PICT: Adaptive GPU Accelerated Differentiable Fluid Simulation for Machine Learning

Aleksandra Franz [1] **Nils Thuerey** [1]

Abstract

In this paper we present *PICT*, our differentiable 2D and 3D fluid simulator for machine learning in the PyTorch framework with support for GPU acceleration. We implemented the PISO algorithm using custom CUDA operations for the core components and Python for the overall algorithm to achieve GPU performance while keeping the simulation easily customizable. To support geometry beyond toy examples we support spatially adaptive multi-block grids using a generalized coordinate system. This allows the user to refine the grid as necessary and align it to boundaries. The forward simulation is validated using analytical and numerical references as well as long rollouts for stability. The gradients of individual components are checked numerically, and we conducted non-trivial optimization and learning tests to verify the usability of our gradients.

1. Introduction

Modeling and numerical simulation of fluid phenomena, governed by the well-known Navier-Stokes (NS) equations, has a long history in computer science. Computational fluid dynamics (CFD) have many applications in weather forecasting [1], medicine [2], and mechanical design [3], but suffer from heavy computational complexity, especially when employing direct numerical simulation (DNS). While established methods like Reynolds-averaged Navier–Stokes (RANS) and large eddy simulation (LES) aim to reduce this computational complexity, one area of research that is attracting increasing attention is to employ machine learning (ML) methods to improve the performance of simulators. One way is to completely replace the solver by a network that has learned the dynamics from data, but this has proven to be unreliable in generalization and long rollouts [4]. A more promising one is replacing or augmenting costly parts of existing approaches with learned components in hybrid solvers. Properly training such hybrid solvers often requires the rest of the simulator to be differentiable to allow for the more stable end-to-end training in which the learned components can correct for their own prediction errors that would otherwise accumulate in longer trajectories. Another avenue that can benefit from differentiable solvers is learned control tasks [5], [6] and design optimization [7]. Despite the apparent interest in differentiable solvers, only very few such methods exist [8], [9]. Bezgin et al. [10] argue that this may be due to the mismatch between the typically low-level implementation of high-performance solvers and high-level machine learning frameworks, and subsequently introduced JAX-Fluids [10] as a python framework that bridges this gap.

We follow a similar avenue, albeit with different choices for algorithms and discretization, and developed *PICT*, implementing the **PI**SO algorithm by Issa [11] for incompressible flows in **C**UDA for GPU acceleration for the Py**T**orch machine learning framework. Our solver is intended as a general DNS solver that can be used with gradient-based methods for various optimization tasks and applications and supports spatially adaptive meshes by using a generalized coordinate system [12], [13] and multi-block grids. This allows the user to refine the grid as necessary and align it to arbitrary boundaries while keeping the grid topology a regular Cartesian grid structure for easier memory handling and straightforward connection to convolutional neural networks (CNN). For the connection to auto differentiation (AD), we implement analytical gradients for the core components of the PISO algorithm, including an implicit Euler step, and integrate them for use with PyTorch's AD. As is common for ML applications, we implement our solver for GPU for better runtime performance. To validate our implementation, we compare forward simulations to existing analytical or numerical references. We also show our solver's stability in long turbulent rollouts. For differentiation, we validated the gradients of individual operations numerically and show that backpropagation through long rollouts is stable and provides meaningful gradients for both direct optimization

[1]Technical University of Munich, Germany; TUM School of Computation, Information and Technology, Department of Computer Science. Correspondence to: Aleksandra Franz <franzer@in.tum.de>.

Published at the 2^{nd} Differentiable Almost Everything Workshop at the 41^{st} International Conference on Machine Learning, Vienna, Austria. July 2024.

of physical quantities, i.e., viscosity and boundary velocity, and for learning a control force with a CNN.

2. Method

2.1. Governing Equations and Algorithm

The governing NS equations to simulate incompressible flows take the form of momentum

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) - \nu \nabla^2 \mathbf{u} = -\nabla p + S \tag{1}$$

and continuity

$$\nabla \cdot \mathbf{u} = 0, \tag{2}$$

with velocity $\mathbf{u}$, pressure p, viscosity ν, time t, and external sources S.

For the numerical simulation we use the PISO algorithm introduced by Issa [11]. It comprises a predictor step to solve the momentum equation (1) and advance the simulation in time, followed by typically 2 corrector steps to enforce continuity (eq. (2)) on the result. For the predictor step

$$\frac{1}{\Delta t}\mathbf{u}^* + \nabla \cdot (\mathbf{u}^n \mathbf{u}^*) - \nu \nabla^2 \mathbf{u}^* = \frac{1}{\Delta t}\mathbf{u}^n - \nabla p + S^{n+1} \tag{3}$$

the velocity is split into velocity from previous step $\mathbf{u}^n$ (advecting) and the velocity guess $\mathbf{u}^*$ (advected). In matrix form this advection is $C\mathbf{u}^* = \frac{1}{\Delta t}\mathbf{u}^n - \nabla p + S$, which we solve with an implicit Euler step.

For the corrector step the matrix C is split into its diagonal A and off-diagonal entries H. With $h = -H\mathbf{u}^* + \frac{\mathbf{u}^0}{\Delta t}$, the pressure correction comes from the linear system

$$\nabla^2 (A^{-1} p^*) = \nabla \cdot \left(A^{-1} h + A^{-1} S^{n+1} \right) \tag{4}$$

which is solved for the pressure p. This pressure is then used to compute the corrected, divergence-free velocity $\mathbf{u}^{**}$ with

$$\mathbf{u}^{**} = A^{-1} h - A^{-1} \nabla p^* + A^{-1} S^{n+1} \tag{5}$$

The pressure correction, equations 4 and 5, are repeated twice [11], with an additional * indicating the second update. The velocity of the next time-step is then $\mathbf{u}^{n+1} := \mathbf{u}^{***}$.

2.2. Discretization

For discretization we use the finite volume method (FVM) based formulation as described by Maliska [13]. At its core, it uses the divergence theorem to convert the divergence operators $\nabla\cdot$ of the governing equations to sums over discrete faces $\mathbf{f}$ of final volume elements.

$$\nabla \cdot \mathbf{u} \approx \sum_{\mathbf{f}} \mathbf{u}_f \cdot \vec{n}_f a_f, \tag{6}$$

where $\vec{n}_f$ is the face normal and a_f its area.

