PERFORMANT NAVIER-STOKES SOLVERS FOR TWO-DIMENSIONAL INCOMPRESSIBLE FLOW

PHILIP MURZYNOWSKI

1. Abstract. The aim of this project was to develop a performant implementation of a Navier-Stokes Solver for single-phase, incompressible, nonisothermal, and Newtonian fluids in a two dimensional enclosed plane with modifiable fluid velocities at the boundaries. The motivations for this particular topic center around the opportunity for exposure to computational fluid dynamics, experience in the implementation of an accurate and efficient partial differential equation solver, and a bias towards simulations with subjectively interesting graphical output. The setup also lends itself easily to comparison with the well know Lid-driven cavity benchmark problem it was modeled after. The problem is constrained to single phase, incompressible, and Newtonian fluids to narrow the scope to a project appropriate for a single semester, and algorithmic and implementation-based optimizations are discussed, until finally performance is compared between two solver implementations.

2. Introduction. Listed below are the two dimensional, non-dimensionalized, Navier Stokes equations which are at the crux of this report. The first is the continuity equation, and the second and third are partial differential equations for the evolution of the fluid velocities in different axes as they are dependent on (going through in the order that they are written) their spacial derivatives, pressure gradient, and stress on the fluid as captured in the viscous terms [7].

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0$$

$$\frac{\partial u}{\partial t} = -\frac{\partial u^2}{\partial x} - \frac{\partial uv}{\partial y} - \frac{\partial p}{\partial x} + \frac{1}{Re}\Big(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\Big)$$

$$\frac{\partial v}{\partial t} = -\frac{\partial v^2}{\partial y} - \frac{\partial uv}{\partial y} - \frac{\partial p}{\partial y} + \frac{1}{Re}\Big(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\Big)$$

With regards to specifics on the simulation, it is carried out in a square, four walled, and enclosed environment, with fluid velocities on the walls configurable by users and working under the no slip boundary condition. Critically, by the no slip boundary condition and the above equations it can be seen that the pressure gradient at the wall boundaries must be zero, so we also obtain a Neumann boundary condition for pressure.

The report will start with a solution implementation arising almost immediately from a discretization of the above equations and close relation to the widely known SIMPLE algorithm, followed by a description of the approach taken for optimization of this first solver and identification of areas for improvement. The following theoretical, algorithmic, and optimization topics with focus on developing a solver that attempts to target the shortcomings of the first. For reference throughout this report, tests are performed on computer with the following specifications: Intel(R) Core(TM) i5-7300HQ CPU @ 2.50 GHZ, and the code is available with an example workflow in Main.jl at https://github.com/PhilMurzynowski/navierstokes18337.

3. The Initial Solver: GridSolver. This section will detail the implementation and optimization of the first functional solver in this project, referred to as GridSolver in the code base. The solver is an implementation, with slight variations in the details of the variable corrections, of the SIMPLE algorithm (Semi-Implicit Method for Pressure-Linked Equations), which is one of the earliest finite volume method algorithms in computational fluid dynamics and has enjoyed widespread use especially in laminar, steady state, and incompressible flow [4]. The core ideas to the algorithm are that it decouples velocity and pressure when solving for one value or the other, for example deriving an equation for pressure by converting relevant velocity values to constants, and later applying a range of corrective schemes to attempt to make up for the inherent assumptions made in keeping dependent values constant [11][3]. More precisely, the coupled system is broken up, or segregated, linearized, and then iteratively solved independently in an alternating manner, first velocity, then pressure, velocity, and so on, until convergence [11][3]. Care must be taken in the discretization and linearization to ensure that the continuity equations and relevant conservation laws are still satisfied with a tolerable margin of error.

