Project 2 Solution: 2D Shallow Water Finite Volume Method

AERO 523 Fall 2021
Author: Erik Rutyna

November 12, 2021

Introduction

The goal of this project was to solve for the height as a function of time using the shallow water equations for fuel sloshing in a tank using the Finite Volume Method (FVM). Since this problem existed in 2-D, the 2-D FVM is used over the domain shown in Figure 1 [2].

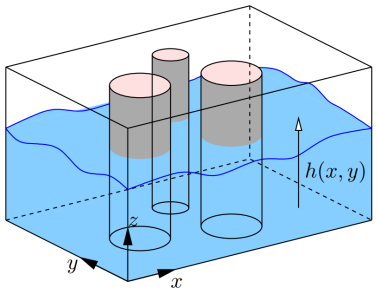

Figure 1: The fuel tank geometry. Only the top slice (height) of the tank is measured.

Because this problem exists in 2-D, our state and fluxes now come in vector form. The full version of these equations is shown in Figure 2 [2].

$$
\begin{aligned}
&\frac{\partial}{\partial t}(h) &+ &\frac{\partial}{\partial x}(hu) &+ &\frac{\partial}{\partial y}(hv) &= 0 \quad &\text{(continuity)} \\
&\frac{\partial}{\partial t}(hu) &+ &\frac{\partial}{\partial x}(hu^2 + gh^2/2) &+ &\frac{\partial}{\partial y}(huv) &= 0 \quad &(x\text{-momentum}) \\
&\frac{\partial}{\partial t}(hv) &+ &\frac{\partial}{\partial x}(hvu) &+ &\frac{\partial}{\partial y}(hv^2 + gh^2/2) &= 0 \quad &(y\text{-momentum})
\end{aligned}
$$

Figure 2: The conservation equations for the shallow water in the tank

However, it is difficult to manipulate and use these equations in this form, so another way of writing it is

$$\frac{\partial \boldsymbol{u}}{\partial t} + \nabla \cdot \vec{\boldsymbol{F}} = \boldsymbol{0} \tag{1}$$

where the two vectors, $\boldsymbol{u}$ and $\vec{\boldsymbol{F}}$, are

$$\boldsymbol{u} = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}, \qquad \vec{\boldsymbol{F}} = \begin{bmatrix} hu & hv \\ hu^2 + gh^2/2 & hvu \\ huv & hv^2 + gh^2/2 \end{bmatrix}. \tag{2}$$

As with all FVMs, our computation domain (mesh) was divided up into triangular cells and then the conservation equations were then applied onto each individual cell. An example of such a cell can be seen in Figure 3 [1]. A flux was then calculated across both boundary and internal edges of each cell as the mesh was marched forward in time. This allowed the state to be updated at each time interval according to the inflow/outflow of the state based on the fluxes. When the mesh was marched all the way to the final time, t_{max}, the simulation had ended and the results were then observed.

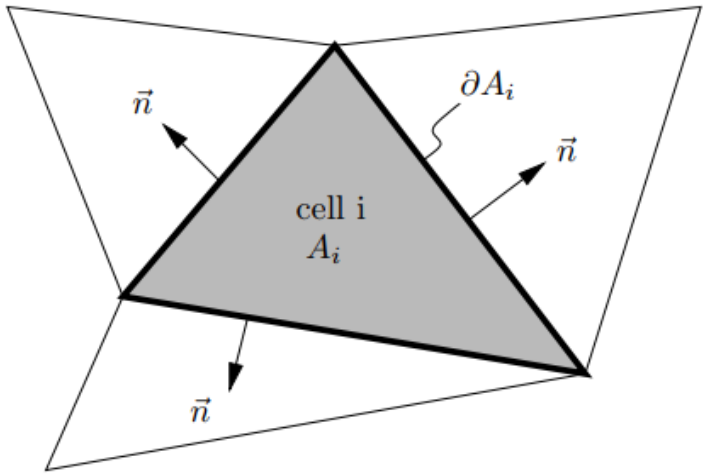

Figure 3: A generic triangular cell with index i, with some of its properties labeled. Notice that the edge-normals always point out of the cell.

Setup

Geometry

For the project I was given mesh files in the form of .gri files, as well as a helper function to read in the mesh and put it in a usable format. Both meshes are unstructured, but the helper function that was given to me made it easy to index nodes (end points of edges), boundary edges (edges on edge of domain against inviscid walls), internal edges (edges between two cells), and the cells themselves. While both meshes are the same geometry, the more coarse mesh (less cells) has been plotted to show what this mesh looked like, and can be seen in Figure 4 [2].

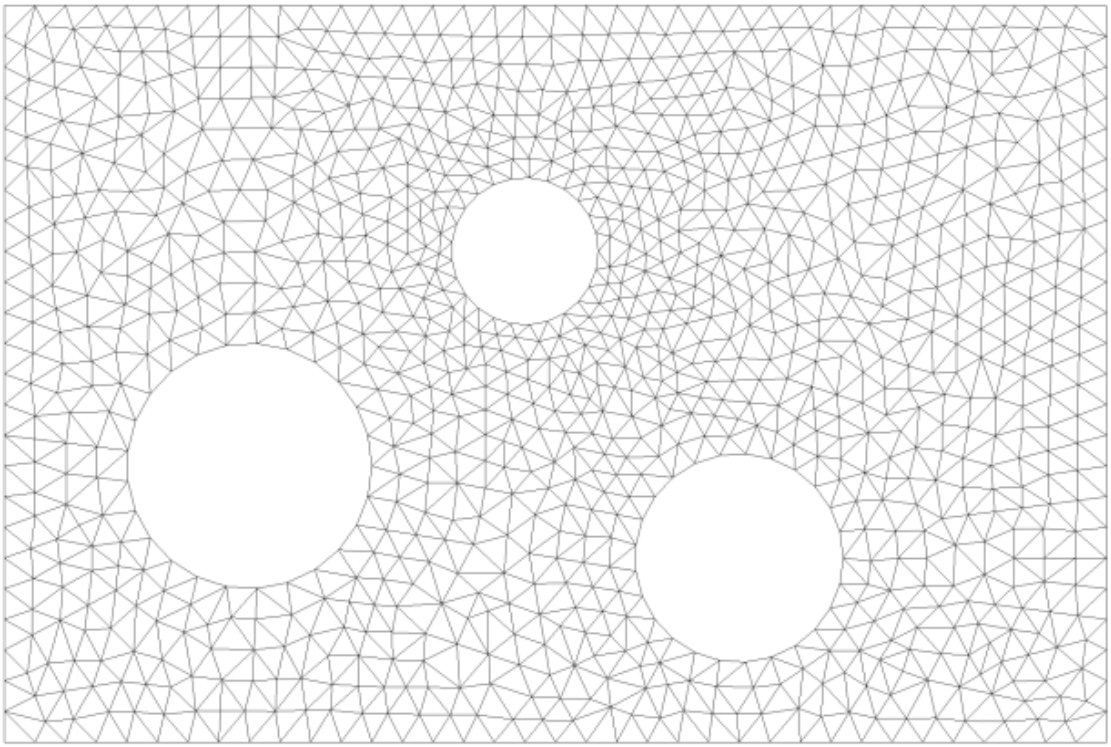

Figure 4: This is a visualization of the coarse (less cells) mesh. The origin in the bottom left corner with positive x to the right and positive y pointing up, just like the axis shown in Figure 1.

Because of the way the mesh was read in via the helper function, the mesh was standardized for computing cell characteristics. As such, anytime geometric properties of a given cell and/or edge needed to be computed, the following formulas could be used,

$$\Delta l_{i,e_1} = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2} \tag{3}$$

$$\vec{n}_{i,e} = \left[\frac{y_B - y_A}{\Delta l_{i,e}}\hat{x}, \frac{x_A - x_B}{\Delta l_{i,e}}\hat{y}\right] \tag{4}$$

$$A_i = \sqrt{s(s-a)(s-b)(s-c)}, \qquad s = \frac{\Delta l_{i,e_1} + \Delta l_{i,e_2} + \Delta l_{i,e_3}}{2} \tag{5}$$

$$c_i = \left[\sum_{k=1}^{3} x_{i,k}, \sum_{k=1}^{3} y_{i,k}\right], \tag{6}$$

where $\Delta l_{i,e}$ is the respective edge length of the cell on edge e, c_i is the centroid of cell i, and $x_{i,k}$ and $y_{i,k}$ are the x and y coordinates of the node k on cell i. All node coordinate positions are measured from the origin in the bottom left corner of the domain. An example cell with some geometric properties can be seen in Figure 5 [1].