Transformations To align grid axes to physical boundaries and support refinement in areas of interest we include the option to transform the vertices of the regular grids. Since we use a FVM-based formulation, the face fluxes created from eq. (6) need to take the new physical size and orientation of the now-transformed faces into account. To handle these mesh transformations we use the generalized coordinate system as described by Kajishima and Taira [12] and Maliska [13], which effectively scales a_f and rotates $\vec{n}_f$, but allows to precompute the required factors from the mesh coordinates. We only support static meshes that do not change during the simulation, but a discussion of temporally changing grids can be found in [13].

Block structure To handle complex geometry, we use a multi-block grid where the domain of interest can be split into multiple blocks, but each block is still a regular grid with its own velocity and pressure tensor that together make up the global field. Advection and pressure are still solved for the whole domain. Each side of a block can have one boundary specified, either a connection with matching resolution or a prescribed quantity. The block connections allow for more complicated meshes while keeping the regular structure for most of the memory. This also fits nicely for the connection to CNNs where connections could be handled by padding with a ghost layer of connected blocks.

Boundary Conditions In addition to the connections between blocks, we support Dirichlet boundaries for the velocity, with Neumann being a possible extension, and an advective outflow boundary. The advective outflow updates the boundary between each PISO step by advecting the block's boundary cell layer into the boundary with some characteristic velocity $\mathbf{u}_m$ to satisfy $\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}_m \frac{\partial \mathbf{u}}{\partial x_i} = 0$. This prevents the boundary from reflecting flow structures back into the domain [14]. During the PISO step the boundary is then treated as a fixed Dirichlet boundary. The pressure boundary conditions for Dirichlet velocity boundaries are implicitly 0-Neumann, for the implementation these prescribed boundaries can be largely ignored as the pressure correction should not change them.

2.3. Implementation

Our solver is implemented as C++/CUDA module for Python containing the individual operations needed to build the PISO algorithm with implicit Euler advection. It comprises the setup of the advection/diffusion linear system (matrix and RHS of eq. (3)), a preconditioned BiCGStab GPU solver for advection, the setup of the pressure system (matrix and RHS of eq. (4)), a simpler CG solver for pressure, and the pressure correction for the predicted velocity field (eq. (5)). It also includes the data structure necessary to store the multi-block structure with its tensors and con-

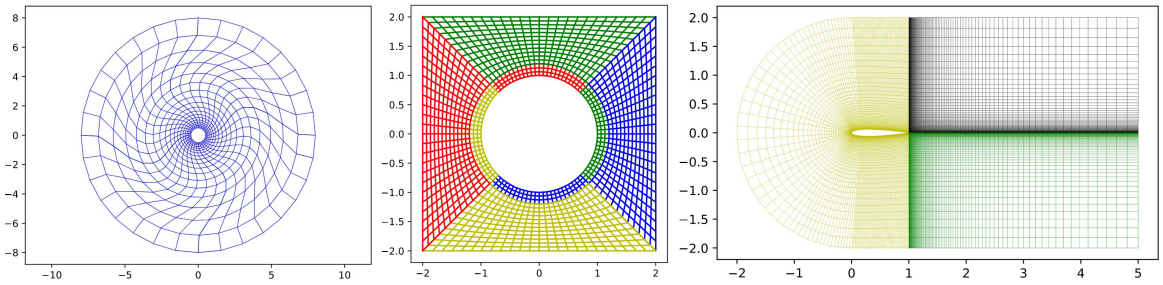

Figure 1. Some example transformed multi-block meshes that can be handled by our simulator. From left to right: a torus grid with rotational distortion, a (rather bad) mesh fitting a round obstacle in a block, a refined C-grid around an airfoil.

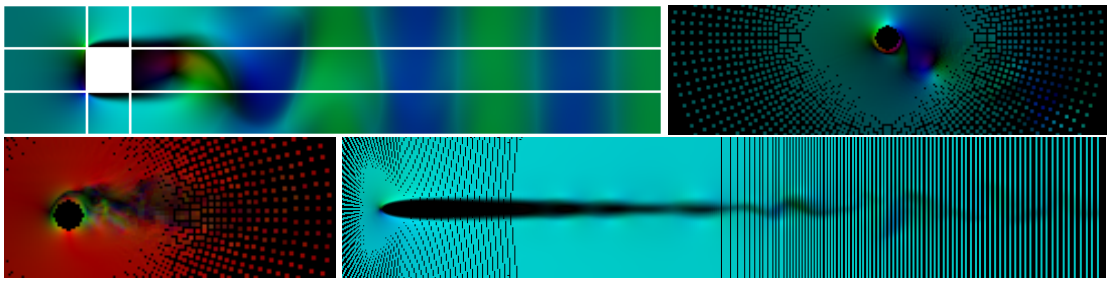

Figure 2. Velocity of the final frame of various forward simulations. For 2D the velocity is mapped to a color circle, for 3D the absolute velocity vector is mapped directly to RGB. Top left: a multi-block vortex street setup with a rectangular obstacle, the white lines show block connections. Top right: a crop of a vortex street using a torus grid and a rotating round obstacle. The cell centers are resampled to a pixel grid, which leaves some empty black areas. Bottom left: a crop of the center slice of a 3D flow around a rotating cylinder. Bottom right: a flow around a NACA 0012 airfoil, also resampled and cropped.

nections and make them accessible on the GPU. The final, combined PISO algorithm is then implemented in Python to allow for easier customization.

2.4. Differentiability

To support the differentiability needed for integration with ML applicated we implement analytical gradients for the individual operations. For the operations that build the linear systems, this is mainly done by ”inverting” the forward compute graph. For the linear solvers we do not backpropagate through the solution procedure of $Ax = b$, but instead solve the system $A^T \partial b = \partial x$ for ∂b [15]. The gradient w.r.t. the matrix entries is then the outer product $\partial A = -\partial b \otimes x$. Since A is a sparse matrix, only elements that exist on A are used for ∂A. For the connection to PyTorch AD the individual differentiable operations are wrapped in Python to enable the necessary tensor tracking.

3. Results

First we test various forward simulations to validate the accuracy and long-term stability of our solver. Then we show optimization and learning setups to verify gradient backpropagation through longer rollouts beyond numerically checked gradients.

