MCG 4139 Final Project

Modeling of a Ferro-Fluid in a Magnetic Field

Student number: 300152010
Name: Matthew Kordasiewicz

Start date: April 27th, 2023

Department of Mechanical Engineering
Faculty of Engineering

Abstract

The goal of this project is to demonstrate the effect of an external magnetic field on a ferro-fluid and to demonstrate the principles of computational and numerical methods for fluid modeling. To accomplish this a 2D unstructured mesh was generated with three sub-domains, the first being a background domain, another for the fluid domain and the final one for the wires. The magnetic field was then generated using the Biot-Savart equation from the centroid of each wire. A finite volume solver was then used for the numerical approximation of the shallow water equations, with a source term to account for the acceleration of the fluid by the external magnetic field.

Table of Figures

Index of Tables

Table of Contents

Introduction

This project is being done to understand how the finite volume solver works, and to simulated the flow of a ferro-fluid in a magnetic field, with the goal of generating a positive flow. This was accomplished by generating a magnetic field and incorporating a source term into the shallow water PDEs to provide acceleration. The unstructured solver from lab 9 was modified to achieve this goal.

Continuous model

For this project only the shallow water equations were solved, however the code is templated in such a way that any set of PDEs can be solved given they follow the same format. The shallow water equations are a set of 3 PDEs with the first equation solving for the height of the fluid, this equation effectively tracks the pressure using the height of the water column in a given cell. The other two equations are momentum equations which are used to track the velocity of the fluid.

$$\frac{\partial}{\partial t}h+\frac{\partial}{\partial x}F_{h_x}+\frac{\partial}{\partial y}F_{h_y}=0$$

$$\frac{\partial}{\partial t}hu_x+\frac{\partial}{\partial x}F_{m_{xx}}+\frac{\partial}{\partial y}F_{m_{xy}}=S_x$$

$$\frac{\partial}{\partial t}hu_y+\frac{\partial}{\partial x}F_{m_{yx}}+\frac{\partial}{\partial y}F_{m_{yy}}=S_y$$

Where the fluxes are:

$$F_{h_x}=hu_x=[\frac{m^2}{s}] \qquad F_{h_y}=hu_y=[\frac{m^2}{s}]$$

$$F_{m_{xx}}=hu_x^2+\frac{gh^2}{2}=[\frac{m^3}{s^2}] \qquad F_{m_{xy}}=hu_xu_y=[\frac{m^3}{s^2}]$$

$$F_{m_{yx}}=hu_xu_y=[\frac{m^3}{s^2}] \qquad F_{m_{yy}}=hu_y^2+\frac{gh^2}{2}=[\frac{m^3}{s^2}]$$

The source terms are:

$$S_x=-gh\frac{\partial}{\partial x}b+ha_x=[\frac{m^3}{s^2}] \quad S_y=-gh\frac{\partial}{\partial y}b+ha_y=[\frac{m^3}{s^2}]$$

For the this simulation the bottom is set to zero thus the source terms are only a function of the height and acceleration terms. These acceleration terms are defined as:

$$a_x=\frac{Force_x}{mass}=[\frac{m}{s^2}] \qquad a_y=\frac{Force_y}{mass}=[\frac{m}{s^2}]$$

The mass is calculated from the density and volume of each cell. Then the force terms are defined as

$$F=\nabla(m_{sc}\odot b) \qquad m_{sc}=V_{core}*M_s$$

m_{sc} is the effective magnetic moment approximation for single crystal iron nanoparticles. b is the magnetic flux density calculated using the Biot-Savart equation for an infinite vertical wire. V_{core} is the volume of the nanoparticles and is defined as a fraction of the cell volume. M_s is the saturation magnetization, a constant from [1] of 375 kA/m (larger of two experimental values). Using these approximations a simple proportional acceleration term can be expressed as.

$$a_x = \frac{\gamma M_s b_x}{\rho} \quad a_y = \frac{\gamma M_s b_y}{\rho}$$

where gamma defines the fraction of nanoparticles in the volume (set to 10%) and rho defines the density (set to $1000 \frac{kg}{m^3}$).

The Biot-Savart equation: $b = \frac{\mu_o I_{enc}}{2\pi r}$ was used for generating the magnetic flux density, where r is the radius from the centroid of the wire. This gives a magnitude in radial coordinates which was transformed into x and y components.

Discrete Model

Explicit Euler $\bar{U}^{n+1} = \bar{U}^n + \Delta t \left(\frac{d\bar{U}^n}{dt} + S\right)$ was used for time marching.

With $\frac{d\bar{U}}{dt}$ being approximated using local Lax-Friedrichs. The source term is calculated using the equations explained in the continuous model, from the PDE the source term should be a part of $\frac{d\bar{U}}{dt}$ but, in the code this is calculated per edge whereas the source is calculated per cell, so these were combined in a later step with the time marching update per cell.

With a finite volume solver the fluxes are calculated across the edges, to do this the solution vector of a given cell must first be rotated so that the flux can be calculated on a normal vector perpendicular to the edge of interest. Thus the flux is always done in a rotated x axis. With this simplification the lambda max for local Lax-Friedrichs can also be done in this rotated frame using only the x velocity since the fluid can only travel perpendicular to the edge. The length of the edge acts as the weight for the flux and the area used in place of the volume (2D volume).

$$\frac{d\bar{U}}{dt} = \frac{-l}{V_i} \sum \widetilde{F}(U_l, U_r, \hat{n}) \qquad \widetilde{F}' = \frac{F_l' + F_r'}{2} - \frac{|\lambda_{max}|}{2}(U_r' + U_l')$$

Here the prime denotes that this is being calculated in the rotated frame of reference, the solution vector is first rotated, flux calculated, then the flux is rotated back to be used in the calculation of $\frac{d\bar{U}}{dt}$. F_l' and F_r' are calculated using the x direction flux equation from the continuous model.

$$\lambda_{max} = u_{rot_x} + \sqrt{g * h} \qquad \Delta t = CFL * \frac{\sqrt{V_i}}{|\lambda_{max}|}$$

Geometry

The fluid domain is 0.4 m by 0.05m, with a background domain of 0.45m by 0.11m, with 34 circles representing the wire coils, this geometry can be seen in figure 1 below. 100 represents the background, 200 the fluid domain and 300 the coil domain. In addition to these sub-domains the edges of the fluid domain have an assigned physical group that defines its behavior in the solver, the edge types are shown in figure 2 below. Only the edge types for the fluid domain are used in the solver, the wire edges are a remnant of an old method for picking up the wires.

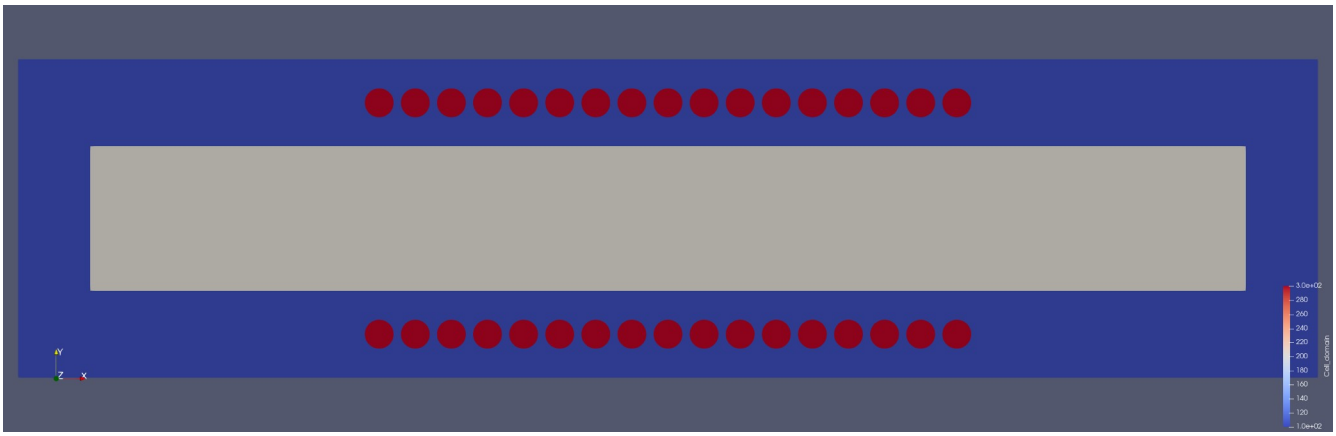

Figure 1: Cell domain types

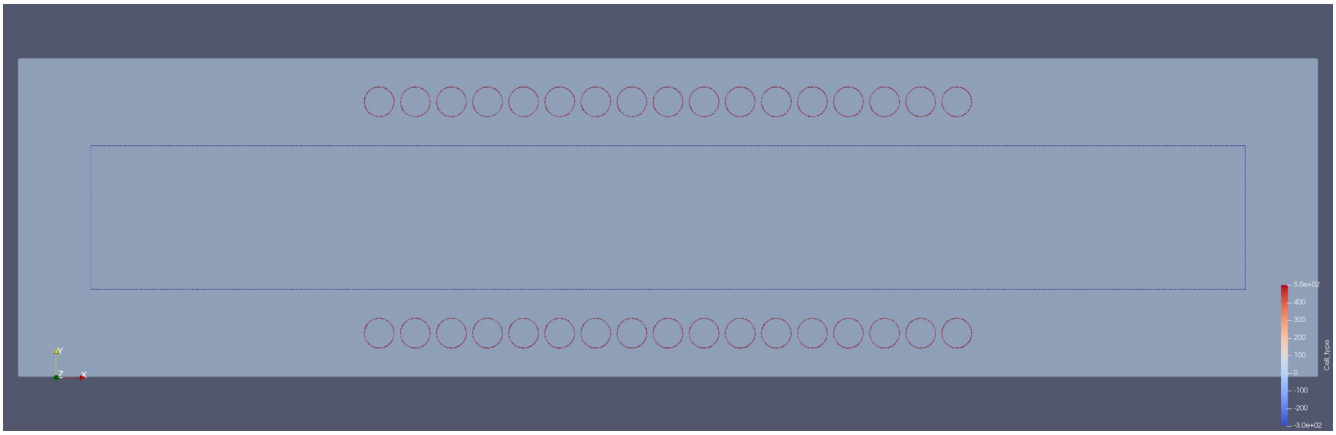

Figure 2: Cell edge types

The code picks up wire cells, groups them, finds their centroids and then generates the magnetic field. The wires are grouped using the wire diameter, first all the wire cells are found and then the distance is calculated between the first of each wire to ensure that it is less than the wire diameter. The magnetic field components can be seen in figures 3 and 4 below. The top and bottom edges of the domain are treated as reflections, then the left and right edges are called inlets and outlets and are setup to mirror the height and the x velocity from the domain. This inlet and outlet setup appear to have led to some issues that will be discussed later.

The magnetic field, specifically the y component is key for understanding the behavior seen later on and brought a lot of trouble in the mesh study. Ultimately it was found that the final time had to be rather small because with larger final times the solution has some issues.

The images shown in figures 3 and 4 are using the 0.0005 characteristic length mesh which will be explained in more detail in the mesh study. The magnetic field was generated to provide a positive x component to achieve a positive flow velocity in the fluid. The domain will split into two regions for terminology, the left half called the “convergent” region and the right is the “divergent” region, this is based on the direction of the y component magnetic field in these regions.

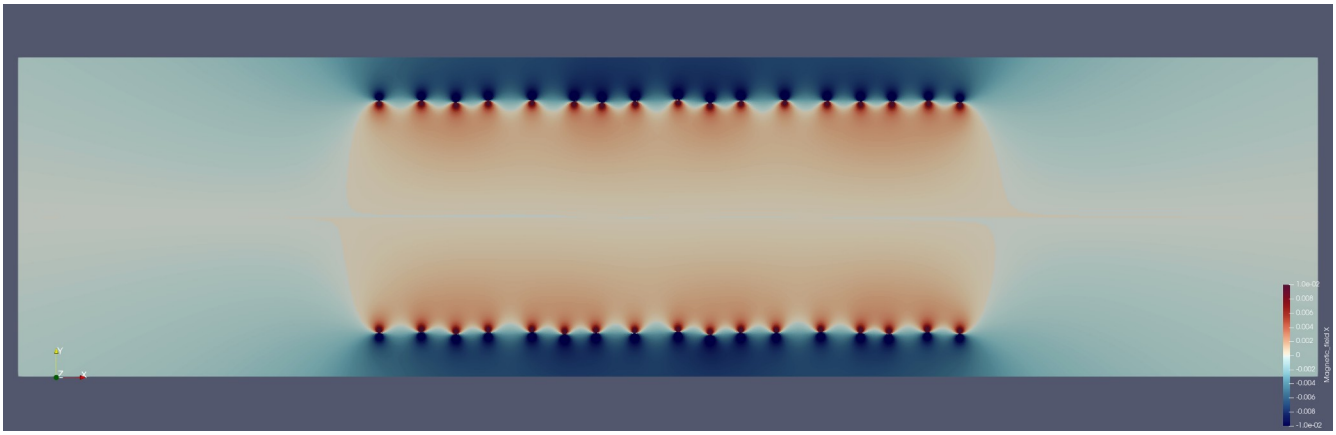

Figure 3: Magnetic Field X component

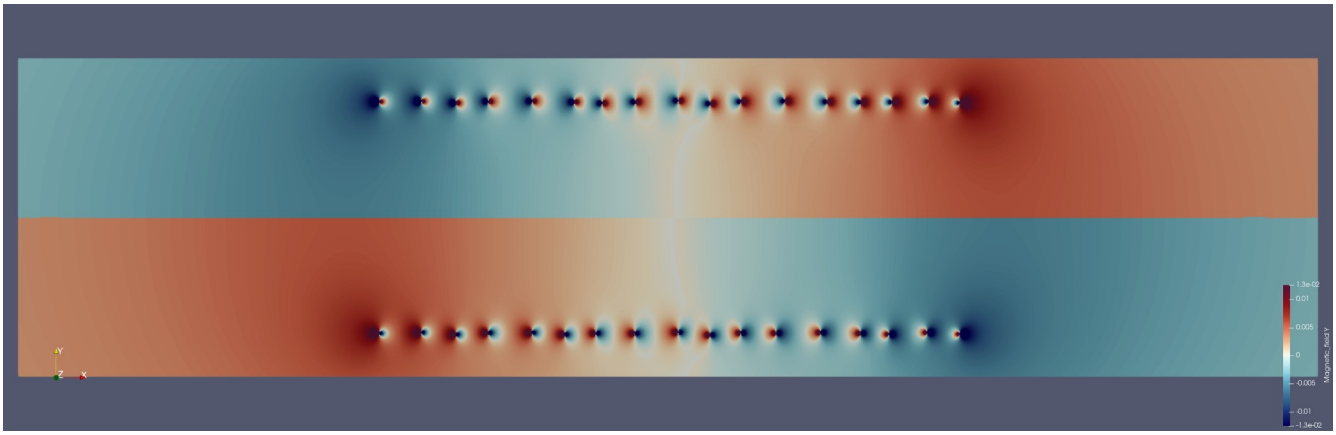

Figure 4: Magnetic Field Y component

In figure 4 the convergent and divergent regions can be seen very clearly, with a split down the middle dividing them. The red indicates a positive component and blue represents a negative component.

Resolution Study

For the resolution studies a constant initial height of one was applied to the entire domain. The top edges of the fluid domain are treated as reflections (labeled no_slip_wall in code for other PDEs). Then the left edge is treated as an inlet and the right edge is treated as an outlet. Final time of 0.025 s.

Mesh Study

For the mesh study 13 different meshes were used, to size these the characteristic length in the “domain.geo” file was modified using the following values: 0.05, 0.025, 0.0175, 0.01, 0.0075, 0.005, 0.0025, 0.00175, 0.001, 0.00075, 0.0005, 0.00025, 0.000175. Using a number CFL of 0.25.

The first and simplest analysis is a visual comparison, images were generated for the final time-step for all three primitive values, a brief comparison can be seen below in figures 5 through 8.

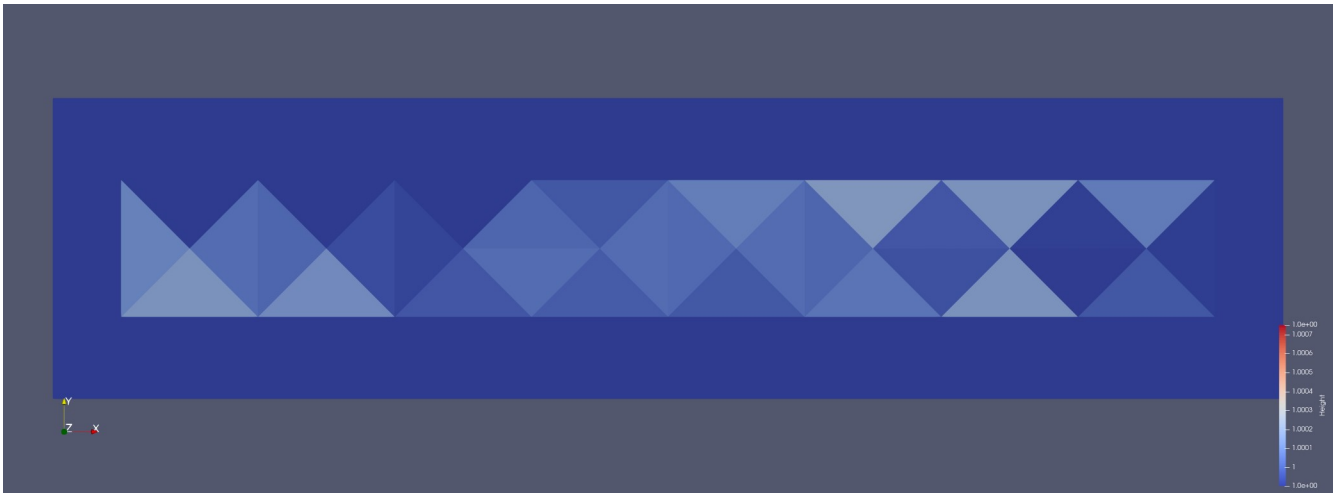

Figure 5: CharLen 0.05 Height

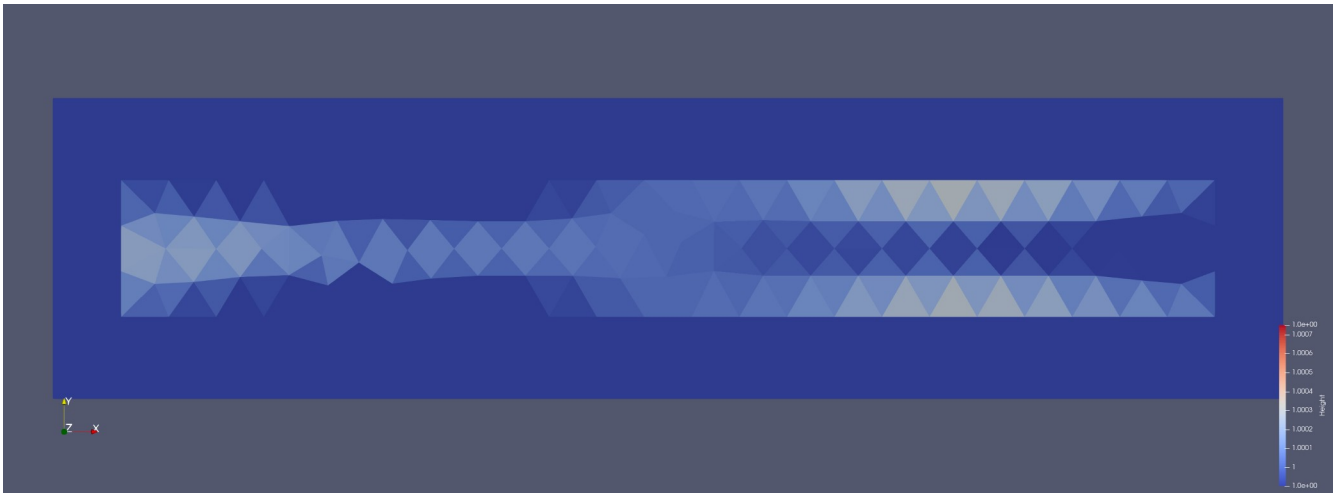

Figure 6: CharLen 0.0175 Height

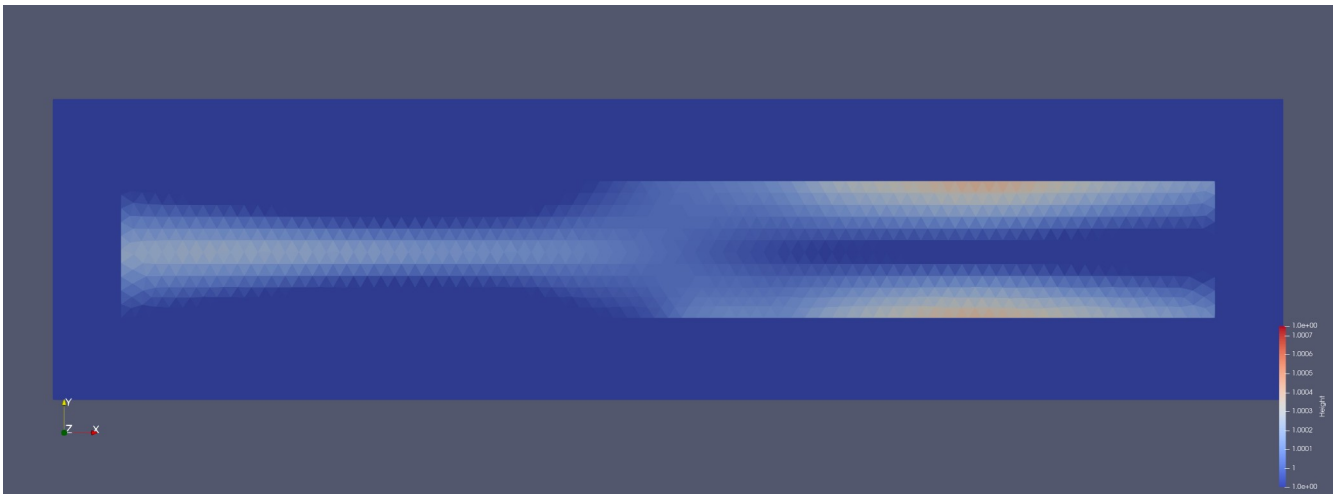

Figure 7: CharLen 0.005 Height

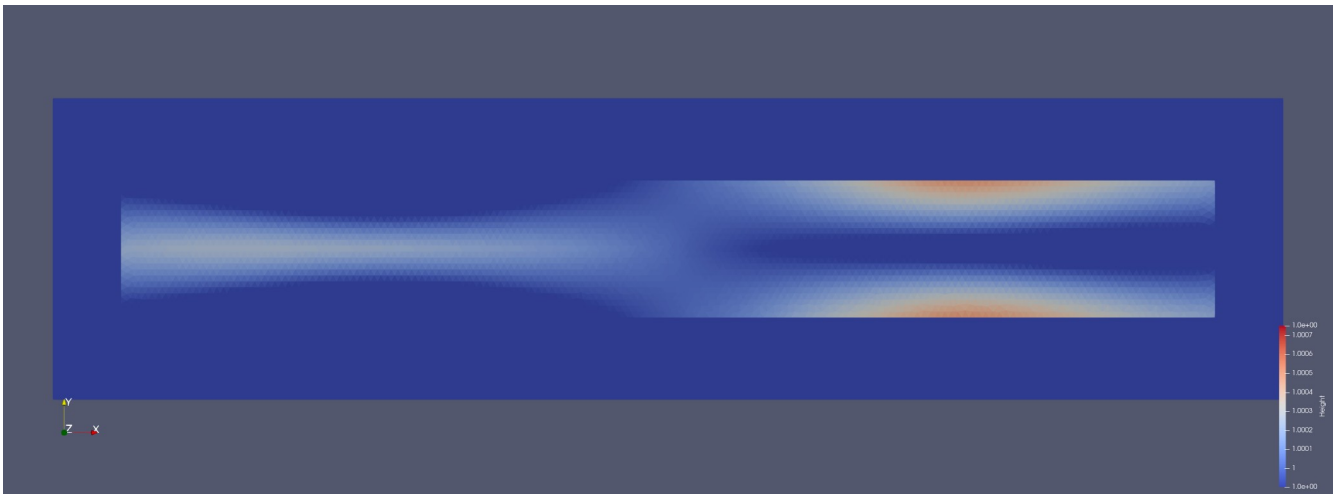

Figure 8: CharLen 0.0025 Height

From these figures it is clear that meshes using characteristic lengths of 0.05 to 0.005/0.0025 are under-resolved. The characteristic length of 0.0025 appears to be the transition point where the solution is visually very similar. Going beyond this the shape doesn't drastically change, only the local values change.

Only once the characteristic length is below 0.0025 does it appear that the solution is properly coming through, inklings of it can be seen earlier as the 0.0175 mesh but these are still under-resolved. To see images for all the characteristic lengths and the other variables see Appendix A.

The second method of analysis is a comparison of the minimum and maximum values for each primitive value in the solution vector. This analysis can be seen below in tables 1 and 2. Table 1 shows the values and table 2 shows the difference between rows. This analysis is inconclusive, the differences bounce around and never consistently decrease.