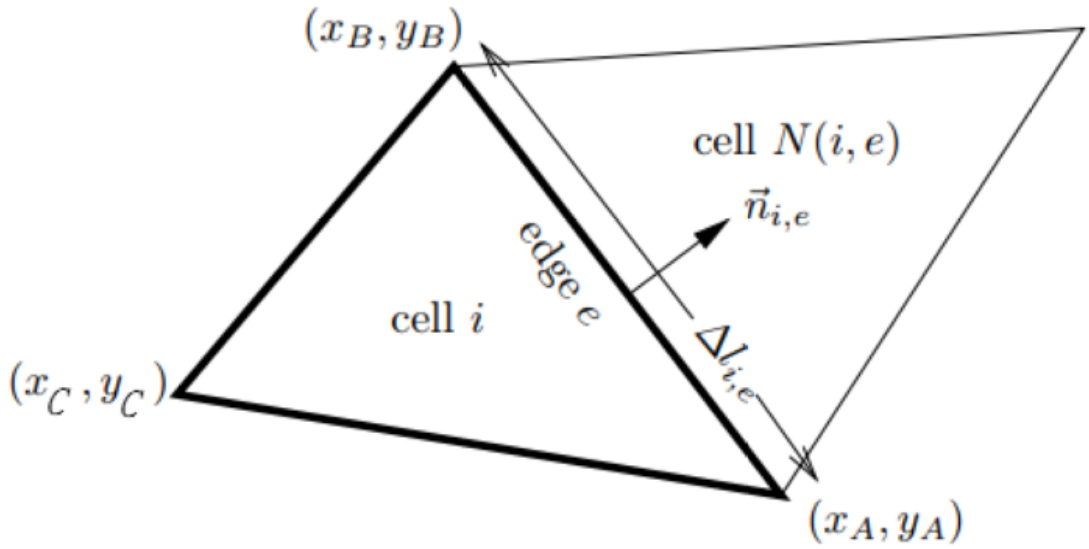

Figure 5: An example cell with the geometric properties labelled.

Initial Condition

The initial condition for the problem is based on the height of the fluid at different coordinates. This height can be described with a formula,

$$h^0(x,y) = 1.0 + 0.3e^{-50(x-1.3)^2-50(y-0.9)^2} \tag{7}$$

and the velocity at all points is zeroed, $\vec{v}^0(x,y) = 0$. This initial condition was plotted on the coarse mesh and can be seen in Figure 6.

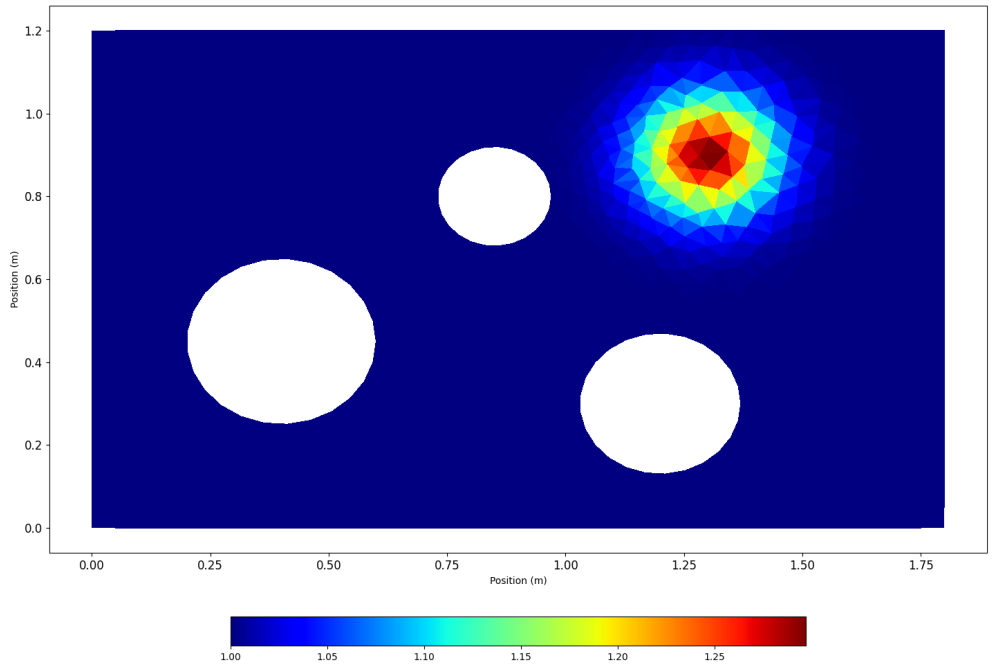

Figure 6: The initial condition plotted on the coarse mesh.

Boundary Conditions

The only boundary conditions present in the problem are that of inviscid walls. That means that there is no flux through the wall, only a pressure term is present. In this case, this pressure term is modelled by the following formula,

$$\vec{\boldsymbol{F}} \cdot \vec{n}\Big|_{wall} = \begin{bmatrix} 0 \\ \frac{n_x g h^2}{2} \\ \frac{n_y g h^2}{2} \end{bmatrix}. \tag{8}$$

These inviscid walls are present at four sections in the domain, along the outer wall, and along the three interior pipes. These walls and pipes have been colored and labelled, and can be seen in Figure 7.

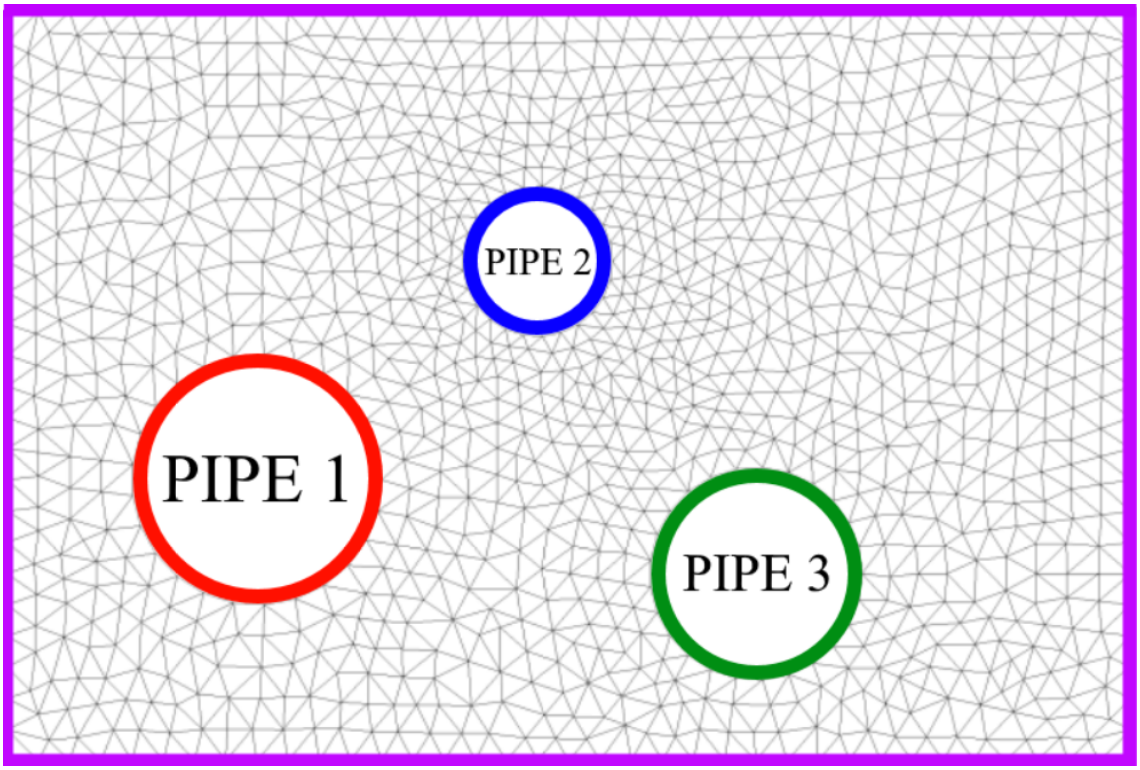

Figure 7: The three pipes are the interior inviscid walls, and the outer, purple boundary is the exterior wall.