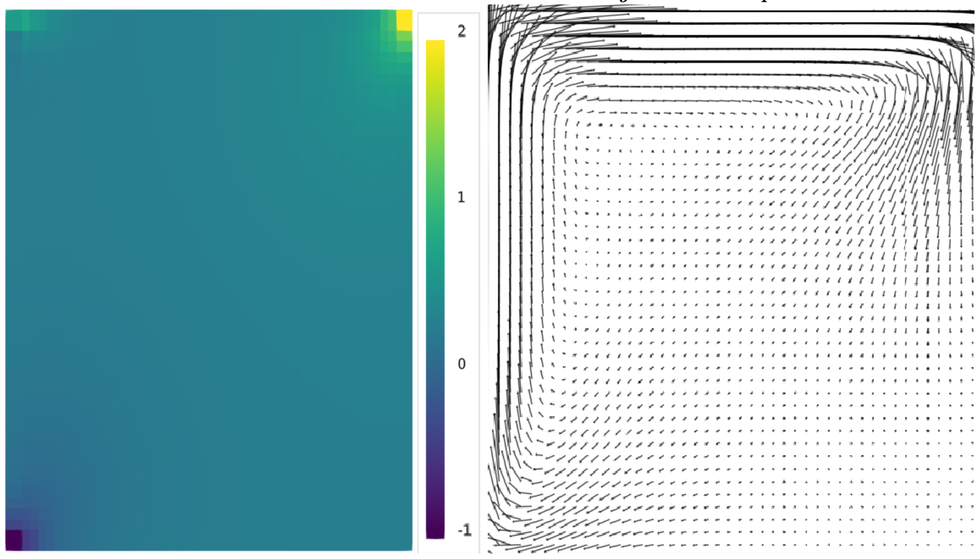
The first choice in implementing the finite volume based algorithm is mesh type, specifically structured vs. unstructured and staggered vs. collocated. A structured grid, which is implemented here, constrains and subdivides the region of interest into an array of cuboids that are uniformly spaced and of equivalent size, shape, and orientation, while an unstructured grid is not bound by these constraints. Next, the staggered vs. collocated decision concerns the frame of reference of some of the quantities of interest. Pressure is almost always in reference to the center of a cuboid, however velocities may be defined at the faces of the cuboids, defining a staggered grid approach. The staggered grid is utilized to circumvent odd-even decoupling between velocity and pressure and therefore decrease potential coarsening of the simulation [9]. The structured grid was chose for simplicity as well as the a fully collocated grid, with more details on how the odd-even decoupling between velocity and pressure is tackled later.

3.1. GridSolver Optimization. The few main focuses in optimizing this first solver code revolved around making as much reuse of preallocated memory as possible, avoiding creating new arrays with slice operations by heavily preferring views, and finally, with these views on arrays, encouraging the compiler to use vectorized operations on the arrays by using broadcast operations whenever possible and sensible. For example, every forward time stepping function was passed in two arrays, one to hold the current value and one the future, which were swapped every iteration, and practically all of the described block operations in the discretization above were accessed as views so that almost the entirety of the matrix could be operated on with vectorized or broadcast operations.

However, after profiling the code, it is evident that the majority of the runtime is spent in the iterative procedure to solve the two dimensional Poisson equation for pressure, which is a main focus of subsequent solver implementation. Though specialized solvers exist for solving this form of equation, a large portion of the objective is to attempt to develop a custom solver, specialized to as great as an extent possible for the Poisson equation, to alleviate the bottleneck of pressure solving on performance.

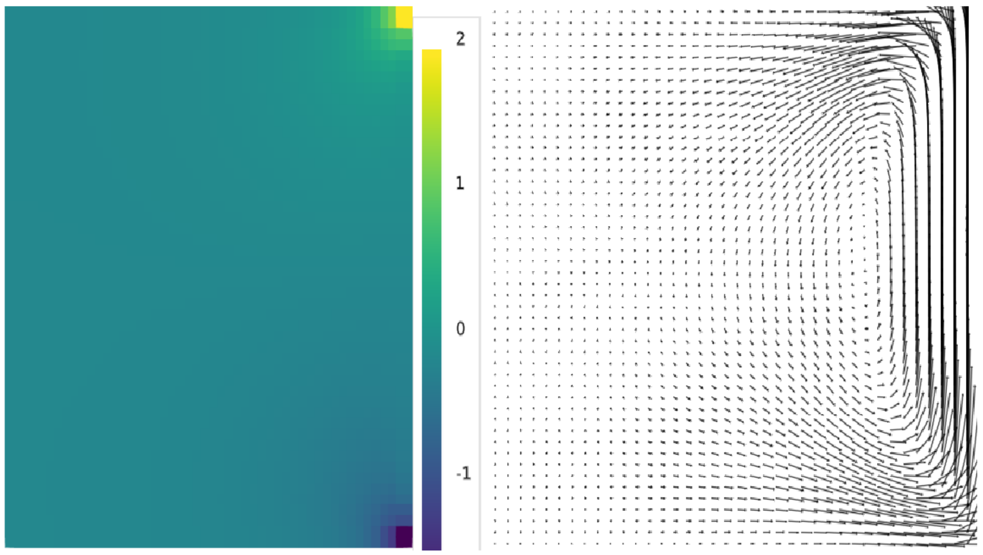
4. Conjugate Gradient Implementation. Serious research in optimizing solving the Poisson equation began with a suggestion from Professor Rackauckas to consider implementation of a specialized Krylov solver. Very briefly, Krylov solvers can