3.1. Validation of the Forward Simulation

We validated our solver using the analytical solution of the Plane Poiseuille flow, and numerical references for 2D [16] and 3D [17] lid-driven cavity setups. Here we also tested different Reynolds numbers, grid refinement towards the closed boundaries, rotational distortions of the grid, and permutations of the lid boundary for the lid-driven cavity setups. See appendix B for more details and result graphs.

Flow Around Obstacle We investigated flows around various obstacles to test solver stability. The first test is a simple vortex street with a box obstacle and 8 computational blocks, as shown in figure 2. Inflow is from the left, and the domain is periodic in y-direction. The results show a consistent and stable vortex shedding behavior. Our second test is another vortex street with a round obstacle inside a torus grid. Here the grid is split into 2 blocks, left and right of the obstacle, to handle the now curved in- and outflow boundaries. We further prescribed a tangential motion at the obstacle boundary to simulate a clockwise rotating obstacle. This induces vortex shedding from the start of the simulation and redirects it from the center line, downwards in our case. Then we extrude this torus vortex street to 3D with inverted obstacle rotation which shows a more chaotic behaviour. Lastly, we simulate the flow on a 3-block C-grid around a

Table 1. Runtime Performance Metrics. The columns contain: sim steps: length of the simulation in PISO-steps. opt it: optimization iterations for learning tasks, each including a full simulation. max mem: peak GPU memory usage. sim time: wall clock time for the forward simulation, total and per simulation step, including network evaluation. opt time: is the same for the backpropagation and variable update. The first 4 tests are forward simulations, followed by our 2 optimization tests. All test were run on a single Nvidia RTX A5000.

Test	resolution	#cells	sim steps	opt it	max mem	sim time	opt time
Block Vortex Street	512×96	48k	4102	-	32MB	13.5m (174ms)	-
Torus Vortex Street	128×60	7680	6585	-	17MB	10m (90ms)	-
3D Vortex Street	$128 \times 60 \times 32$	245k	5755	-	181MB	43m (451ms)	-
Airfoil	791×143	113k	14788	-	200MB	162m (658ms)	-
Direct Optimization	32×32	1024	70 - 400	100	190MB	7m (45ms)	3m (22.5ms)
Learned Control	64×32	2048	500	500	605MB	3h (40ms)	1h (14ms)

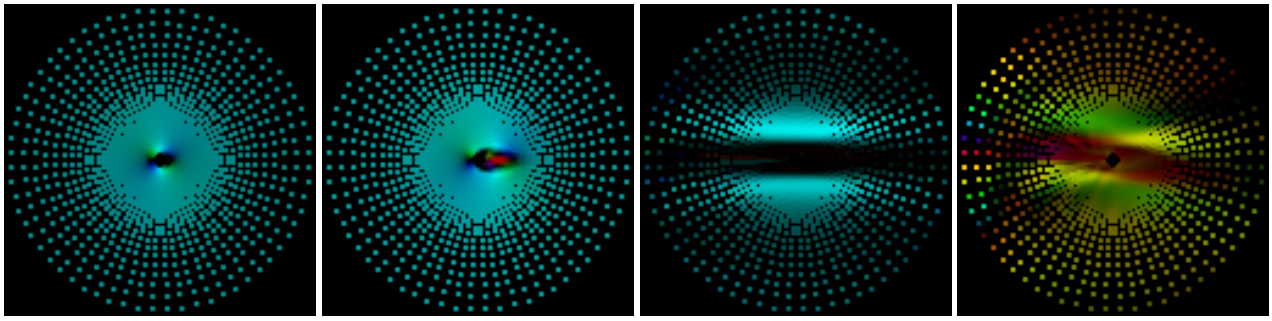

Figure 3. The final frame of the learned control test. From left to right: the first velocity of the simulation, the final velocity of a simulation without control force, the final velocity with learned control force applied over the simulation, the control force of the final frame. The velocity is mapped to a color circle and normalized in each visualization.

NACA 0012 airfoil. The left, upper, and lower boundaries are prescribed with the inflow velocity, while we use the advective outflow condition on the right. Visualizations of the results are in fig. 2, sequences can be found in the appendix.

3.2. Optimization and Learning

The gradients of our individual operations are validated numerically using PyTorch's gradcheck [18]. We further investigated a simple direct optimization of a flow quantity before training a neural network with gradients from our solver. For the direct optimization we use a lid-driven cavity setup and optimize both viscosity and lid velocity using simple gradient descent (GD). As loss we use a L2 loss to a reference simulation, using only the final velocity at the horizontal and vertical center lines, similar to the velocity profiles of the lid-driven cavity validation. Over the course of the optimization both quantities converge towards their targets, showing that the gradients point consistently in the right direction even when backpropagating through a up to 400 steps long rollout. More information is in appendix D.

Learned Control To showcase training a CNN within our solver we use the torus vortex street setup with a slightly rotating obstacle to induce vortex shedding behavior from the start. With this we train a simple 3-layer CNN Θ (3x3 kernel, stride 1, ReLU activation) that maps the velocity to the forcing term of the next step $\Theta(\mathbf{u}^n) = S^{n+1}$. The objective is to minimize the y-velocity after the following PISO step, i.e. $L^{n+1} = \left(\mathbf{u}_y^{n+1}\right)^2$. The simulation is run for 500 iterations, and Θ and L are evaluated and applied at every step. Gradients are then backpropagated through the 500 iterations rollout. In the end the network learns to keep the flow away from the obstacle, which is shown in fig. 3.

4. Conclusion

In summary, we presented our differentiable fluid simulator and its integration with ML applications. We showed its stability and accuracy in forward simulations and the efficacy of the provided gradients for optimization tasks. While our simulator is robust and works well in the investigated scenarios, there are still some limitations: not every quantity is made differentiable, most notably the mesh transformations. The quality of the mesh is also very important for stable results, as large variations between neighboring cells, such as seen in the middle mesh in fig. 1, quickly deteriorate the results. We anticipate that the combination of a powerful, efficient, and differentiable flow solver will enable a large variety of interesting learning objectives that are highly relevant for practical applications. We plan to make the solver publicly available as open-source.

References

[1] P. Bauer, A. Thorpe, and G. Brunet, "The quiet revolution of numerical weather prediction," *Nature*, vol. 525, no. 7567, pp. 47–55, 2015. DOI: 10.1038/nature14956.

