Eigen Library Implementation for Pressure Poisson Equation

Group G: Replacing PETSc with EIGEN

Batuhan Erden, Erfan Mashayekh, and Nigel Bruce Khan

Technical University of Munich

January 20, 2022

Abstract — An EIGEN library solver was created here to replace the PETSc solver for the pressure-Poisson equations of fluid dynamics modeling. A custom algorithm was written for covering both 2D and 3D cases that was verified to surpass the original PETSc solver in terms of performance and accuracy in certain cases. Lastly, ideas to parallelize the code to solve large systems efficiently is discussed.

1 Introduction

This paper records, how PETSc was replaced with the C++ EIGEN library in order to solve the pressure-Poisson equation as the computational fluid dynamics solver for 2D and 3D grids of a template implementation. The code was written from scratch using theory and only followed the original PETSc solver for verification purposes. A sparse matrix system is being solved, with relevant result comparisons at the end for verification and validation. We found our results to be similarly accurate and faster in most cases. Most importantly the new code is more understandable, greatly optimized, and more efficiently uses matrix patterns to solve the system. A link to the Merge Request containing our changes can be found here.

2 Numerical Modeling

The governing equations for the fluid flow considered in the present study are incompressible Navier-Stokes equations. The equations of continuity and momentum conservation are given by:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{1}$$

$$\begin{aligned} \frac{\partial u}{\partial t} &= -\frac{\partial p}{\partial x} + \tilde{F} \\ \frac{\partial v}{\partial t} &= -\frac{\partial p}{\partial y} + \tilde{G} \\ \frac{\partial w}{\partial t} &= -\frac{\partial p}{\partial z} + \tilde{H} \end{aligned} \tag{2}$$

$$\tag{3}$$

Where $\tilde{F}$, $\tilde{G}$, and $\tilde{H}$ are defined by:

$$\begin{aligned} \tilde{F} &:= -\frac{\partial(u^2)}{dx} - \frac{\partial(uv)}{dy} - \frac{\partial(uw)}{dz} \\ &\quad + \frac{1}{Re}\left(\frac{\partial(u^2)}{dx^2} + \frac{\partial(u^2)}{dy^2} + \frac{\partial(u^2)}{dz^2} + g_x\right) \\ \tilde{G} &:= -\frac{\partial(uv)}{dx} - \frac{\partial(v^2)}{dy} - \frac{\partial(vw)}{dz} \\ &\quad + \frac{1}{Re}\left(\frac{\partial(v^2)}{dx^2} + \frac{\partial(v^2)}{dy^2} + \frac{\partial(v^2)}{dz^2} + g_x\right) \end{aligned}$$

$$\begin{aligned} \tilde{H} &:= -\frac{\partial(uw)}{dx} - \frac{\partial(vw)}{dy} - \frac{\partial(w^2)}{dz} \\ &\quad + \frac{1}{Re}\left(\frac{\partial(w^2)}{dx^2} + \frac{\partial(w^2)}{dy^2} + \frac{\partial(w^2)}{dz^2} + g_x\right) \end{aligned} \tag{4}$$

The pressure Poisson equation (PPE) is derived from the momentum equation. By evaluating the continuity equation using the momentum equation, one can drive the PPE equation as the following.

$$\begin{aligned} &\frac{\partial^2 p^{(n+1)}}{\partial x^2} + \frac{\partial^2 p^{(n+1)}}{\partial y^2} + \frac{\partial^2 p^{(n+1)}}{\partial z^2} \\ &= \frac{1}{\Delta t}\left(\frac{\partial F^{(n)}}{\partial x} + \frac{\partial G^{(n)}}{\partial x} + \frac{\partial H^{(n)}}{\partial x}\right) \end{aligned} \tag{5}$$

where $F^{(n)}$, $G^{(n)}$, and $H^{(n)}$ can be written as the following taking to account a first-order discretization in time.

$$\begin{aligned} F^{(n)} &= u^{(n)} + \Delta t \tilde{F}^{(n)} \\ G^{(n)} &= v^{(n)} + \Delta t \tilde{G}^{(n)} \\ H^{(n)} &= w^{(n)} + \Delta t \tilde{H}^{(n)} \end{aligned} \tag{6}$$

By discritizing the left-hand-side of the PPE equation and substitute the right-hand-side with $RHS^{(n)}$, the equation changes to the following linear system of equation.

$$Ap^{(n)} = RHS^{(n)} \tag{7}$$

Where A is the coefficients matrix and p and RHS are the vectors containing the pressure and right-hand-side values on the physical domain. C++ Libraries such as PETSc, and Eigen can solve such linear systems of equations. As mentioned in the previous section, in this project, the Eigen library has been used to solve the above linear system of equations, and the results have been compared with the PETSc's KSP solver.