FIG. 3.1. ***GridSolver Pressure and Velocity Field Example 1***



Sample pressure and velocity output for a problem initialized with a positive y velocity on the left boundary and a positive x velocity on the top boundary. The point of highest pressure is in the top right corner with the velocity field forming a triangular pattern is it mostly circulates between the bottom left, top left, and top right corners.

FIG. 3.2. ***GridSolver Pressure and Velocity Field Example 2***



Sample pressure and velocity output for a problem initialized with a positive y velocity on the right boundary. The point of highest pressure is in the top right corner with the velocity field largely sticking to the right wall due to the no slip condition.

be thought of as methods to iteratively find a least squares solution over an expanding Krylov subspace, until finally the optimal solution is found when the final Krylov subspace spans the space of the solution vector [2]. However, if the objective is to specialize on the matrix equation for the two dimensional Poisson equation, then the solver can be specialized much further than simply plugging in a multipurpose Kyrlov Solver such as GMRES, the generalized minimum residual method. In particular, aside from the block band structure of the Poisson matrix and overall sparsity, one key feature in relation to Krylov Solvers is the Poisson matrix's symmetry, as for symmetric matrices a more specialized method such as the Conjugate Gradient method can be implemented, which was the implementation of choice in this project. Other methods, including stationary methods such as Jacobi Iteration and Gauss-Seidel were considered, however were disregarded do to the lack of a guarantee for convergence for any positive definite and higher dependency on the spectral radius of the matrix of interest, or the possibility of exceedingly high numbers of iterations, respectively [6][13]. On the other hand, the rate of decrease for the error in each iteration in the Conjugate Gradient method can be shown to be at most $(k-1)/(k+1)$, for a condition number k, with a guarantee to converge in n iterations excepting floating point error for a matrix of dimensions n by n, and an overall asymptotic runtime of $O(m\sqrt{k})$, where m is the number of nonzero entries and k the condition number [6].

4.1. Conjugate Gradient Optimization. An immediate optimization that can be made in the Conjugate Gradient method is to cut down the number of matrix multiplications, which can be accomplished by updating the residual using the residual from the previous timestep, instead of recomputing the residual with the current best solution to the problem. Next, if the method is to be only used with the Poisson equation, then in fact no matrix needs to be passed into the method at all, leading to noticeable memory savings, as the matrix is known, and the pentadiagonal sparse structure additionally only uses two two values scaled by the spacing between cells, meaning the multiplication can be easily manipulated into looping code. However, as will be discussed later in an algorithmic optimization, preconditioning, this is also an inherently limited approach as to be scaled it would require hard coding and unrolling any additional matrix operations which may be desirable to include. This optimization is implemented but in fact discouraged in favor of simply using the available SparseArrays.jl package, as in the small sacrifice made when generalizing from the two dimensional Poisson matrix to a general sparse matrix, access is granted to already heavily optimized sparse matrix math routines. SparseArrays was compared to BandedMatrices.jl as well, however, in terms of representing the Poisson matrix SparseArrays proved to be more effective. Lastly, in the case in which it may be desirable to pass a view of a section of an array to the solver, for example if the section of the grid excluding the boundaries is passed in, there is a noticeable difference in performance compared to immediately copying and passing in a standalone vector. For at least smaller matrices, of size N equal to 32 where the total number of discretization points is N^2, creating a sliced vector is at least 50% faster than passing in a view, as the price of the single slice is more than accounted for subsequently when the vector has to be accessed and manipulated every iteration of the Conjugate Gradient solve, and the more compact form leads to fewer cache misses. However, as a final note on the base Conjugate Gradient method, there are also deep rooted problems in a potential combination of the Conjugate Gradient method and a SIMPLE algorithm variant such as GridSolver, as there is no clean way to represent the Neumann boundary conditions on the sides of region of interest within the Poisson matrix without losing symmetry [10].