Residual Calculations

The flux function used for this project was one that mirrored the Roe Flux. The function to calculate this was provided by Professor Fidkowski. This makes residual calculation relatively trivial, as I all to do is give the function the state and geometric properties of the cell and the state of the neighboring cell. As such, the residual calculations at each timestep for a cell i are as follows,

$$\boldsymbol{R_i} = \sum_{e=1}^{3} \hat{\boldsymbol{F}}(\boldsymbol{u_i}, \boldsymbol{u}_{N(i,e)}, \vec{n}_{i,e}) \Delta l_{i,e}, \tag{9}$$

where $\hat{\boldsymbol{F}}(\boldsymbol{u_i}, \boldsymbol{u_{N(i,e)}}, \vec{n}_{i,e})$ is the flux calculated via the helper function provided by Professor Fidkowski. Cell i is the cell the flux is being computed at and cell $N(i,e)$ is the adjacent cell for that specific edge.

Global Timestepping

In order to maintain stability, a properly chosen global timestep must be selected. However, due to the unsteady nature of the problem, this timestep had to be chosen after each update in time. Because the fluxes act independently from one another in this method, it means that each cell is going to have a unique timestep. This local timestep for cell i is

$$\Delta t_i = \frac{2A_i}{\sum_{e=1}^{3} |s|_{i,e} \Delta l_{i,e}} CFL, \tag{10}$$

where $|s|_{i,e}$ is the absolute maximum wave propogation speed along the cell edge $l_{i,e}$. Then to guarantee stability for the system, the timesteps were arranged into a vector and the minimum value was taken,

$$\Delta t|_{global} = minimum(\Delta \boldsymbol{t}). \tag{11}$$

Task 1 - Wall Fluxes

Task one asked me to implement a wall flux function for Eqn. 8, and I was successful in doing so. However, I did it in an in-direct way. I chose to make a single residual function that computes $\boldsymbol{R}$ for an entire timestep, so it calculates the residuals for both the interior and boundary edges. The general algorithm can be seen in Algorithm 1. In this case $\boldsymbol{R}$ is a represents a matrix/vector of state vectors for the residuals at time t.

Algorithm 1 Residual Calculation

1: **for all** e in IE **do**
2: Calculate geometric properties for the cell that has edge e (Area, edge norm, etc)
3: Calculate flux and wave speed via Prof. Fidkowski's function for edge e
4: Calculate local timestep for edge e
5: Update residual vector
6: Update timestep vector with local timestep
7: **end for**
8: **for all** e in BE **do**
9: Calculate geometric properties for the cell that has edge e (Area, edge-normals, etc)
10: Calculate flux via Eqn. 8 for edge e
11: Calculate local timestep for edge e
12: Update residual vector
13: Update timestep vector with local timestep
14: **end for**
15: **return** residual vector, minimum($\Delta \boldsymbol{t}$)

Here IE $\equiv$ internal edges and BE $\equiv$ boundary edges. Additionally, $\Delta \boldsymbol{t}$ is a vector of all the timesteps for each edge. By taking the minimum of this, we enforce stability as the simulation marches in time.

This looping process is known as "residual assembly". My secret to it being so successful is to have a series of working helper-functions to calculate things like geometric properties, like those listed in Eqn. 3 to Eqn. 6, for edge e.

In order to test this algorithm, I wanted a simple mesh that would be very easy to check by hand. I used a 4-node and 2-element mesh with unit length known as "test.gri" that can be seen in Figure 8. While testing this mesh I used what is known as "Free-stream Conditions" which is a $h = 1,\ u = v = 0$ for all elements.

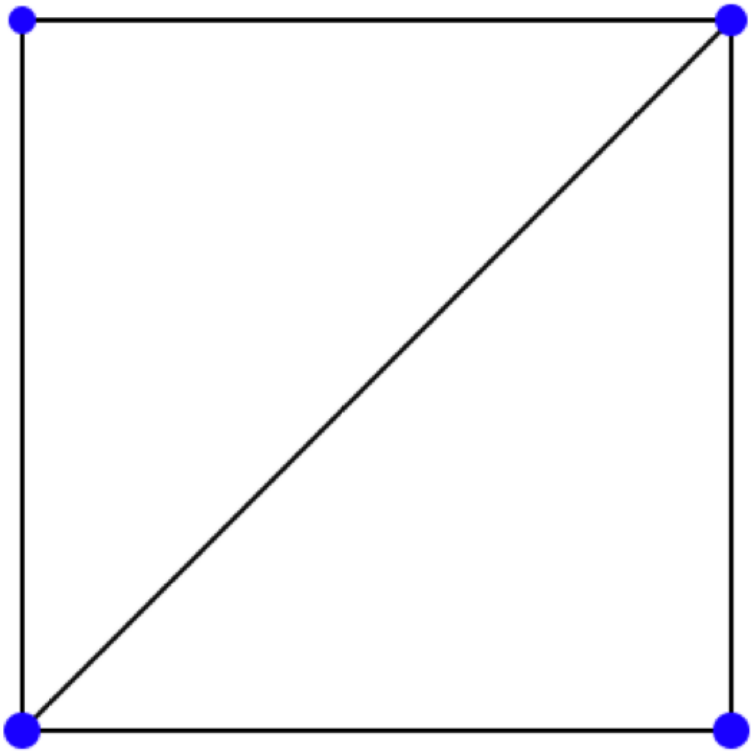

Figure 8: The simple 4-node, 2-element test mesh. Element 1 is in the top left, and element 2 in the bottom right.

Due the simplicity of this test mesh, I can do calculations by hand in order to check their validity and know that my loop over the boundary edges (this loop mimicking the wall flux function) works how its supposed to.

Under the assumption that Professor Fidkowski's flux computation function is accurate, the residual from the only internal edge at $t = 0$ under the free-stream condition for the two elements is,

$$\boldsymbol{R} = \begin{bmatrix} 0 & -4.9 & 4.9 \\ 0 & 4.9 & -4.9 \end{bmatrix}. \tag{12}$$

Starting at the top and going clockwise, the four residuals from the four boundary edges are,

$$\begin{aligned} \boldsymbol{R_{1}}_{(top)} &= [0, 0, -4.9] \\ \boldsymbol{R_{2}}_{(right)} &= [0, 4.9, 0] \\ \boldsymbol{R_{2}}_{(bottom)} &= [0, 0, 4.9] \\ \boldsymbol{R_{1}}_{(left)} &= [0, -4.9, 0] \end{aligned}. \tag{13}$$

When all of these residuals are summed together it ends up with $\boldsymbol{R} = \vec{0}$, as it should. In the case of steady state free-stream condition, there should be no net change in the system anywhere, and so the end result is that the boundary check passes the test.

Task 2 - FVM Solver & Algorithm Design

Task 2 asks me to write a first-order finite-volume solver for the given problem, as well as describe my code structure. I was able to successfully create a finite-volume solver using the given helper functions provided by Professor Fidkowski. A higher level generalized approach can been in the flowchart in Figure 9. A more in-dept pseudo-code variant of this algorithm can be seen in Algorithm 2. Please note that emboldened vectors for u, A, and R represent the entire group of state vectors, areas, and residuals for all cells, not the individual state vector/residual vector at cell i.