Table 1: Mesh Study Min Max Values

CharLen	Max Height	Min Height	Min Vx	Max Vx	Min Vy	Max Vy
0.05	0.999958	1.000172	0.000183	0.000705	-0.000533	0.000480
0.025	1.000125	1.000189	-0.000071	0.000656	0.000169	0.000557
0.0175	1.000018	1.000277	0.000553	0.000653	-0.000041	0.000462
0.01	1.000043	1.000326	-0.000159	0.000717	-0.000118	0.000224
0.0075	1.000147	1.000408	-0.000066	0.000775	0.000109	0.000266
0.005	0.999950	1.000407	-0.000133	0.000845	0.000074	0.000153
0.0025	0.999947	1.000496	-0.000166	0.000680	0.000009	0.000081
0.00175	1.000124	1.000529	-0.000095	0.000664	-0.000028	0.000130
0.001	1.000088	1.000557	-0.000288	0.000678	0.000026	0.000205
0.00075	0.999771	1.000579	-0.000145	0.000726	-0.000003	0.000219
0.0005	0.999968	1.000575	-0.000195	0.000664	-0.000023	0.000237
0.00025	1.000148	1.000592	-0.000416	0.000794	0.000011	0.000231
0.000175	0.999907	1.000610	-0.000361	0.000698	0.000017	0.000261

Table 2: Mesh Study Min Max Differences

CharLen	Max Height	Min Height	Min Vx	Max Vx	Min Vy	Max Vy
0.05	N/A	N/A	N/A	N/A	N/A	N/A
0.025	0.000167	1.617071e-05	-0.000253	-0.000049	0.000702	0.000077
0.0175	-0.000108	8.822941e-05	0.000623	-0.000003	-0.000210	-0.000095
0.01	0.000025	4.898590e-05	-0.000711	0.000063	-0.000077	-0.000238
0.0075	0.000104	8.191994e-05	0.000093	0.000058	0.000227	0.000042
0.005	-0.000197	3.581101e-07	-0.000067	0.000070	-0.000036	-0.000112
0.0025	-0.000003	8.823477e-05	-0.000033	-0.000165	-0.000065	-0.000072
0.00175	0.000177	3.371693e-05	0.000071	-0.000016	-0.000036	0.000049
0.001	-0.000036	2.730336e-05	-0.000193	0.000014	0.000053	0.000075
0.00075	-0.000317	2.201450e-05	0.000142	0.000048	-0.000028	0.000014
0.0005	0.000196	-3.942848e-06	-0.000050	-0.000062	-0.000020	0.000018
0.00025	0.000180	1.697995e-05	-0.000221	0.000130	0.000034	-0.000006
0.000175	-0.000241	1.791092e-05	0.000055	-0.000097	0.000006	0.000030

The third method is a more complicated scheme involving interpolation, since the meshes are all different the values cannot directly be compared. For this reason each of the solution vectors was interpolated onto a common grid so that these could directly be compared (1250 by 1250 nodes). Several different interpolation schemes were tried but the simplest and cheapest was chosen, this has its limitations in that the grid has the same number of points in both x and y axis so there is some loss in the x axis (I tried a better scheme that used and irregular grid but it ate up all 128 GB of my ram, took a very very long time and never finished). With the interpolated vectors the error was found between subsequent meshes, then the sum of the absolute values was taken to reduce the error vector to a single number for each variable. This was also normalized by the change in the number of cells to get a relative performance metric. This analysis below in table 3 shows a clear trend.

Table 3: Mesh Study Interpolated Differences

CharLen	Height error	Vx error	Vy error
0.05 vs. 0.025	2.871125	3.231844	8.249233
0.025 vs. 0.0175	0.523408	1.185493	2.011303
0.0175 vs. 0.01	0.139165	0.351731	0.437897
0.01 vs. 0.0075	0.112927	0.293304	0.219977
0.0075 vs. 0.005	0.028970	0.141943	0.089518
0.005 vs. 0.0025	0.008483	0.021047	0.018638
0.0025 vs. 0.00175	0.002371	0.003355	0.006441
0.00175 vs. 0.001	0.000693	0.001346	0.001653
0.001 vs. 0.00075	0.000366	0.001385	0.000895
0.00075 vs. 0.0005	0.000141	0.000381	0.000319
0.0005 vs. 0.00025	0.000020	0.000147	0.000086
0.00025 vs. 0.000175	0.000011	0.000107	0.000048

From all this analysis it is clear that a characteristic length of less than 1mm is necessary to be close to mesh independence, however, due to the complex nature of the flow that will be explored later. The solution might need even more of cells to truly be mesh independent. Below in figures 9 and 10 two examples of the interpolated solutions are shown. For the coarse meshes the interpolation does not work nearly as well as for the fine meshes which is going to be another source of error in the comparison. However, for the finer meshes this problem disappears, which is the region of interest, thus this source of error should be minor and the analysis still valid.

*Note: Upon review this analysis should have been done slightly differently mathematically (log[err1/err0]/log[size_mesh1/size_mesh0]), but the results still show the correct trend.

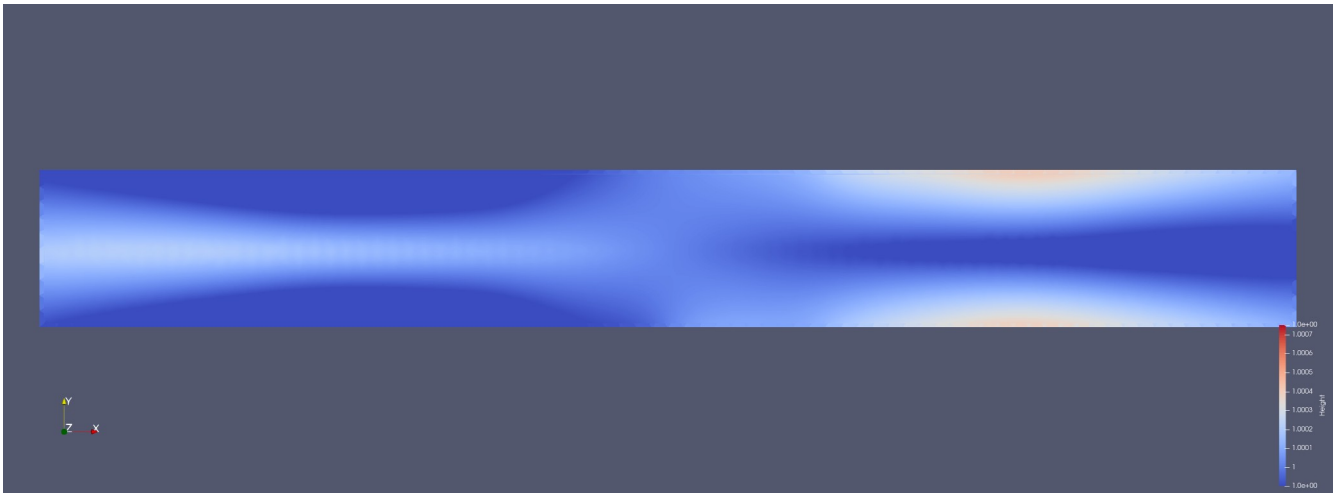

Figure 9: CharLen 0.005 Interpolated Height

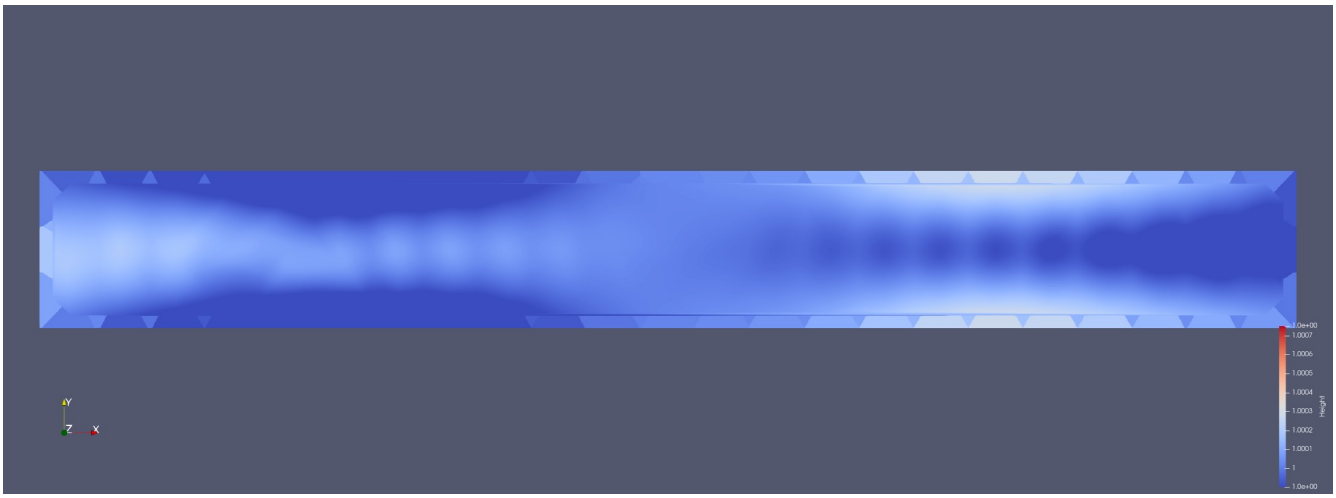

Figure 10: CharLen 0.0175 Interpolated Height

The error vector can be visualized by exporting it to a VTK file and then saving images in ParaView. Not all the error vectors were visualized in this report but all of the VTKs can be found in the attached folder, see “Project_submission/ReadMe.txt” for details.

Notes: First between different images the scales changes by at least one order of magnitude.
Second these error vector values were normalized with the change in the number of cells, which is why the values are so small. Here only the height for the two finest meshes will be compared, 2 other error vectors with the other variables can be found in Appendix A.

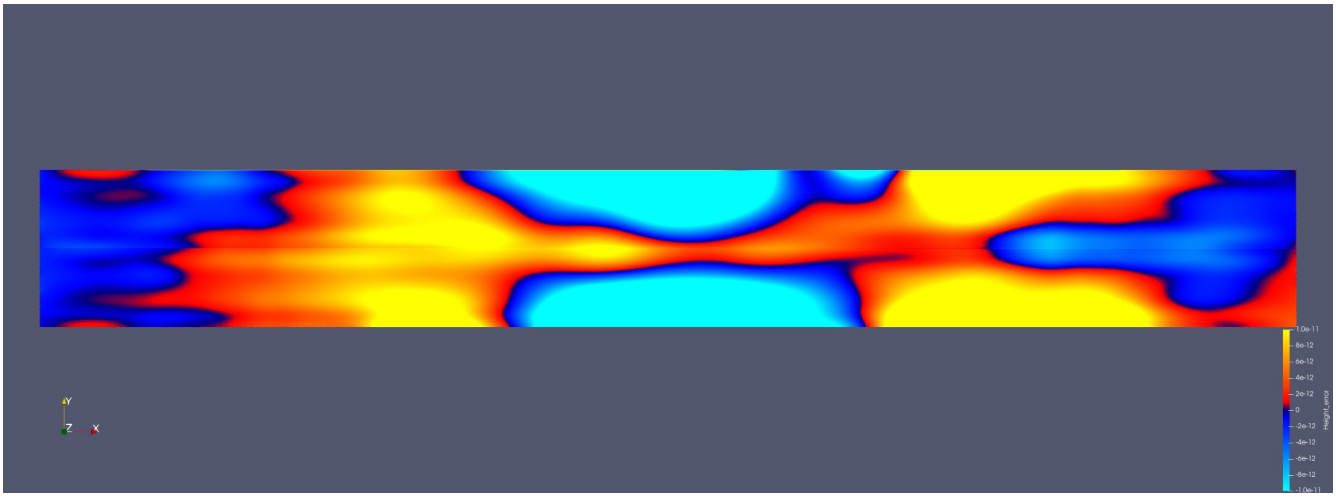

Figure 11: CharLens 0.00025 vs. 0.000175 Height error

The image in figure 11 shows that the error is generally occurring where there is large flow. In the middle of the convergent region there is a large amount of error, along the top and bottom edges of the divergent region there is a large amount of error, and there is also a large amount of both positive and negative error around the expansion from the convergent to the divergent region. The reasoning for why this occurs will become more clear in the results section as the behavior of the flow is explored in more depth, in short this has to do with the y velocity oscillating between positive and negative.

Time Study

For the time study six different CFL numbers were used: 1.01, 0.99, 0.9, 0.75, 0.5, 0.25. For this study the mesh with a characteristic length of 0.0005 was used since it was deemed fine enough and wouldn't take several hours to run all the cases. Since all of the time study cases used the same mesh these can be directly compared but, due to time constraints the interpolation analysis from the mesh study was used. The CFL numbers 1.01, 0.99, 0.9 will not be analyzed since there were unstable and resulted in all NANs. Thus it is immediately known that the CFL number has to less than 0.9. This reduced CFL number is likely due the fast localized flows, particularly in the y direction which will be discussed later in the results section.

Table 4: Time Study Interpolated Differences

CFL	Height error	Vx error	Vy error
0.75 vs. 0.5	206.96417	1946.994932	1339.694239
0.5 vs. 0.25	2.82339	0.780513	6.138498

From table 4 it is clear that the CFL number needs to be below 0.75, around 0.5 the solution begins to converge and 0.25 which was used to the mesh study is a good CFL number.

Briefly comparing the location of the error in the height also reviles that the CFL 0.75 case is unstable as well, however it remained bounded on this time-scale. In the 0.75 vs 0.5 CFL case there is now rhyme or reason to the error, it appears to be completely random. Whereas the 0.5 vs 0.25 CFL case is coherent and shows a very clear pattern, and the error is orders of magnitude smaller.

Figure 12: CFLs 0.75 vs. 0.5 Height error

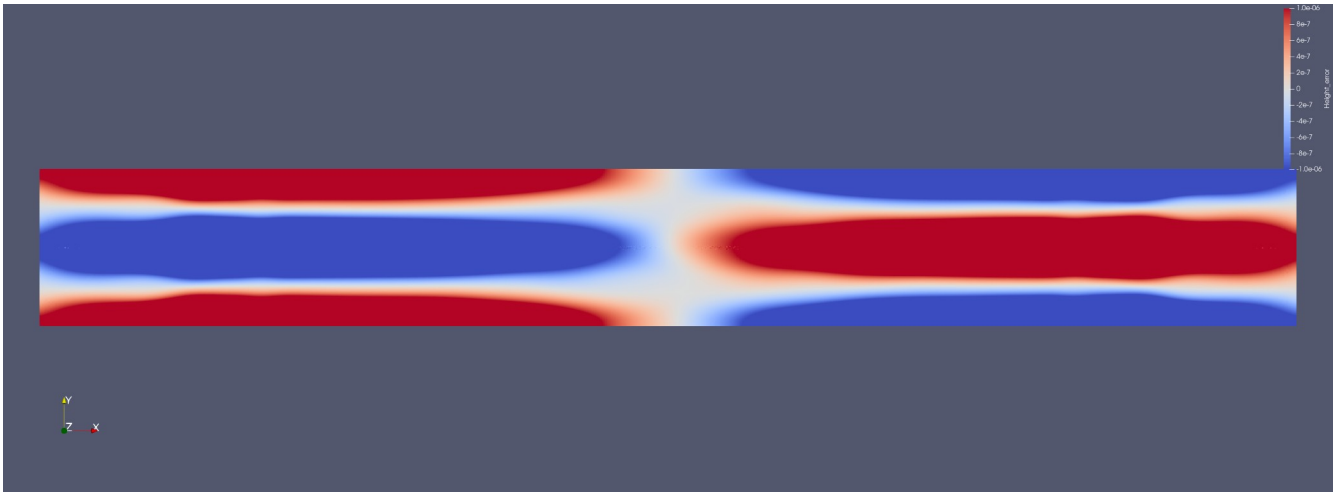

Figure 13: CFLs 0.5 vs. 0.25 Height error

Results

In this section the details of the finest mesh study will be explored further, some have been brought up already but will be discussed in greater detail here. This solution was ran out to 0.025 seconds with a CFL number of 0.25 which we now know to be safe from the time study.

The first discussion point is the height variable, a movie showing the development of this variable can be found in the submission files. The height variable is the simplest of the three and can be fairly simply be understood with the final time-step image. Seen below in figure 14, the left side of the domain is convergent with the fluid being squeezed together due to the convergent y component magnetic field, and in the left region the fluid is pushed apart due to the divergent y component magnetic field. This flow develops fairly steadily but there is some interesting pulsing which can be seen in the movie file (again found in submission files). This pulsing is most likely due to the oscillating y component velocity which will be discussed soon.

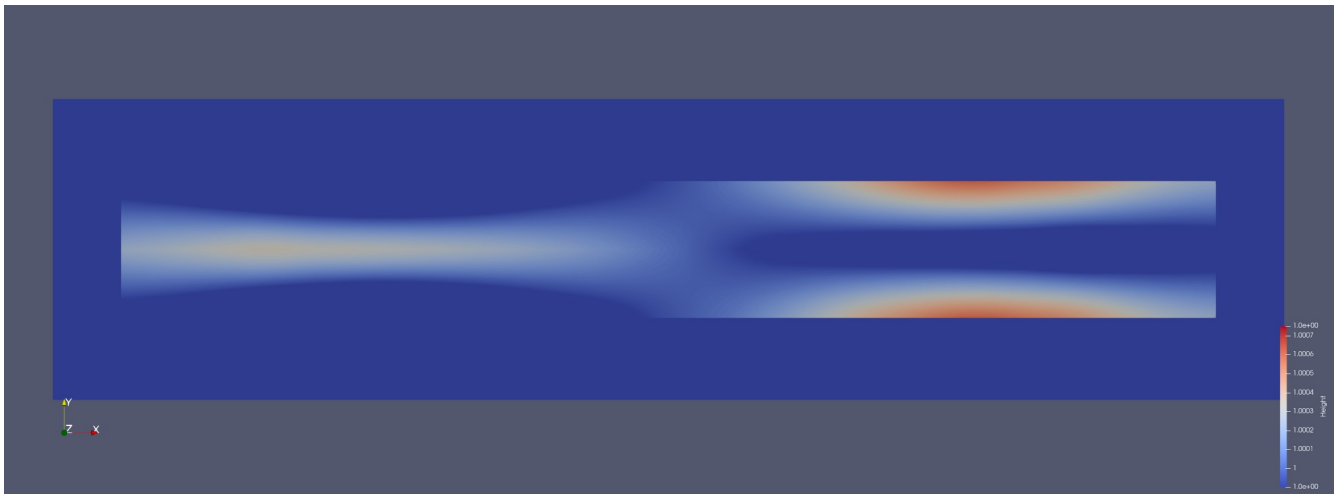

Figure 14: CharLen 0.000175 Height

The next point is the x component velocity, this was the main variable of interest. The goal of this project was to see how the flow develops in a magnetic field with the intention of generating a positive flow. This was achieved to some degree, mostly between the coils as can be seen in figure 15 below. In between the coils there is a positive velocity, but around this there is a negative velocity. This at first doesn't really make sense, but upon further analysis it is likely due to the oscillating y component velocity and/or due to the inlet/outlet condition which allows for a feedback loop of the velocity. There are also some regions around the corners of the domain that develop with a positive velocity and in between a negative velocity. These edge effects once again may be due to the oscillating y component velocity. Similar to the height there is a movie "Vx" showing the x component velocity development, this does not exhibit the same pulsing but is still interesting to watch develop.

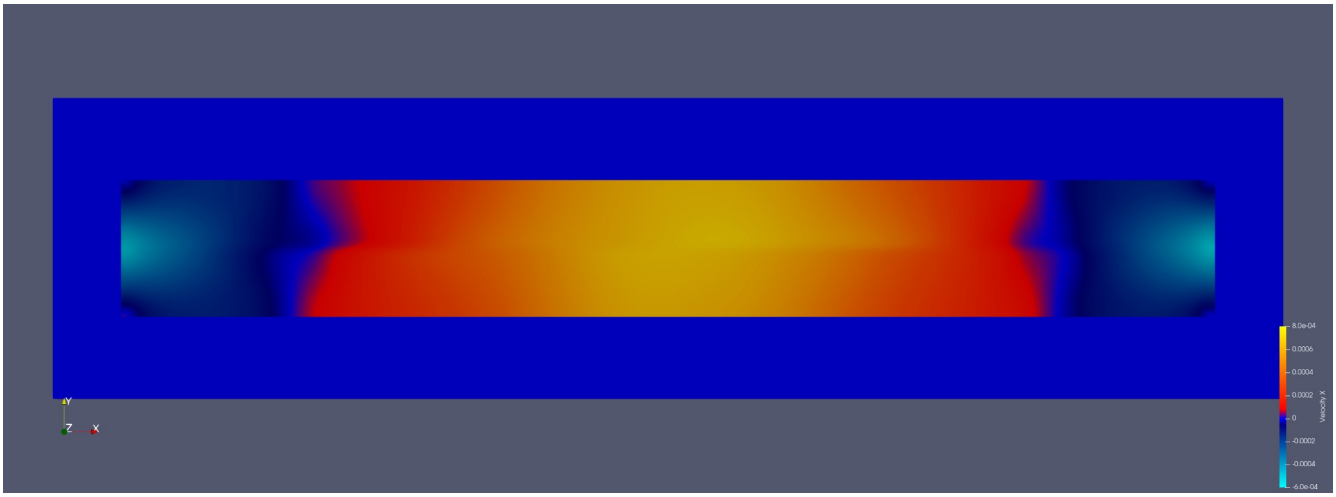

Figure 15: CharLen 0.000175 Vx

Most interesting of all is the y component velocity (one I did not even consider to be important at first), from these simulations it has become clear it is very important. For this velocity it is useful the reader views the attached “Vy” movie in the “movies/Results” folder. Shortly after t = 0 the expected velocity field develops, convergent on the left (top is negative velocity and bottom is positive) and divergent on the right (top is positive and bottom is negative). These velocities increase for a short while until eventually they begin to dissipate, and ultimately swap places (left half becomes divergent right half becomes convergent). To demonstrate this four images were taken at time-steps: 0.00625 s, 0.0125 s, 0.01875 s, 0.025 s. See figures 16 through 19 below.

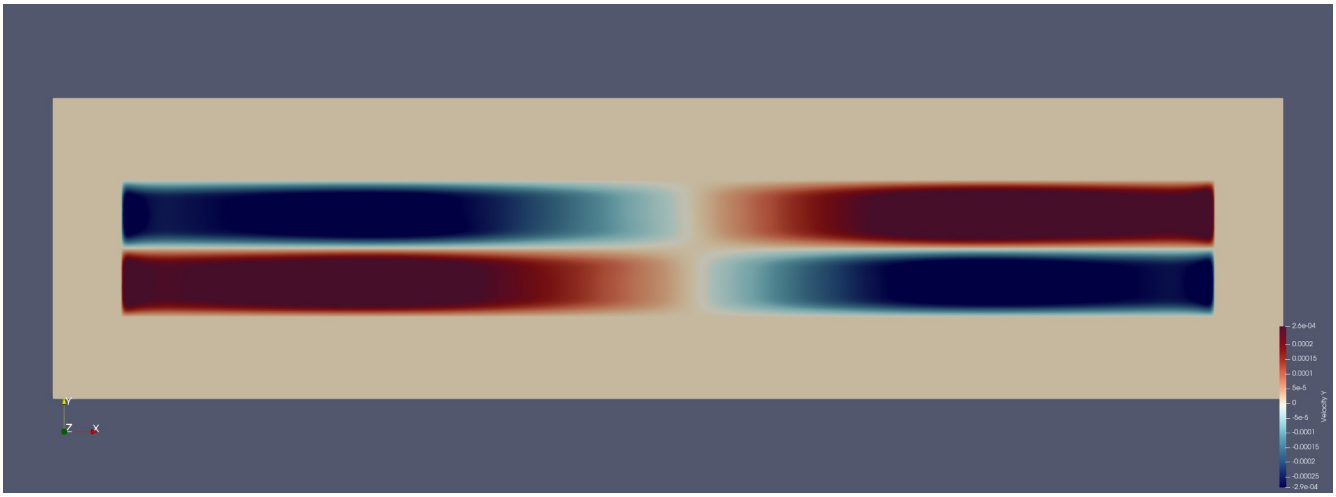

Figure 16: CharLen 0.000175 Vy [t = 0.00625 s]

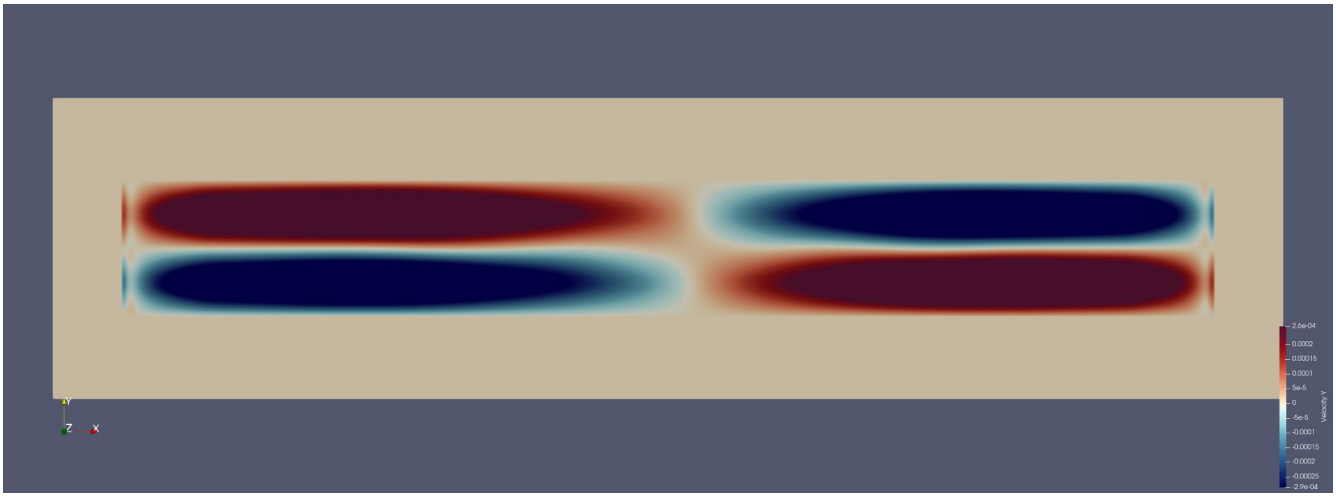

Figure 17: CharLen 0.000175 Vy [t = 0.0125 s]

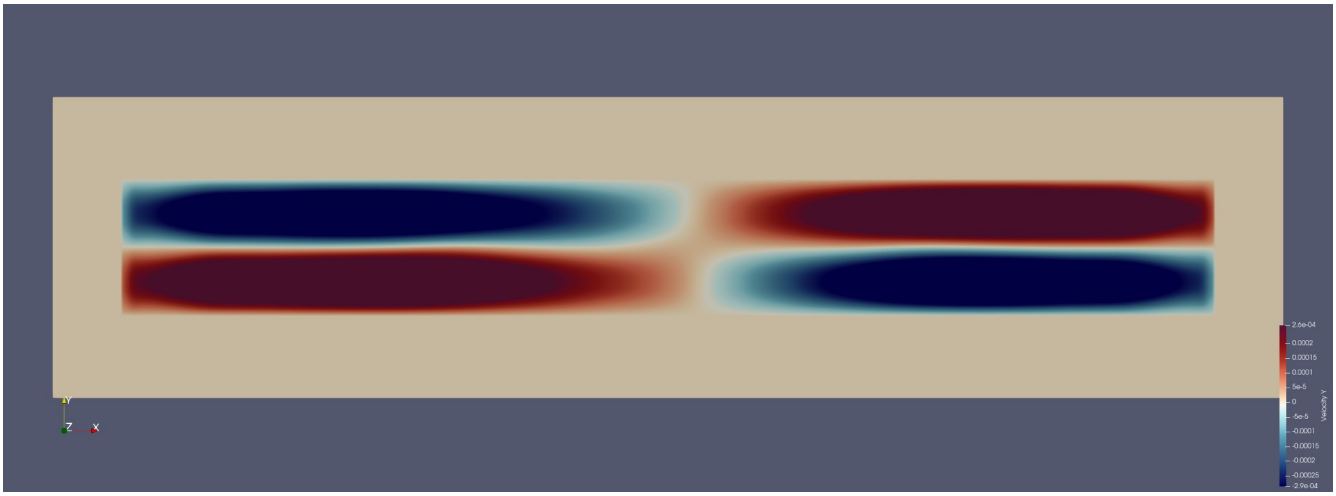

Figure 18: CharLen 0.000175 Vy [t = 0.01875 s]

An additional bit of interesting information that can be gleamed from this is that the “bridge” between the regions of positive velocity seems to grow larger and faster as time progresses. Additionally regions of opposite velocity develop at the inlet and outlet which might be the explanation for the odd flow see in the x component velocity.

