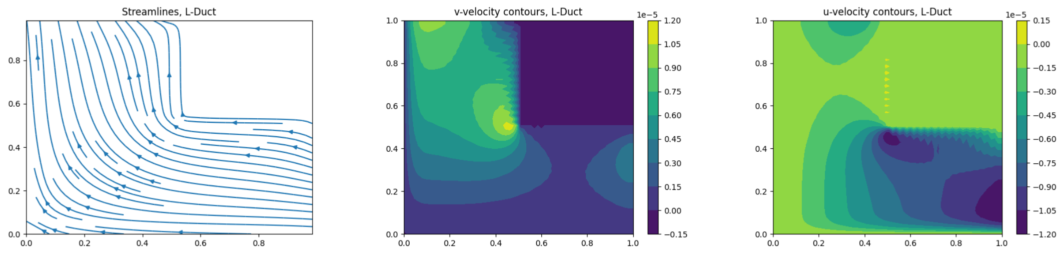

Figure 25: L-duct flow plots.

Airfoil

Using the geometry meshes generated by the algorithm detailed above, we attempt to run our Navier Stokes Solver on an airfoil geometry. For velocity boundary conditions, we use Neumann boundary conditions on the computational boundary, and zero velocity on the mesh geometry. For pressure, we use Neumann boundary conditions on the computational boundary. We simulate the flow for Re= 100. We initialize all velocities in the space to be the same, except on the geometry mesh boundary, to simulate flight. We used a 150 x 150 computational grid, and the mesh is shown in Figure 26. For zero angle of attack, the results are shown in Figure 27

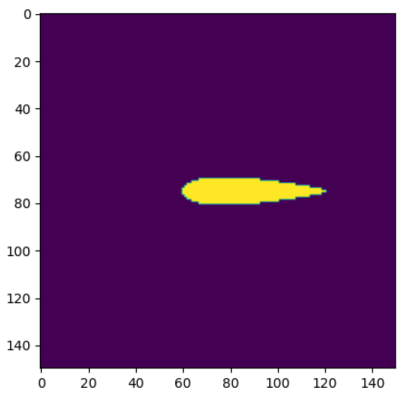

Figure 26: Mesh for airfoil used for further testing.

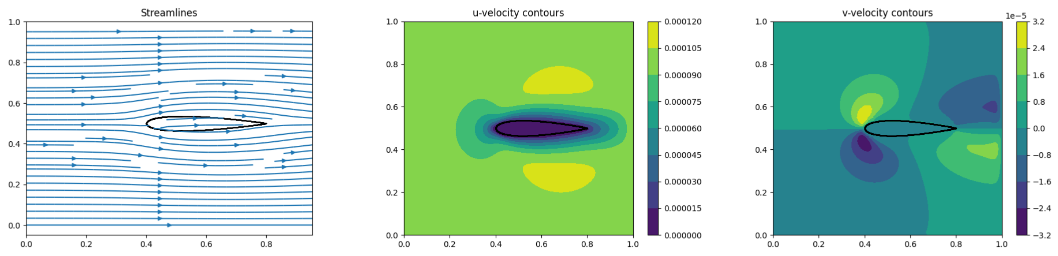

Figure 27: Streamlines and velocity contours for airfoil geometry at zero angle of attack.

For a 10° angle of attack, we get the following plots:

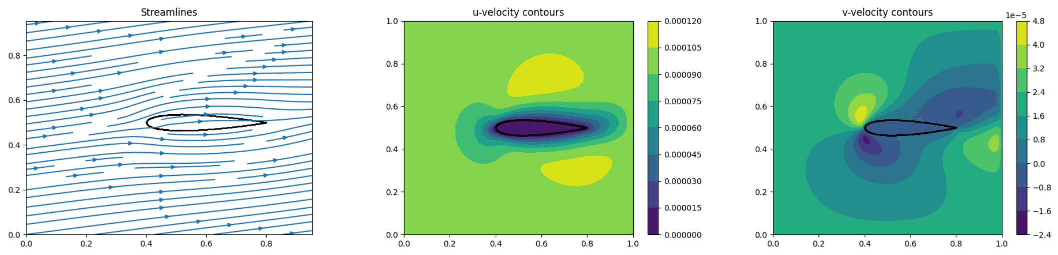

Figure 28: Streamlines and velocity contours for airfoil geometry at 10° angle of attack.