[2] B. M. Johnston, P. R. Johnston, S. Corney, and D. Kilpatrick, "Non-newtonian blood flow in human right coronary arteries: Steady state simulations," *Journal of Biomechanics*, vol. 37, no. 5, pp. 709–720, 2004, ISSN: 0021-9290. DOI: 10.1016/j.jbiomech.2003.09.016.

[3] J. D. Denton and W. N. Dawes, "Computational fluid dynamics for turbomachinery design," *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, vol. 213, no. 2, pp. 107–124, 1998. DOI: 10.1243/0954406991522211.

[4] B. List, L.-W. Chen, and N. Thuerey, "Learned turbulence modelling with differentiable fluid solvers: Physics-based loss functions and optimisation horizons," *Journal of Fluid Mechanics*, vol. 949, A25, 2022. DOI: 10.1017/jfm.2022.738.

[5] F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and J. Z. Kolter, "End-to-end differentiable physics for learning and control," *Advances in neural information processing systems*, vol. 31, 2018.

[6] M. Li, R. Antonova, D. Sadigh, and J. Bohg, "Learning tool morphology for contact-rich manipulation tasks with differentiable simulation," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 1859–1865. DOI: 10.1109/ICRA48891.2023.10161453.

[7] K. R. Allen, T. Lopez-Guevara, K. Stachenfeld, *et al.*, *Physical design using differentiable learned simulators*, 2022. arXiv: 2202.00728 [cs.LG].

[8] Y. Hu, L. Anderson, T.-M. Li, *et al.*, "Difftaichi: Differentiable programming for physical simulation," *ICLR*, 2020.

[9] P. Holl and N. Thuerey, "Phiflow: Differentiable simulations for pytorch, tensorflow and jax," in *International Conference on Machine Learning*, 2024.

[10] D. A. Bezgin, A. B. Buhendwa, and N. A. Adams, "Jax-fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows," *Computer Physics Communications*, vol. 282, p. 108 527, 2023, ISSN: 0010-4655. DOI: 10.1016/j.cpc.2022.108527.

[11] R. I. Issa, "Solution of the implicitly discretised fluid flow equations by operator-splitting," *Journal of computational physics*, vol. 62, no. 1, pp. 40–65, 1986.

[12] T. Kajishima and K. Taira, *Computational fluid dynamics: incompressible turbulent flows*. Springer, 2016.

[13] C. R. Maliska, *Fundamentals of computational fluid dynamics: the finite volume method*. Springer Nature, 2023, vol. 135.

[14] T. Poinsot and S. Lelef, "Boundary conditions for direct simulations of compressible viscous flows," *Journal of Computational Physics*, vol. 101, no. 1, pp. 104–129, 1992, ISSN: 0021-9991. DOI: 10.1016/0021-9991(92)90046-2.

[15] M. Giles, "An extended collection of matrix derivative results for forward and reverse mode automatic differentiation," 2008.

[16] U. Ghia, K. Ghia, and C. Shin, "High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method," *Journal of Computational Physics*, vol. 48, no. 3, pp. 387–411, 1982, ISSN: 0021-9991. DOI: 10.1016/0021-9991(82)90058-4.

[17] S. Albensoeder and H. Kuhlmann, "Accurate three-dimensional lid-driven cavity flow," *Journal of Computational Physics*, vol. 206, no. 2, pp. 536–558, 2005, ISSN: 0021-9991. DOI: 10.1016/j.jcp.2004.12.024.

[18] PyTorch. "Gradcheck mechanics." (2023), [Online]. Available: https://pytorch.org/docs/stable/notes/gradcheck.html (visited on 06/18/2024).

A. Implementation Details

Fig. 4 shows a visualization of the data flow in our solver. The boxes are individual differentiable operations implemented in C++/CUDA and wrapped in Python while the remaining structure of the PISO algorithm is realized directly in Python.

B. Validation Results

B.1. Plane Poiseuille flow

The plane Poiseuille flow is a simple 2D test case in which the NS equations simplify to have the analytic solution $u = \frac{G}{2*\nu} y(1-y)$. It is a flow through a periodic channel with closed no-slip boundaries and a constant forcing G. In our test we use $\nu = 1$ and $G = 1$ and tested growing resolutions and refinement towards the closed boundaries. All resolutions agree well with the analytic solution, as can be seen in fig. 5. For non-orthogonal grid transformations we also tested a grid with rotational distortion in the center of the grid (not shown).

B.2. Lid-Driven Cavity

We compare a converged lid-driven cavity simulation to high-res DNS references for 2D [16], see Fig. 6 and 7 and 3D [17], see Fig. 8, for different Reynolds numbers and with grid refinement towards the boundaries. With increasing resolution, the solution converges to the reference. For higher Reynolds numbers (Fig. 7 and 8) the refinement, shown in the right pair, further improves the results, while at lower Reynolds numbers the uniform grid performs better. Additionally, we tested permutations of the lid and its velocity direction and rotational distortions of the grid (not shown). The results on a distorted grid are impacted by the worse mesh quality but are still stable and close to the reference.

C. Simulation Sequences

Figures 9 to 13 show several interesting frames from different simulations. Note that compared to the 'sim steps' from Table 1 a frame can consist of multiple simulation steps. The last frame shown is always the final step of the simulation.

D. Optimization

As mentioned in sec. 3.2, we run direct optimizations (no network training) on two different low-dimensional flow quantities, namely viscosity and lid velocity, in the same lid-driven cavity setup that we also used for the simulation validation above. Here, we use a 2D setup with a resolution of 32×32 and closed no-slip boundaries. The boundary at the lower y-border moves in x-direction as the driving lid. As objective for the optimization we use a L2 loss to the velocity of a reference simulation. This loss is evaluated only on the last frame of the simulation and only on the horizontal and vertical center lines of the grid, meaning that this sparse supervision is backpropagated through the complete simulation rollout. We use simple gradient descent without momentum for the optimization. The learning rate depends on the quantity optimized. When optimizing a quantity, it is initialized as $u_{init} = 1$ for lid velocity or $\nu_{init} = 0.005$ for viscosity. The target values are $u_{tar} = 0.2$ and $\nu_{tar} = 0.001$ respectively. This results in both the initial and target state having $Re = 200$. A quantity that is not optimized in a test is set to its target value. We run the optimization for 100 iterations (variable updates). Each iteration runs a simulation for 10 time units with adaptive time-steps sizes based on the current lid velocity. When optimizing the quantities individually they converge against their respective value used in the reference simulation, as expected, see fig. 14 and 15. However, there is no unique solution given the simple objective when jointly optimizing viscosity and lid velocity, where a higher velocity can compensate for a lower viscosity and vice versa. While this results in flows of different Reynolds number that are visually distinct, it still converges to a solution with low loss. The exact solution found depends on the relative learning rates used for the optimization, as shown in fig. 16.