5. Vorticity-Streamfunction Solver. Converting our problem formulation from one of solving for velocities and pressure to vorticity and streamfunction is one of the most straightforward ways to retain symmetry in the Poisson matrix for our problem and thereby allow use of the Conjugate Gradient method. While the full derivation will not be belabored here, the new problem formulation and discretization are provided below, and the key ideas are as follows: first, the Poisson equation is used to solve for streamfunction given vorticity instead of pressure given velocity, and second, all pressure terms are eliminated from the equations, so the Neumann boundary conditions for pressure disappear, the boundary conditions for streamfunction are Dirchlet, and finally the boundary conditions for vorticity are dependent on only the streamfunction and the preset constant velocities at the walls of the region. The intuitive explanation for the boundary conditions follows from that idea that walls can be considered streamlines as fluid flows in parallel to them, and therefore their streamfunction can be set to a constant [1].

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = -\omega$$

$$\frac{\partial \omega}{\partial t} = -\frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} + \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} + \frac{1}{Re}\Big(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\Big)$$

$$\frac{\psi^n_{j,i+1} - 2\psi^n_{j,i} + \psi^n_{j,i-1}}{\Delta x^2} + \frac{\psi^n_{j+1,i} - 2\psi^n_{j,i} + \psi^n_{j-1,i}}{\Delta y^2} = -\omega^n_{j,i}$$

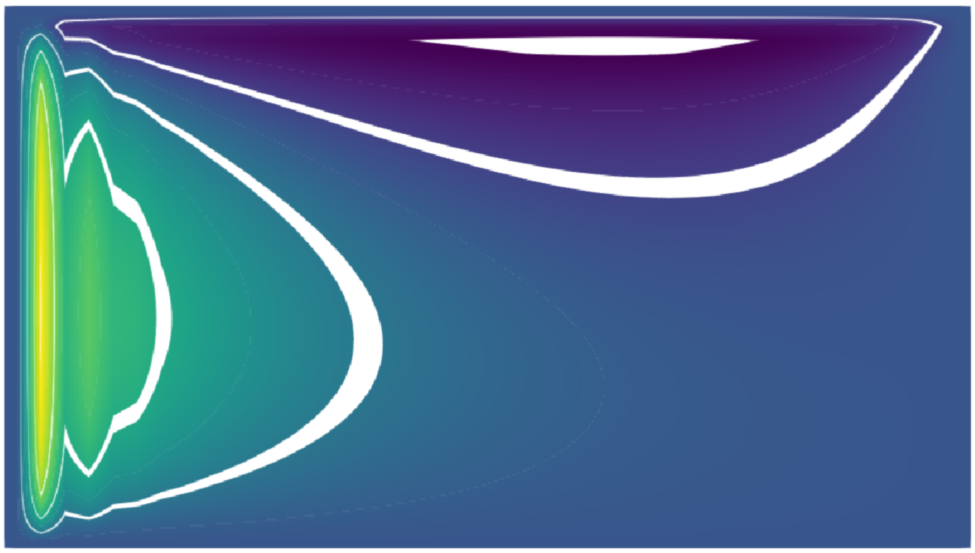
$$\begin{aligned}\omega^{n+1}_{j,i} = \omega_{j,i} + \Delta t\Big(- u_{j,i}\frac{\omega^n_{j,i+1} - \omega^2_{j,i-1}}{2\Delta x} - v_{j,i}\frac{\omega^n_{j+1,i} - \omega^2_{j-1,i}}{2\Delta y}\Big)\\ + \frac{\Delta t}{Re}\Big(\frac{\omega^n_{j,i+1} - 2\omega^n_{j,i} + \omega^n_{j,i-1}}{\Delta x^2} + \frac{\omega^n_{j+1,i} - 2\omega^n_{j,i} + \omega^n_{j-1,i}}{\Delta y^2}\Big)\end{aligned}$$

To describe the equations presented just above, the first is the Poisson Equation relating vorticity, ω and streamfunction, ψ the second the equation for vorticity under incompressible conditions in two dimensions. Below those first two are their discretizations using a FTCS (forward time centered space) first order method, with the last having a slight caveat that the code also includes second order upwind differencing for increased accuracy with the nonlinear terms related to fluid stress or viscosity, omitted for space [1].