2.1 Algorithm and Setup

The basis of the algorithm is to solve the pressure Poisson equation which involves a linear system of equations for each grid point represented as a matrix, the grid's unknown pressures in a vector and a right hand side of terms accumulated together in a vector. The velocity and other terms are solved for in other parts of the code and their implementation will not be changed. Our goal is to solve for the pressure. To do so an iterative method will be selected from the Eigen Library, but first the matrix and right and side must be created properly. The algorithm to do this is described below for a 2D system first and then the 3D explanation is added to it, then we discuss how the boundary conditions. Finally, the obstacle conditions for cases such as a backward-step are handled.

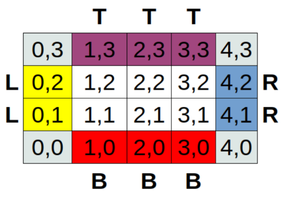

Figure 1 2D grid example from the physical domain

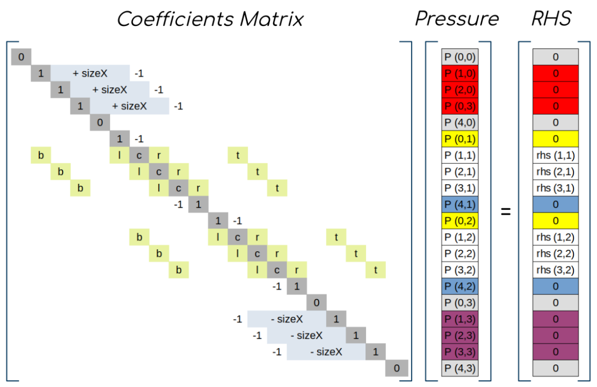

Figure 2 The patterns in the coefficient matrix populated using the 2D grid

2.2 Grid to Matrix

For any type of system here, we know that there are the internal nodes and the ghost cells which can be used to hold the boundary conditions. For 2D, we need a matrix of size dimension one multiplied by dimension two. Thus, each row of the matrix accounts for all the grid points. Initially we were traversing the grid column wise to fill the matrix, but after running the case we realized this would slow down the computations, and it is discussed later. For a 2D system, traversing the grid row-wise, as can be seen in the figures 1 and 3, a pattern is noticed first for the internal (fluid) cells nodes. Each fluid cell has in its row, a cell to account for its own value, on the matrix diagonal, cells to the left and right to account for the left and right neighbours, and finally, cells a distance of size X (horizontal dimension) away from them on either side of the diagonal entry. For 3D there will also be neighbours at a distance of dimensionX*dimensionY. Before, discussing the pattern further, lets shift to the actual values in the matrix entries. The calculation for the matrix entries of the cell node and its neighbours is carried out using the same formulas in PETSc. Using *parameters.meshsize* with the correct dimension and cell index input gives the cell width in the appropriate direction. Next, for each side in one dimension, top-bottom, left-right, or front-back, we use the formulas

$$dx = \frac{x_i + x_{i-1}}{2}, \quad dx = \frac{x_i + x_{i+1}}{2} \tag{8}$$

These give the average size of the cell. Next we use the formulas for arbitrary mesh-sizes already provided in the given PETSc code. An example is shown below for the top neighbour (all other neighbors will look similar), and for the center cell.

$$TopNeighbour = \frac{2}{dx_T \cdot (dx_T + dx_{Bo})}, \tag{9}$$

$$CenterCell = -\frac{2}{dx_R \cdot dx_L} - \frac{2}{dx_T \cdot dx_{Bo}}, \tag{10}$$