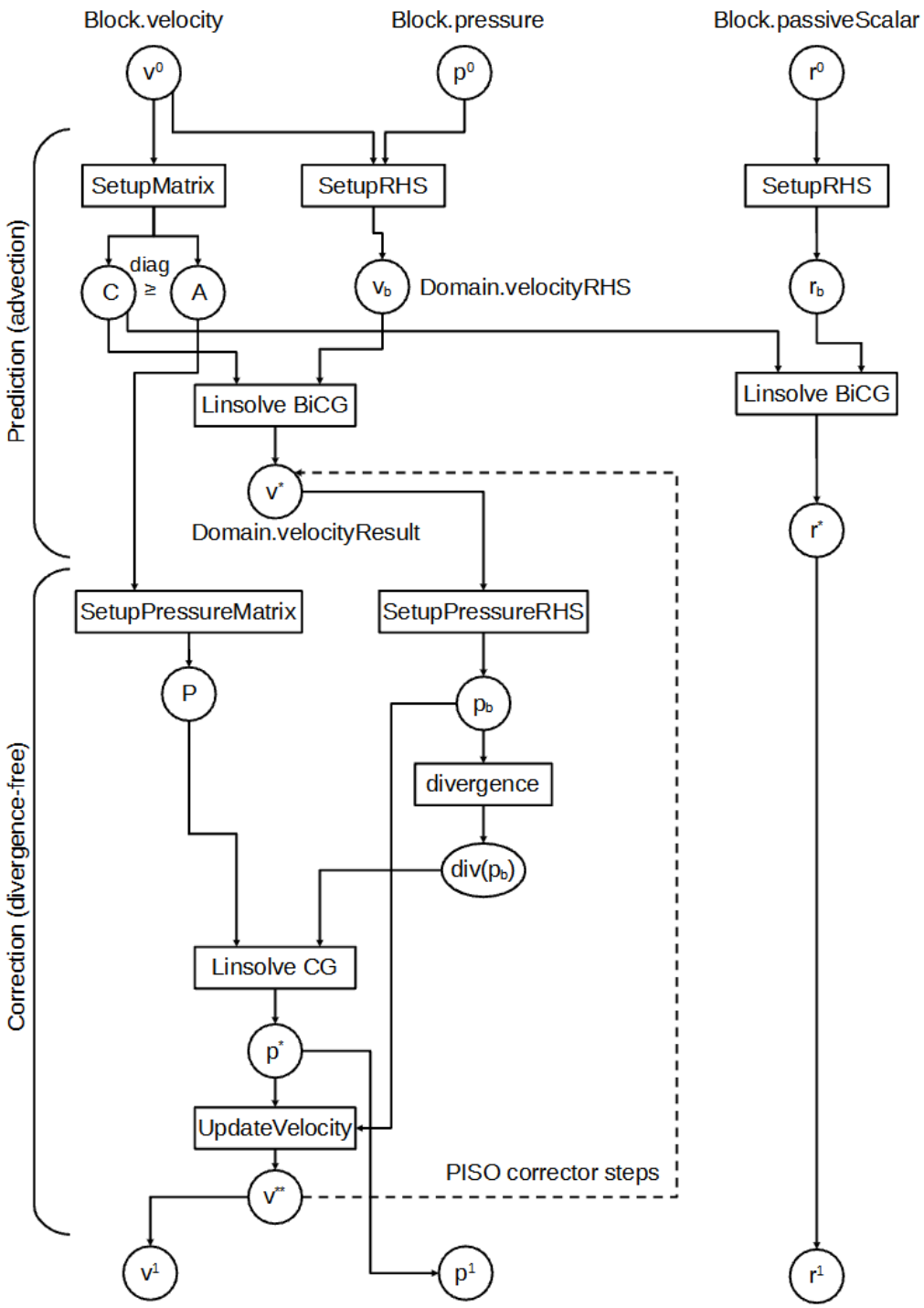

Figure 4. A flow chart showing the implemented components of our solver and their interaction.

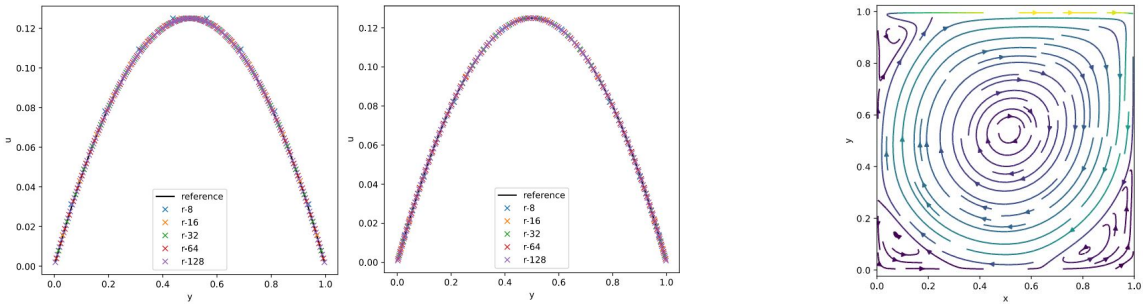

Figure 5. The 2 graphs on the left show vertical u-velocity profiles for the plane Poiseuille flow for increasing resolution from 8×8 to 128×128. Left is a uniform grid, middle uses a grid refined towards the closed boundaries. On the right is a stream plot of the 2D lid-driven cavity with Re 5000 and resolution 128×128 with the lid at the top moving to the right.