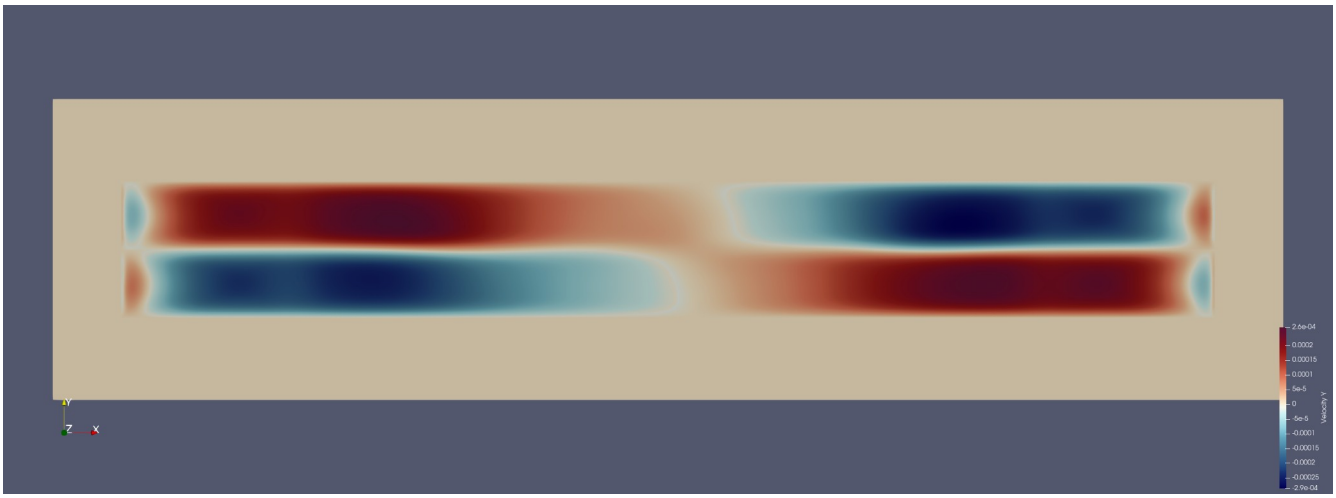

Figure 19: CharLen 0.000175 Vy [t = 0.025 s]

What appears to be happening is the fluid is initially propelled by the magnetic field as expected, then these regions swap with their neighbors because the “information” is traveling in that given direction, then it the fluid comes into the “wrong” region based on the velocity and local magnetic field. In addition to this the fluid traveling towards the wall will bounce off since it is treated as a reflection. The fluid begins to oscillate continuously in this manner with some transition region in the middle. Translating this concept to 3D, the fluid would likely have a spin imparted on it, traveling in a cylinder, the y and z components would push and pull the fluid towards their respective walls. This push and pull would likely impart a spin around the x axis where the convergent and divergent regions would be pulsing as seen in this simulation (now in 3D), and there would be a transition region in the middle of the coil. The oscillation seen in the y component likely is a form of a 2D spin around the x axis. Almost like a set of bike peddles being cranked around the x-axis, here however there is no z axis for the fluid to rotate around so it is a sudo spin.

One other oddity worth exploring is running the simulation to larger final times, even just 0.5 seconds rather that 0.1, the flow reverses and the x component velocity is completely negative. A movie of this can also be found in the “movies/results” folder. For demonstration purposes images of the x velocity have been included at 3 time-steps: the original 0.025 s, 0.1 s and 0.5s, see figures 18 through 20 below, all three using the same scale for ease of use. Another piece that makes thing more puzzling is that this effect does not appear with coarser meshes above 0.0025, however this can likely be explained away by not having the resolution to capture all the details for the complex flow. What might be happening is a feedback loop with the inlet outlet conditions, these conditions repeat the flow properties from inside the domain, so there could be a feedback loop where the inlet and outlet velocities grow until they overpower the force of the magnetic field. For this reason the future work and recommendations section exists to explore a potential solution for this, which was not done in this project due to it’s complexity.

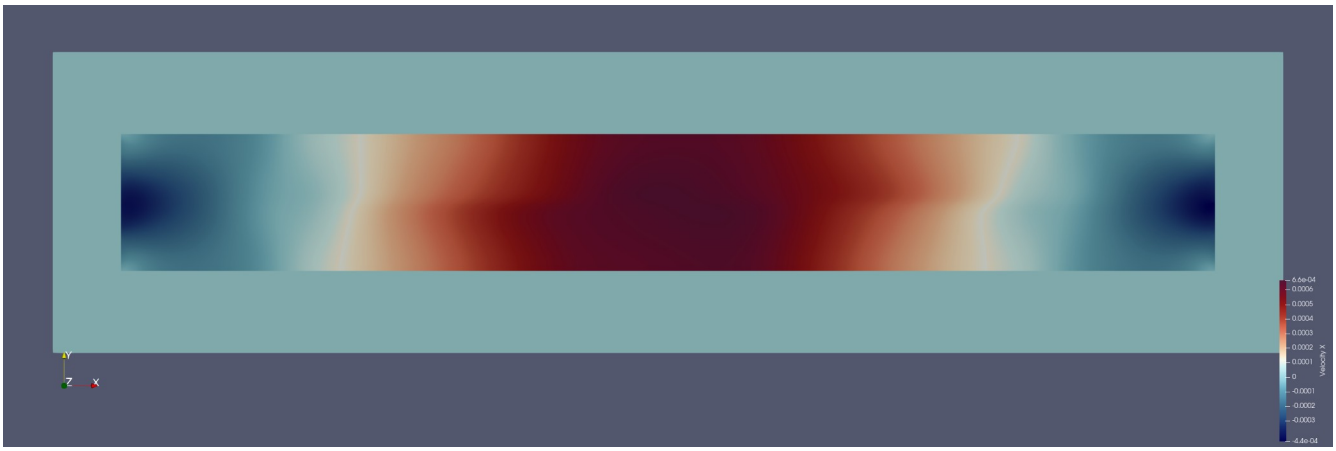

Figure 20: CharLen 0.0005 Vx [t = 0.025 s]

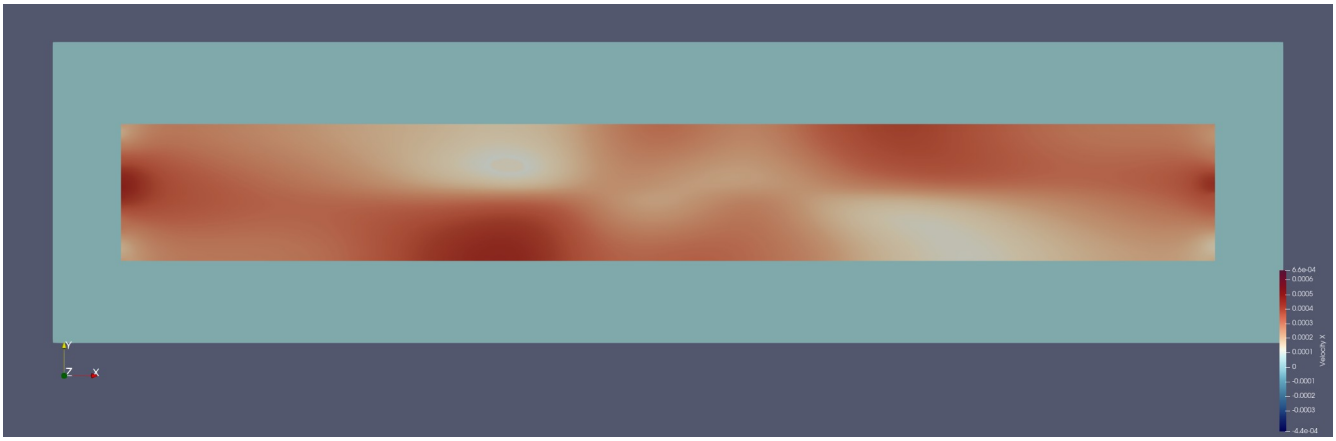

Figure 21: CharLen 0.0005 Vx [t = 0.1 s]

Figure 22: CharLen 0.0005 Vx [t = 0.5 s]

Future Work and Recommendations

In hindsight rather than having the inlet/outlets setup the way they were where the height is fixed at the inlet and the velocity copied from inside, the domain should have instead been made periodic (i.e. fluid that goes out comes in and vice versa). This however, would have required some complicated mapping since the cells on either inlet and outlet are not guaranteed to be the same. To do this the momentum and height fields at a given edge (inlet and outlet) would need to be mapped to a temporary vector as a function of y, then this vector would be passed to the other (from outlet to inlet, inlet to outlet) this map would then be decoded and those properties applied to that edge. This would likely achieve the desired goal of the entire fluid domain having a positive x velocity. This would likely also eliminate the feedback issue that seems to plague the larger final times where the fluid moves in the “wrong” direction.

Another modification that could be made would be to reduce the y height of the fluid domain even further, this would be to increase the overall x component magnetic field within the fluid domain since it falls off quickly as was seen in figure 3.

Another modification would be to have the fluid re-directed around the outside of the coils, in this case the fluid would enter and leave on the same side of the domain (enters left side with positive velocity leaves with negative velocity). This would allow the fluid to take full advantage of the magnetic field rather than having one half of the magnetic field doing no useful work. This case would also make sense to have a periodic boundary condition but it would be much more complicated since the inlet and outlet do not occupy the same region of the y axis, and the outlet would be split into two segments, so these would need to be combined in the mapping.

References

[1] T. Kahmann and F. Ludwig, “Magnetic field dependence of the effective magnetic moment of multi-core nanoparticles,” *Journal of Applied Physics*, vol. 127, no. 23, pp. 233901–233901, Jun. 2020, doi: https://doi.org/10.1063/5.0011629.

[2] "Biot-Savart Law," Geosci.xyz, [Online]. Available: https://em.geosci.xyz/content/maxwell1_fundamentals/formative_laws/biot_savart.html?highlight=biot%20savart. [Accessed: April 20, 2023].