Algorithm 2 Finite Volume Solver

load in mesh
compute centroids of cells
initialize state to initial condition
compute areas of cells
set start time to $t = 0$
while $t < t_{max}$ **do**
    $\boldsymbol{R}$, Δt = residual and timestep calculation (see Algorithm 1)
    $\boldsymbol{u}^{n+1} = \boldsymbol{u}^n - \frac{\Delta t}{\boldsymbol{A}}\boldsymbol{R}$
    Calculate forces across walls
    $t = t + \Delta t$ from residual calculation
    Update arrays for desired quantities: forces on walls, L_2 norm of residuals, timesteps, etc.
end while
Save forces, timesteps, residual norm, and state vectors to disk

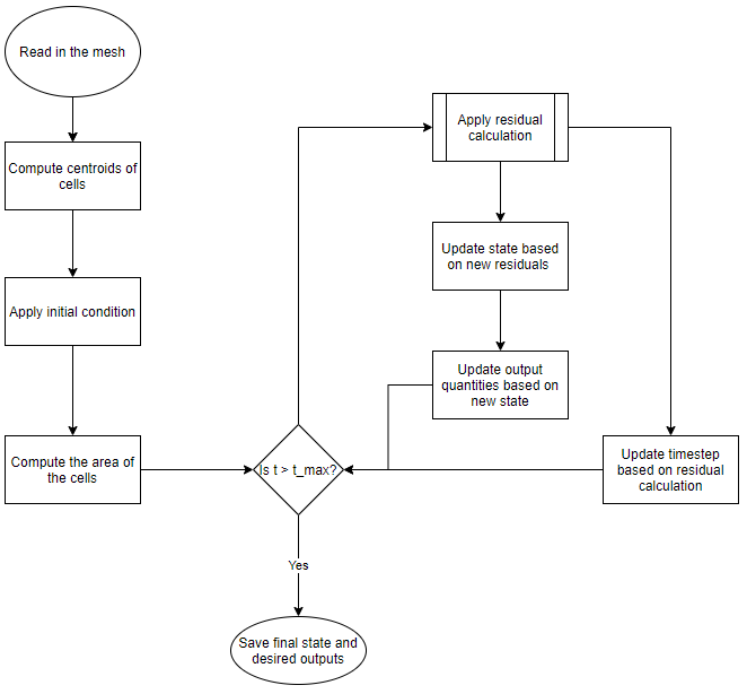

Figure 9: A flow chart visualizing the algorithm in Algorithm 2.

The code itself makes heavy use of helper-functions that do things such as computing areas, edge lengths and normal-vectors, calculating net forces, as well as plotting. This makes the over-arching algorithm relatively simple as whenever I need a given quantity, like area, I just call my function "areaCalculator" with an input of the cell I need the area of. This creates level of abstraction that helped me to streamline the design and debugging of the solver.

The downside to my methodology is that I am constantly computing geometric properties, instead of passing them around as arguments. This makes my code inefficient computational speed wise, but I would argue that it makes up for that in clarity and debugging. By functionalizing everywhere that I could, it made it simple for me to test and debug various my various helper functions without having to run the entire time integration portion of the solver.

Helper Functions

This subsection contains a brief description of the helper functions that I wrote that are used in the time marching section of the finite-volume solver.

1. *centroid*: When given the nodes of the cell and the coordinates for each node in the cell it returns the geometric center of the cell.

2. *initialCondition*: When given two arrays of equal length corresponding to X-Y coordinates, it returns the height based on the initial condition function in Eqn. 7.

3. *initialFreestream*: When given two arrays of equal length corresponding to X-Y coordinates, it returns the height based on the freestream initial condition.

4. *plotConditionState*: When given the mesh and a vector of a states it produces a contour mapping of the state at each cell in the mesh.

5. *edgePropertiesCalculator*: When given two nodes in the counter-clockwise order it calculates the length of the edge and the normal vector pointing out of cell i into cell N as shown in Figure 3 using Eqn. 3 and Eqn. 4.

6. *areaCalculator*: When given the mesh and a cell index, it is able to calculate the area of the cell using Heron's formula (Eqn. 5).

7. *sCalculatorKFID*: This is a copy-paste of the calculations of $|s|$ from Professor Fidkowski's flux.py function, but adjusted to take in the state for a cell, and an edge-normal. I know that it calculates the three possible wave propagation speeds, $c + \vec{n} \cdot \vec{v}$, $c - \vec{n} \cdot \vec{v}$, $\vec{n} \cdot \vec{v}$, and returns the maximum.

8. *timestepCalculator*: When given the mesh, the local state at cell i, the index i itself, and the CFL number, it returns the local timestep for the cell Δt_i. This is then appended to $\Delta \boldsymbol{t}$.

9. *boundaryForce*: When given the mesh and the state vector, it will return the forces along all boundary edges at time t.

10. *residualCalculator*: This computes the residuals at time t. An in-depth analysis for this can be found in Algorithm 1.

11. *timeIntegration*: This is the primary driving function that marches the solution forward in time by updating the stated and residuals at each timestep. It takes in the mesh, the initial state, the CFL number, the time to terminate the simulation at, and an additional time to record the outputs of the simulation. Upon termination of the function it returns the final state vector $\boldsymbol{u}$, the additional state vector at the other described time, the forces along the inviscid walls, the timesteps, and the L_2 norm of the residuals.

Here is a brief description of the helper functions written by Professor Fidkowski that were given to me that I actually used. There are a few more, but did not use them.

1. *FluxFunction*: When given the two states of two adjacent cells and the outward pointing normal for this edge, it approximately computes the Roe Flux across this boundary. It can only be used on internal edges.

2. *readgri*: When given a .gri filename, it is able to read this file and turn this data into the Mesh Dictionary designed by Professor Fidkowski.

3. *edgehash*: This takes in a list of elements and edges and organizes them to be either internal or boundary edges.

4. *plotmesh*: When given a mesh, a state vector, anda file name, it saves a contour mapping of the states on the mesh.

Here is a brief description of the helper functions used in the post-processing plotting file.

1. *loadFile*: Loads in the data in the given filename as a numpy array.

2. *plotForces*: When given an array of force values for the inviscid walls it plots them and saves the figure to the data's directory.

Data Structures

For this project I chose to avoid complex data structures as I felt that they could be avoided via the use of clever indexing of the results from Professor Fidkowski's mesh reading function and liberal usage of helper functions. In fact, the only unique data structure in the project is the output from the "readmesh" function given to us. This data structure is a Python Dictionary, which is just a clever organization of arrays represented by keywords that can easily be indexed. This allows me to access certain types of information from the mesh when needed, like nodes, or internal edges.

Task 3 - Freestream Condition

Task 3 requires me to run a "Free-stream Condition" test on my code. This condition is where the state is set to $h = 1,\ u = v = 0$ for all cells in either the coarse or the fine mesh. This should result in machine precision residuals at the first timestep, and subsequent timesteps for any and all t. This test was run on the coarse mesh as it makes no difference which mesh it was run on.

The residuals were plotted against iteration number in Figure 10. The fact that it hovers around machine precision validates the simulation because the sum of all residuals under

free-stream condition should always be zero. The pressures are balanced by the walls, and the lack of any perturbations means there will be any fluxes between cells, and result in steady state conditions for all time.

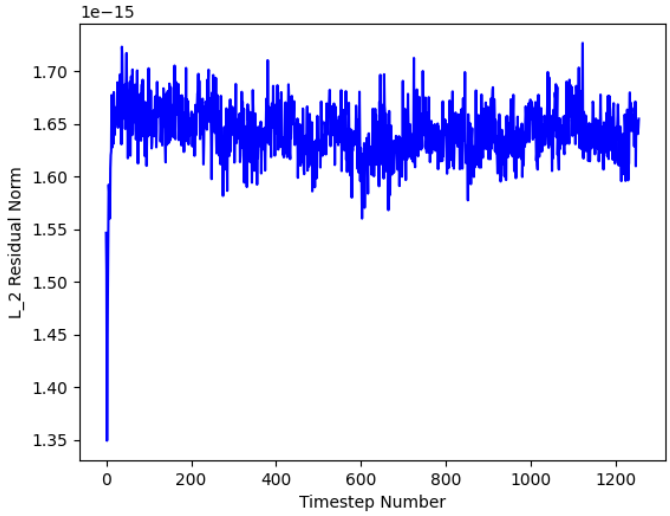

Figure 10: The residuals hover at machine precision for all time, as is expected for a steady state free-stream Condition

Task 4 - State Contour Mappings

Task 4 asked me to implement a way to plot contour maps of the height, x-velocity and y-velocity at the final time, and at about halfway through the simulation for both the coarse and fine meshes. I was successfully able to do so, and the results can be seen in Figures 11 through 22. The final time is at $t = 0.5$ seconds, and the halfway time is approximately 0.25 seconds.