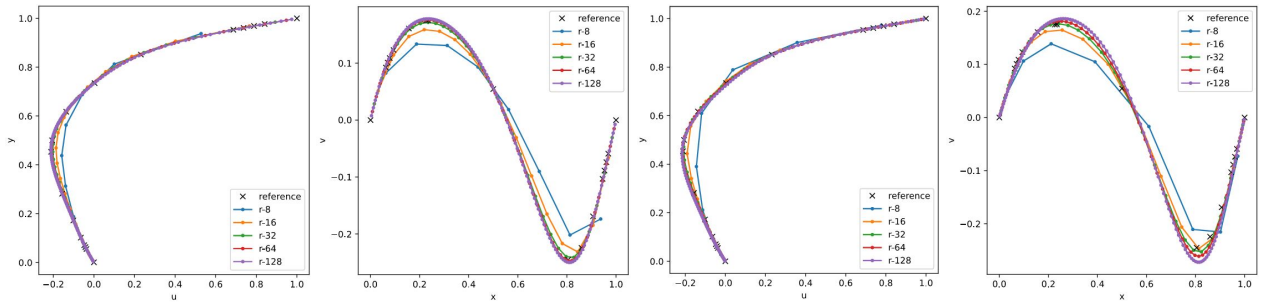

Figure 6. Velocity profiles for the 2D lid-driven cavity with Re 100 for increasing resolutions. The left image of a pair is the u-velocity on the vertical center line, and the right is the v-velocity on the horizontal center line. The left pair uses a uniform grid, the right a grid that was refined towards all boundaries.

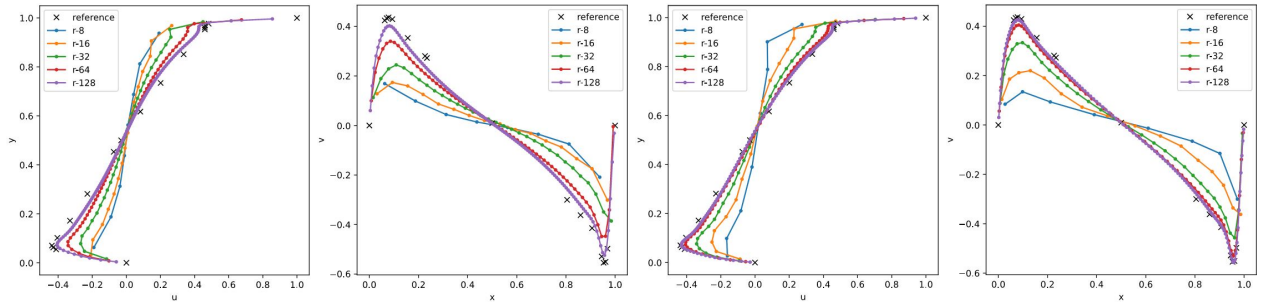

Figure 7. Velocity profiles for the 2D lid-driven cavity with Re 5000 for increasing resolutions. The left image of a pair is the u-velocity on the vertical center line, and the right is the v-velocity on the horizontal center line. The left pair uses a uniform grid, the right a grid that was refined towards all boundaries.

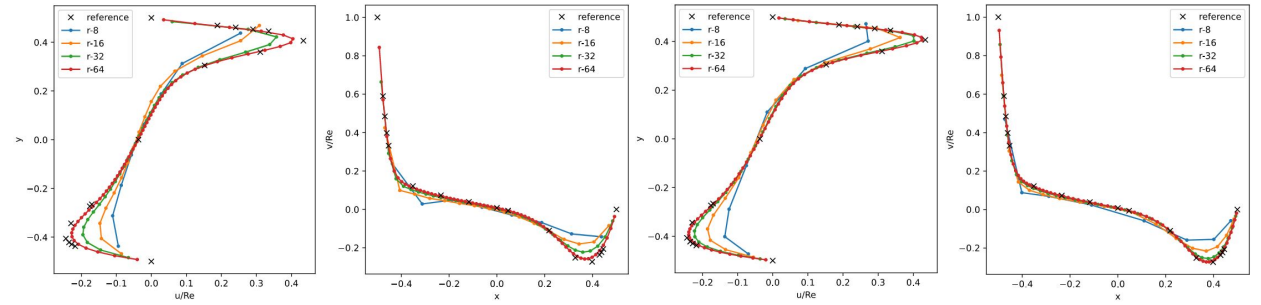

Figure 8. Velocity profiles for the 3D lid-driven cavity with Re 1000 for increasing resolutions. The left image of a pair is the u-velocity on the vertical center line, and the right is the v-velocity on the horizontal center line. The left pair uses a uniform grid, the right a grid that was refined towards all boundaries.

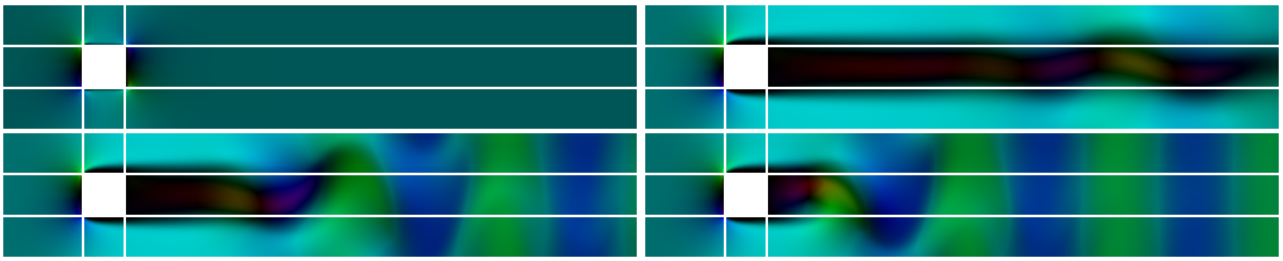

Figure 9. Simulation of a 2D flow around a box obstacle. Top left: frame 1, top right: frame 260, bottom left: frame 350, bottom right: frame 500.