Coming back to the pattern again, it is realized that all internal fluid cells will have the same matrix row entries, on and around the diagonal at the same distances. Thus, we skip the cells on the bottom boundary and a left ghost cell in the grid row before writing for the patter the first time. Note, unlike the PETSc implementation we did not use 2 ghost layers, because we do not employ a parallel implementation yet. We then fill the pattern on each consecutive row by finding the diagonal and placing the neighbours entries around it. The pattern for each fluid node is repeated over all corresponding fluid cell rows. The algorithm then skips the left and right ghost layer row entries at each grid row, as well as the top layer at the end of the run. To skip the bottom and top layer, the code needs to know the length of the x-dimension. To know how many times to repeat the pattern it takes the y-dimension and subtracts 2 for a ghost layer on each side of the grid. This method however, is only applicable for a rectangular/square grid. Using this pattern method, the appropriate matrix entries can be filled.

2.3 Boundary conditions

Each boundary, meaning the walls, inlet flow, or outflow, has either the Dirichlet or Neumann boundary condition in velocity and pressure, depending on the case be examined. In this project, for Neumann and Dirichlet boundary conditions, the below conditions hold.

$$\frac{\partial \phi}{\partial n} = 0 \quad : Neumann \tag{11}$$

$$\phi = Const. \quad : Dirichlet \tag{12}$$

where n is the normal unit vector to the surface of the boundary. The boundary conditions are elements in the coefficients matrix in figure 3. The coloured grids are all boundaries that need to be implemented in a way that the above equations hold for them in the linear system. Considering this for Neumann boundary condition in pressure, $p_1 = p_0$ is used.

2.4 Obstacles

After setting up the basic matrix, we also need to consider grids with obstacle cells in them. For an obstacle we also need to consider whether it is surrounded by fluid cells on some sides. We want the some of the values from the fluid and the cell itself to equal zero. Hence in the obstacle cell row in the matrix we specify a value of 1 in the neighbor fluid entry and a negative of the sum of all neighbor fluid cells to the diagonal. The algorithm and code for this was directly used from the current PETSc implementation. The only addition to the normal code is to check for each internal fluid cell whether it is a fluid or not before implementing the normal pattern stencil or the obstacle stencil.

3 Eigen

Our motivation for this project stems from the fact that PETSc is a heavyweight library and it would not be optimal to run it on systems that are not workstations. Eigen, however, is a modern and lightweight library that can work well on simpler machines, since Eigen's matrix-matrix product kernel is fully optimized and already uses almost 100% of the CPU capacity [2]. Our first impression was that Eigen would be much faster, and it really is as the results are visible in the corresponding Results section. We also checked the CPU usage during runs, and Eigen really does utilize almost all resources. Let's talk about what Eigen is and how we used its great features.

Eigen is a high-level C++ library for linear algebra and numerical solvers. The most important feature of the library is its ease of use. That is, developers can perform matrix operations as they can in Python with the numpy library. Basic arithmetic operations are also defined in the library. For instance, users can easily perform matrix multiplication using just the "*" operator.

Eigen consists of more than just these functions, however. It is also a versatile library that supports not only small fixed-size matrices, but also arbitrarily large dense matrices and even sparse matrices. In this project, we used its great functions for sparse matrices because our coefficient matrix A is a huge sparse matrix and not using it as a sparse matrix would hurt both performance and memory usage. On the other hand, Eigen is also fast, as can be seen in the Results section. We have exploited vectorization as much as possible and let Eigen do the optimization, since it performs explicit vectorization for most instruction sets such as AVX, AVX2, FMA, etc. Moreover, Eigen implicitly avoids dynamic memory allocation, which results in fully optimizing fixed-size matrices, and it can also handles large matrices by making optimal use of the cache[3]. We tested this property of cache-friendliness by trying two different variants: *column-major and row-major*. Since Eigen stores data in a row-major format, accessing the data in a column-major format came at the cost of performance. After re-implementing everything that required redefining our coefficient matrix and right-hand side vector to be compatible with row-major access, we observed far better results when we accessed the data in row-major format. In a nutshell, row-major layout is way faster than column-major layout, and for larger matrices, the performance difference is even more notable.