Appendix

A. Mesh Study Plots

Height

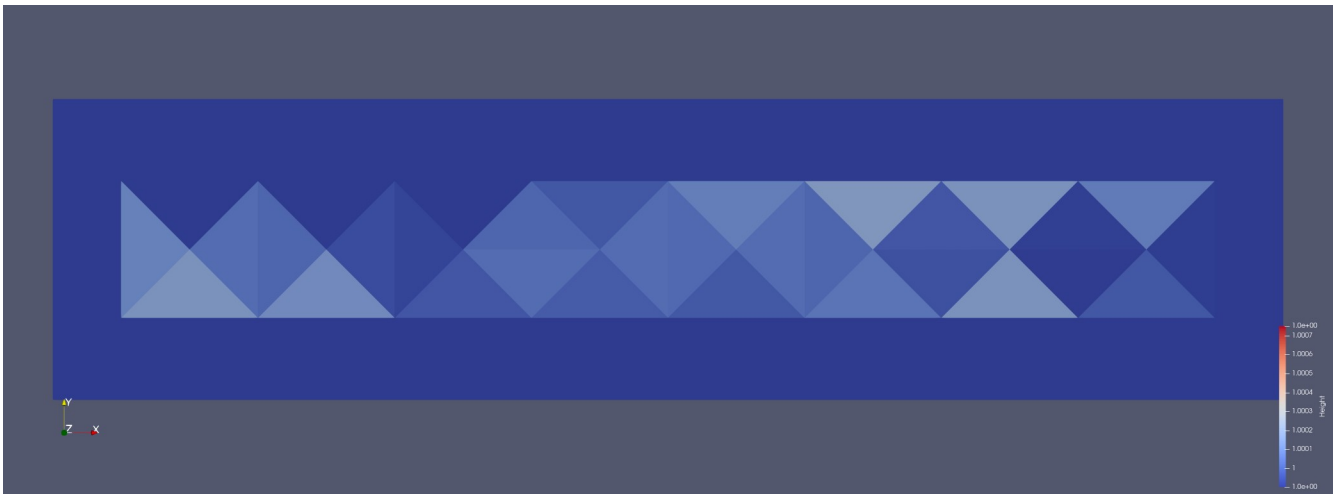

Figure 23: CharLen 0.05 Height

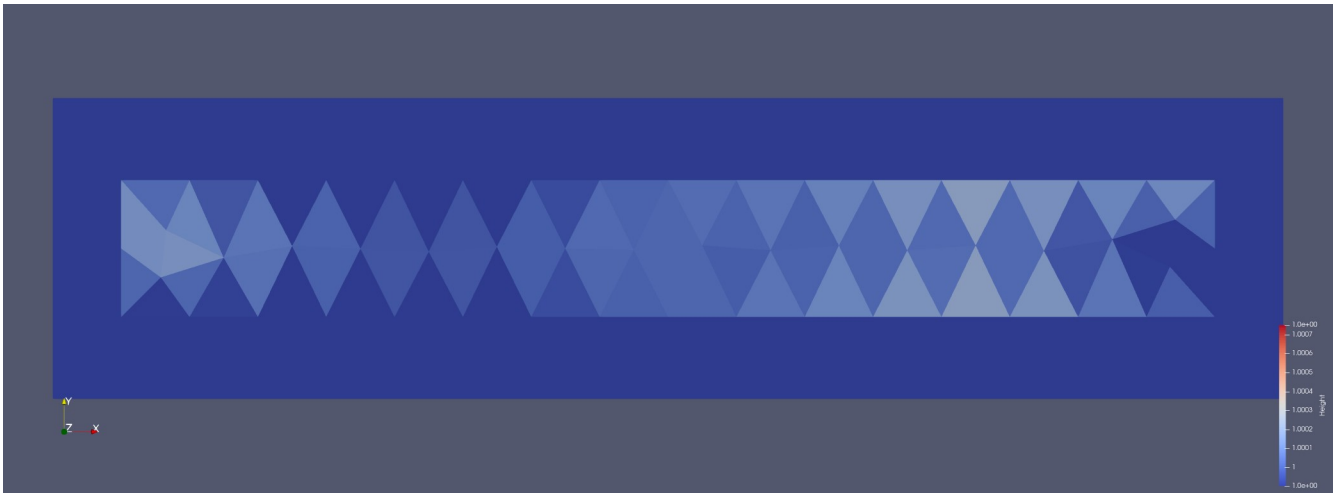

Figure 24: CharLen 0.025 Height

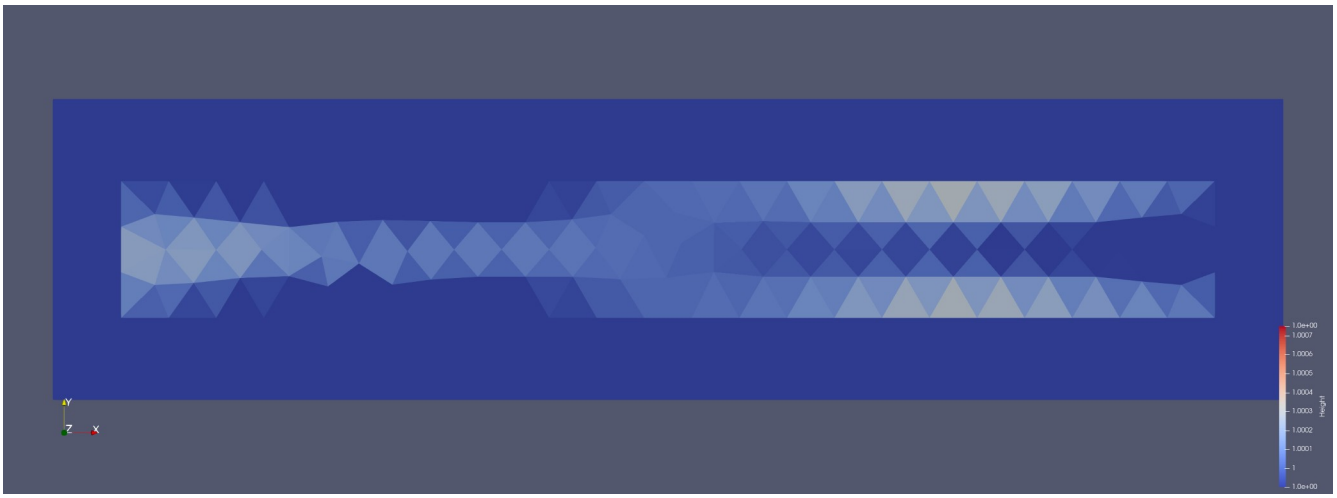

Figure 25: CharLen 0.0175 Height

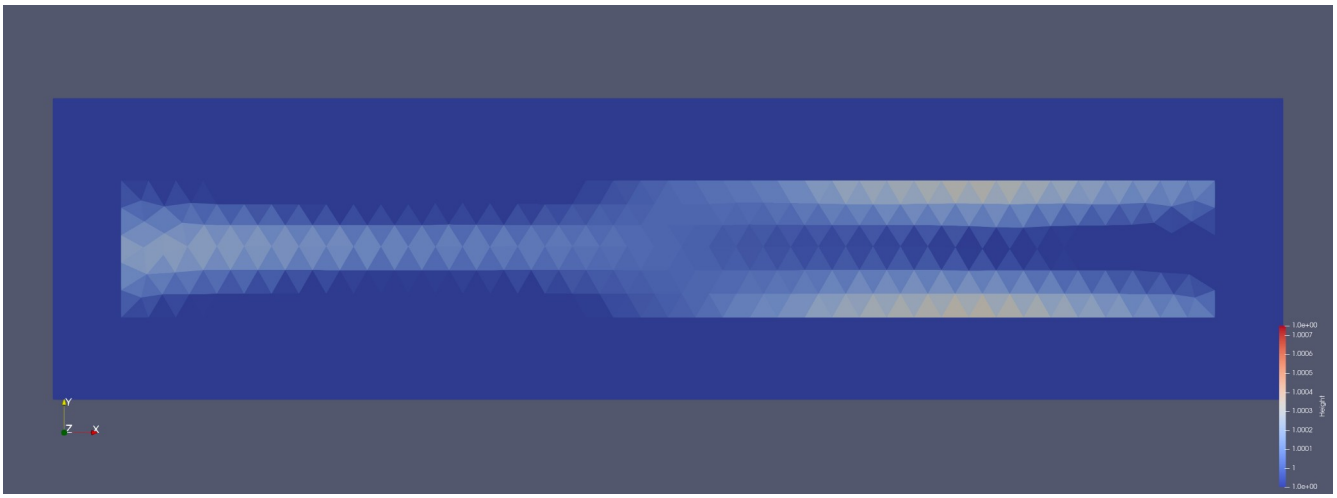

Figure 26: CharLen 0.01 Height

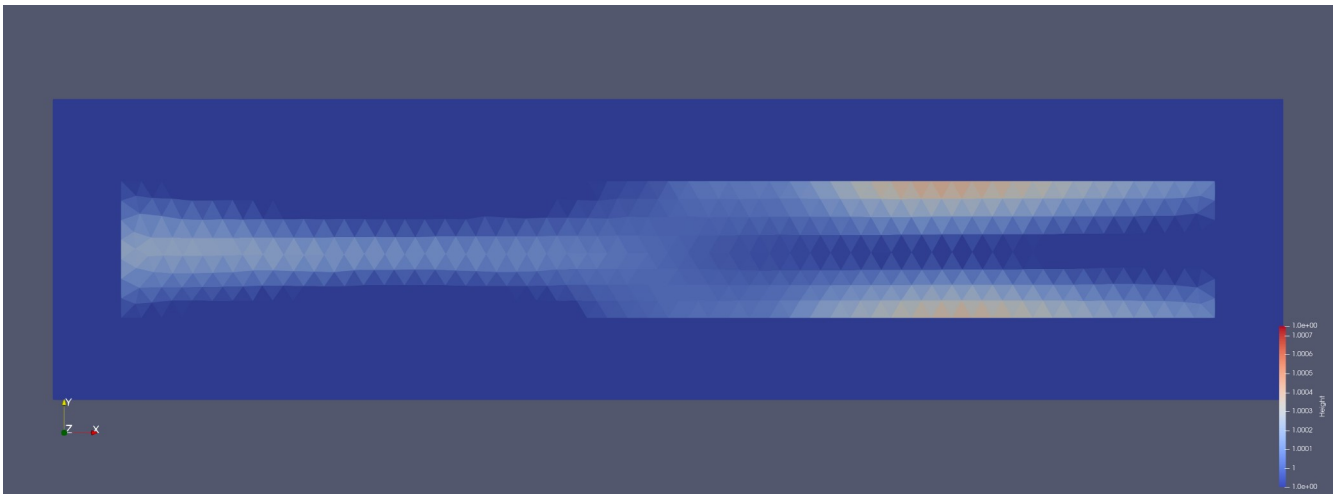

Figure 27: CharLen 0.0075 Height

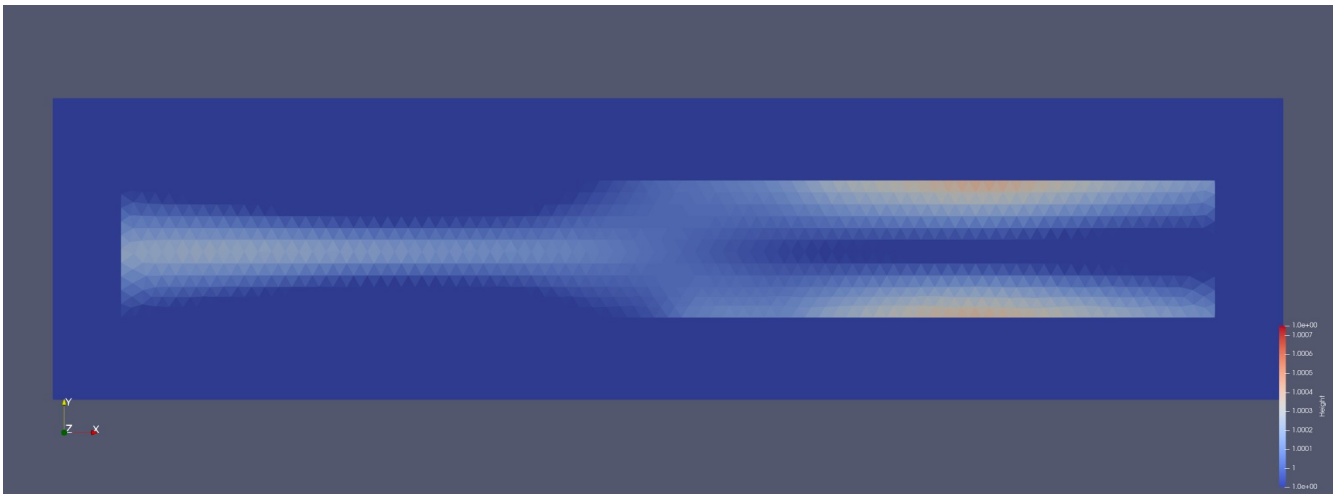

Figure 28: CharLen 0.005 Height

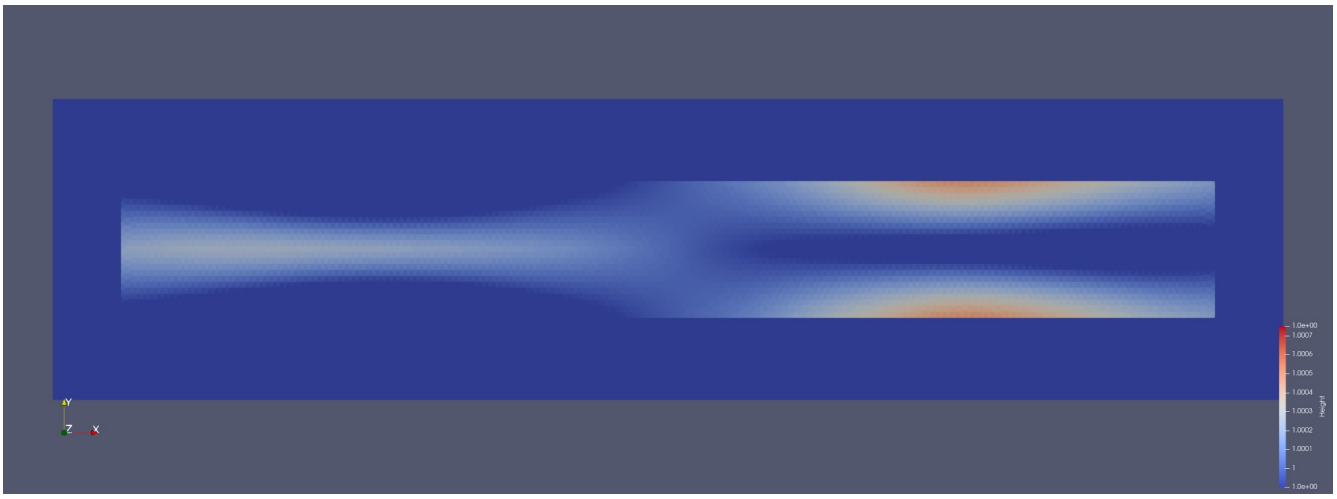

Figure 29: CharLen 0.0025 Height

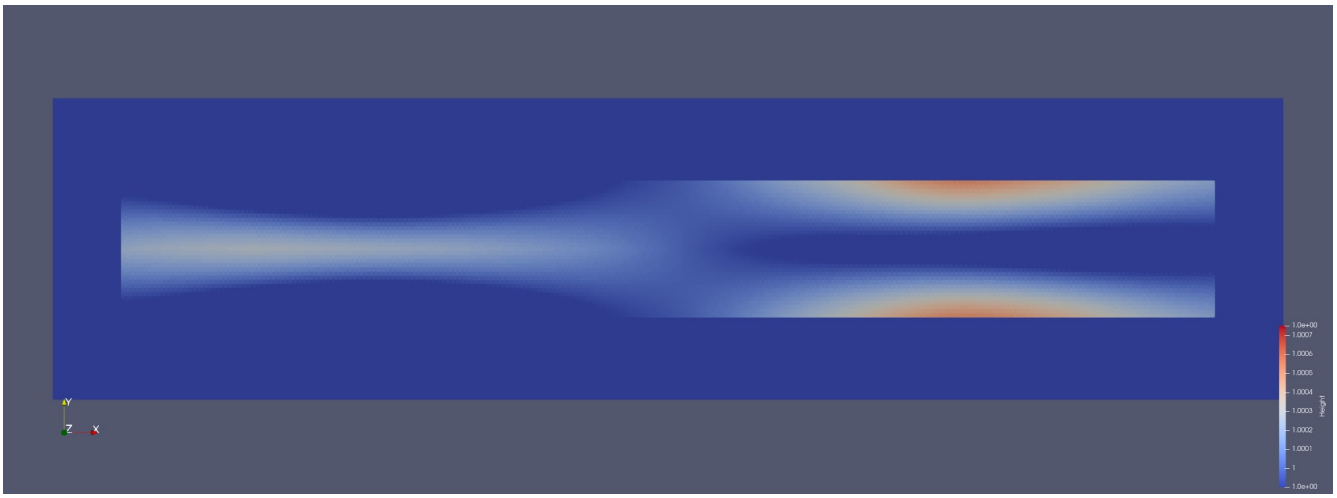

Figure 30: CharLen 0.00175 Height

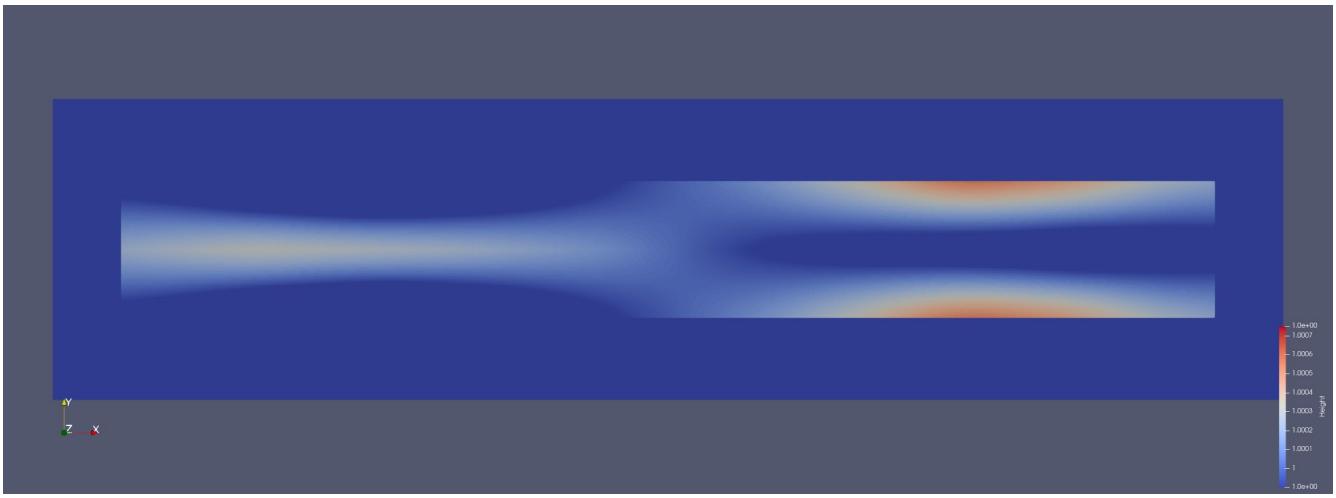

Figure 31: CharLen 0.001 Height

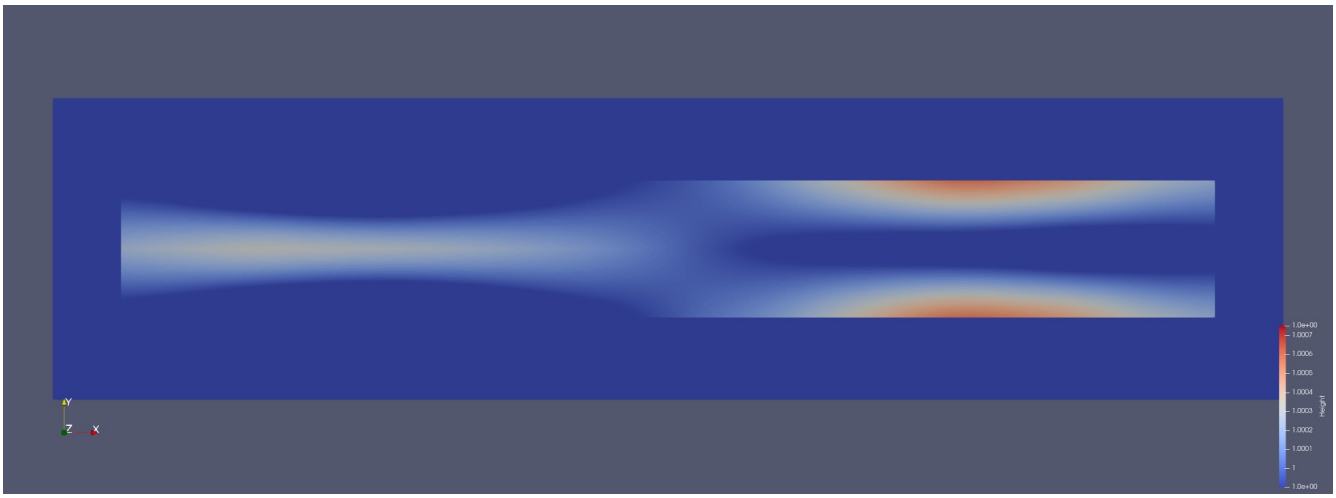

Figure 32: CharLen 0.00075 Height

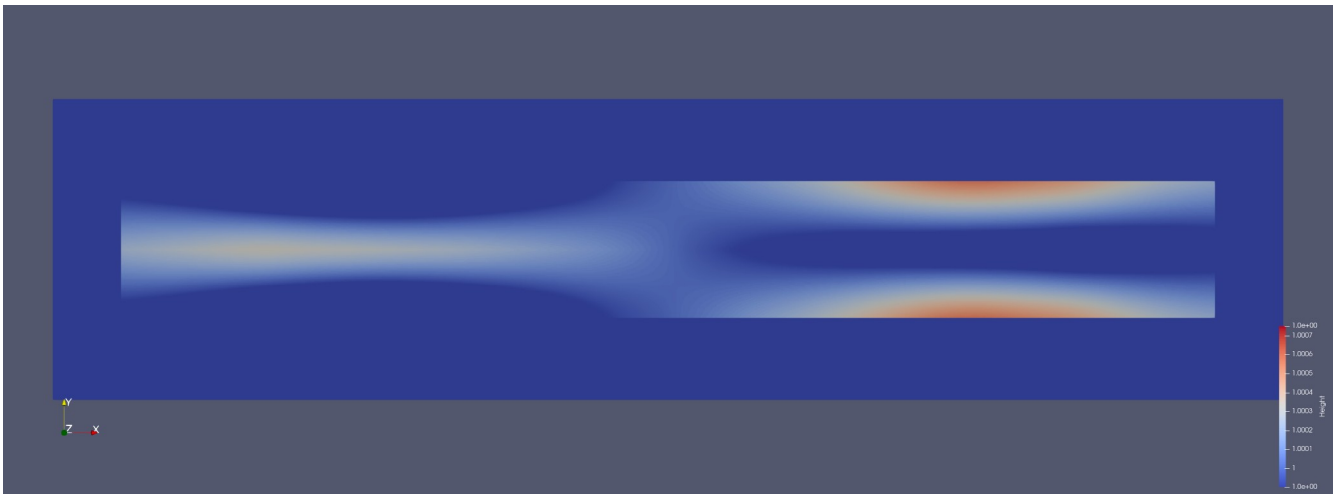

Figure 33: CharLen 0.0005 Height

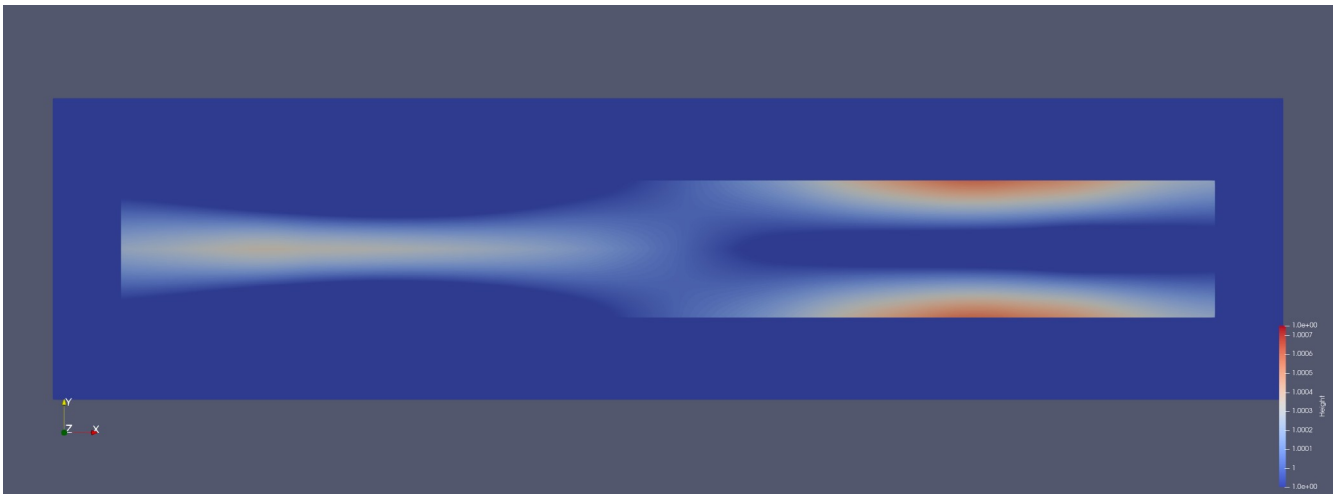

Figure 34: CharLen 0.00025 Height

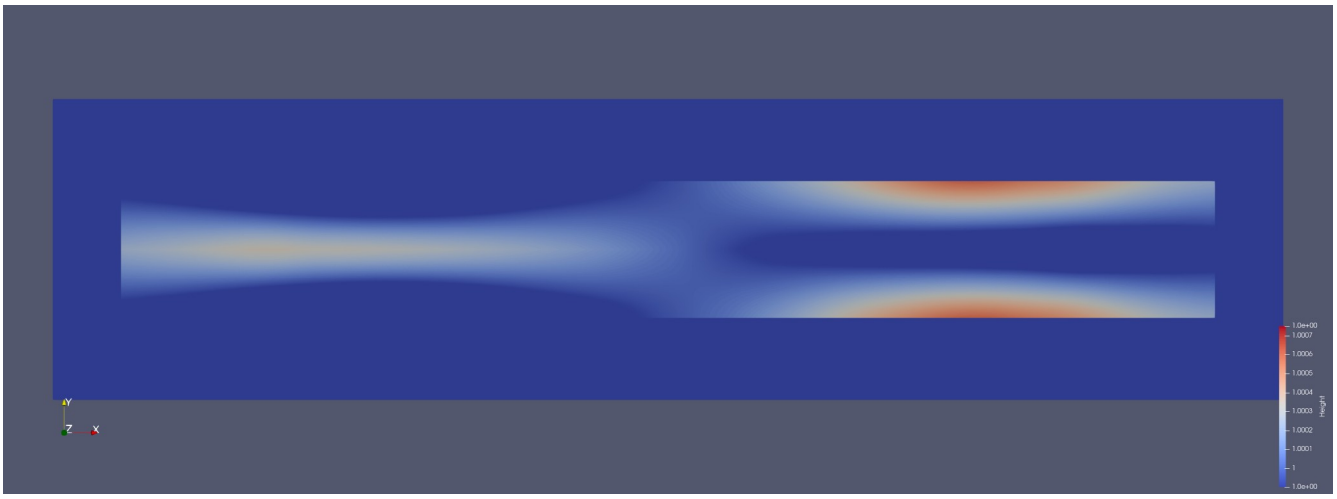

Figure 35: CharLen 0.000175 Height

X Velocity

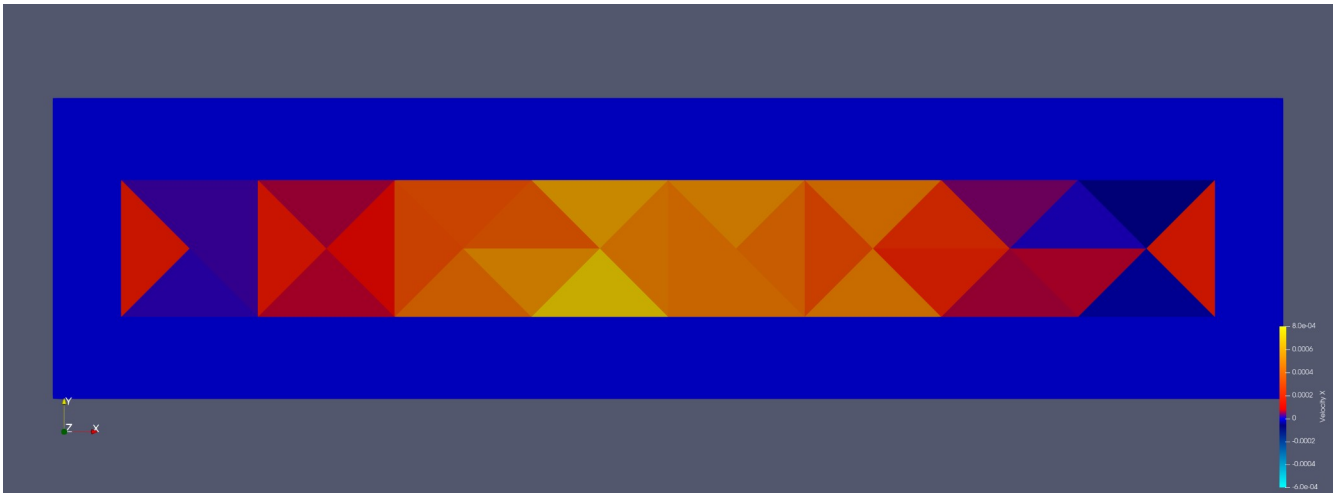

Figure 36: CharLen 0.05 Vx

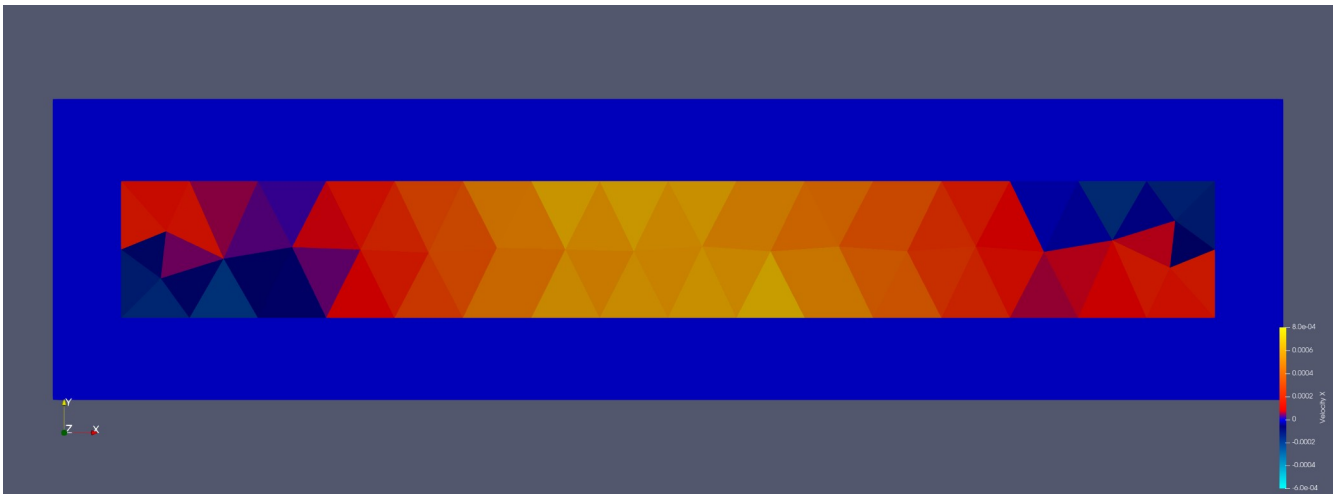

Figure 37: CharLen 0.025 Vx

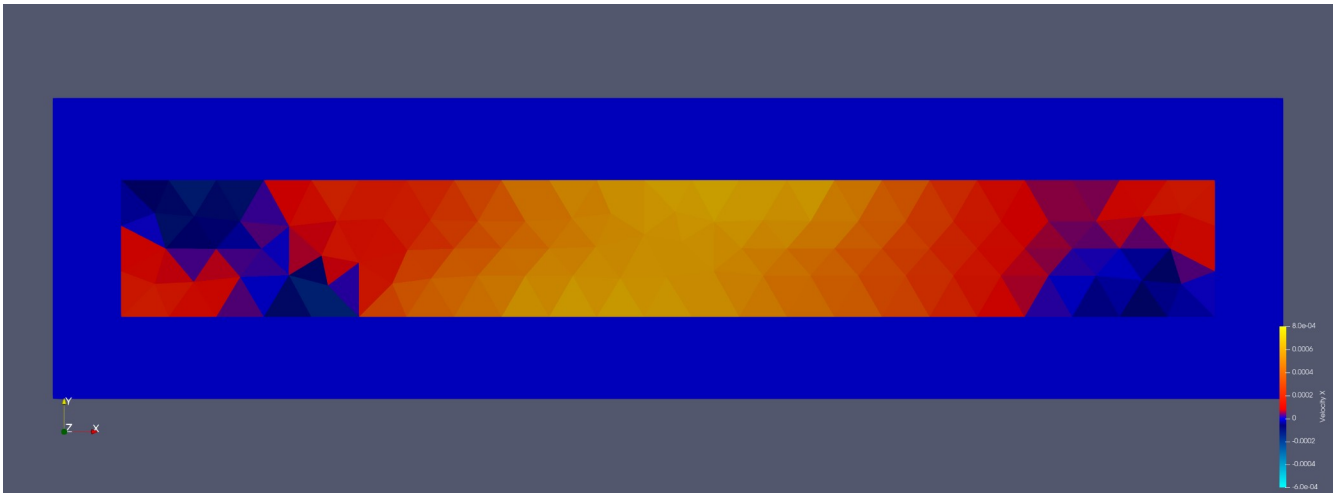

Figure 38: CharLen 0.0175 Vx

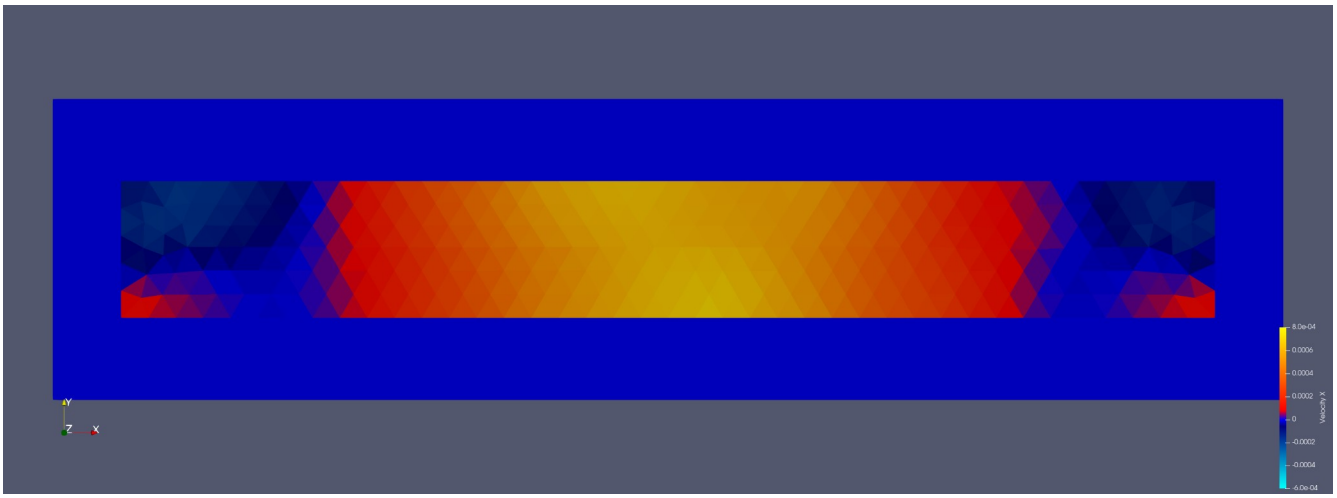

Figure 39: CharLen 0.01 Vx

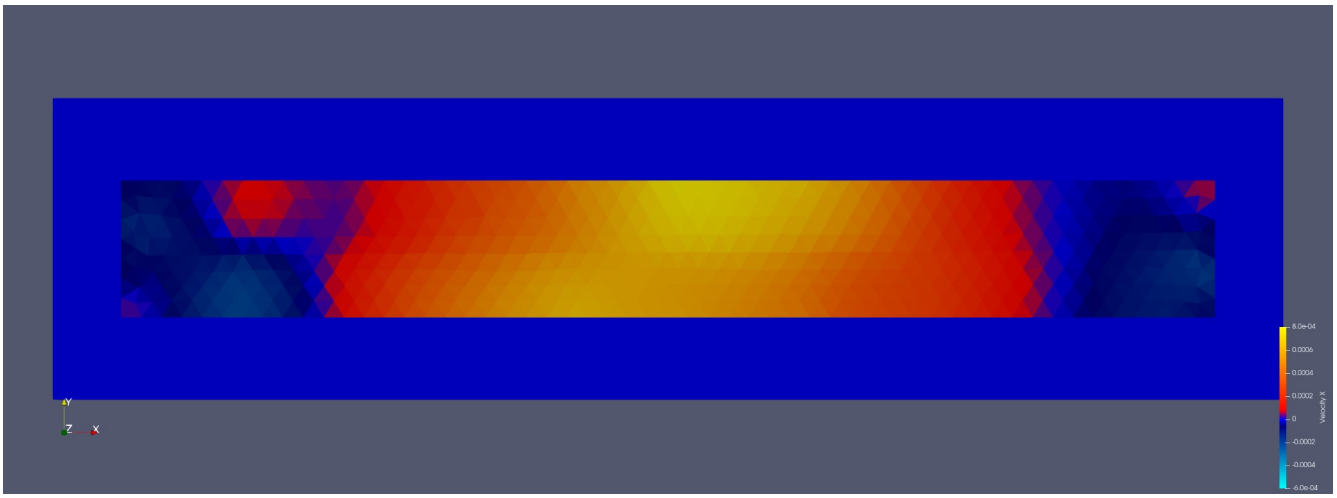

Figure 40: CharLen 0.0075 Vx

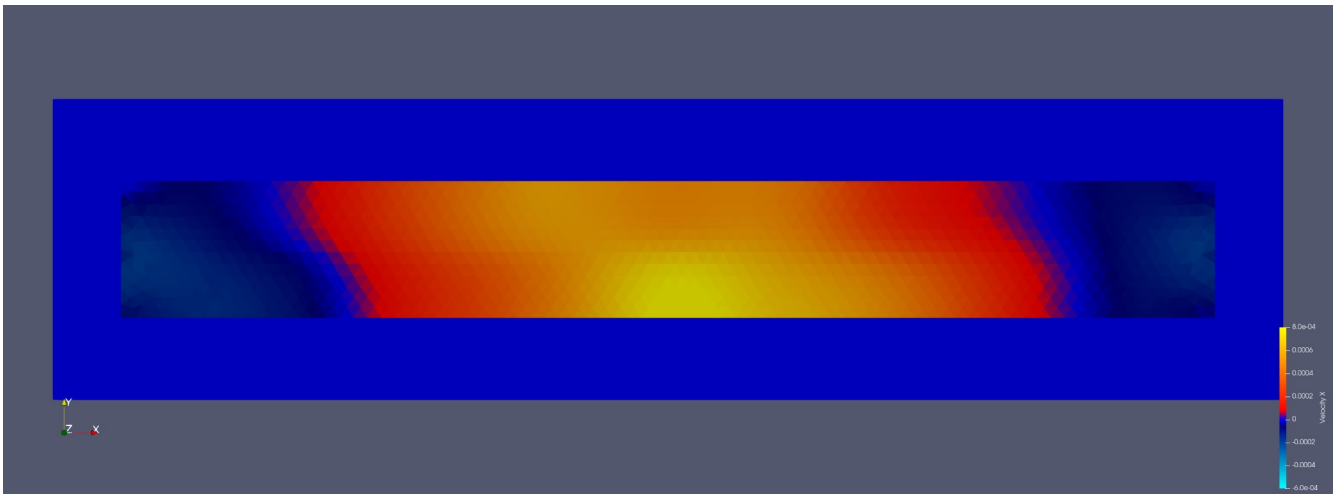

Figure 41: CharLen 0.005 Vx

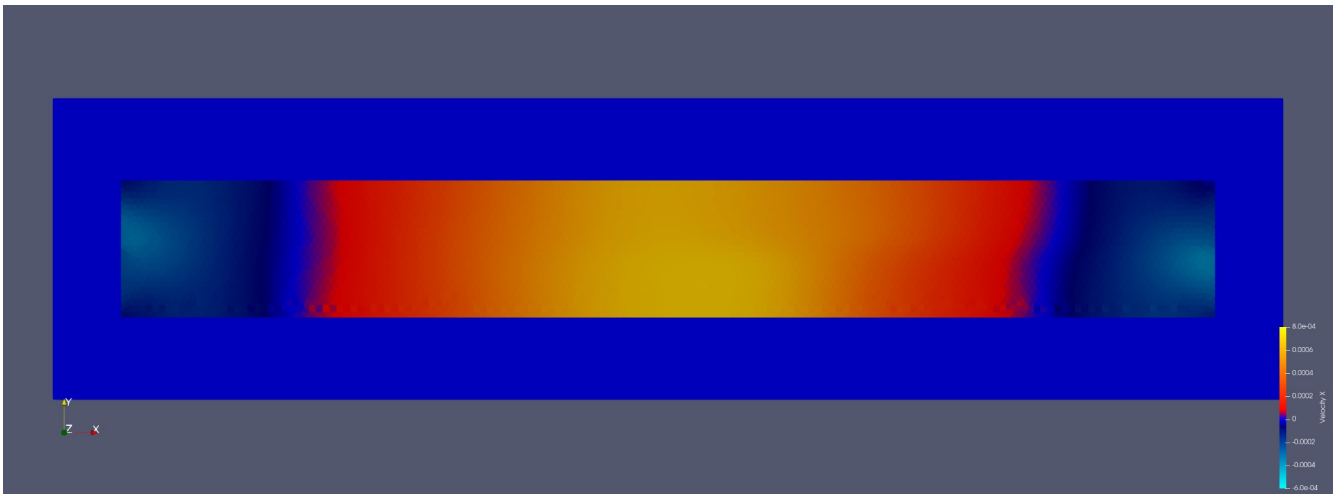

Figure 42: CharLen 0.0025 Vx

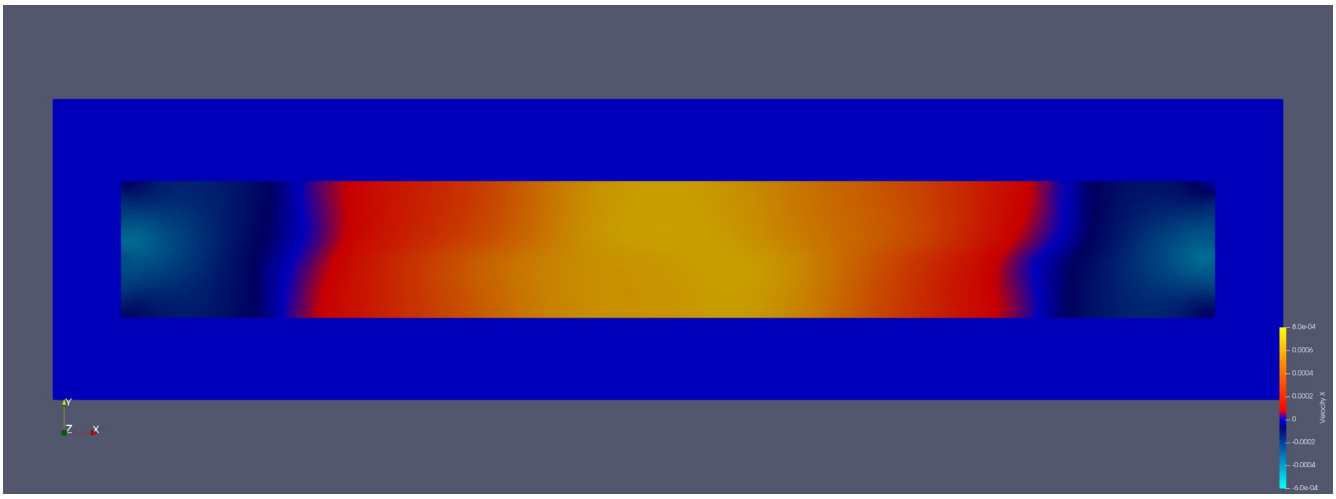

Figure 43: CharLen 0.00175 Vx

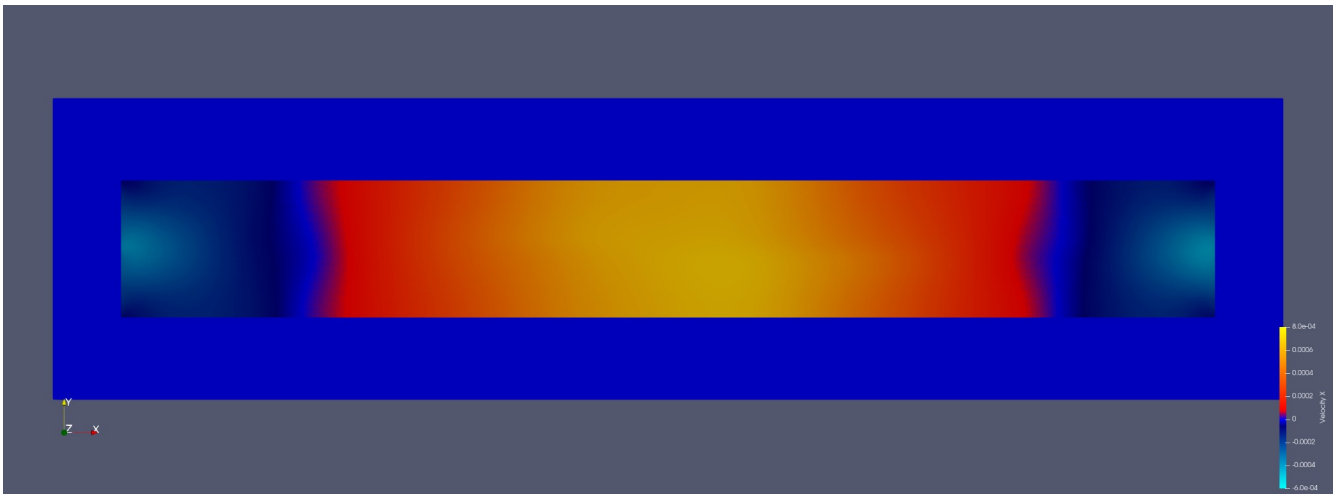

Figure 44: CharLen 0.001 Vx

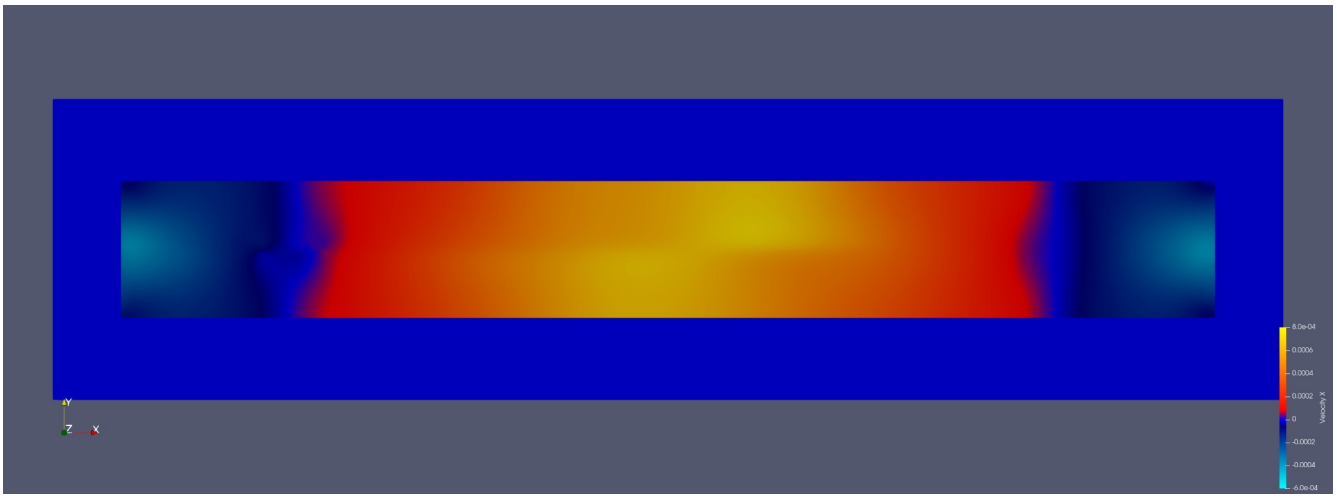

Figure 45: CharLen 0.00075 Vx

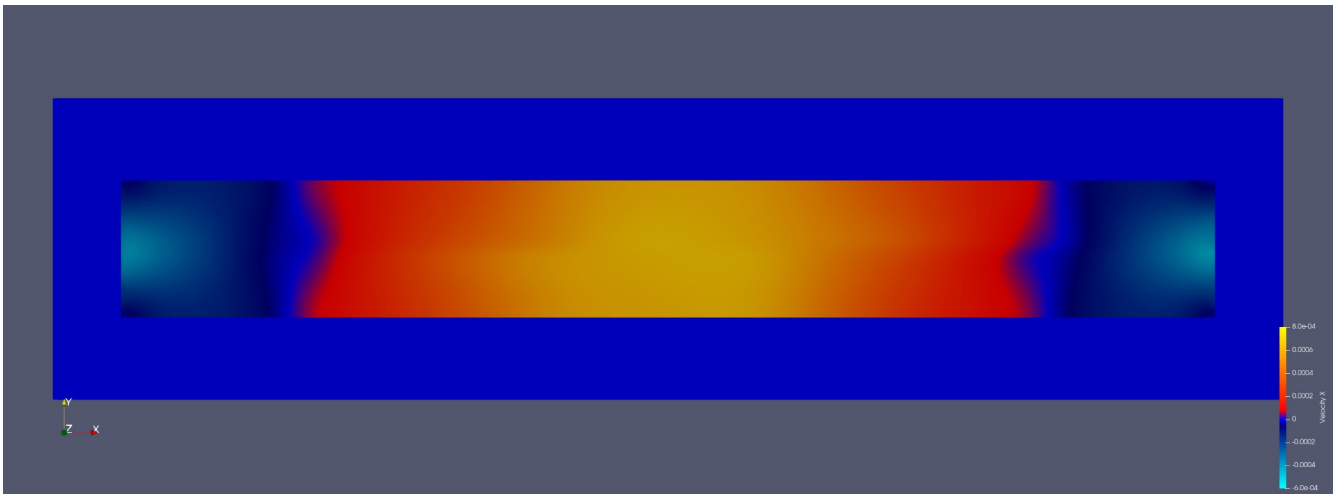

Figure 46: CharLen 0.0005 Vx

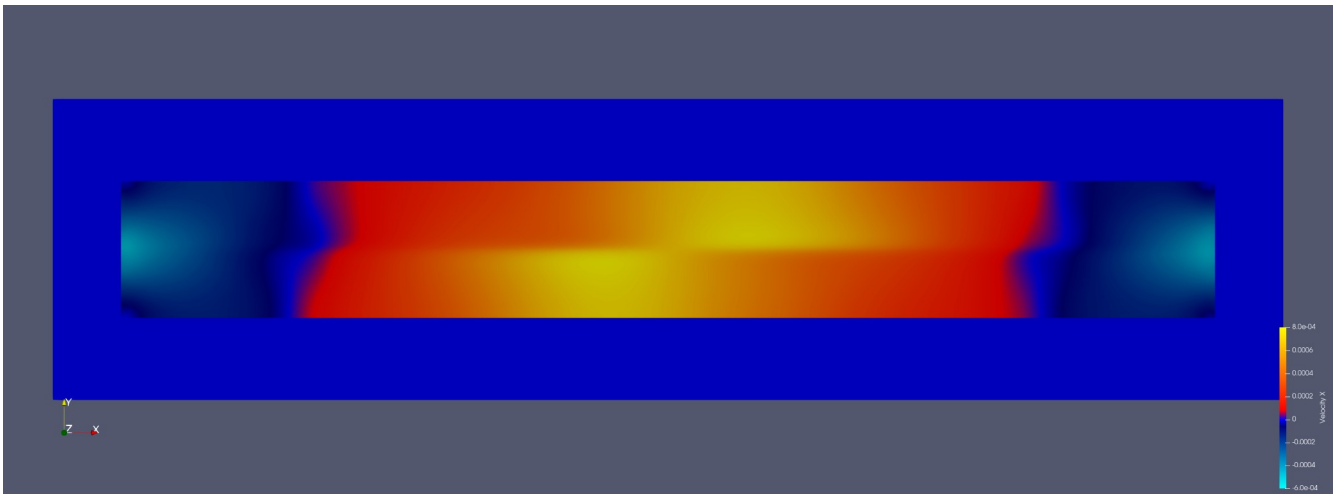

Figure 47: CharLen 0.00025 Vx

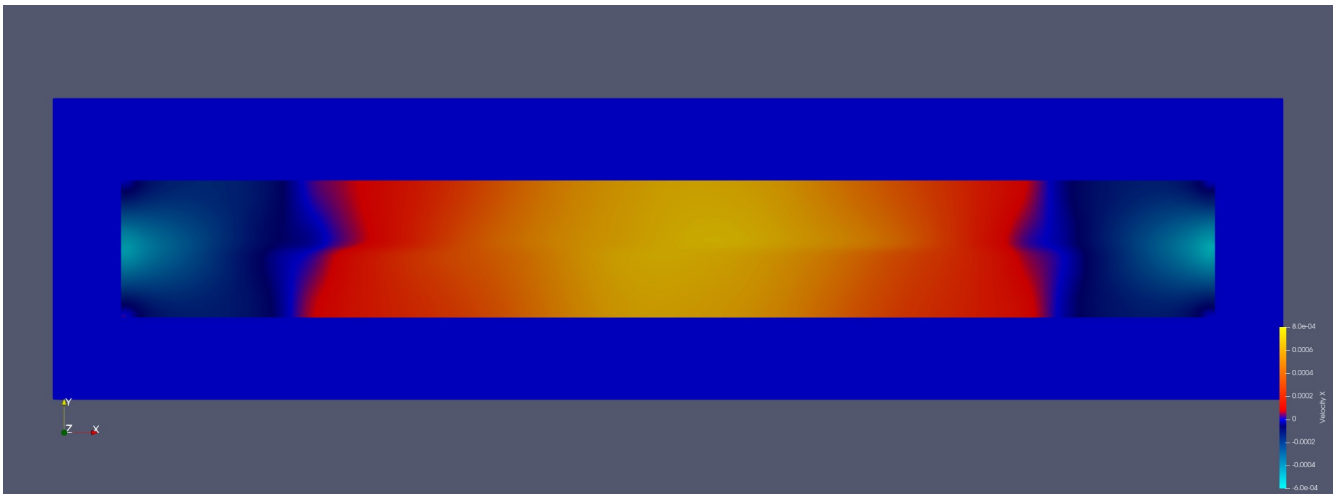

Figure 48: CharLen 0.000175 Vx

Y Velocity

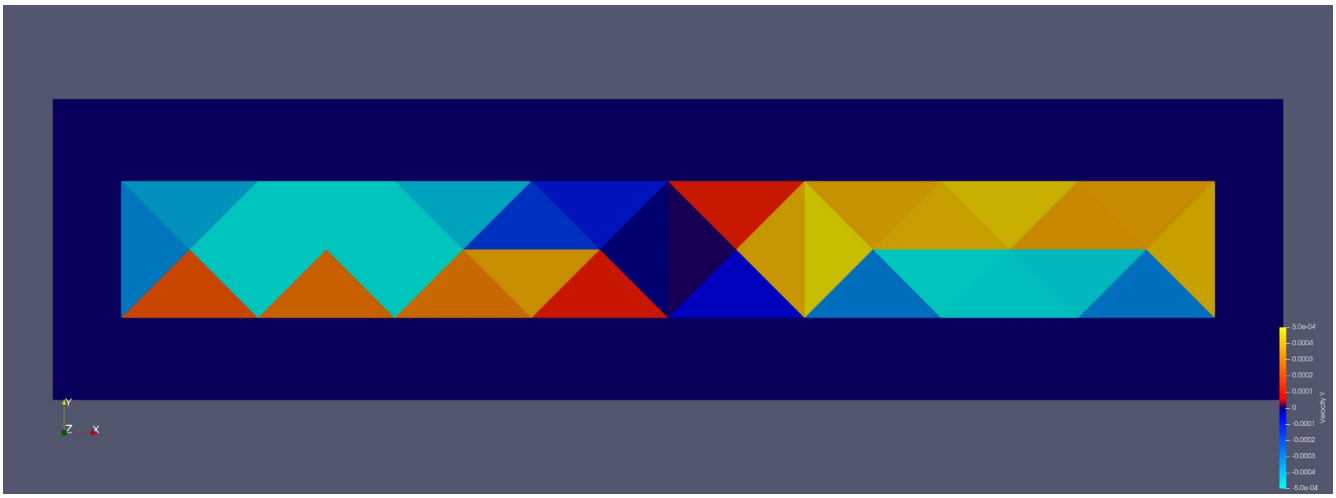

Figure 49: CharLen 0.05 Vy

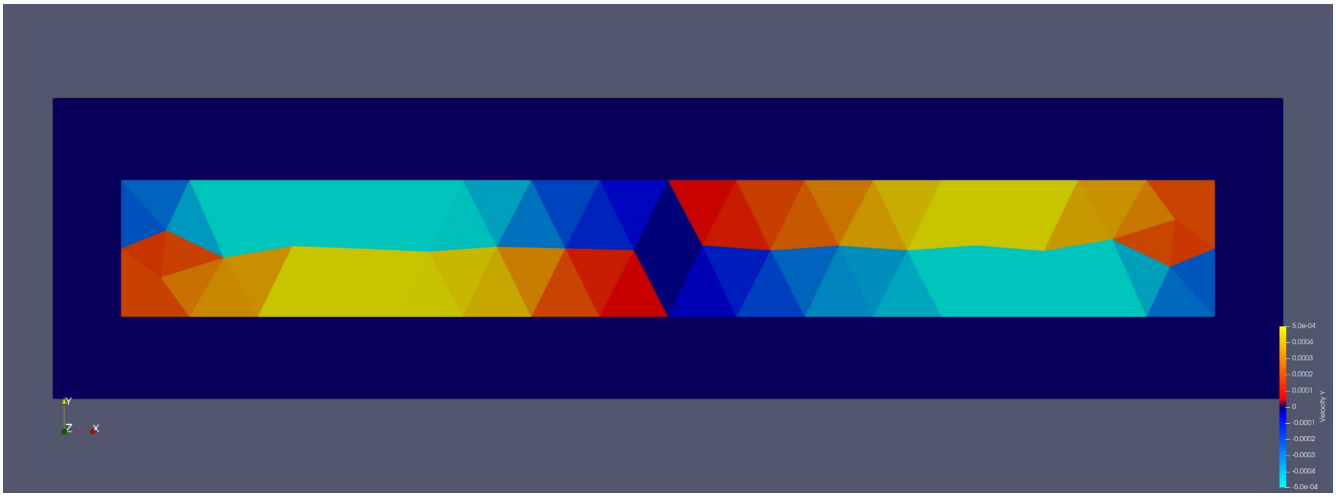

Figure 50: CharLen 0.025 Vy

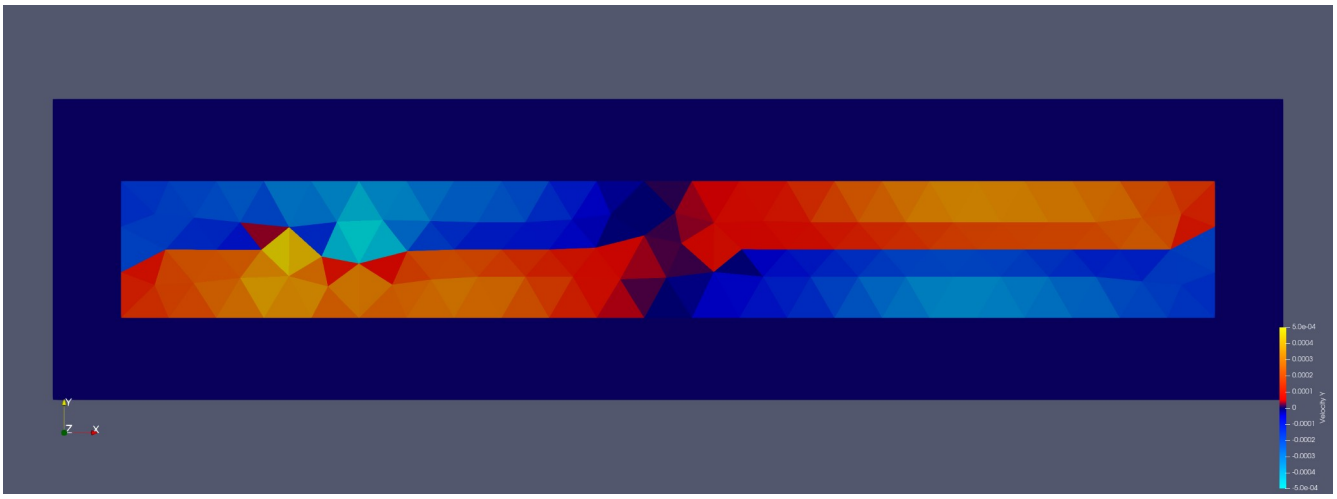

Figure 51: CharLen 0.0175 Vy

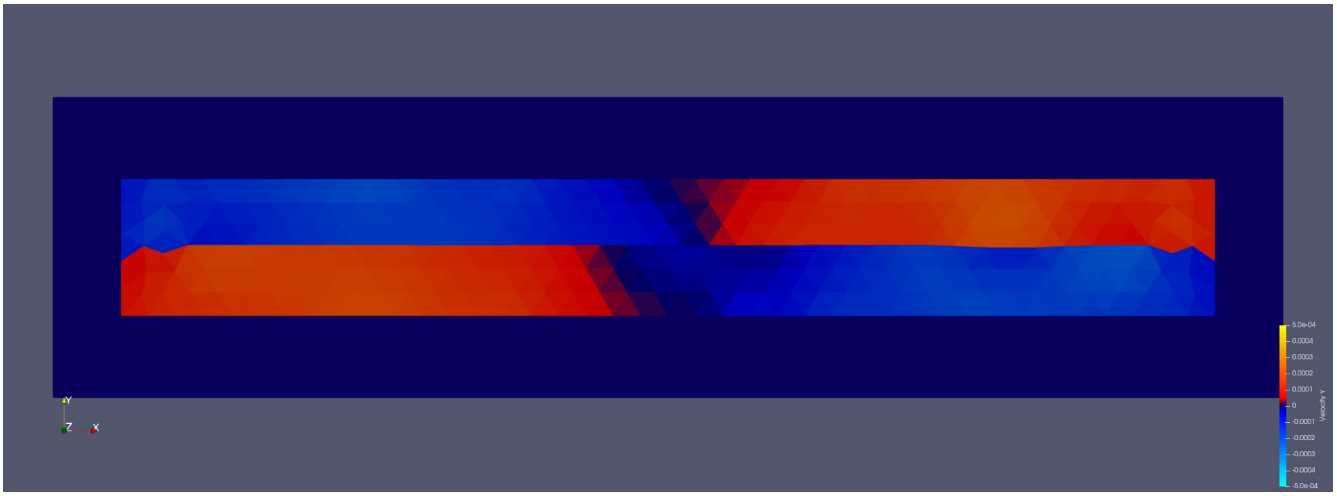

Figure 52: CharLen 0.001 Vy

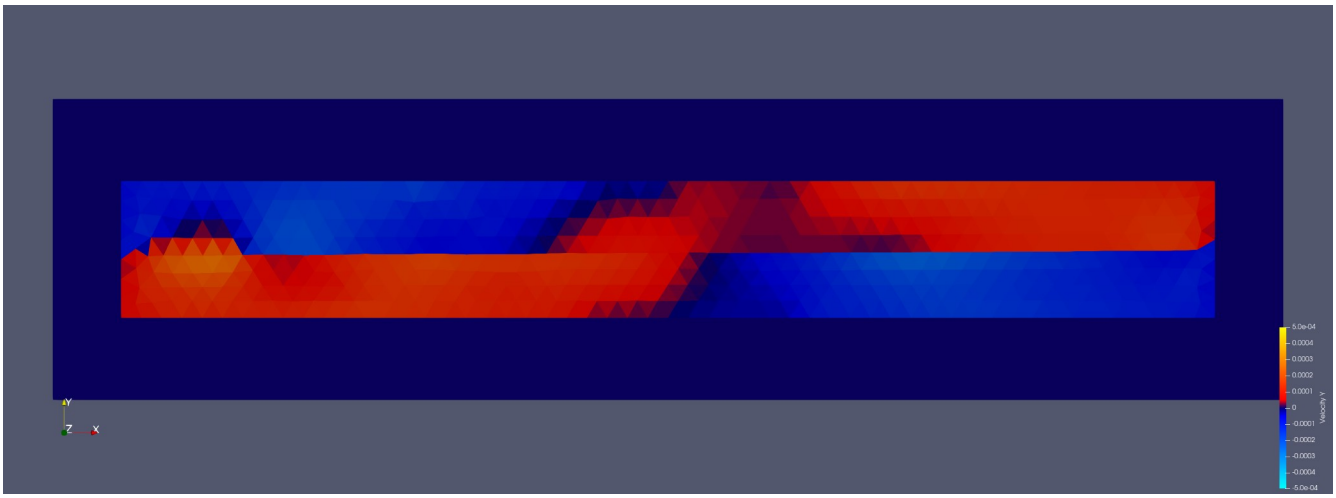

Figure 53: CharLen 0.0075 Vy

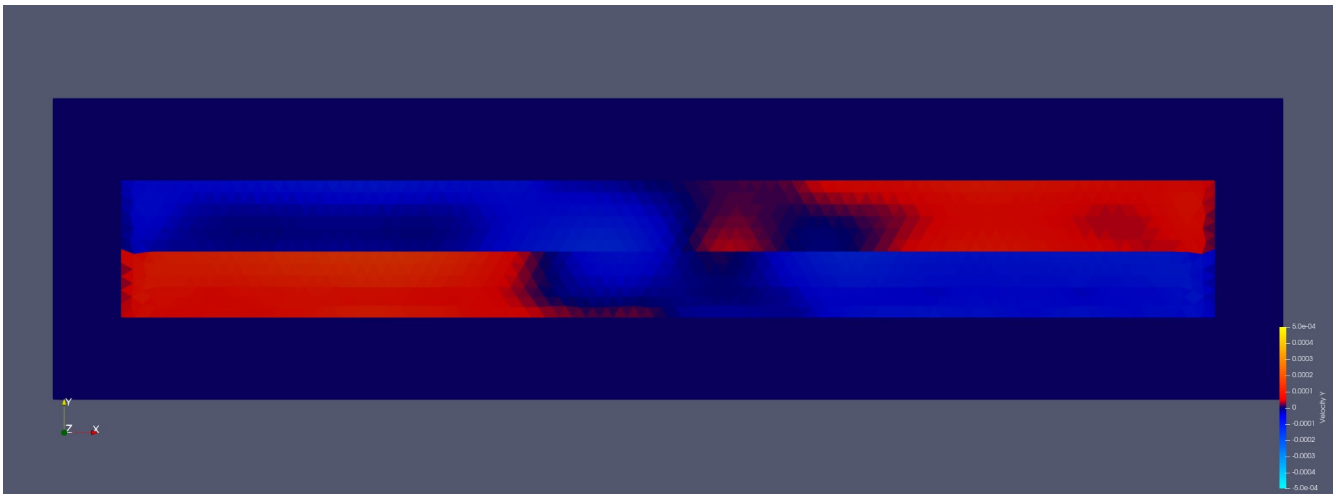

Figure 54: CharLen 0.005 Vy

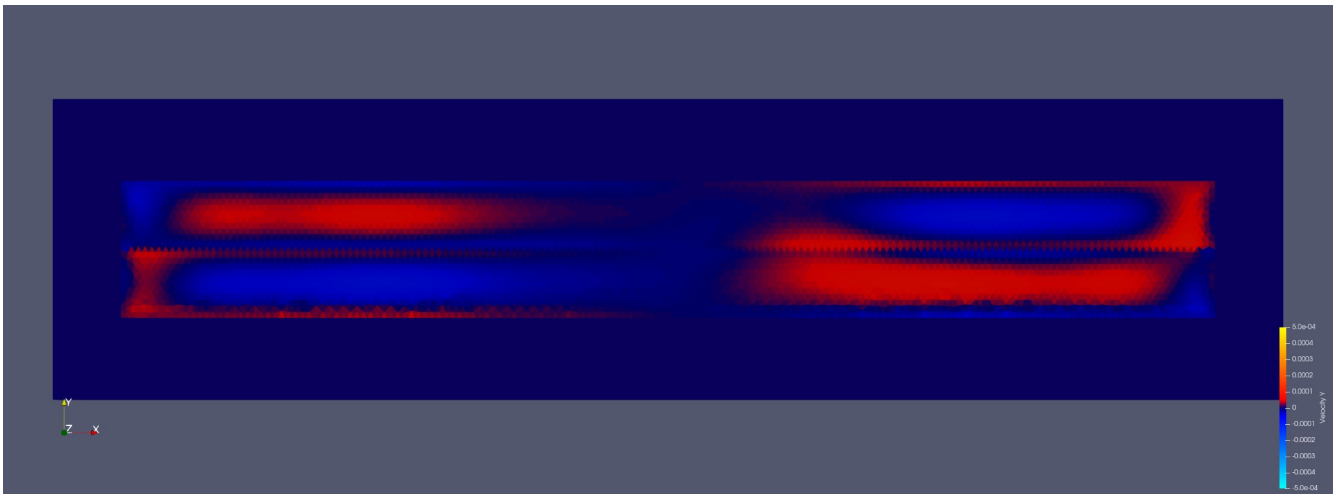

Figure 55: CharLen 0.0025 Vy

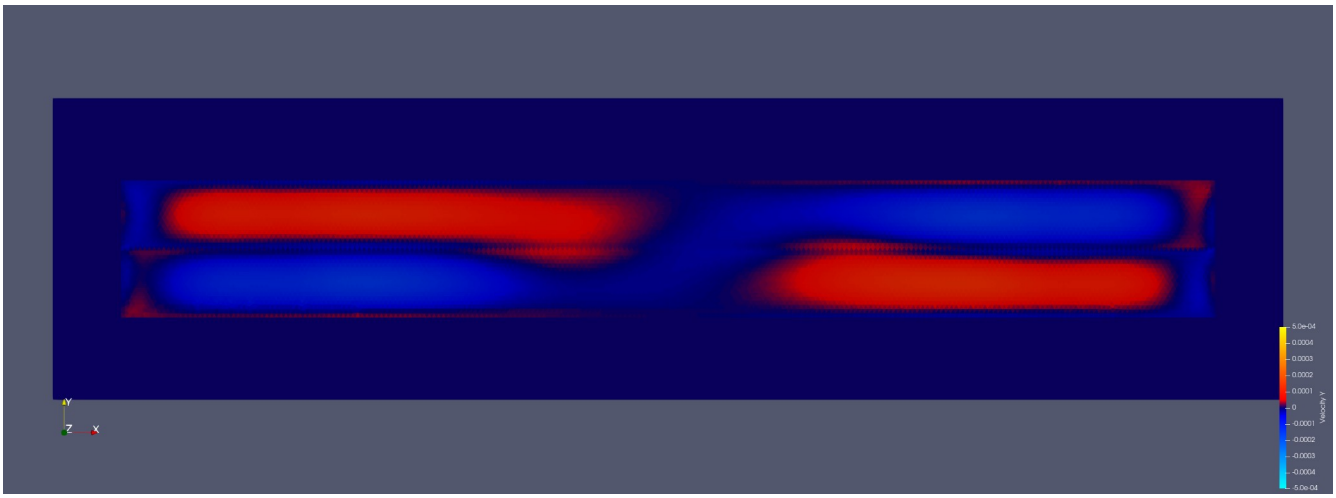

Figure 56: CharLen 0.00175 Vy

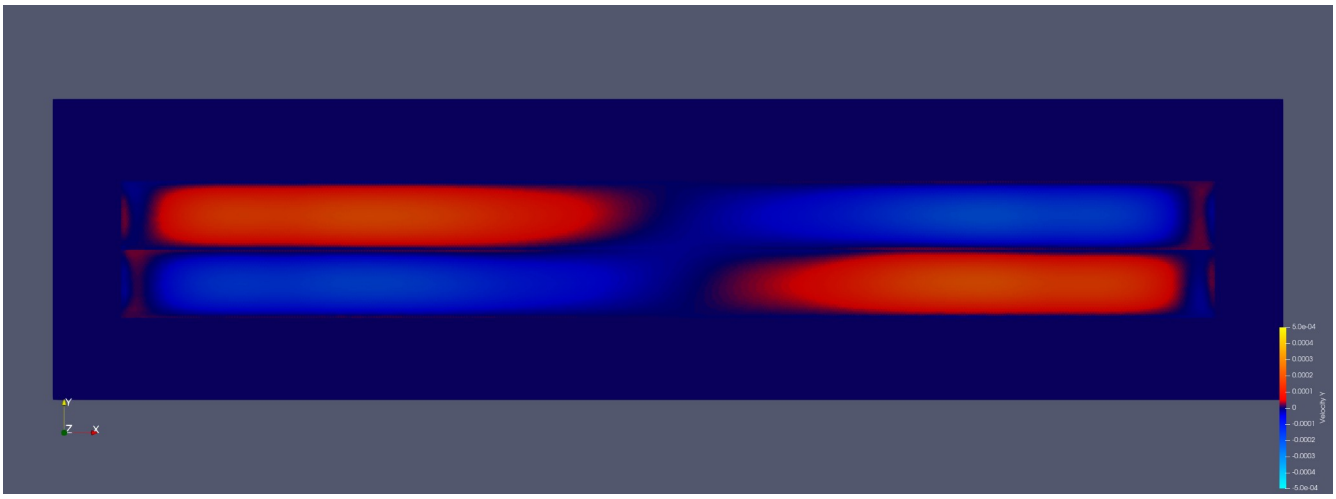

Figure 57: CharLen 0.001 Vy

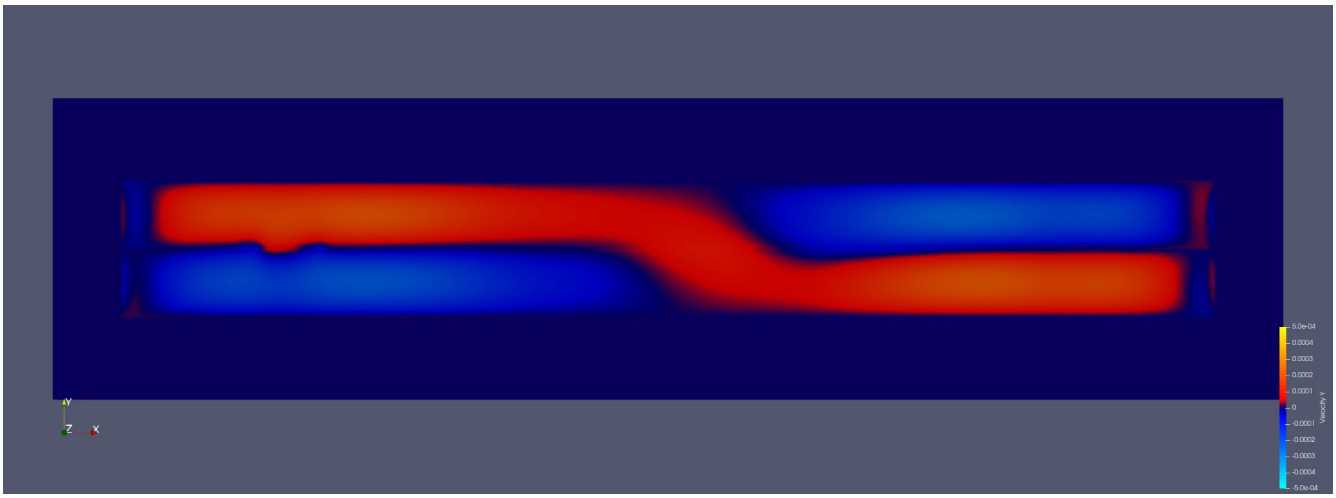

Figure 58: CharLen 0.00075 Vy

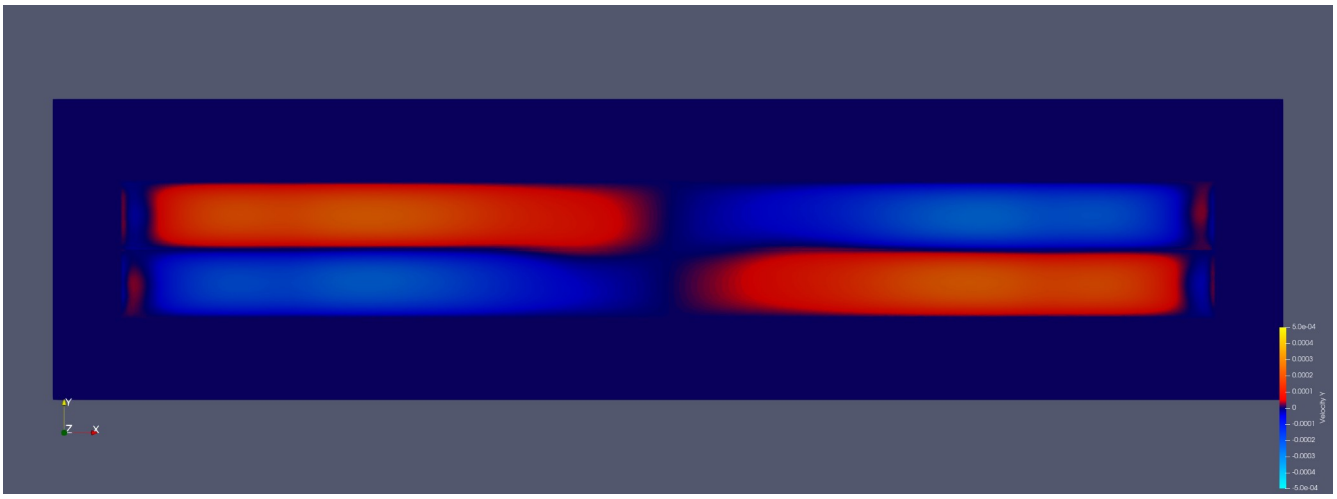

Figure 59: CharLen 0.0005 Vy

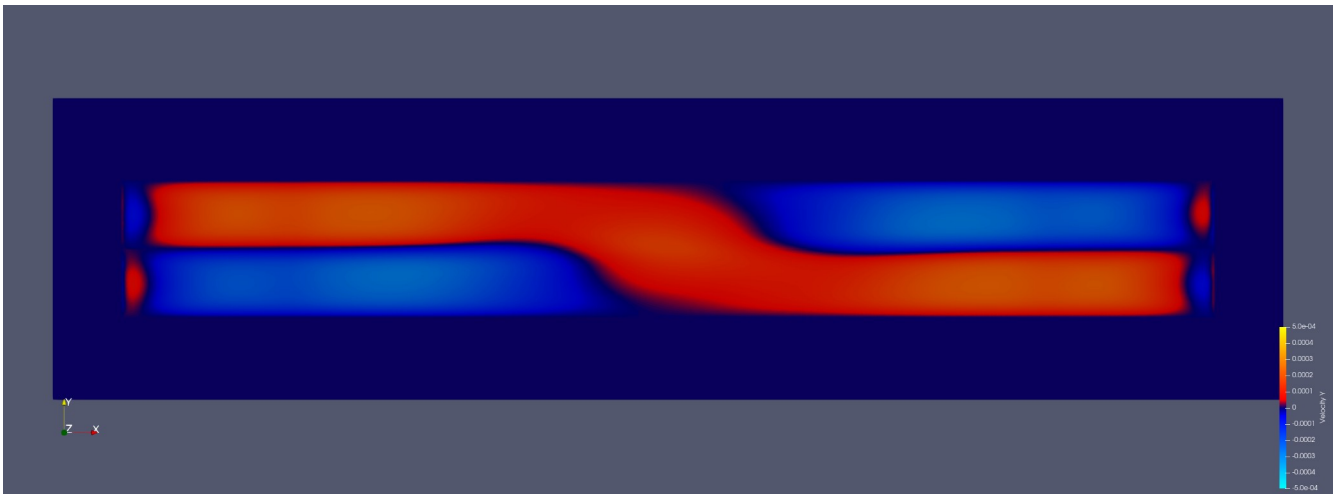

Figure 60: CharLen 0.00025 Vy

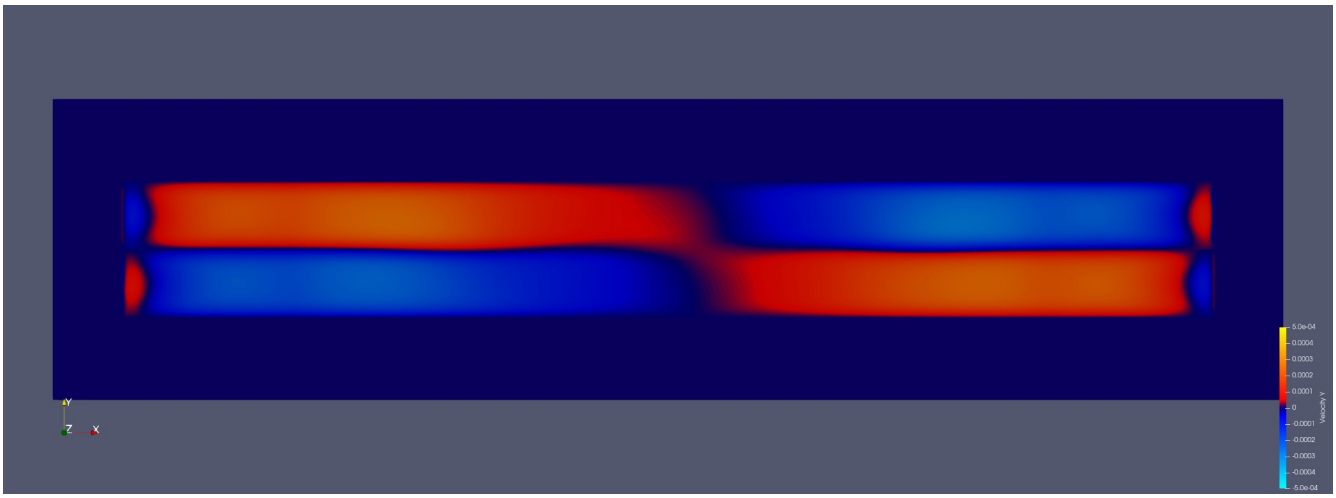

Figure 61: CharLen 0.000175 Vy

Height Error

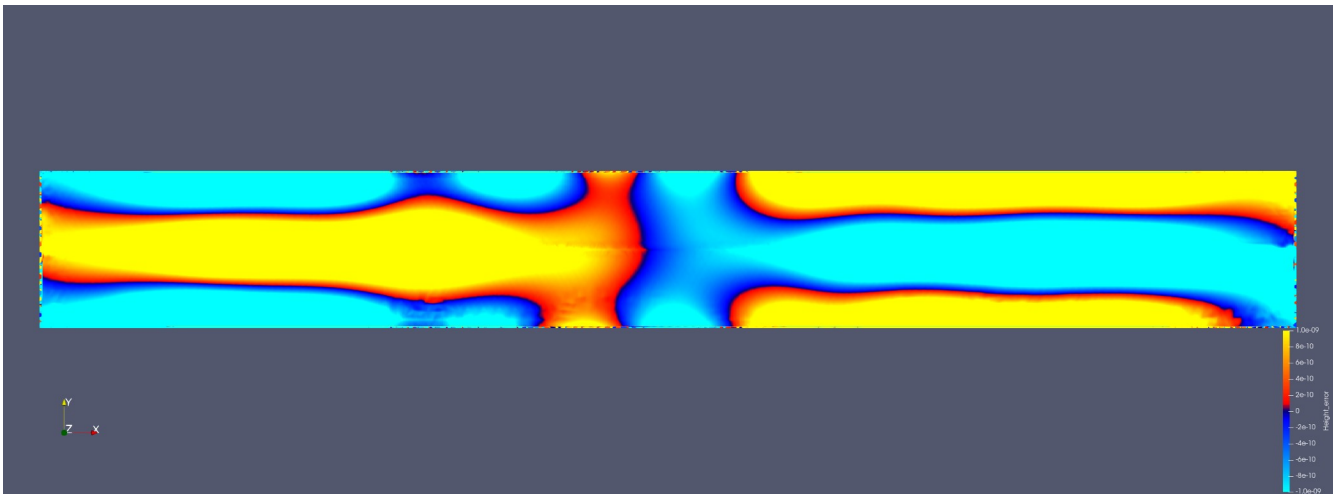

Figure 62: CharLens 0.0025 vs. 0.00175 Height error

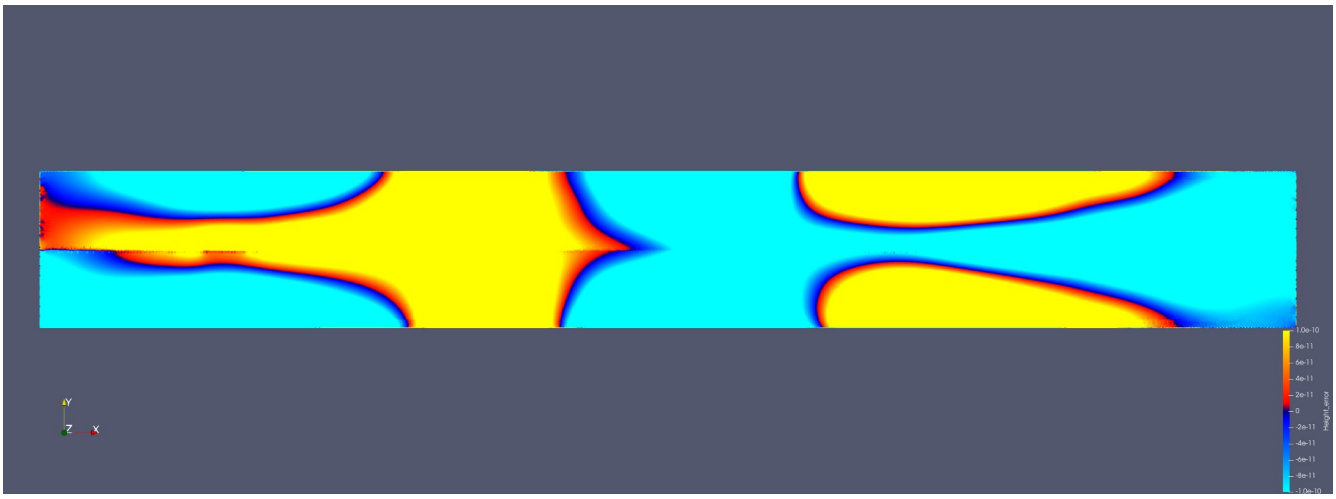

Figure 63: CharLens 0.001 vs. 0.00075 Height error

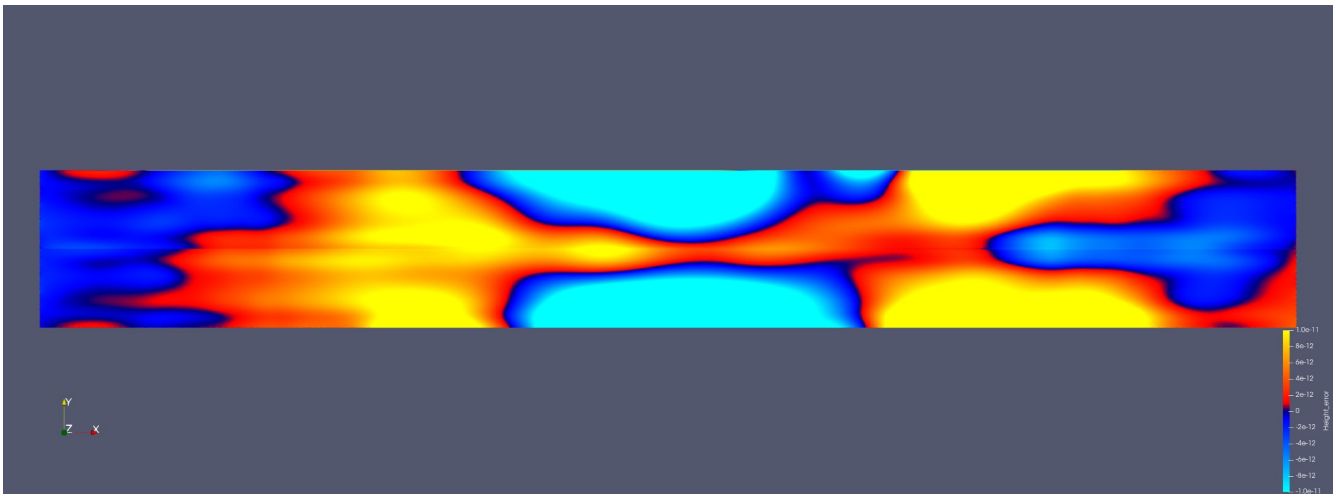

Figure 64: CharLens 0.00025 vs. 0.000175 Height error

X Velocity Error

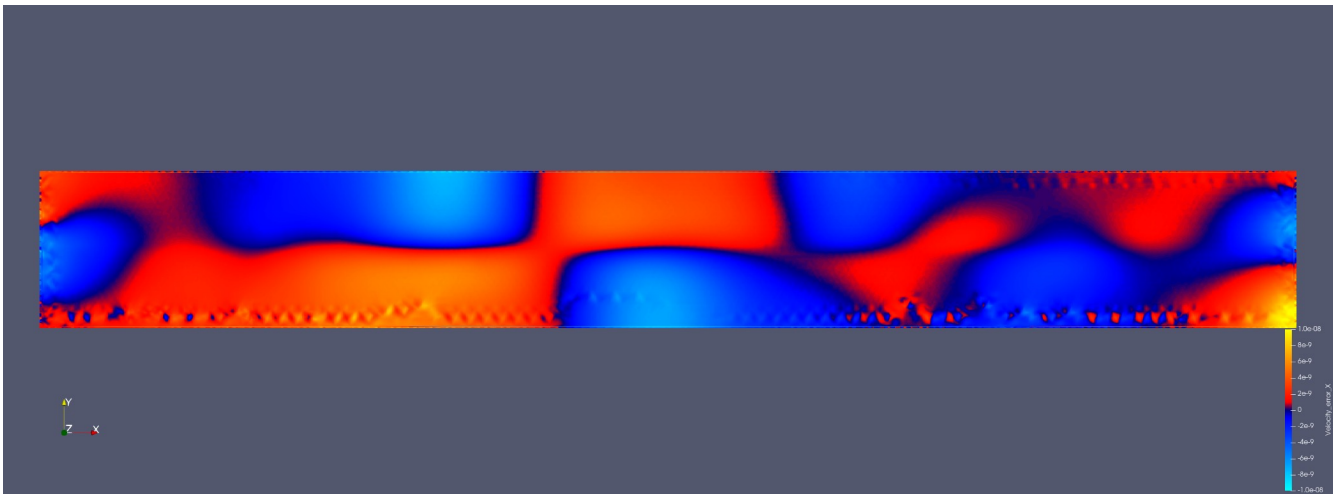

Figure 65: CharLens 0.0025 vs. 0.00175 Vx error

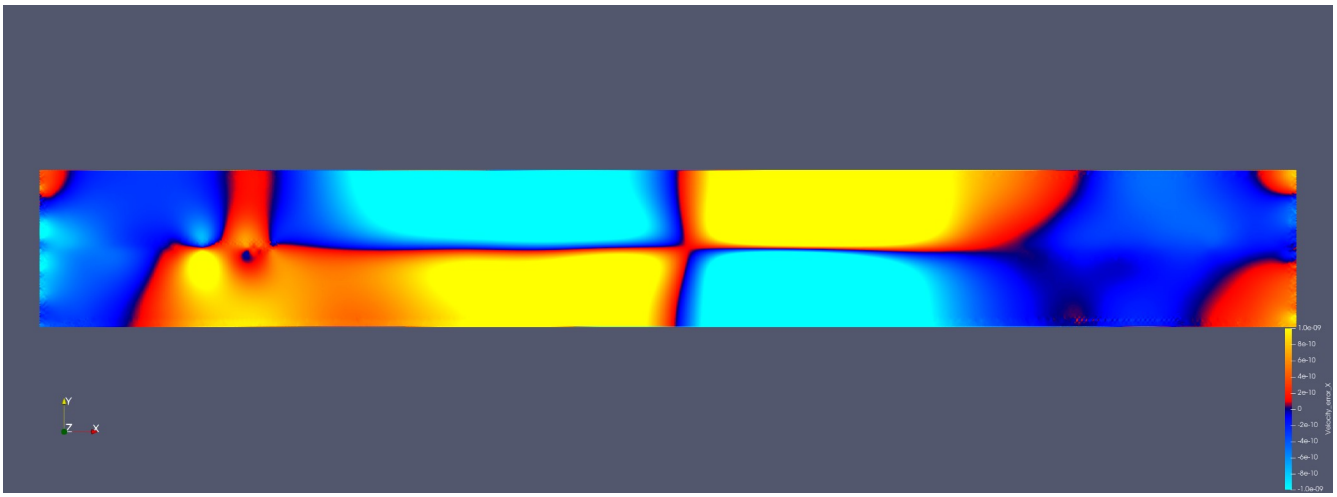

Figure 66: CharLens 0.001 vs. 0.00075 Vx error

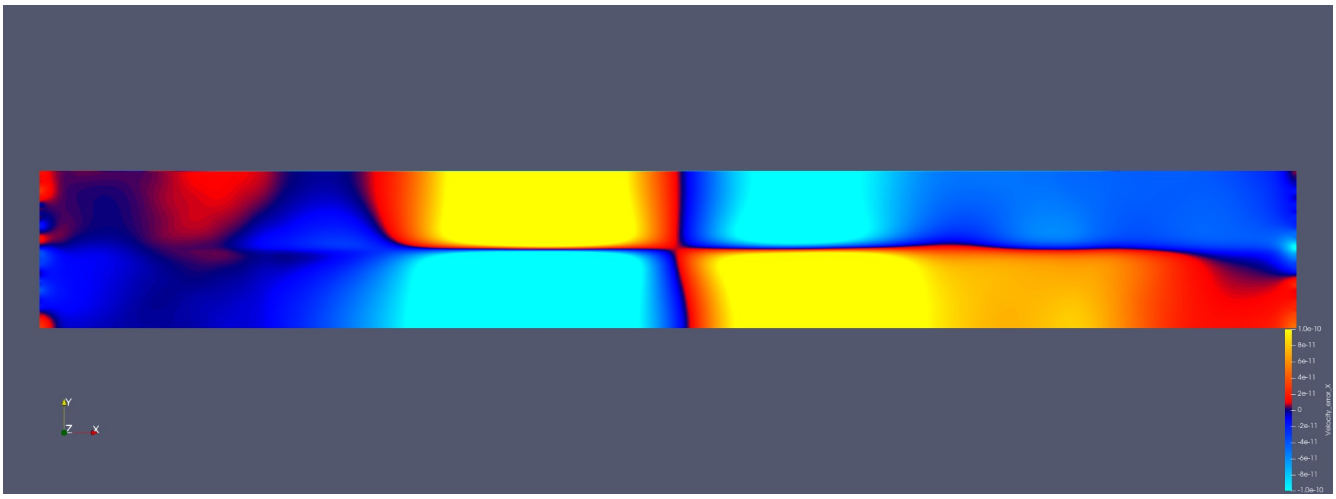

Figure 67: CharLens 0.00025 vs. 0.000175 Vx error

Y Velocity Error

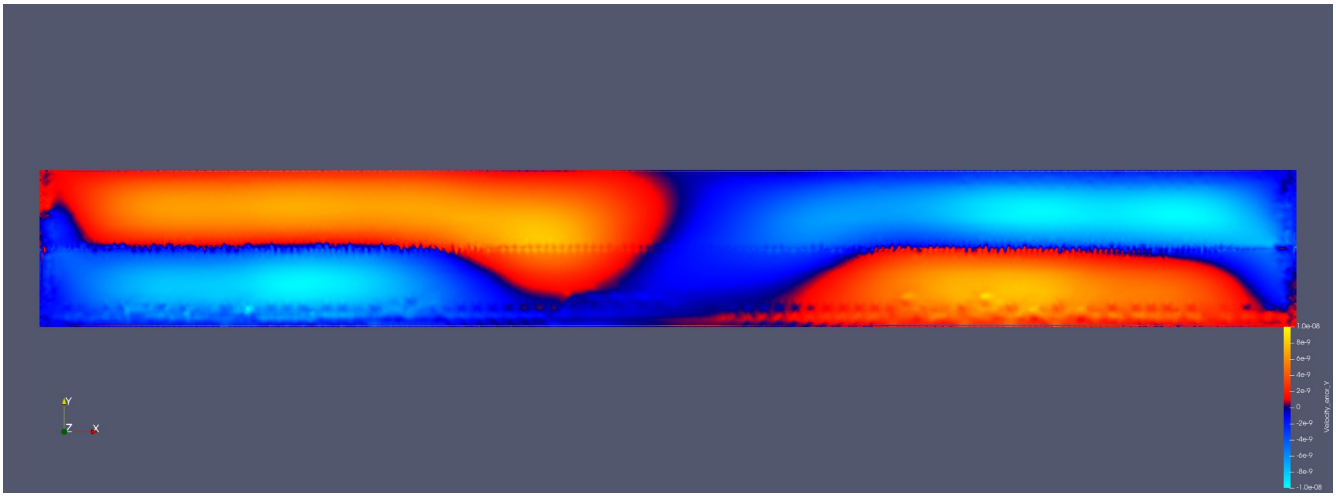

Figure 68: CharLens 0.0025 vs. 0.00175 Vy error

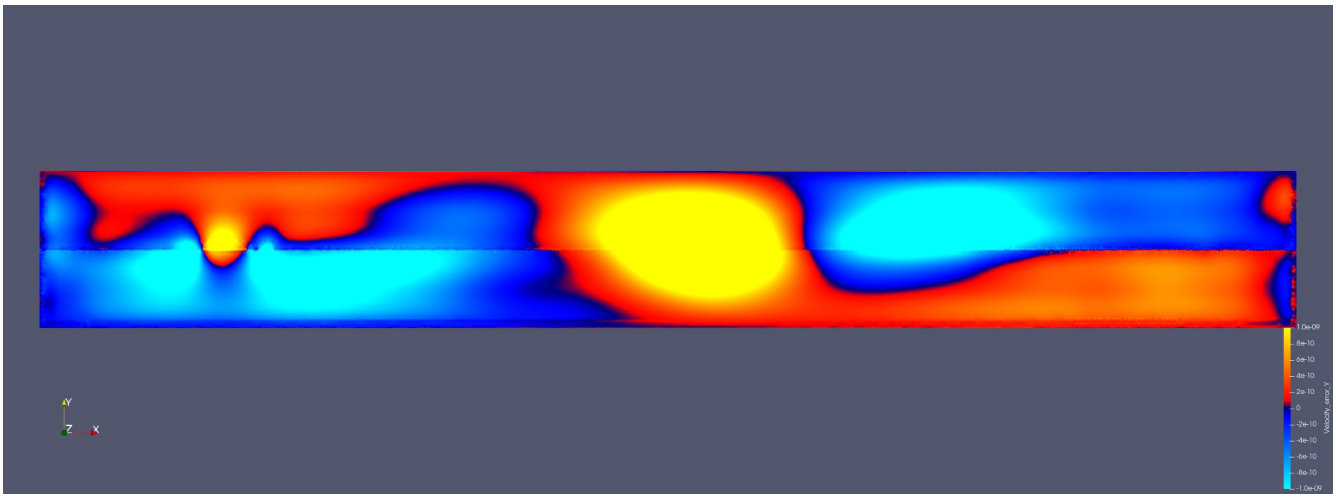

Figure 69: CharLens 0.001 vs. 0.00075 Vy error

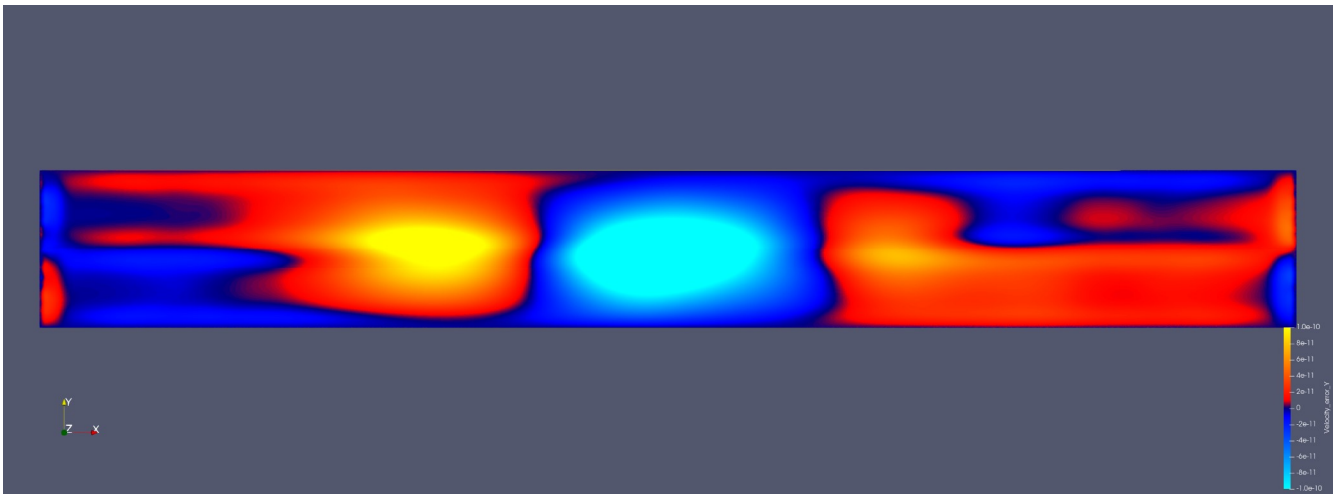

Figure 70: CharLens 0.00025 vs. 0.000175 Vy error

B. Code

UNSTRUCTURED MESH

```
#ifndef UNSTRUCTURED_MESH_H
#define UNSTRUCTURED_MESH_H
//////////////////////////////////////////////////////////////////////
//              Unstructured_Mesh.h
//              MCG4139 Winter 2021
//////////////////////////////////////////////////////////////////////
#include<vector>
#include<iostream>
#include<iomanip>
#include<fstream>
#include<string>
#include<stdexcept>
#include<sstream>
#include<Eigen/Core>
//////////////////////////////////////////////////////////////////////
//              Some useful types (useful everywhere)
//////////////////////////////////////////////////////////////////////
struct Cell {
  int node0;
  int node1;
  int node2;
  int line01_type;
  int line12_type;
  int line20_type;
  int id;
};
//////////////////////////////////////////////////////////////////////
struct Edge {
  int node0;
  int node1;
  int cell_l;
  int cell_r;
};
//////////////////////////////////////////////////////////////////////
struct Boundary {
  int node0;
  int node1;
  int type;
};
//////////////////////////////////////////////////////////////////////
```

```
using Node2D = Eigen::Vector2d;

////////////////////////////////////////////////////////////////////////