Halfway Time Contour Maps

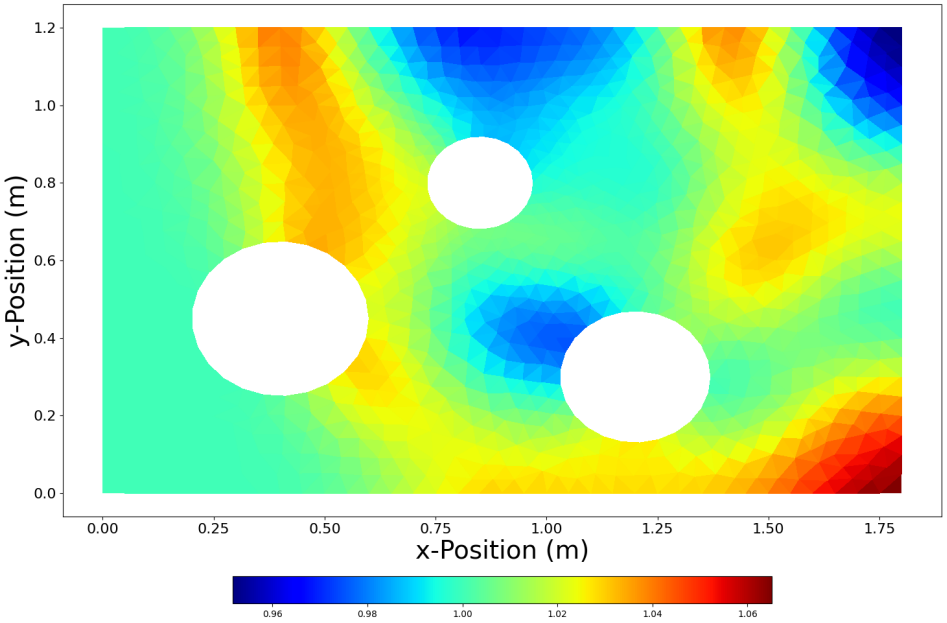

Figure 11: The height on the coarse mesh at $t \approx \frac{t_{max}}{2}$.

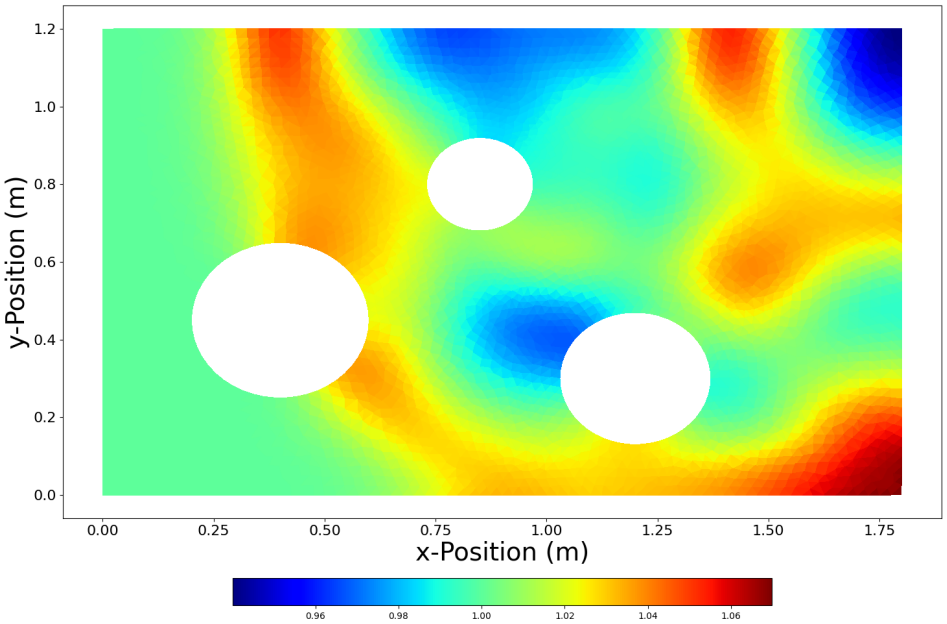

Figure 12: The height on the fine mesh at $t \approx \frac{t_{max}}{2}$.

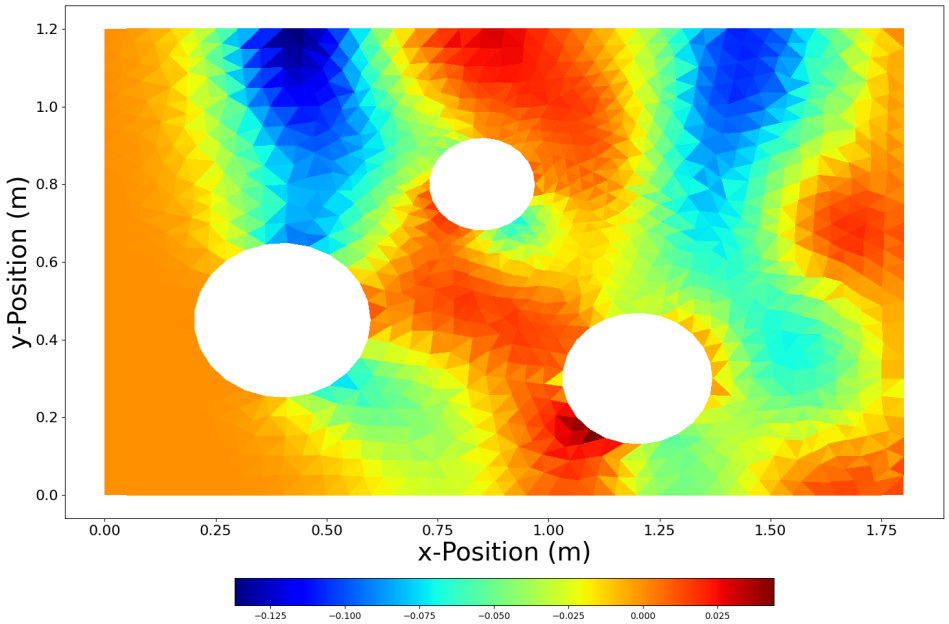

Figure 13: The x-velocity on the coarse mesh at $t \approx \frac{t_{max}}{2}$.

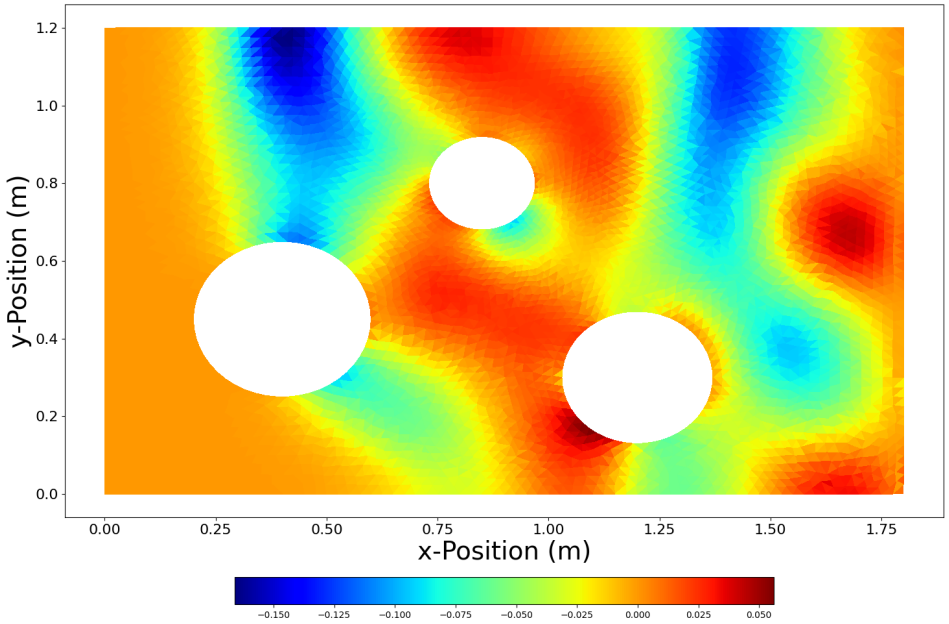

Figure 14: The x-velocity on the fine mesh at $t \approx \frac{t_{max}}{2}$.