Eigen is not only versatile and fast, but also reliable, as it has been extensively tested in many different applications and the algorithms have been carefully selected for their reliability. Another good aspect of Eigen is its elegance. By taking advantage of Eigen's capabilities of vectorizing every possible for loop, we managed to write a very optimized solver code that is both clean and maintainable. For example, in the PETSc solver, there were two functions for 2D and 3D that calculated the values of obstacle cells around fluids. To achieve this functionality, 250 lines of code were written, but in Eigen's solver we managed to re-implement the same functionality with only 20 lines of code in total for both 2D and 3D, with the function *"computeStencilRowForObstacleCellWithFluidAround"*.

Last but not least, Eigen also provides good compiler support and is open-source, so developers can contribute to the project, if needed.

3.1 Optimization

In the previous sections, we talked about how we converted the grid into our coefficient matrix. First, everything was on paper, and we had to implement it with Eigen. To check if the idea worked, we first implemented the code with more for loops and less vectorizations. Of course, this didn't use the full potential of Eigen because we didn't take full advantage of the vectorization by creating as many vectors and matrices as possible.

In order to use all of Eigen's features and leave the optimizations to the Eigen, we started looking for patterns in the coefficient matrix so that we could vectorize them. In the end, we vectorized everything and used a minimal number of for loops. In the Results section you can see the great performance we achieved.

3.1.1 Patterns

Using the patterns we found, we created vectors and matrices, and then inserted them into the matrix and moved them in X, Y and Z directions using the grid size. We created 3 patterns in 2D and 4 patterns in 3D.

The first pattern is the blue boxes on the matrix that are filled by the fluid region. We created vectors for these blue boxes, inserted them into the matrix and shifted them. To be more specific, in the 2D case, the number of cells in the Y direction defines how many of these consecutive blue vectors we have while the number of cells in the X direction defines how many of these consecutive blocks (blocks of consecutive blue vectors) we have.

The second and third patterns are the black and red boxes on the matrix filled with boundary conditions, where the black boxes are matrices containing the values calculated using the left and right wall conditions, and the red boxes are vectors containing the values calculated using the bottom and top wall conditions. In the 3D case, we also have the fourth pattern where the values are calculated using the front and back wall conditions. In the 2D case, the number of cells in the Y direction defines the size of these black matrices and how many of them we have, while for the red vectors, the number of cells in the X direction defines how many of these red vectors we have.

For simplicity, we only show the patterns for 2D, which can be seen in figure below. The patterns for 3D also follows the same structure. The coefficients matrix we have created for the 3D can be found in the appendix section.

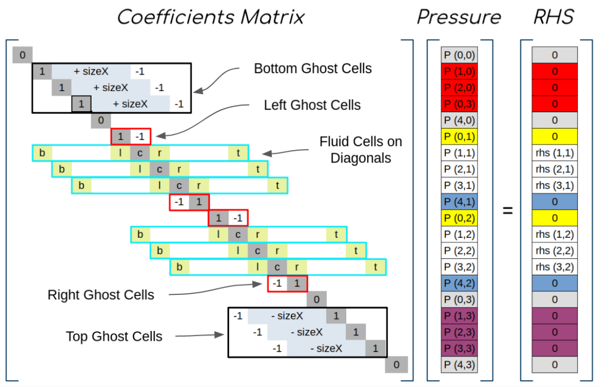

Figure 3 The patterns in the coefficient matrix populated using the 2D grid

With these patterns, we inserted them into the matrix using only the necessary for loops. This way, we were able to create a fully-vectorized code. In the Results section, it is clearly visible how all these optimizations paid off.

3.2 Solver

For a linear system with a coefficient matrix, Eigen has two types of solvers available, direct and iterative. We go with the iterative solver, because they provide the needed efficiency and only minimally compromise on accuracy. Moreover, PETSc also uses an iterative solver. From the three iterative ones available *ConjugateGradient* is recommended for large 3D sparse matrices. However, we compared it against the *BiCGSTAB*, which is the bi-conjugate gradient stabilized solver, which is used to speed up convergence and found it to have similar accuracy and much higher speed. In the end we chose *BiCGSTAB*. By default this solver uses x = 0 as the initial solution but it can be altered if needed. This solver is used for sparse problems and allows control over the maximum convergence iterations as well as the tolerance i.e. relative residual error [4]. According to its description on the Eigen website, the solver gives the best performance with a row-major sparse matrix format, which we were able to confirm in our results.