//   n_hat_and_length (usefull function [anywhere given two nodes])
////////////////////////////////////////////////////////////////////////
auto n_hat_and_length(const Node2D& n0, const Node2D& n1) {

  auto length = (n1-n0).norm();
  Node2D n_hat;
  n_hat.x() =  (n1.y()-n0.y())/length;
  n_hat.y() = -(n1.x()-n0.x())/length;
  return std::make_pair(n_hat, length);
}

////////////////////////////////////////////////////////////////////////
//          Output BC to .dat file (diagnostic)
////////////////////////////////////////////////////////////////////////
auto print_BC(const std::vector<Boundary>& BC) {

  std::string filename = "BC.dat";
  std::cout << "Writing output to " << filename << ".\n";
  std::ofstream fout(filename);
  if(!fout) {
    throw std::runtime_error("Could not open file: " + filename);
  }

  fout.precision(16);
  fout << std::setw(30) << "node0"
       << std::setw(30) << "node1"
       << std::setw(30) << "type" << "\n";

  for(int i = 0; i < static_cast<int>(BC.size()); ++i){
    Boundary b = BC[i];
    fout << std::setw(30) << b.node0
            << std::setw(30) << b.node1
            << std::setw(30) << b.type << "\n";
  }
}

////////////////////////////////////////////////////////////////////////
//        Output cells to .dat file (diagnostic)
////////////////////////////////////////////////////////////////////////
auto print_cells(const std::vector<Cell>& cells) {

  std::string filename = "cells.dat";
  std::cout << "Writing output to " << filename << ".\n";
  std::ofstream fout(filename);
```

```
  if(!fout) {
    throw std::runtime_error("Could not open file: " + filename);
  }

  fout.precision(16);

  fout << std::setw(30) << "node0"
       << std::setw(30) << "node1"
       << std::setw(30) << "node2"
       << std::setw(30) << "line01_type"
       << std::setw(30) << "line12_type"
       << std::setw(30) << "line20_type" << "\n";

  for(int i = 0; i < static_cast<int>(cells.size()); ++i){
    Cell cell = cells[i];
    fout << std::setw(30) << cell.node0
            << std::setw(30) << cell.node1
            << std::setw(30) << cell.node2
            << std::setw(30) << cell.line01_type
            << std::setw(30) << cell.line12_type
            << std::setw(30) << cell.line20_type << "\n";
  }
}

////////////////////////////////////////////////////////////////////////
//                Read gmsh file (for solver)
////////////////////////////////////////////////////////////////////////
auto read_gmsh_file(const std::string& filename) {

  std::ifstream fin(filename);
  if(!fin) {
    throw std::runtime_error("Cannot open file: " + filename + ".");
  }
  ////////////////////////////////////////////////////
  // Lambda function to consume lines that are expected
  // but should be ignored
  auto expect_line = [filename] (std::ifstream& fin, const std::string& expected) {
    std::string s;
    do {
      std::getline(fin,s);
    } while (s.empty());

    if(s != expected) {
      throw std::runtime_error("Error reading file: " + filename + ".\n" +
                               "Expected \"" + expected + "\", but got \"" + s +
                               "\".");
    }
```

```
};
std::cout << "\n\nReading gmsh file: " << filename << '\n';

//////////////////////////////////////////////////////
// Read file
expect_line(fin, "$MeshFormat");
expect_line(fin, "2.2 0 8");
expect_line(fin, "$EndMeshFormat");
expect_line(fin, "$Nodes");

int number_of_nodes;
fin >> number_of_nodes;

std::vector<Node2D> nodes(number_of_nodes);
for(int i = 0; i < number_of_nodes; ++i) {
  int    dummy_index;
  double dummy_z_coordinate;
  fin >> dummy_index
      >> nodes[i].x()
      >> nodes[i].y()
      >> dummy_z_coordinate;
  if(dummy_index != i+1) {
    throw std::runtime_error("Error with node index.");
  }
  if(fabs(dummy_z_coordinate) > 1.0e-12) {
    throw std::runtime_error("Error, node has z component.");
  }
}

expect_line(fin, "$EndNodes");
expect_line(fin, "$Elements");

int number_of_elements; //not all will be cells
fin >> number_of_elements;

std::vector<Cell> cells;
std::vector<Boundary> BC;
for(int i = 0; i < number_of_elements; ++i) {
  std::string s;
  do {
    std::getline(fin,s);
  } while (s.empty());

  std::istringstream ss(s);

  int element_num;
  ss >> element_num;
```

```
  if(element_num - 1 != i) {
    throw std::runtime_error("Error reading element number.");
  }

  int element_type;
  ss >> element_type;

  if(element_type == 1){
    Boundary tmp_b;
    int dummy;
    ss >> dummy >> tmp_b.type >> dummy >> tmp_b.node0 >> tmp_b.node1;
    tmp_b.node0 -= 1;
    tmp_b.node1 -= 1;
    BC.push_back(tmp_b);
  }

  if(element_type == 2) {
    //triangular cell
    Cell c;
    int dummy;
    int cell_id;
    ss >> dummy >> cell_id >> dummy >> c.node0 >> c.node1 >> c.node2;
    c.node0 -= 1;
    c.node1 -= 1;
    c.node2 -= 1;
    c.id = cell_id;

    auto assign_bc = [&c](int node_a, int node_b, int &bc_value_a, const std::vector<Boundary>
&bc_data) {
        auto it = std::find_if(bc_data.begin(), bc_data.end(), [node_a, node_b](const Boundary &bc){