The vorticity streamfunction approach to Navier Stokes is in general a very attractive approach to solving the two dimensional problem, and though it does experience complications in extension to higher dimensions it is quite a suitable fit for a project limited to two dimensions [1]. Furthermore, though the use of a staggered grid was avoided in writing of GridSolver for ease of implementation, the importance of such an approach is diminished as with the elimination of pressure from the momentum equation, odd-even coupling between pressure and velocity is no longer present [13]. The simpler and cleaner collocated grid can be used without issue. The transformation back to a velocity and pressure is also an easy and familiar task, as to obtain the velocity the streamfunction is simply differentiated, and from that point the Poisson equation can be solved to retrieve pressure.

5.1. Vorticity-Streamfunction Solver Optimization. Optimizations in this section of the code share many common themes with optimization of grid solver, preallocating memory when possible and reusing preallocated memory in the vorticity timestep update and Poisson solve using ICCG (discussed in the next section), inlining smaller functions with few line expressions, and using @inbounds when loops are guaranteed to remain within the desired array and specifically at the targeted indices. As a slight discrepancy, the vorticity update function includes behavior to only use second order upwind differences on non boundary points as zero valued sentinels do not pad the grid by another layer in order to save on memory. In order to limit branching/conditional behavior in the code, the loops are broken up into their separate regions, sides, corner, inner, in order to provide appropriate updates.

FIG. 5.1. ***Vorticity Streamfunction Solver Streamlines Example 1***



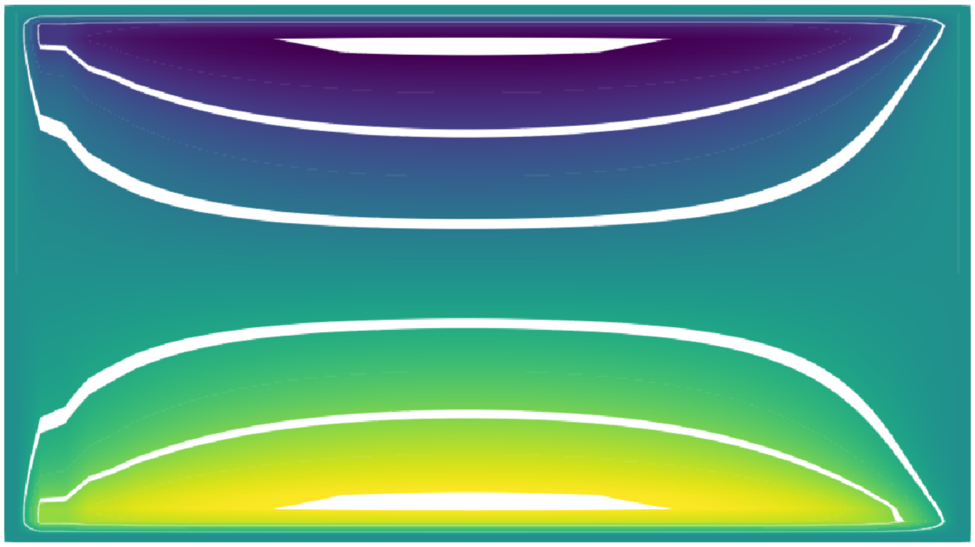
Sample streamlines for a diverging flow, top boundary moving in positive x direction, left boundary moving in negative y direction.

6. Preconditioning.

The final central component towards writing a fast Navier Stokes simulator is preconditioning the conjugate gradient method. As the convergence of the Conjugate Gradient method is related to the condition number of the matrix, the goal of a preconditioner is to bring the eigenvalues closer together, clustering them, and effectively reducing the condition number [8]. However, this is not the sole criteria for preconditioners, as from this perspective A^{-1} would simply prove to be the perfect preconditioner for A, but would come at the large cost of computing the inverse of A as well, which is exactly what an efficient solver most likely looks to avoid. Therefore, peak performance requires a balance to be struck between the extra cost of applying the preconditioner and the accelerated convergence. If the preconditioner is applied with standard multiplication, ideally it would approximate the inverse of its target yet be computed when multiplying quickly, either through sparsity or some other favorable structure, as must be multiplied every iteration of the iterative solve [12]. The standard Conjugate Gradient algorithm in the code base, CG, is modified to PCG to provide this functionality by simply passing in an additional preconditioner matrix.