3.2.1 Error Threshold and Iteration Correction

We keep the error as low as *1e-7*. However, judging from the difference in performance for larger matrices, we believe that PETSc Solver defines a higher error threshold for large matrices. In Eigen Solver, we always aim for *1e-7*, which requires us to perform higher number of iterations for larger matrices as the error usually is higher. This is why observed lower performance numbers for larger matrices. However, in the Results section below, it is easy to see that Eigen still performs better than PETSc in all cases except for the 80x80 grid used in the Backward Facing Step channel. The reason why this might be the case is that in Eigen solver, the average number of iterations is dynamically set to 650 to keep the error around *1e-7*, while PETSc solver might be using higher convergence error resulting in fewer iterations and a faster run-time.

4 Results and Comparison

To test our the Eigen solver, we compared its computational time and accuracy against PETSc for both 2D and 3D in all cases available to us, with a variety of grid sizes. We have run the experiment for each grid size 3 times and taken the average. This was done because, PETSc's performance was drastically changing between different runs, while Eigen managed to keep the computational time difference lower between the runs, proving itself to be a more reliable solver in terms of usability. The tests were conducted on a system with the following configuration:

Intel(R) Core(TM) i5-8279U @ 2.40GHz (4c/8t) 16 GB

The performance findings are summarized below on a log graph in Figure 4 showing the grid sizes and the time taken by each solver in log scales. From small (10x10) to medium size grids(80x80), it was found that EIGEN performs better as it takes less time for both the Cavity and Backward Channel cases in 2D. Going, bigger than that they seem to have comparable results. The results from both solvers had identical accuracy to around 5 significant figures.

Similarly, with a 3D implementation of different grid sizes, Eigen still performed better in terms of computational time while providing precision almost exactly equally to that of PETSc. The computational

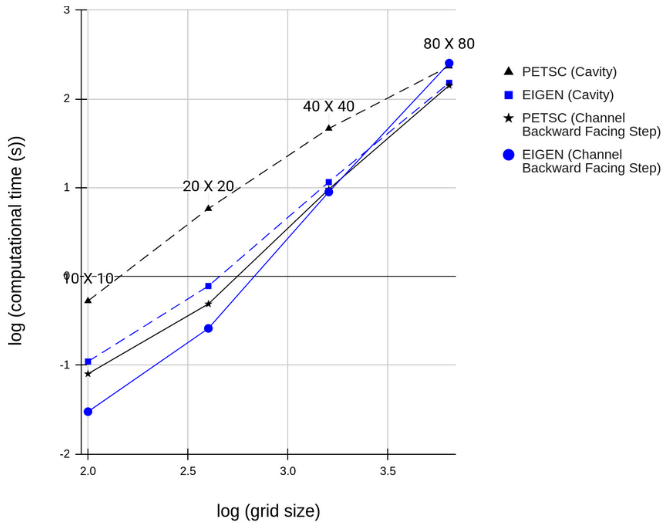

Figure 4 Comparison the computation times of Eigen and PETSc in 2D cases in log scale

time comparisons for 3D can be seen in figure 5 below.

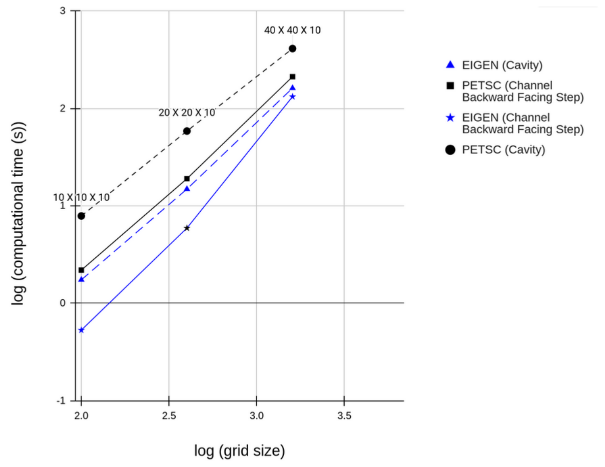

Figure 5 Comparison the computation times of Eigen and PETSc in 3D cases in log scale

The accuracy of our implementations is also checked by comparing with PETSc solver, and results are added inside the figures 7 and 8.