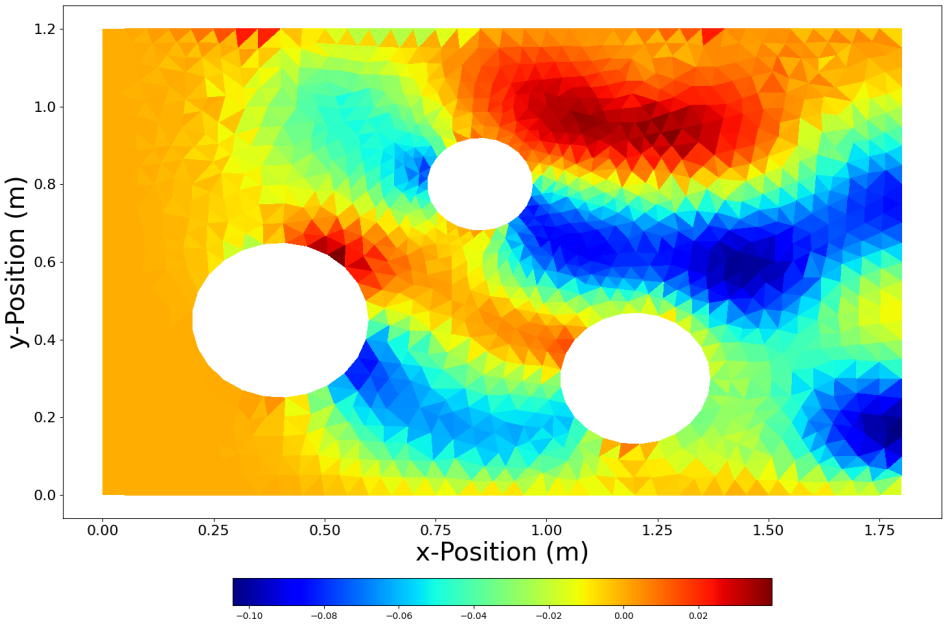

Figure 15: The y-velocity on the coarse mesh at $t \approx \frac{t_{max}}{2}$.

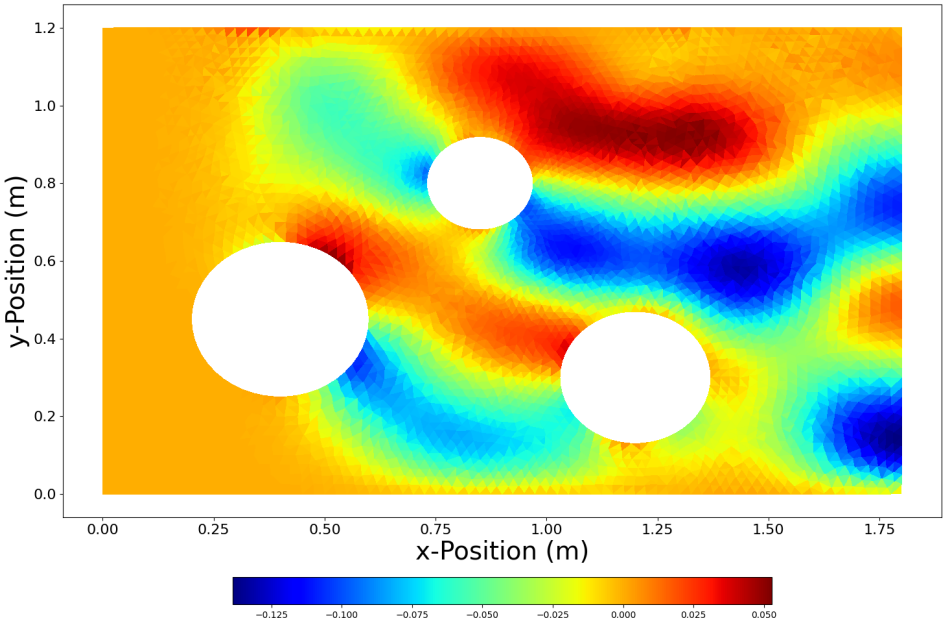

Figure 16: The y-velocity on the fine mesh at $t \approx \frac{t_{max}}{2}$.

Final Time Contour Maps

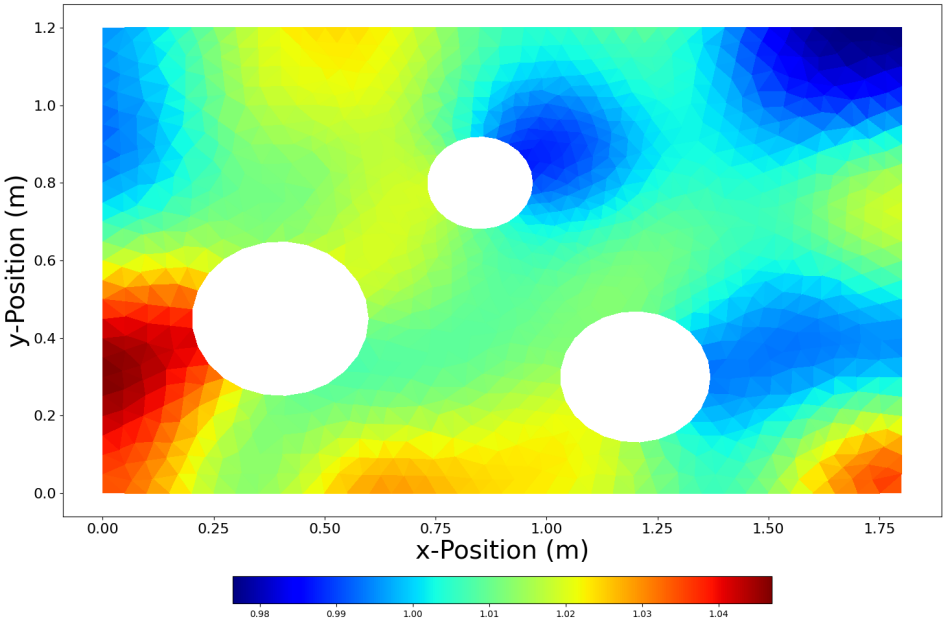

Figure 17: The height on the coarse mesh at $t = t_{max}$.

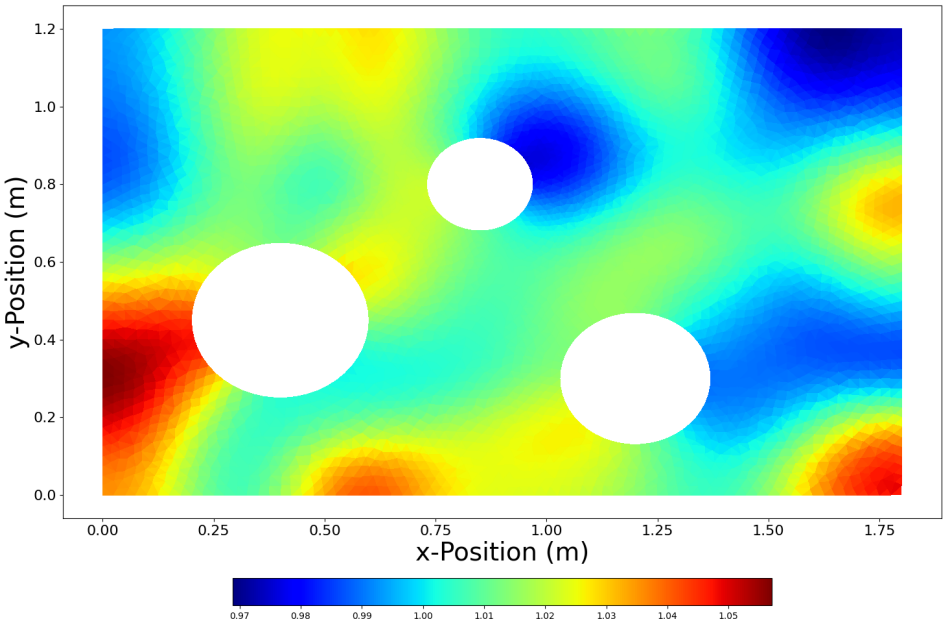

Figure 18: The height on the fine mesh at $t = t_{max}$.