Two preconditioners were investigated, the simple diagonal preconditioner and the Incomplete Cholesky preconditioner. Unfortunately, the simple diagonal preconditioner, though sometimes noticeably effective despite is simplicity, has absolutely no effect on the Poisson problem. This is a direct consequence of the fact that the diagonal preconditioner only multiplies a matrix by the inverse of its diagonal entries,

Fig. 5.2. ***Vorticity Streamfunction Solver Streamlines Example 1***



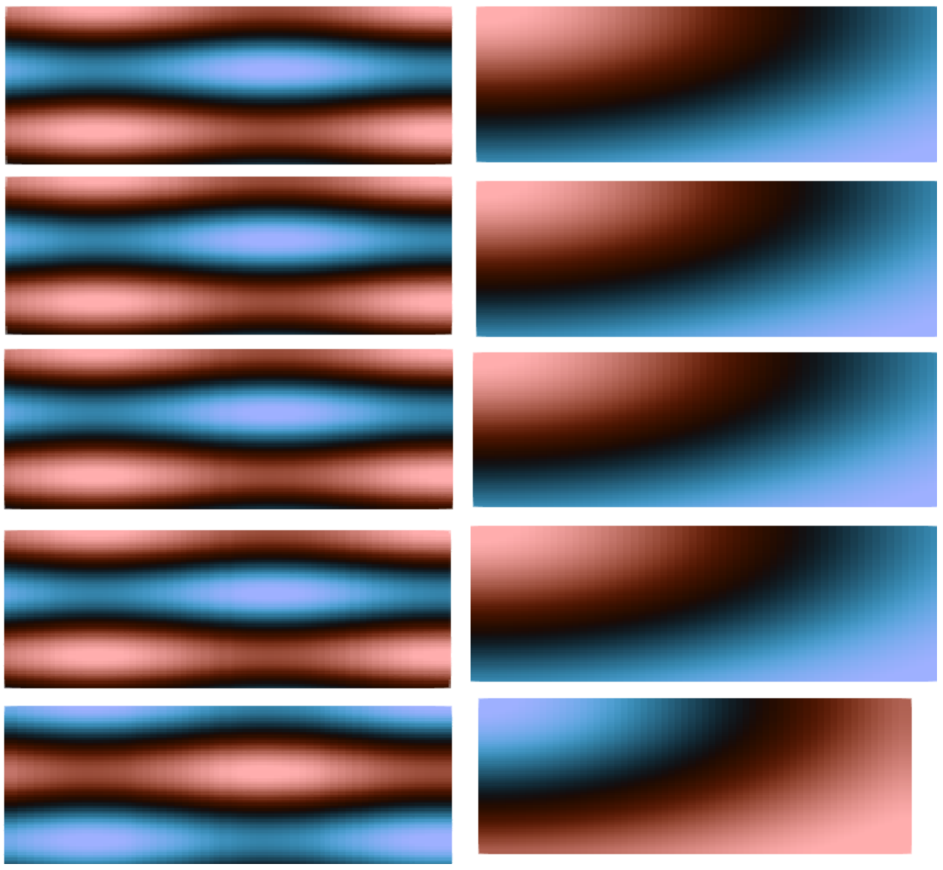
Sample streamlines for equal magnitude top and bottom boundaries moving in positive x direction.

yet since the Poisson matrix always has a constant value along its main diagonal, all rows of the matrix will be scaled equally and as a result the condition number will remain unchanged. However, the Incomplete Cholesky Conjugate Gradient method, one of the first well known preconditioners used for Conjugate Gradient method, shows far more promise both in theory and in practice [5]. The Incomplete Cholesky factorization of A is equivalent to a Cholesky factorization of A with all entries in the factorized triangular matrices equal to zero where A is originally equal to zero. Either the upper or lower triangular matrices can be used, in this implementation the upper triangular matrices are utilized, and since the operation only need to be performed once at the initialization of simulation a standard Cholesky factorization is used and then entries are subsequently zeroed out, though more efficient algorithms can be found. Additionally, what is especially exciting about the preconditioner it outputs is that the factors will be both sparse, having only three diagonal bands, and triangular, making them quite suitable for quick application, one of the above criteria. Multiplying them together and inverting to form a single preconditioner matrix would be needlessly computationally expensive, and a powerful optimization is to perform two triangular solves instead whenever applying the preconditioner.