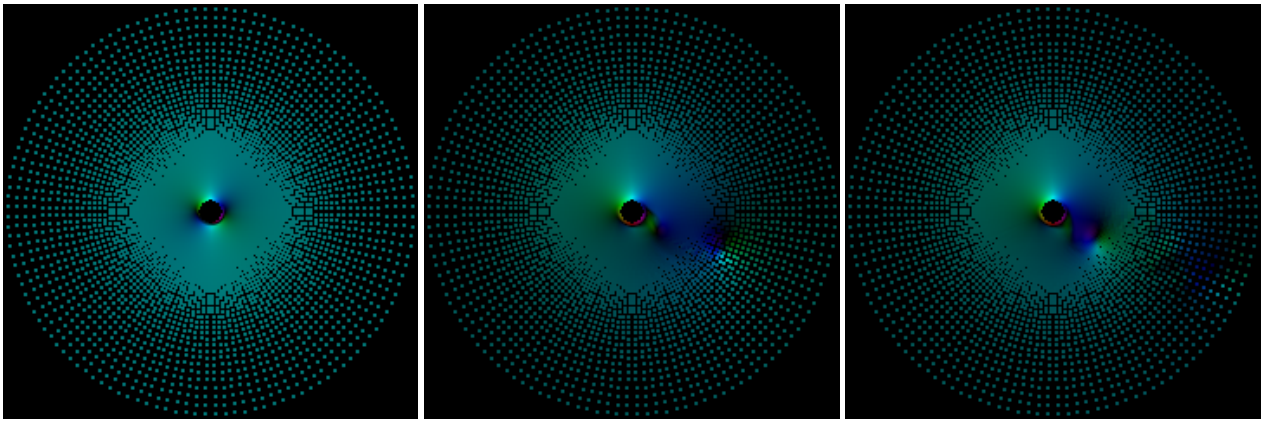

Figure 10. Simulation of a 2D flow around a round obstacle that is rotating clockwise. Left to right: frame 1, 25, 200.

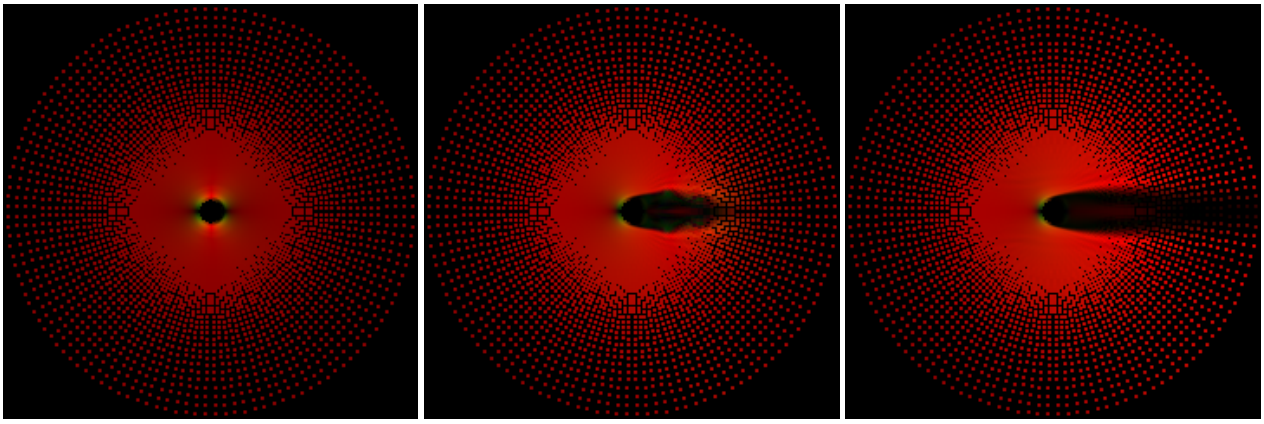

Figure 11. Simulation of a 3D laminar flow around a round obstacle, middle slice in z-dimension. Left to right: frame 1, 25, 200.

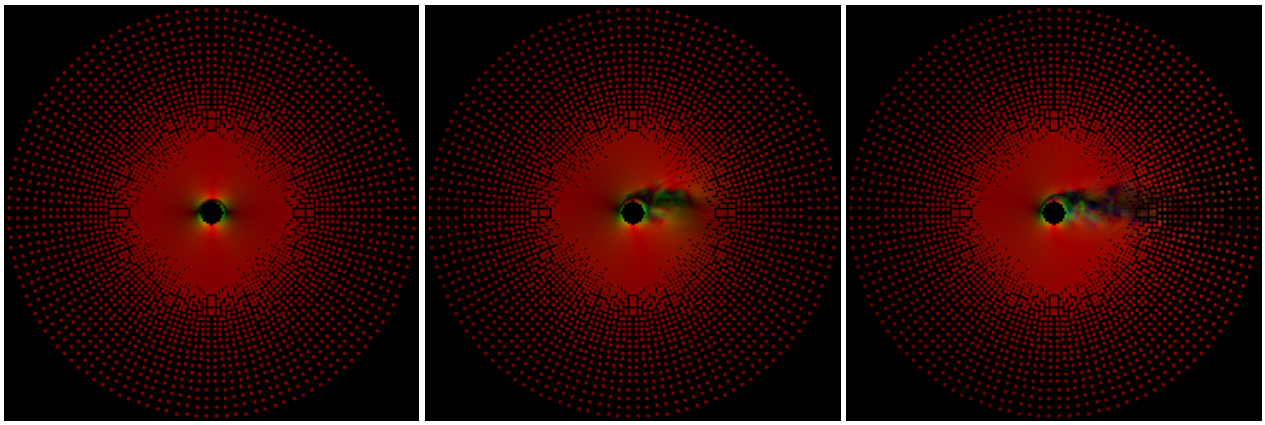

Figure 12. Simulation of a 3D turbulent flow around a round obstacle that is rotating counter-clockwise, middle slice in z-dimension. Left to right: frame 1, 25, 200.