          if(bc.node0 == node_a && bc.node1 == node_b) return true;
          if(bc.node0 == node_b && bc.node1 == node_a) return true;
          return false;
        });
        if(it != bc_data.end()){
          bc_value_a = -it->type;
        }else {
          bc_value_a = -1;
        }
    };

    assign_bc(c.node0, c.node1, c.line01_type, BC);
    assign_bc(c.node1, c.node2, c.line12_type, BC);
    assign_bc(c.node2, c.node0, c.line20_type, BC);

    cells.push_back(c);
  }
```

```
  }
  ////////////////////////////
  // Diagnostic
  //print_BC(BC);
  //print_cells(cells);

  expect_line(fin, "$EndElements");

  std::cout << "done.\n";

  return std::make_pair(nodes, cells);

}

//////////////////////////////////////////////////////////////////////
//        Compute Edges (for read_gmsh -> for solver)
//////////////////////////////////////////////////////////////////////
auto compute_edges(const std::vector<Node2D>& nodes, const std::vector<Cell>& cells) {

  std::cout << "Computing Edges.\n";
  double target = 0.01;

  std::vector<Edge> edges;
  std::vector<Edge> background_edges;

  for(int i = 0; i < static_cast<int>(cells.size()); ++i) {

    const auto& cell = cells[i];

    //////////////////////////////////////////////////////////////////
    //   fluid domain edges
    if(cell.id == 200){
      auto e = std::find_if(edges.begin(), edges.end(), [&cell] (const Edge& edge) {
          if(cell.node0 == edge.node0 && cell.node1 == edge.node1) return true;
          if(cell.node0 == edge.node1 && cell.node1 == edge.node0) return true;
          return false;
      });

      if(e != edges.end()) { //it was found
          e->cell_r = i;
      } else { // new edge
          edges.push_back({cell.node0, cell.node1, i, cell.line01_type});
      }

      e = std::find_if(edges.begin(), edges.end(), [&cell] (const Edge& edge) {
          if(cell.node1 == edge.node0 && cell.node2 == edge.node1) return true;
          if(cell.node1 == edge.node1 && cell.node2 == edge.node0) return true;
```

```
        return false;
    });

    if(e != edges.end()) { //it was found
        e->cell_r = i;
    } else { // new edge
        edges.push_back({cell.node1, cell.node2, i, cell.line12_type});
    }

    e = std::find_if(edges.begin(), edges.end(), [&cell] (const Edge& edge) {
        if(cell.node2 == edge.node0 && cell.node0 == edge.node1) return true;
        if(cell.node2 == edge.node1 && cell.node0 == edge.node0) return true;
        return false;
    });

    if(e != edges.end()) { //it was found
        e->cell_r = i;
    } else { // new edge
        edges.push_back({cell.node2, cell.node0, i, cell.line20_type});
    }
  } else if(cell.id == 100){
    ////////////////////////////////////////////////////////////////////
    //    background edges
    auto e = std::find_if(background_edges.begin(), background_edges.end(), [&cell] (const Edge&
edge) {
        if(cell.node0 == edge.node0 && cell.node1 == edge.node1) return true;
        if(cell.node0 == edge.node1 && cell.node1 == edge.node0) return true;
        return false;
    });

    if(e != background_edges.end()) { //it was found
        e->cell_r = i;
    } else { // new edge
        background_edges.push_back({cell.node0, cell.node1, i, cell.line01_type});
    }

    e = std::find_if(background_edges.begin(), background_edges.end(), [&cell] (const Edge& edge) {
        if(cell.node1 == edge.node0 && cell.node2 == edge.node1) return true;
        if(cell.node1 == edge.node1 && cell.node2 == edge.node0) return true;
        return false;
    });

    if(e != background_edges.end()) { //it was found
        e->cell_r = i;
    } else { // new edge
        background_edges.push_back({cell.node1, cell.node2, i, cell.line12_type});
```

```
    }

    e = std::find_if(background_edges.begin(), background_edges.end(), [&cell] (const Edge& edge) {
        if(cell.node2 == edge.node0 && cell.node0 == edge.node1) return true;
        if(cell.node2 == edge.node1 && cell.node0 == edge.node0) return true;
        return false;
    });

    if(e != background_edges.end()) { //it was found
        e->cell_r = i;
    } else { // new edge
        background_edges.push_back({cell.node2, cell.node0, i, cell.line20_type});
    }

    if(static_cast<double>(i+1)/static_cast<double>(cells.size()) >= target) {
        std::cout << i+1 << "/" << cells.size() << " cells processed.\n";
        std::cout.flush();
        target += 0.01;
    }
  }
}

std::cout << "All cells processed.\n";

//make sure all edges are properly aligned
// (that the unit normal points from cell_l to cell_r)
for(auto& edge: edges) {
    using std::swap;

    const auto edge_centroid = 0.5*(nodes[edge.node0]+nodes[edge.node1]);

    const auto& cell_left = cells[edge.cell_l];  //cell_left is never -1
    const auto left_cell_centroid = ( nodes[cell_left.node0]
                                    +nodes[cell_left.node1]
                                    +nodes[cell_left.node2])/3.0;