5 Future Work

As can be seen in the Results section above, Eigen can outperform PETSc, and at the same time, the code written with Eigen is more readable and maintainable. Of course, we can also take this project to the next level.

We have explored several ways to make the Eigen solver executable for the parallel case. Currently, the project can only run sequentially. We have tried to use

2D				
Grid size	Cavity		Channel Backward Facing Step	
	PETSc	Eigen	PETSc	Eigen
10x10	0.53	0.11	0.08	0.03
20x20	5.80	0.78	0.49	0.26
40x40	46.32	11.57	9.59	8.94
80x80	203.12	151.58	141.60	252.53
3D				
Grid size	Cavity		Channel Backward Facing Step	
	PETSc	Eigen	PETSc	Eigen
10x10x10	7.87	1.74	2.19	0.53
20x20x10	58.57	14.86	19.01	5.91
40x40x10	411.44	161.65	211.78	132.16

Figure 6 Comparison the computation times of Eigen and PETSc

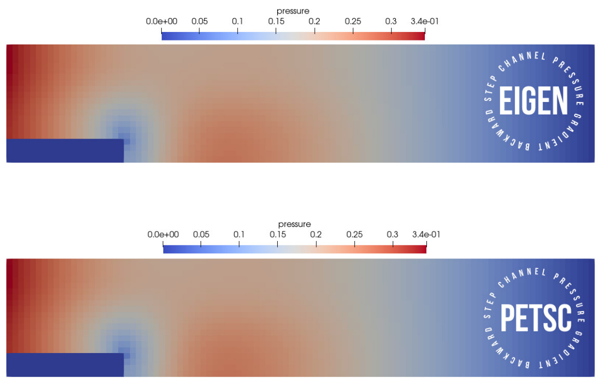

Figure 7 3D Channel backward facing step pressure gradient with grid size (100 X 20)

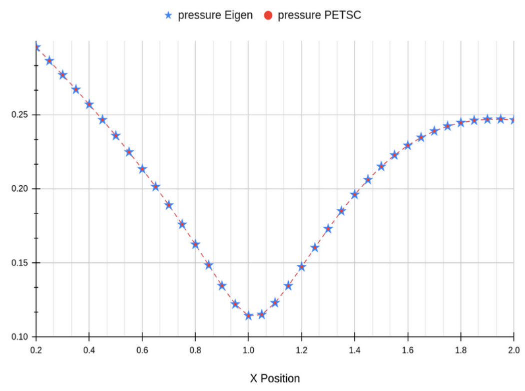

Figure 8 Pressure Comparison of Eigen and PETSc near vortex at Y = 0.3

the OpenMP library and paralellize the entire eigen solver as well as some of the for loops. However, in order for this project to run in parallel with MPI, some functionality needs to be in place to make this possible. In the next step, we will talk about our ideas and how we thought parallelization was appropriate. Remember that we do not have access to the solver in Eigen and the only way to parallelize it is to use OpenMP.

5.1 Parallelization

Initially, we tried to use OpenMP according to Eigen's guidelines. We created the Eigen using the threads in the system and also parallelized some for loops. However, we did not observe any change in the performance on our test system that has 8 software threads.

Second, we tried to work with MPI parallelization as it is the case in the PETSc solver, which already uses parallelization with its built-in functions. However, Eigen does not provide built-in functions like PETSc that can be used immediately.

The first idea that comes to mind is to decompose the grid in the X, Y, or Z direction, or in all directions, as shown in the figure above. Since we are using Eigen to create a linear system, we can't just decompose it and communicate the boundaries as needed, as we did in Worksheet 2 to communicate pressure, velocity, and viscosity values.