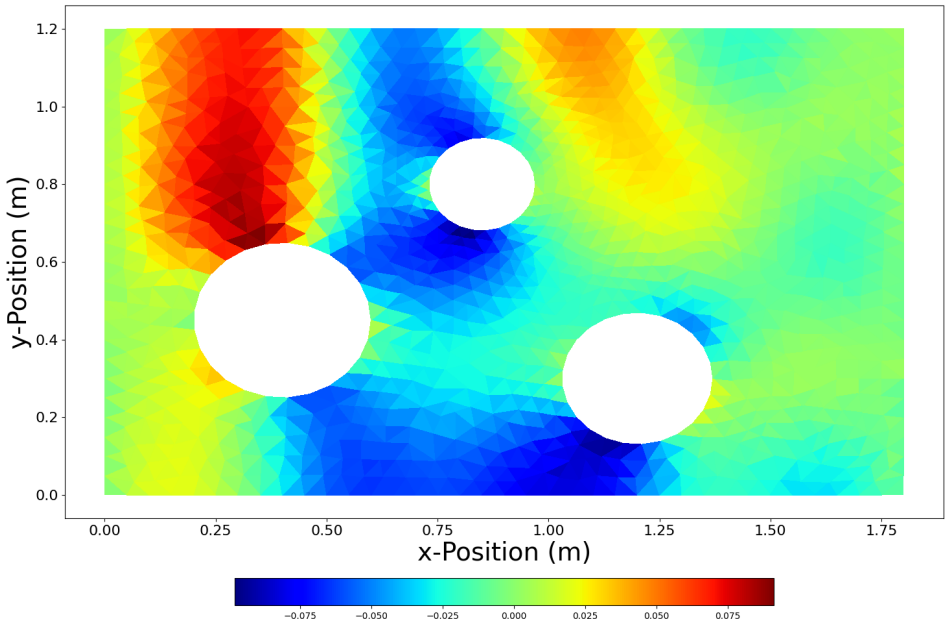

Figure 19: The x-velocity on the coarse mesh at $t = t_{max}$.

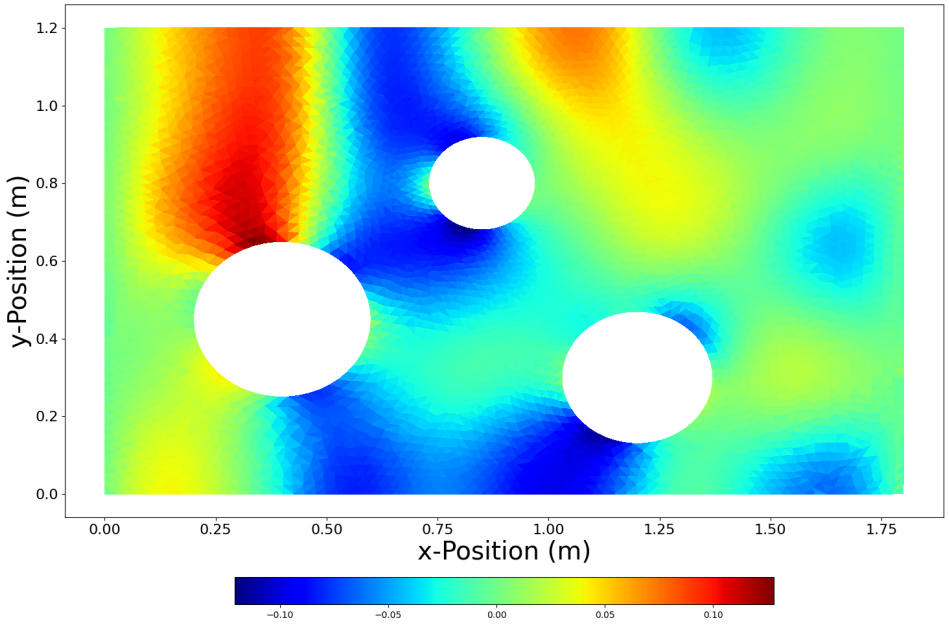

Figure 20: The x-velocity on the fine mesh at $t = t_{max}$.

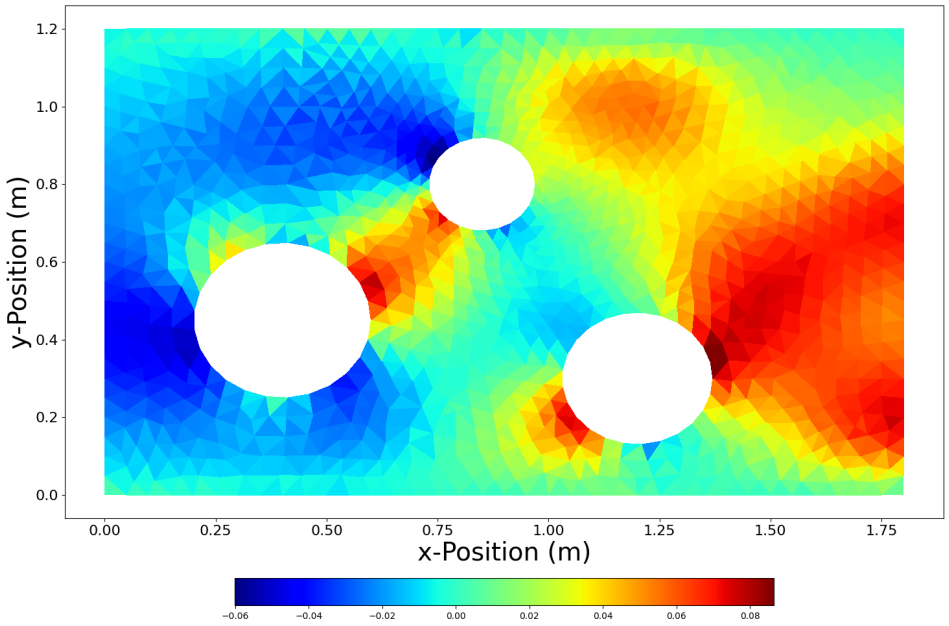

Figure 21: The y-velocity on the coarse mesh at $t = t_{max}$.

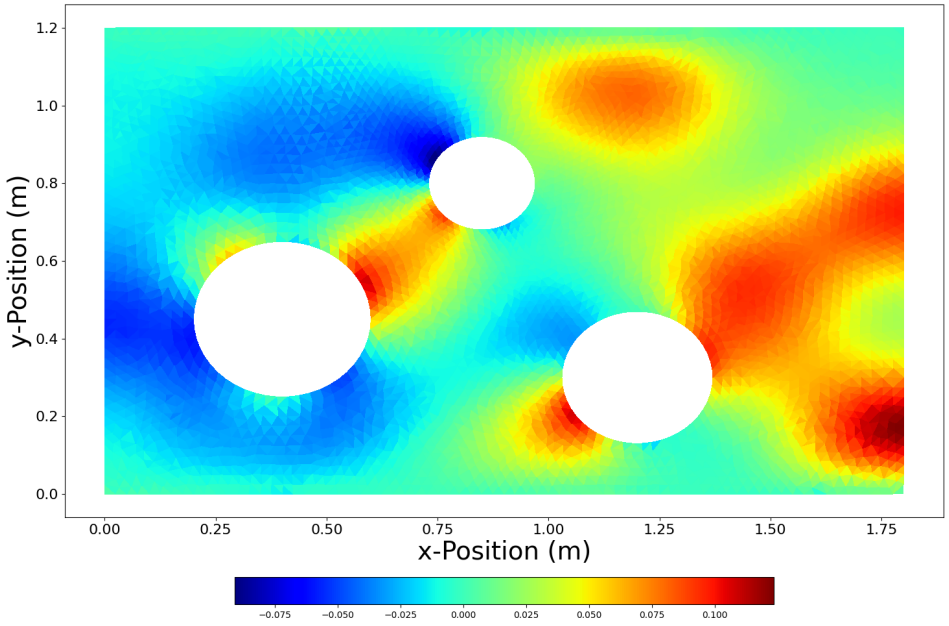

Figure 22: The y-velocity on the fine mesh at $t = t_{max}$.

In all six sets between coarse and fine for any time or state variable, there are two trends between the two meshes. These two trends are: moving from coarse to fine results in smoother gradients and larger extrema.

By introducing more cells as we go from coarse to fine, we can more accurately depict how the state changes over space. This results in smoother gradients of a respective state variable, and a more accurate representation of the fluid in the tank.

When comparing the two meshes, one could argue that the coarse mesh acts like a aggregate version of the fine mesh. This averaging causes cells to not have as large of magnitude of extrema as they the physics would dictate that they have. This is reversed for the fine mesh as it has more cells so it can more accurately model the physics and produce more larger extrema by minimizing this aggregation/smearing effect.

Task 5 - Boundary Edge Forces

Task 5 asked me to implement the force calculations along the boundary edges for the inviscid walls and plot them over time for the two meshes, and for both the x and y directions. I was successfully able to do this, and the results can be seen in Figures 23 and 24 for the x-direction forces, and Figures 25 and 26 for the y-direction forces.