The implementation of ICCG in the repository uses exactly this ideal, two triangular back solves, as well as additionally tagging the factorized upper triangular matrices as sparse to provide more information to the compiler and accelerate the solving process.

7. Results. Included are tables of runtime and iteration count, taken as the minimum of multiple runs as noise is additive in a system, for the different Poisson equation solving methods previously described when given a specified ϵ residual tolerance parameter. Though a trend can be seen in decreasing iteration count in a mostly expected order among the CG solvers, with PCG having an equal iteration count to ICCG as it is simply inefficiently applying the Incomplete Choleksy factorization to

Fig. 7.1. ***Poisson Equation Solving Tests***



Poisson Equation solvers with were tested for correctness by generating a pressure or streamline, multiplying the pressure field by the Poisson matrix to obtain a source vector, and then checking that the solvers could generate the proper pressure or streamline given the source. Two different inputs are show in the left and right columns with the top entries being the actual values, and in descending order solutions found with CG, PCG, ICCG, and the Poisson Solver from GridSolver, which gives slightly different values as it must incorporate Neumann boundary conditions, while the others and generated input assumed Dirchlet boundary conditions.

Table 7.1
N=64 Poisson Solver Iteration Comparison

ϵ	Grid	CG	PCG	ICCG
1e-4	102	243	73	73
1e-5	102	256	77	77
1e-6	102	265	81	81
1e-7	102	277	85	85
1e-8	102	288	88	88
1e-9	102	297	93	93

Table 7.2
N=64 Poisson Solver Runtime Comparison

ϵ	Grid	CG	PCG	ICCG
1e-4	0.019066	0.044509	0.21656	0.027253
1e-5	0.007698	0.095235	0.744288	0.017501
1e-6	0.08428	0.021050	0.797563	0.020389
1e-7	0.017465	0.033517	0.846498	0.031289
1e-8	0.017893	0.048644	0.943054	0.030626
1e-9	0.022400	0.052771	0.951253	0.042294

Table 7.3
N=32 Poisson Solver Iteration Comparison

ϵ	Grid	CG	PCG	ICCG
1e-4	100	121	38	38
1e-5	99	127	41	41
1e-6	99	133	43	43
1e-7	99	138	46	46
1e-8	99	143	48	48
1e-9	99	148	50	50

Table 7.4
N=32 Poisson Solver Runtime Comparison

ϵ	Grid	CG	PCG	ICCG
1e-4	0.018659	0.001626	0.054517	0.025315
1e-5	0.005894	0.007745	0.033539	0.005081
1e-6	0.006191	0.007203	0.038014	0.004204
1e-7	0.006736	0.005831	0.044570	0.005129
1e-8	0.005580	0.007082	0.043993	0.011221
1e-9	0.008356	0.008070	0.036385	0.005279

the same effect, the consistency of the Grid Solver is almost as surprising as the unpreconditioned Conjugate Gradient's method extremely high iteration count. In terms of runtime, PCG is unsurprisingly the slowest as it is multiplying a dense matrix, CG is slightly better, and out of all of the three conjugate gradient methods, Incomplete Cholesky Conjugate Gradient performs best with its sparse solves, multiplications, and decreased iteration count due to conditioning. However, rather unexpectedly none of the methods outperform the pressure solve for the initial GridSolver, with ICCG matching it occasionally for lower matrix sizes. The memory allocations however around the following values for Grid Solver with N = 32 : (3.38 k allocations: 6.517 MiB), Grid Solver with N = 64 :(4.40 k allocations: 27.525 MiB), Vorticity Stream Solver with N = 32 : (377 allocations: 2.960 MiB), and the Vorticity Stream Solver for N = 64 : (1.27 k allocations: 19.861 MiB).

Even though ICCG uses two and a half times fewer allocations for N=64, almost an order of magnitude fewer allocations for N=32, and overall almost twice as less memory usage, GridSolver still outperforms ICCG by approximately a factor of over 2 with regards to runtime. Possible explanations include build up of floating point error in the Conjugate Gradient method leading to extra unecessary iterations, good vectorization and use of vector operations in GridSolver, a subpar choice of preconditioner (meaning there may be more benefits to be gained trying another preconditioner such as multigrid), or finally perhaps given the extensive sparsity of the Poisson matrix, the optimal path towards solving the Poisson equation more quickly than the vectorized operations GridSolver uses may belong to sparse linear solvers. However, as the discrepancy in runtime only increases for larger N, there is a multitude of potential explanations which remain to be tested and no definite, conclusive statements can yet be drawn as to the main underlying causes.