However, what we tried to do, and what could be done instead, is to pass the right-hand side values calculated by each process before the solving part and send them to the master process. In the master process, the linear system is solved exactly as in the sequential version, while the other processes remain idle. Here, one can also use the remaining threads on the system that are sitting idle (The threads that are not currently being used by MPI). After the linear system is solved, the master process can send the solution to the processes. It should be noted that only the solution required by each process needs to be sent, not the entire solution. That is, each process receives only the pressure values it needs to calculate the velocity in its subdomain.

We believe that it is possible to run the code in parallel if the above steps are followed carefully. Eigen has already proven to be faster, and with the parallel implementation, it really has an advantage against PETSc.

This way we can also run the code in parallel. However, in the future, we would also like to try to create and solve the linear system in each process. With the code we wrote in Worksheet 2, the pressure values are already communicated at the boundaries. As a result, solving just the sub-linear system in each subprocess would achieve the best parallelization.

6 Conclusion

Our project was to re-implement the solver for the fluid dynamics simulations, by re-writing it to work with the Eigen Library instead of PETSc. We started the code from scratch, beginning with the pressure-Poisson equation for a linear 2D system to work with a cavity and channel case. Next, we built on top of it for 3D cases and obstacles. The code was then optimized for efficiency and performance, using patterns observed in the grid and matrices, to surpass the PETSc implementation. Lastly, we compared the accuracy from both solvers and found our results to be quite satisfying. Towards, the end we have also attempted parallelization but further work must be done to get it working. The project did well to provide a code that is more versatile and easy to customize in the future when needed. If used as a teaching tool, this code will also help improve the understanding of the students who take this course in the future providing them a platform to build more complex projects.

References

[1] Griebel, M., Dornseifer, T., & Neunhoeffer, T. (1998). Numerical simulation in fluid dynamics: a practical introduction. Society for Industrial and Applied Mathematics.

[2] Make Eigen run in parallel. Eigen. (n.d.). Retrieved January 20, 2022, from https://eigen.tuxfamily.org/dox/TopicMultiThreading.html

[3] Eigen. (n.d.). Retrieved January 20, 2022, from https://eigen.tuxfamily.org/index.php

[4] Eigen::BiCGSTAB< MatrixType, Preconditioner > Class Template Reference Eigen. (n.d.). Retrieved January 20, 2022, from https://eigen.tuxfamily.org/dox/classEigen_1.html

7 Appendix

In this section the result of the three-dimensional cavity and channel backward facing step of Eigen solver have been added. As can be seen from the figures, Eigen solver class is able to model the three dimensional cases.

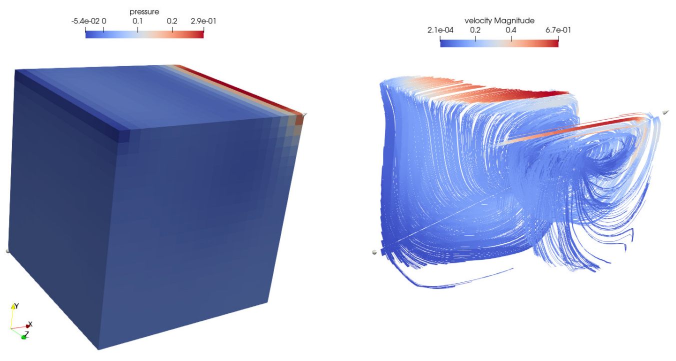

Figure 9 3D cavity model, pressure and velocity fields with streamlines

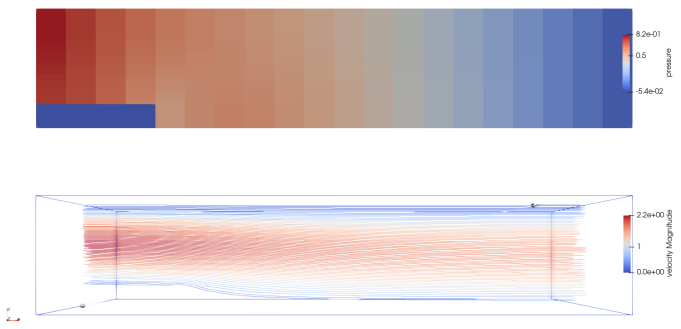

Figure 10 3D Channel backward facing step model, pressure and velocity fields with streamlines