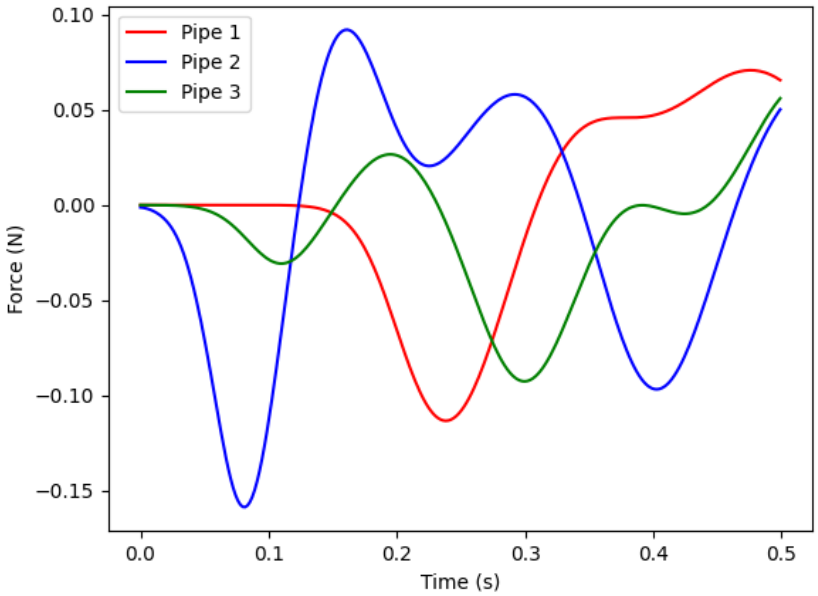

Figure 23: The forces in the x-direction over the 0.5s time period for the coarse mesh.

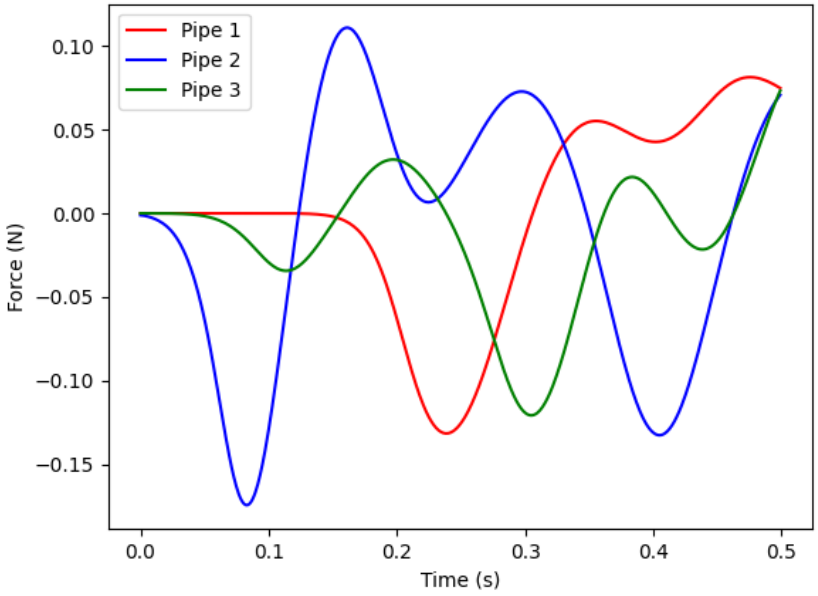

Figure 24: The forces in the x-direction over the 0.5s time period for the fine mesh.

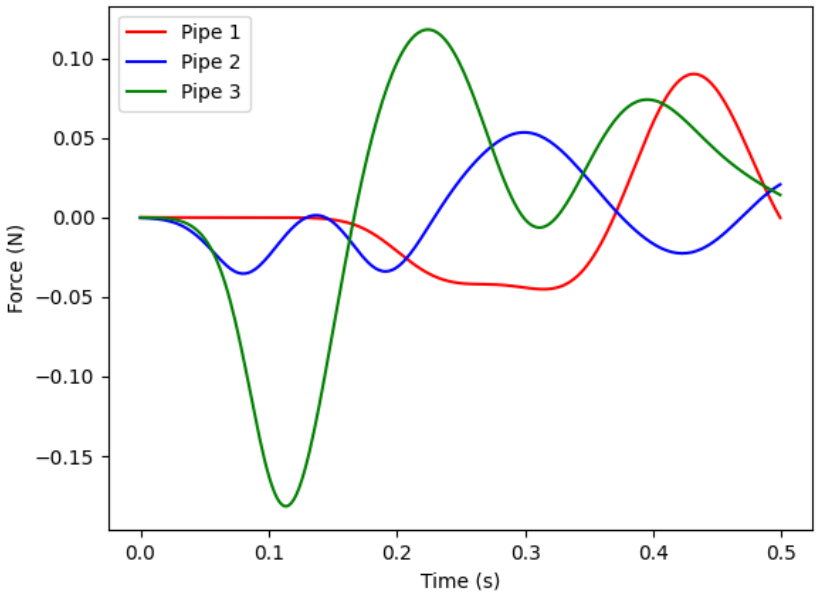

Figure 25: The forces in the y-direction over the 0.5s time period for the coarse mesh.

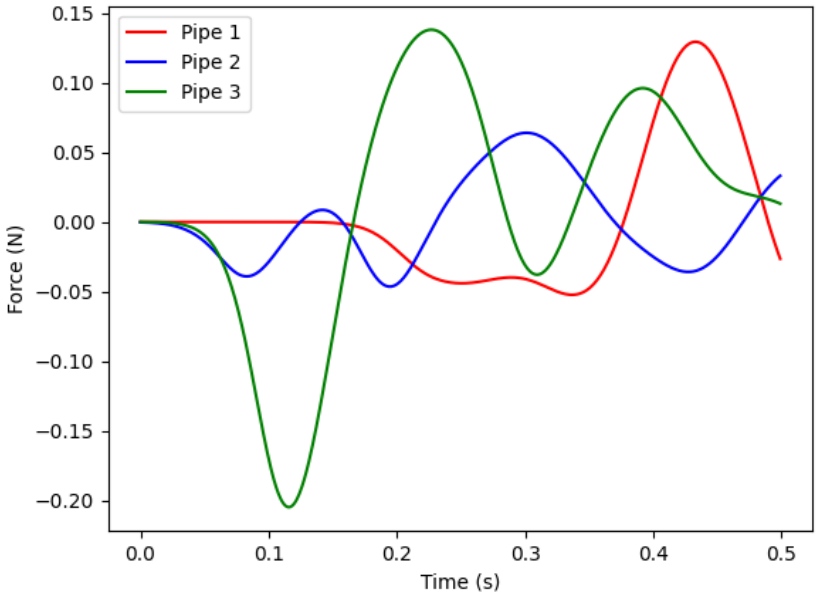

Figure 26: The forces in the y-direction over the 0.5s time period for the fine mesh.

In both meshes the forces on each pipe are similar as the force evolves over time. This is a good thing as it is helps reaffirm the fact that the simulation is working as intended (assuming no systematic errors are present). However, there are two primary differences when looking at the differences in the simulation results.

The first is that the amplitudes are smaller for the coarse mesh. This means that the size of the cells are big enough such that our first-order approximation of what is happening in each cell is not quite capturing the true nature of what is actually happening. Had I used a more accurate finite-volume solver, or had an even more refined mesh, I would likely see additional changes in the extrema of the forces over time.

The second difference I want to highlight is the oscillatory nature of the forces and how they don't line up. The oscillations come from the sloshing of the fuel as it moved back and forth against the pipes over time. In the case of the fine mesh, by having more cells we are able to get a better approximation of what is happening at any given time by knowing what the state is at more points in space. This means we can more accurately model the physics which results in a slight variance when compared to the coarse mesh results. As previously mentioned, the coarse mesh could be viewed like an an aggregated version of the fine mesh, so I should have more smeared results. Smeared results should look something like smoother force functions with less oscillatory behavior, and you can see exactly that when looking at both the x-direction and y-direction force plots for the coarse mesh when compared to the fine mesh.

REFERENCES

[1] Krzysztof Fidkowski. *Computational Fluid Dynamics*. University of Michigan, Aug. 2021, pp. 139–140.

[2] Krzysztof Fidkowski. *Project 2: 2D Shallow Water Equations*. Oct. 2021.