    const auto vec1 = left_cell_centroid - edge_centroid;

    auto n_hat_l = n_hat_and_length(nodes[edge.node0],nodes[edge.node1]);
    const auto& n_hat = n_hat_l.first;

    if(n_hat.dot(vec1) > 0.0) swap(edge.cell_l, edge.cell_r);

}


return std::make_pair(edges, background_edges);
```

```
}

////////////////////////////////////////////////////////////////////////
//        Output edges to .dat file (diagnostic)
////////////////////////////////////////////////////////////////////////
auto print_edges(const std::vector<Edge>& edges) {

  std::string filename = "edges.dat";
  std::cout << "Writing output to " << filename << ".\n";
  std::ofstream fout(filename);
  if(!fout) {
    throw std::runtime_error("Could not open file: " + filename);
  }

  fout.precision(16);


  fout << std::setw(30) << "node0"
       << std::setw(30) << "node1"
       << std::setw(30) << "cell_l"
       << std::setw(30) << "cell_r" << "\n";

  for(int i = 0; i < static_cast<int>(edges.size()); ++i){
    Edge edge = edges[i];
    fout << std::setw(30) << edge.node0
            << std::setw(30) << edge.node1
            << std::setw(30) << edge.cell_l
            << std::setw(30) << edge.cell_r << "\n";
  }
}

#endif //o#ifndef UNSTRUCTURED_MESH_H
```

SHALLOW WATER

```
#ifndef SHALLOW_WATER_H
#define SHALLOW_WATER_H
//////////////////////////////////////////////////////////////////////
//                Shallow_Water.h
//                MCG4139 Winter 2021
//////////////////////////////////////////////////////////////////////
#include<Eigen/Core>
#include<cmath>

//////////////////////////////////////////////////////////////////////
//                 Shallow_Water_Equations
//////////////////////////////////////////////////////////////////////
class Shallow_Water_Equations {
public:

  using Vector_type = Eigen::Vector3d;
  static constexpr int number_of_unknowns = 3;

  ////////////////////////////////////
  // get_primitive
  template<typename Vec_in>
  static Vector_type get_conserved(const Vec_in& U, const int& cell_id){
    if(cell_id == 200){
      return {U[0],
              U[1]/U[0],
              U[2]/U[0]};
    } else {
      Vector_type U_zero;
      U_zero.setZero();
      return U_zero;
    }

  }

  ////////////////////////////////////
  //  Flux x
  template<typename Vec_in>
  static Vector_type Fx(const Vec_in& U) {
    Vector_type F;
    F[0] = U[1];
    F[1] = U[1]*U[1]/U[0] + 0.5*g*U[0]*U[0];
    F[2] = U[1]*U[2]/U[0];
```

```
  return F;
}

////////////////////////////////////
//  Rotate
template<typename Vec_in>
static Vector_type rotate(const Vec_in& U, Node2D n_hat) {
  Vector_type U_rot;
  U_rot[0] =  U[0];
  U_rot[1] =  U[1]*n_hat.x() + U[2]*n_hat.y();
  U_rot[2] = -U[1]*n_hat.y() + U[2]*n_hat.x();
  return U_rot;
}

////////////////////////////////////
//  Rotate back
template<typename Vec_in>
static Vector_type rotate_back(const Vec_in& U, Node2D n_hat) {
  Vector_type U_rot;
  U_rot[0] = U[0];
  U_rot[1] = U[1]*n_hat.x() - U[2]*n_hat.y();
  U_rot[2] = U[1]*n_hat.y() + U[2]*n_hat.x();
  return U_rot;
}

////////////////////////////////////
//  reflect_x
template<typename Vec_in>
static Vector_type reflect_x(const Vec_in& U) {
  Vector_type U_ref = U;
  U_ref[1] *= -1.0;
  return U_ref;
}

////////////////////////////////////
//  inlet_x (also just reflection)
template<typename Vec_in>
static Vector_type inlet_x(const Vec_in& U) {
  Vector_type U_in;
  U_in[0] = U[0];
  U_in[1] = U[1];//0.0;//1.0e-3;
  U_in[2] = 0.0;
  return U_in;
}

////////////////////////////////////
//  outlet_x (also just reflection)
```

```
  template<typename Vec_in>
  static Vector_type outlet_x(const Vec_in& U) {
    Vector_type U_out;
    U_out[0] = U[0];
    U_out[1] = U[1];
    U_out[2] = 0.0;
    return U_out;
  }

  ////////////////////////////////////
  //  no_slip_wall_x (just another reflection for shallow water eq.)
  template<typename Vec_in>
  static Vector_type no_slip_wall_x(const Vec_in& U) {
    Vector_type U_wall = U;
    const double ff = 1.0;
    U_wall[1]  *= -ff;
    return U_wall;
  }

  ////////////////////////////////////
  //       Source term
  template<typename Vec_in, typename EM_in>
  static Vector_type source(const Vec_in& U, const EM_in& em) {
    Vector_type S;
    const double source_multiplier = U[0]*magnetic_saturization*magnetic_fraction/rho;
    S[0] = 0.0;
    S[1] = source_multiplier*em[0];
    S[2] = source_multiplier*em[1];
    return S;
  }

  ////////////////////////////////////
  //  max_lambda_x
  template<typename Vec_in>
  static double max_lambda_x(const Vec_in& U) {
    return fabs(U[1]/U[0])+sqrt(U[0]*g);
  }

private:
  static constexpr double             tolerance = 1.0e-12;
  static constexpr double                   g = 9.81;
  static constexpr double                 rho = 1000;
  static constexpr double   magnetic_saturization = 375000.0;
  static constexpr double      magnetic_fraction = 0.1;
  static constexpr double permeability_free_space = (4*acos(-1.0))*0.0000001;
};
```

```
#endif //#ifndef SHALLOW_WATER_H
```