These results seem plausible, but future work will be devoted to comparing these results to empirical and analytical results.

Cylinder

We attempt to solve for flow around a cylinder using the same conditions from the airfoil case. The mesh tested is shown in Figure 29. The results are displayed in Figure 30.

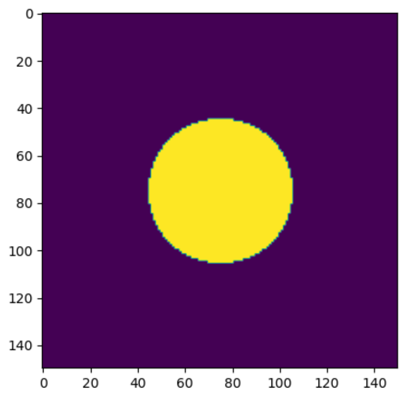

Figure 29: Mesh for cylinder used for further testing.

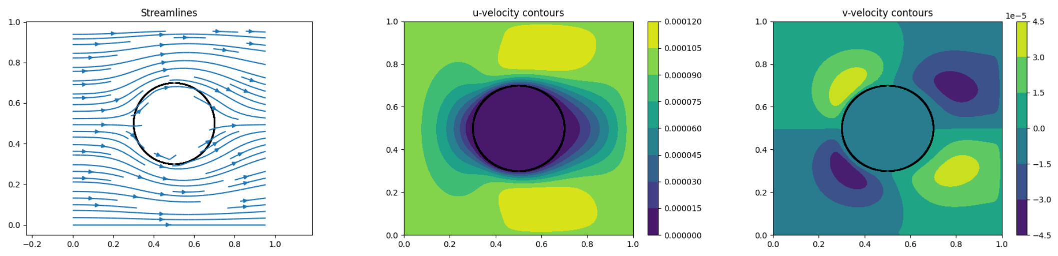

Figure 30: Streamlines and velocity contours for cylinder geometry.

Similar to the airfoil case, future work into this project will be devoted toward analyzing the efficacy of these results, and the efficacy of the boundary conditions used.

References

[1] Araujo Bitencourt, P. Algorithm for Two-dimensional Airfoil Automatic Structured Mesh Generation. 2021 Nov.
https://repositorio.unesp.br/bitstream/handle/11449/215833/bitencourt_pha_tcc_ilha.pdf?sequence=4

[2] Gimeno J. Incompressible Navier-Stokes solver in Python. 2021 June 22.
https://upcommons.upc.edu/bitstream/handle/2117/360887/TFM__Incompressible
_Navier_Stokes_solver_in_Python.pdf?sequence=1

[3] Hultmark M. Fluid Mechanics Lecture 14. 2018

[4] Marchi C, et al. The Lid-Driven Square Cavity Flow:Numerical Solution with a 1024 x 1024 Grid. 2009.

[5] Massachusets Institute of Technology. Lecture 18: Numerical Fluid Mechanics. 2015.
https://ocw.mit.edu/courses/mechanical-engineering/2-29-numerical-fluid-mechanics-spring-2015/lecture-notes-and-references/MIT2_29S15_Lecture18.pdf

[6] Narayanan S., et al. Visualization Analysis of Numerical Solution With 32x32 and 64x64 Mesh Grid Lid-Driven Square Cavity Flow. 2020.

[7] Owkes M. A guide to writing your first CFD solver. 2017 June 2.
https://www.montana.edu/mowkes/research/source-codes/GuideToCFD.pdf

[8] Seibold B. A compact and fast Matlab code solving the incompressible Navier-Stokes equations on rectangular domains. 2008 March 31. https://math.mit.edu/ gs/cse/-codes/mit18086_navierstokes.pdf

[9] Stanford University. Solution methods for the Incompressible Navier-Stokes Equations.
https://web.stanford.edu/class/me469b/handouts/incompressible.pdf

[10] U Ghia; K.N Ghia; C.T Shin (1982). High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. , 48(3), 387–411.

[11] Work D. 12 Parallel Gauss Seidel & Efficient OpenMP.
https://www.youtube.com/watch?v=ZAnjKj-3v7o