8. Conclusion. Two separate two dimensional incompressible Navier Stokes Solvers were developed, GridSolver, a solver based off of the SIMPLE algorithm decoupling the pressure and velocity equations and iterating to provide corrective steps, with the majority of its speed relying on Julias vectorized array operations, and the Vorticity Streamfunction solver, created in an attempt to optimize the Poisson equation solving bottleneck in GridSolver using an implementation of the Conjugate Gradient algorithm and its more advanced successors, namely the Incomplete Cholesky Conjugate Gradient method. Both are functional and ready to use in the code, though as discussed in the results, while the Vorticity Streamfunction solver was successful in decreasing the number of iterations needed to solve the Poisson equation and uses a factor of roughly 2.5 times fewer allocations, the GridSolver overall remained faster. As such, they may be best viewed as two very different approaches to solving the same fluid flow problem, each with their separate trade-offs, and finally, as two opportunities to investigate unique solution methods and possible routes for optimization.

REFERENCES

[1] A. Salih, *Streamfunction-Vorticity Formulation*, Indian Institute of Space Science and Technology, Thiruvananthapuram, (2013), pp. 1–10, https://doi.org/10.1017/9781108855297.004.

[2] Alex Townsend, *Krylov subspace methods*, 2019, http://pi.math.cornell.edu/~web6140/TopTenAlgorithms/KrylovSubspace.html (accessed 2020/12/01).

[3] Ansys Inc., *Ansys Pressure-Velocity Coupling*, 2009, https://www.afs.enea.it/project/neptunius/docs/fluent/html/th/node373.htm (accessed 2020/10/25).

[4] Barton, I.E, *Comparison of SIMPLE- and PISO-type algorithms for transient flows*, Int. J. Numer. Meth., 26 (1998), pp. 459–483, https://doi.org/10.1002/(SICI)1097-0363(19980228)26:4⟨459::AID-FLD645⟩3.0.CO;2-U.

[5] I. Arany, *The Preconditioned Conjugate Gradient Method with Incomplete Factorization Preconditioners*, Elsevier, 31, https://doi.org/10.1016/0898-1221(95)00210-3.

[6] Jonathan Richard Shewchuk, *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*, School of Computer Science Carnegie Mellon University, (1994), pp. 1–58, https://doi.org/6491967.

[7] Maciej Matyka, *Solution to two-dimensional Incompressible Navier-Stokes Equations with SIMPLE, SIMPLER and Vorticity-Stream Function Approaches. Driven-Lid Cavity Problem: Solution and Visualization*, University of Wroclaw, (2004), pp. 1–13, https://doi.org/arXiv:physics/0407002.

[8] Mihai Anitescu, *Conjugate Gradient Convergence/Preconditioning*, 2012, https://www.mcs.anl.gov/~anitescu/CLASSES/2012/LECTURES/S310-2012-lect6.pdf (accessed 2020/12/08).

[9] Oceananigans Documentation, *Oceananigans.jl*, 2020, https://clima.github.io/OceananigansDocumentation (accessed 2020/10/20).

[10] Qiqi Wang, *Numerical Methods of Partial Differential Equations*, 2014, https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-90-computational-methods-in-aerospace-engineering-spring-2014/lecture-videos/session-7-numerical-methods-of-partial-differential-equations-introduction/ (accessed

2020/11/26).

[11] QuickerSim Inc., *Tutorial 2 - Numerics SIMPLE Scheme*, 2017, https://quickersim.com/tutorial/tutorial-2-numerics-simple-scheme/ (accessed 2020/10/28).

[12] Stanford University, *Conjugate Gradient Method*, https://web.stanford.edu/class/ee364b/lectures/conj_grad_slides.pdf (accessed 2020/12/02).

[13] Suraj Pawar and Omer San, *CFD Julia: A Learning Module Structuring an Introductory Course on Computational Fluid Dynamics*, MDPI Open Access Journals, 4 (2019), pp. 1–77, https://doi.org/10.3390/fluids4030159.