EM BACKGROUND

```
#ifndef EM_BACKGROUND_H
#define EM_BACKGROUND_H
//////////////////////////////////////////////////////////////////////
//                Shallow_Water.h
//                MCG4139 Winter 2021
//////////////////////////////////////////////////////////////////////
#include<vector>
#include<Eigen/Core>
#include"Unstructured_Mesh.h"

struct wire{
  std::vector<int> cells;
  Node2D centroid;

  void set_zero(){
    cells.clear();
    centroid.setZero();
  }
};

//////////////////////////////////////////////////////////////////////
//                Electric Magnetic Background
//////////////////////////////////////////////////////////////////////
class EM_Background {
public:

  ////////////////////////////////////////////////////////////////////
  //  Default Constructors, etc.
  EM_Background() = default;
  EM_Background(const EM_Background&) = default;
  EM_Background(EM_Background&&) = default;
  EM_Background& operator=(const EM_Background&) = default;
  EM_Background& operator=(EM_Background&&) = default;

  using Vector_type = Eigen::Vector3d;
  static constexpr int number_of_unknowns = 3;

  ///////////////////////////////////
  //  Solution Initialization

  std::vector<Vector_type> solution(const std::vector<Cell>& edges, const std::vector<Node2D>&
```

```
centroids);

private:
  static constexpr double                current = 1.0e2;       // Magnitude of the current
  static constexpr int         magnetic_direction = 1;       // Which way do we want the field lines to go - follows right hand rule - positive wires in +y will have current (*) if this is positive
  const double                     wire_diameter = 0.01;       // Needs to be changed based on wire diameter from the mesh
  static constexpr double permeability_free_space = (4*acos(-1.0))*0.0000001;
};

//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
//          Generate magnetic field within domain based on the wires.
////////////////////////////////////////////////////////////////////////
std::vector<EM_Background::Vector_type> EM_Background::solution(const std::vector<Cell>& cells, const std::vector<Node2D>& centroids) {

  std::vector<Vector_type> magnetic_field;
  std::vector<wire> wires;
  std::vector<int> wire_cells;

  ////////////////////////////////////////////////////////////////////
  //              Find cells bordering a wire
  for(int i = 0; i < static_cast<int>(cells.size()); ++i) {
    if(cells[i].id == 300) {
      wire_cells.push_back(i);
    }
  }

  ////////////////////////////////////////////////////////////////////
  //            Function declerations
  auto distance = [](const Node2D& node0, const Node2D& node){
    const double x = (node0.x()-node.x());
    const double y = (node0.y()-node.y());
    const double distance = sqrt(x*x+y*y);
    //std::cout << "distance = " << distance << "\n" << std::flush;
    return distance;
  };
```

```
  auto find_centroid = [&centroids](const std::vector<int>& cells){
    Node2D wire_centroid;
    const int precision = 1;
    wire_centroid.setZero();
    for(const auto& cell : cells){
      wire_centroid.x() += centroids[cell].x();
      wire_centroid.y() += centroids[cell].y();
    }
    wire_centroid.x() = std::round(wire_centroid.x()*std::pow(10, precision))/(cells.size()*std::pow(10,
precision));
    wire_centroid.y() = std::round(wire_centroid.y()*std::pow(10, precision))/(cells.size()*std::pow(10,
precision));
    return wire_centroid;
  };

  auto set_magnetic_field = [&wires, &magnetic_field, &centroids](){
    const auto PI = acos(-1.0);
    magnetic_field.resize(static_cast<int>(centroids.size()));
    for(int i = 0; i < static_cast<int>(centroids.size()); ++i) {
      for(auto& wire : wires){
          Vector_type local_field;
          const double radius = (wire.centroid - centroids[i]).norm();
          const auto cell_direction = (wire.centroid - centroids[i])/radius;
          auto field_direction = Eigen::Vector2d(cell_direction.y(), -cell_direction.x());
          if(magnetic_direction < 0){
            field_direction.x() *= -1.0;
            field_direction.y() *= -1.0;
          }
          if(centroids[i].y() < 0.0){
            field_direction.x() *= -1.0;
            field_direction.y() *= -1.0;
          }
          const double magnitude = permeability_free_space*current/(2*PI*radius);
          local_field[0] = magnitude*field_direction.x();
          local_field[1] = magnitude*field_direction.y();
          local_field[2] = 0.0;
          magnetic_field[i] += local_field;
      }
    }
  };

  ////////////////////////////////////////////////////////////////////////
  //          Seperating wires and finding centroids
  wire new_wire;
  int node0;

  for(int i = 0; i < static_cast<int>(wire_cells.size()); ++i){
```

```
    if(i == 0){
      new_wire.cells.push_back(wire_cells[i]);
      wires.push_back(new_wire);
    }

    if(i != 0){

      bool added_new_wire = false;

      for(auto& wire : wires){

            node0 = wire.cells[0];
            if(distance(centroids[node0], centroids[wire_cells[i]]) <= wire_diameter){

              wire.cells.push_back(wire_cells[i]);
              added_new_wire = true;
              break;
            }
      }

      if(!added_new_wire){

            new_wire.set_zero();
            new_wire.cells.push_back(wire_cells[i]);
            wires.push_back(new_wire);
      }
    }
  }

  for(auto& wire : wires){
    wire.centroid = find_centroid(wire.cells);
  }
  ////////////////////////////////////////////////////////////////////////
  //         Print the centroids (diagnostic)
  std::cout << "\n" << "# of wires: " << static_cast<int>(wires.size()) << "\n" << std::flush;
  //std::cout << "\nWire centroids:\n" << std::flush;
  //for(const auto& wire : wires){
  //  std::cout << "wire centroid x = " << wire.centroid.x() << " , y = " << wire.centroid.y() << "\n" <<
std::flush;
  //}
  std::cout << "\n" << std::flush;

  ////////////////////////////////////////////////////////////////////////
  //          Setup and return of the magnetic field
  set_magnetic_field();
```

```
  return magnetic_field;
}
#endif //#ifndef EM_BACKGROUND_H
```

UNSTRUCTURED FV SOLVER

```
#ifndef UNSTRUCTURED_FV_SOLVER_H
#define UNSTRUCTURED_FV_SOLVER_H
////////////////////////////////////////////////////////////////////////
//                Unstructured_FV_Solver.h
//                MCG4139 Winter 2021
////////////////////////////////////////////////////////////////////////
#include<cmath>
#include<string>
#include<functional>
#include<Eigen/Core>
#include"Unstructured_Mesh.h"
#include"EM_Background.h"

////////////////////////////////////////////////////////////////////////
//                  Local Lax-Friedrichs
////////////////////////////////////////////////////////////////////////
template<typename PDE_type, typename Vec_type>
auto Local_Lax_Friedrichs(const Vec_type& Ul, const Vec_type& Ur) {
  const auto Fl = PDE_type::Fx(Ul);
  const auto Fr = PDE_type::Fx(Ur);
  const auto max_lambda = std::max(PDE_type::max_lambda_x(Ul),PDE_type::max_lambda_x(Ur));
  typename PDE_type::Vector_type F = 0.5*(Fl+Fr-max_lambda*(Ur-Ul));
  return F;
}

////////////////////////////////////////////////////////////////////////
//            Unstructured Finite-Volume Scheme
////////////////////////////////////////////////////////////////////////
template<typename PDE_type>
class Unstructured_FV_Solver : public EM_Background {
public:


  using SolutionVector_type = typename PDE_type::Vector_type;
  using BackgroundVector_type = EM_Background::Vector_type;

  ////////////////////////////////////////////////////////////////////
  //  Default Constructors, etc.
  Unstructured_FV_Solver() = default;
  Unstructured_FV_Solver(const Unstructured_FV_Solver&) = default;
```

```
Unstructured_FV_Solver(Unstructured_FV_Solver&&) = default;
Unstructured_FV_Solver& operator=(const Unstructured_FV_Solver&) = default;
Unstructured_FV_Solver& operator=(Unstructured_FV_Solver&&) = default;

////////////////////////////////////////////////////////////////////////
//  Constructor taking a filename and initial condition
Unstructured_FV_Solver(const std::string& mesh_filename,
                       const std::function<SolutionVector_type(const Node2D&)>& ic);

////////////////////////////////////////////////////////////////////////
//  Solution vector in cell "i"
auto& U(int i) {
  return Global_U[i];
}

////////////////////////////////////////////////////////////////////////
//  Solution vector in cell "i"
auto& S(int i) {
  return Global_S[i];
}

////////////////////////////////////////////////////////////////////////
//  dUdt in cell "i"
auto& dUdt(int i) {
  return Global_dUdt[i];
}

////////////////////////////////////////////////////////////////////////
//  Background  vector in cell "i"
auto& EM(int i) {
  return Global_EM[i];
}

auto& EM() {
  return Global_EM;
}

auto& centroids(int i){
  return Global_centroids[i];
}

////////////////////////////////////////////////////////////////////////
//  number of cells, nodes, and edges.
auto number_of_cells() {return static_cast<int>(cells.size());}
auto number_of_nodes() {return static_cast<int>(nodes.size());}
auto number_of_edges() {return static_cast<int>(edges.size());}
auto number_of_background_edges() {return static_cast<int>(background_edges.size());}
```

```
  ////////////////////////////////////////////////////////////////////
  //  time march to time
  void time_march_to_time(double final_time, double CFL);

  ////////////////////////////////////////////////////////////////////
  //  Make movie
  void make_movie(double final_time, double CFL, int num_frames, const std::string& file_location,
const std::string& filename);

  ////////////////////////////////////////////////////////////////////
  //  Output two vector max and fluid solution vector
  void Output_Vector_min_max(const std::string& file_location, const std::string& filename);
  void Output_Vector(const std::string& file_location, const std::string& filename);

  ////////////////////////////////////////////////////////////////////
  //  write to VTK
  void write_to_vtk(const std::string& filename);

private:

  ////////////////////////////////////////////////////////////////////
  //  Member variables
  double                              h_vel;
  double                              h_amp;
  double                              e_amp;
  std::vector<Node2D>                     nodes;
  std::vector<Cell>                     cells;
  std::vector<Edge>                     edges;
  std::vector<Edge>             background_edges;
  std::vector<double>                     areas;
  std::vector<Node2D>           Global_centroids;
  std::vector<SolutionVector_type>       Global_U;
  std::vector<SolutionVector_type>       Global_S;
  std::vector<SolutionVector_type>    Global_dUdt;
  std::vector<BackgroundVector_type>    Global_EM;
  //EM_base_vector_type             Global_EM_base;
  double                               time;

  ////////////////////////////////////////////////////////////////////
  //  compute all areas
  void compute_areas() {
    areas.resize(number_of_cells());
    for(int i = 0; i < number_of_cells(); ++i) {
      const Cell& cell = cells[i];
      const Node2D& n0 = nodes[cell.node0];
      const Node2D& n1 = nodes[cell.node1];
```

```
      const Node2D& n2 = nodes[cell.node2];
      areas[i] = 0.5*fabs( n0.x()*n1.y()-n0.y()*n1.x()
                          +n1.x()*n2.y()-n1.y()*n2.x()
                          +n2.x()*n0.y()-n2.y()*n0.x());
    }
  }
};

////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////
//  Constructor
template<typename PDE_type>
Unstructured_FV_Solver<PDE_type>::Unstructured_FV_Solver(const std::string& mesh_filename,
                                        const std::function<SolutionVector_type(const Node2D&)>& ic) {

  ////////////////////////////////////////////////////////////
  // Read gmsh file & compute mesh vectors for solution
  std::tie(nodes, cells) = read_gmsh_file(mesh_filename);
  std::tie(edges, background_edges) = compute_edges(nodes,cells);

  ////////////////////////////////////////////////////////////
  // Diagnostics
  //print_edges(edges);
  //print_cells(cells);

  ////////////////////////////////////////////////////////////////////////////
  // Resizing of vectors for our solver
  Global_U.resize(number_of_cells());
  Global_dUdt.resize(number_of_cells());
  Global_S.resize(number_of_cells());
  Global_EM.resize(number_of_cells());
  Global_centroids.resize(number_of_cells());
  ////////////////////////////////////////////////////////////////////////////
  // Initialization
  time = 0.0;
  for(int i = 0; i < number_of_cells(); ++i) {
    centroids(i) = (nodes[cells[i].node0]+nodes[cells[i].node1]+nodes[cells[i].node2])/3.0;
    if(cells[i].id == 200){
```

```
      U(i) = ic(centroids(i));
    } else {
      SolutionVector_type U_zero;
      U_zero.setZero();
      U(i) = U_zero;
    }
  }
  EM() = this->EM_Background::solution(cells, Global_centroids);
  compute_areas();
}

////////////////////////////////////////////////////////////////////
//  time march to time
template<typename PDE_type>
void Unstructured_FV_Solver<PDE_type>::time_march_to_time(double final_time, double CFL) {

  constexpr double tolerance = 1.0e-12;

  while(time < final_time - tolerance) {

    double dt = final_time-time;

    for(auto& entry : Global_dUdt) {
      entry.fill(0.0);
    }

    for(const auto& edge : edges) {

      double l;
      Node2D n_hat;
      std::tie(n_hat,l) = n_hat_and_length(nodes[edge.node0], nodes[edge.node1]);

      typename PDE_type::Vector_type Ul;
      typename PDE_type::Vector_type Ur;
      typename PDE_type::Vector_type Ul_rot;
      typename PDE_type::Vector_type Ur_rot;

      if(edge.cell_l != -100 && edge.cell_l != -101 && edge.cell_l != -102 && edge.cell_l != -105) {
        Ul = U(edge.cell_l);
        Ul_rot = PDE_type::rotate(Ul, n_hat);
      }

      if(edge.cell_r != -100 && edge.cell_r != -101 && edge.cell_r != -102 && edge.cell_r != -105) {
        Ur = U(edge.cell_r);
        Ur_rot = PDE_type::rotate(Ur, n_hat);
      }
```

```
    if(edge.cell_l == -100) {
      Ul_rot = PDE_type::inlet_x(Ur_rot);
          //Ul_rot = PDE_type::reflect_x(Ur_rot);
    }

    if(edge.cell_r == -100) {
      Ur_rot = PDE_type::inlet_x(Ul_rot);
          //Ur_rot = PDE_type::reflect_x(Ul_rot);
    }

    if(edge.cell_l == -101) {
      Ul_rot = PDE_type::outlet_x(Ur_rot);
          //Ul_rot = PDE_type::reflect_x(Ur_rot);
    }

    if(edge.cell_r == -101) {
      Ur_rot = PDE_type::outlet_x(Ul_rot);
          //Ur_rot = PDE_type::reflect_x(Ul_rot);
    }

    if(edge.cell_l == -102) {
      Ul_rot = PDE_type::no_slip_wall_x(Ur_rot);
    }

    if(edge.cell_r == -102) {
      Ur_rot = PDE_type::no_slip_wall_x(Ul_rot);
    }

    auto F_rot = Local_Lax_Friedrichs<PDE_type>(Ul_rot, Ur_rot);
    auto F = PDE_type::rotate_back(F_rot, n_hat);


    if(edge.cell_l != -100 && edge.cell_l != -101 && edge.cell_l != -102 && edge.cell_l != -105) {
          dUdt(edge.cell_l) -= F*l/areas[edge.cell_l];
          double max_lambda = PDE_type::max_lambda_x(Ul);
          dt = std::min(dt, CFL*sqrt(areas[edge.cell_l])/max_lambda);
    }

    if(edge.cell_r != -100 && edge.cell_r != -101 && edge.cell_r != -102 && edge.cell_r != -105) {
          dUdt(edge.cell_r) += F*l/areas[edge.cell_r];
          double max_lambda = PDE_type::max_lambda_x(Ur);
          dt
= std::min(dt, CFL*sqrt(areas[edge.cell_r])/max_lambda);
    }

   } //end loop over edges
```

```
    for(int i = 0; i < number_of_cells(); ++i) {
      if(cells[i].id == 200){
        typename PDE_type::Vector_type source = PDE_type::source(U(i), EM(i));
          U(i) += dt*(dUdt(i) + source);
      }
    }

    time += dt;
    std::cout << "Time = " << time << "    dt = " << dt << '\n';
  }
}


////////////////////////////////////////////////////////////////////////
//  Make movie
template<typename PDE_type>
void Unstructured_FV_Solver<PDE_type>::make_movie(double final_time,
                                        double CFL,
                                        int num_frames,
                                        const std::string& file_location,
                                                const std::string& filename) {

  const auto dt = final_time/static_cast<double>(num_frames-1);

  const std::string filename_base = file_location + "movie/movie" + filename + "_";

  auto get_name = [&filename_base] (int i) {
    std::stringstream ss;
    ss << filename_base << std::setfill('0') << std::setw(10) << i << ".vtk";
    return ss.str();
  };

  write_to_vtk(get_name(0));

  for(int i = 1; i < num_frames; ++i) {
    std::cout << "Time Marching to time = " << dt*static_cast<double>(i) << '\n';
    time_march_to_time(dt*static_cast<double>(i), CFL);
    write_to_vtk(get_name(i));
  }
  Output_Vector_min_max(file_location, filename);
  Output_Vector(file_location, filename);
}

////////////////////////////////////////////////////////////////////////
//          Output cell_types to .dat file (diagnostic)
////////////////////////////////////////////////////////////////////////
```

```
auto print_cell_types(const std::vector<int>& cell_types) {

  std::string filename = "cell_types.dat";
  std::cout << "Writing output to " << filename << ".\n";
  std::ofstream fout(filename);
  if(!fout) {
    throw std::runtime_error("Could not open file: " + filename);
  }

  fout.precision(16);
  fout << std::setw(30) << "cell #"
       << std::setw(30) << "type" << "\n";

  for(int i = 0; i < static_cast<int>(cell_types.size()); ++i){
    fout << std::setw(30) << i
         << std::setw(30) << cell_types[i] << "\n";
  }
}

////////////////////////////////////////////////////////////////////////
//          Output Solution vector min and max to .dat
////////////////////////////////////////////////////////////////////////
template<typename PDE_type>
void Unstructured_FV_Solver<PDE_type>::Output_Vector_min_max(const std::string& file_location,
const std::string& filename) {

  const std::string filename_base = file_location + "min_max/vector_min_max" + filename + ".dat";

  typename PDE_type::Vector_type maxValues = PDE_type::Vector_type::Zero();
  typename PDE_type::Vector_type minValues = PDE_type::Vector_type::Zero();

  for(int i = 0; i < number_of_cells(); ++i){
    if(cells[i].id == 200){
      const auto W = PDE_type::get_conserved(U(i), cells[i].id);

      maxValues[0] = std::max(maxValues[0], W[0]);
      maxValues[1] = std::max(maxValues[1], W[1]);
      maxValues[2] = std::max(maxValues[2], W[2]);

      minValues[0] = std::min(maxValues[0], W[0]);
      minValues[1] = std::min(maxValues[1], W[1]);
      minValues[2] = std::min(maxValues[2], W[2]);
    }
  }

  std::cout << "Writing min and max to " << filename_base << "\n";
  std::ofstream fout(filename_base);
```

```
  if(!fout) {
    throw std::runtime_error("Could not open file: " + filename_base);
  }

  fout.precision(16);
  fout << std::setw(30) << "Height_min"
       << std::setw(30) << "Height_max"
       << std::setw(30) << "X_velocity_min"
       << std::setw(30) << "X_velocity_max"
       << std::setw(30) << "Y_velocity_min"
       << std::setw(30) << "Y_velocity_max" << "\n";

  fout << std::setw(30) << minValues[0]
       << std::setw(30) << maxValues[0]
       << std::setw(30) << minValues[1]
       << std::setw(30) << maxValues[1]
       << std::setw(30) << minValues[2]
       << std::setw(30) << maxValues[2] << "\n";

}


////////////////////////////////////////////////////////////////////////
//          Output Solution vector  to .dat
////////////////////////////////////////////////////////////////////////
template<typename PDE_type>
void Unstructured_FV_Solver<PDE_type>::Output_Vector(const std::string& file_location, const
std::string& filename) {

  const std::string filename_base = file_location + "vector/vector" + filename + ".dat";

  std::cout << "Writing vector to " << filename_base << "\n";
  std::ofstream fout(filename_base);
  if(!fout) {
    throw std::runtime_error("Could not open file: " + filename_base);
  }

  fout.precision(16);

  fout << std::setw(30) << "X"
       << std::setw(30) << "Y"
       << std::setw(30) << "Height"
       << std::setw(30) << "X_velocity"
       << std::setw(30) << "Y_velocity" << "\n";;

  for(int i = 0; i < number_of_cells(); ++i){
    if(cells[i].id == 200){
```

```
    const auto W = PDE_type::get_conserved(U(i), cells[i].id);

    fout << std::setw(30) << centroids(i).x()
         << std::setw(30) << centroids(i).y()
         << std::setw(30) << W[0]
         << std::setw(30) << W[1]
         << std::setw(30) << W[2] << "\n";
   }
  }
}

////////////////////////////////////////////////////////////////////////
//  write to VTK
template<typename PDE_type>
void Unstructured_FV_Solver<PDE_type>::write_to_vtk(const std::string& filename) {

  std::ofstream fout(filename);
  if(!fout) {
    throw std::runtime_error("error opening vtk file.");
  }

  fout << "# vtk DataFile Version 2.0\n"
       << "Unstructured Solver\n"
       << "ASCII\n"
       << "DATASET UNSTRUCTURED_GRID\n"
       << "POINTS " << number_of_nodes() << " double\n";

  for(const auto& node : nodes) {
    fout << node.x() << " " << node.y() << " 0\n";
  }

  fout << "\nCELLS " << number_of_cells() << " " << 4*number_of_cells() << '\n';
  for(const auto& cell : cells) {
    fout << "3 " << cell.node0 << " " << cell.node1 << " " << cell.node2 << '\n';
  }

  fout << "\nCELL_TYPES " << number_of_cells() << '\n';
  for(int i = 0; i < number_of_cells(); ++i) {
    fout << "5\n";
  }

  fout << "\nCELL_DATA " << number_of_cells()  << '\n'
       << "SCALARS Height double 1\n"
       << "LOOKUP_TABLE default\n";

  for(int i = 0; i < number_of_cells(); ++i) {
    const auto W = PDE_type::get_conserved(U(i), cells[i].id);
```

```
  fout << W[0] << '\n';
}

fout << "VECTORS Velocity double\n";

for(int i = 0; i < number_of_cells(); ++i) {
  const auto W = PDE_type::get_conserved(U(i), cells[i].id);
  fout << W[1] << " " << W[2] << " 0.0" << '\n';
}

fout << "VECTORS Magnetic_field double\n";

for(int i = 0; i < number_of_cells(); ++i) {
  fout << EM(i)[0] << " " << EM(i)[1] << " 0.0" << '\n';
}

fout << "SCALARS Cell_type double 1\n"
     << "LOOKUP_TABLE default\n";

std::vector<int> cell_types(number_of_cells(), 0);

for(int i = 0; i < number_of_edges(); ++i) {
  const Edge edge = edges[i];
  if(edge.cell_l == -100){
    cell_types[edge.cell_r] = -100;
  }
  if(edge.cell_l == -101){
    cell_types[edge.cell_r] = -200;
  }
  if(edge.cell_l == -102){
    cell_types[edge.cell_r] = -300;
  }
  if(edge.cell_r == -100){
    cell_types[edge.cell_l] = 100;
  }
  if(edge.cell_r == -101){
    cell_types[edge.cell_l] = 200;
  }
  if(edge.cell_r == -102){
    cell_types[edge.cell_l] = 300;
  }
}

for(int i = 0; i < number_of_background_edges(); ++i) {
  const Edge edge = background_edges[i];
  if(edge.cell_l == -105){
    cell_types[edge.cell_r] = -500;
```

```
    }
    if(edge.cell_r == -105){
      cell_types[edge.cell_l] = 500;
    }
  }

  for(int i = 0; i < number_of_cells(); ++i) {
    fout << cell_types[i] << '\n';
  }

  fout << "SCALARS Cell_domain double 1\n"
       << "LOOKUP_TABLE default\n";

  for(int i = 0; i < number_of_cells(); ++i) {
    fout << cells[i].id << '\n';
  }
}

#endif //#ifndef UNSTRUCTURED_FV_SOLVER_H
```

FINAL PROJECT

```
#include"Unstructured_FV_Solver.h"
#include"Unstructured_Mesh.h"
#include"Shallow_Water.h"


struct Mesh_Study {
  std::vector<std::string>            meshes;
  std::string               location_output;
  std::vector<std::string>  output_file_names;
};

struct Time_Study {
  std::string                      mesh;
  std::vector<double>                CFLs;
  std::string              location_output;
  std::vector<std::string> output_file_names;
};

////////////////////////////////////////////////////////////////////////
//                      Main
////////////////////////////////////////////////////////////////////////
int main() {

  ////////////////////////////////////////////////////
  // Output console to .dat file for diagnostics
  //std::freopen("diagnostic.dat", "w", stdout);

  ////////////////////////////////////////////////////
  // Fun time
  std::cout << " ------------------------------------------------------\n"
               " |                                                    |\n"
               " |         unstructured shallow-water solver          |\n"
               " |          ferro-fluid in a magnetic field           |\n"
               " |                                                    |\n"
               " ------------------------------------------------------\n\n";

  auto ics = [] (const Node2D& n) {

    Eigen::Vector3d U;
    U[1] = 0.0;
```

```
  U[2] = 0.0;
  if(n.squaredNorm() < 0.05) {
    U[0] = 1.5;
  } else {
    U[0] = 1.0;
  }
  return U;
 };

 auto icc = [] (const Node2D& n) {

  Eigen::Vector3d U;
  U[0] = 1.0;
  U[1] = 0.0;
  U[2] = 0.0;
  return U;
 };

 const double      final_time =        0.5;
 double                 CFL =       0.25;
 const int          n_frames =        401;
 std::string            mesh =
"mesh_study/meshes/domain_0.0005.msh";//"mesh_study/meshes/domain_0.005.msh";
 std::string  location_output = "output/";//"mesh_study/";
 std::string output_file_name = "_CL0.0005_0.5s";//"_CharLen_0.005";

 const bool mesh_study_on = false;
 Mesh_Study mesh_study;
 mesh_study.meshes = {"mesh_study/meshes/domain_0.05.msh",
"mesh_study/meshes/domain_0.025.msh", "mesh_study/meshes/domain_0.0175.msh",
"mesh_study/meshes/domain_0.01.msh",
                      "mesh_study/meshes/domain_0.0075.msh",
"mesh_study/meshes/domain_0.005.msh", "mesh_study/meshes/domain_0.0025.msh",
"mesh_study/meshes/domain_0.00175.msh",
                      "mesh_study/meshes/domain_0.001.msh",
"mesh_study/meshes/domain_0.00075.msh"}; // ~8 GiB
 //mesh_study.meshes = {"mesh_study/meshes/domain_0.0005.msh",
"mesh_study/meshes/domain_0.00025.msh"}; // ~47 GiB
 //mesh_study.meshes = {"mesh_study/meshes/domain_0.000175.msh"};


 mesh_study.location_output = "mesh_study/";
 mesh_study.output_file_names = {"_CharLen_0.05", "_CharLen_0.025","_CharLen_0.0175",
"_CharLen_0.01",
                                 "_CharLen_0.0075", "_CharLen_0.005", "_CharLen_0.0025",
"_CharLen_0.00175",
                                 "_CharLen_0.001", "_CharLen_0.00075"};
```

```
  //mesh_study.output_file_names = {"_CharLen_0.0005", "_CharLen_0.00025"};
  //mesh_study.output_file_names = {"_CharLen_0.000175"};


  const bool time_study_on = false;
  Time_Study time_study;
  time_study.mesh = "time_study/domain.msh";
  time_study.CFLs = {1.01, 0.99, 0.9, 0.75, 0.5, 0.25};
  time_study.location_output = "time_study/";
  time_study.output_file_names = {"_CFL_1.01", "_CFL_0.99", "_CFL_0.9", "_CFL_0.75",
"_CFL_0.5", "_CFL_0.25"}; // ~89 GiB


  if(mesh_study_on){

    for(int i = 0; i < static_cast<int>(mesh_study.meshes.size()); ++i){

      mesh = mesh_study.meshes[i];

      auto solver = Unstructured_FV_Solver<Shallow_Water_Equations>(mesh, icc);
      solver.make_movie(final_time, CFL, n_frames, mesh_study.location_output,
mesh_study.output_file_names[i]);
    }
  } else if(time_study_on){

    for(int i = 0; i < static_cast<int>(time_study.CFLs.size()); ++i){

      CFL = time_study.CFLs[i];
      mesh = time_study.mesh;

      auto solver = Unstructured_FV_Solver<Shallow_Water_Equations>(mesh, icc);
      solver.make_movie(final_time, CFL, n_frames, time_study.location_output,
time_study.output_file_names[i]);
    }
  } else {

    auto solver = Unstructured_FV_Solver<Shallow_Water_Equations>(mesh, icc);
    solver.make_movie(final_time, CFL, n_frames, location_output, output_file_name);
  }

  return 0;
}
```