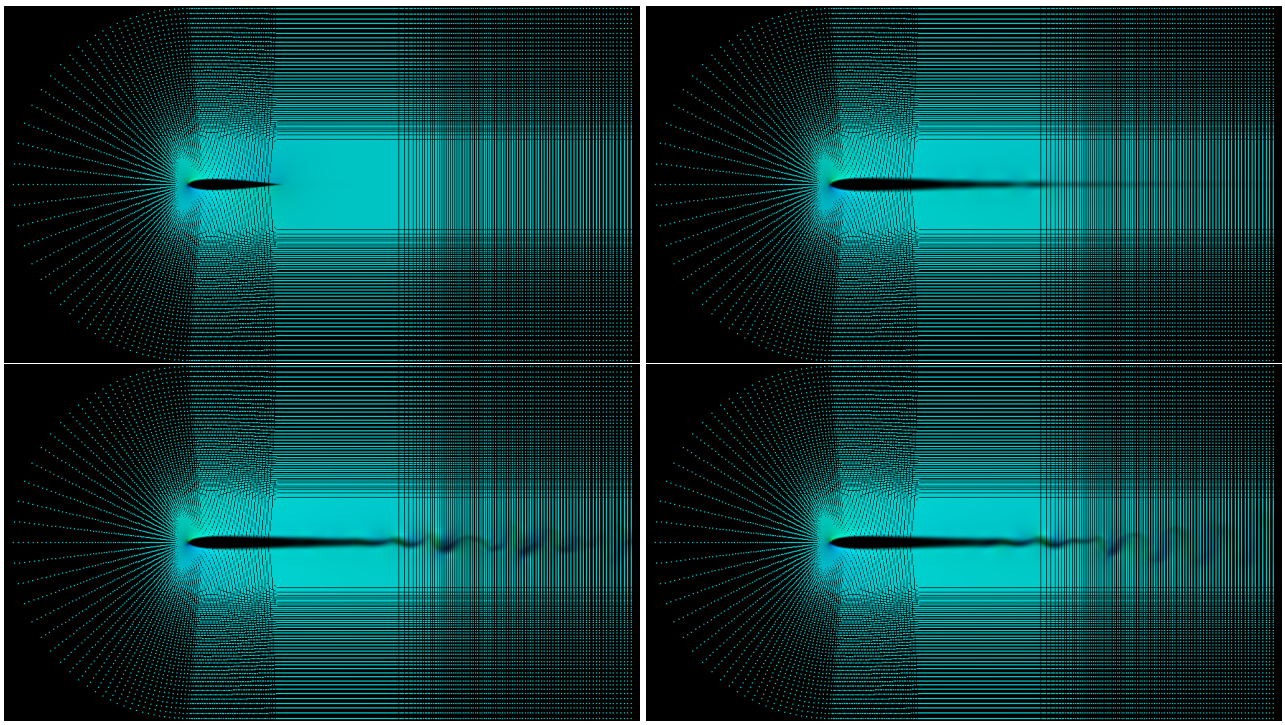

Figure 13. Simulation of a 2D flow around a NACA 0012 airfoil. Top left: frame 1, top right: 70, bottom left: 120, bottom right: 200.

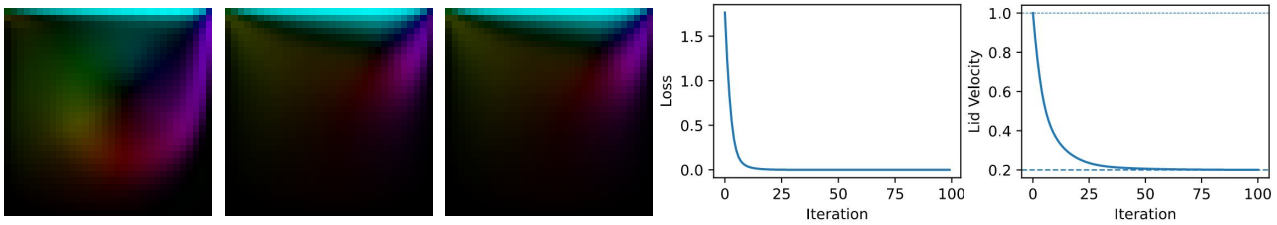

Figure 14. Optimization of the lid velocity with learning rate 4e-2. From left to right: final frame of the simulation with initial parameters, with optimized parameters, with the reference parameters, loss over the optimization iterations, lid u-velocity over the optimization iterations.

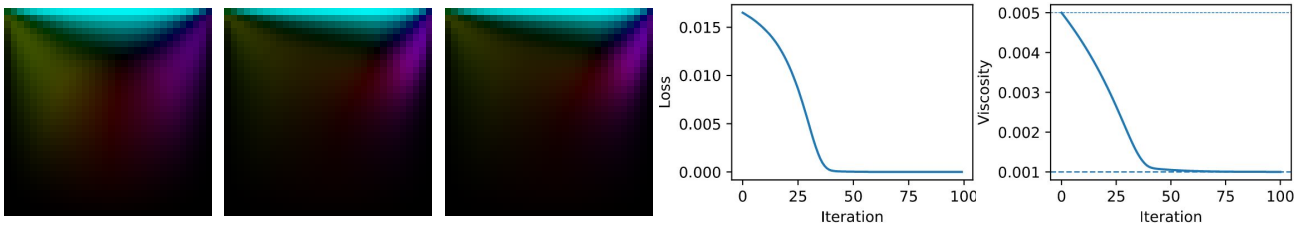

Figure 15. Optimization of the viscosity with learning rate 1e-5. From left to right: final frame of the simulation with initial parameters, with optimized parameters, with the reference parameters, loss over the optimization iterations, viscosity over the optimization iterations.

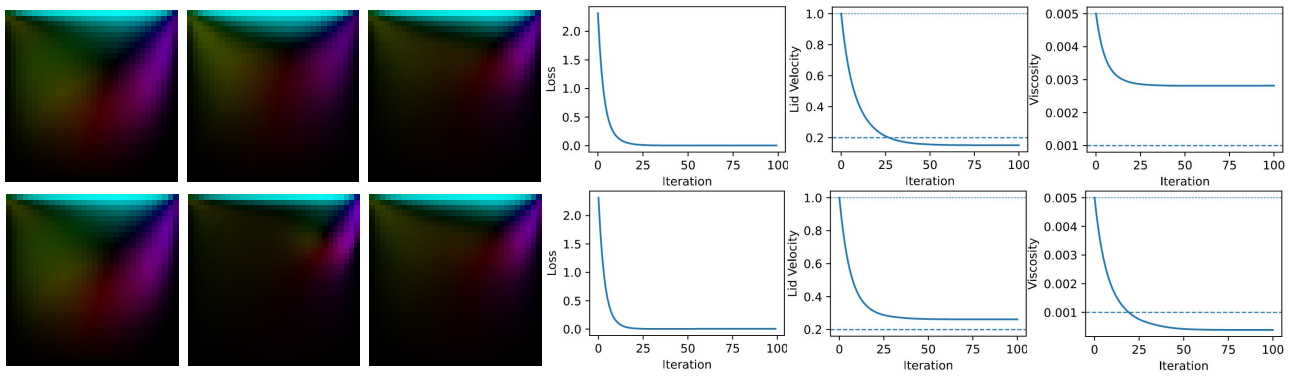

Figure 16. Joint optimization of lid velocity and viscosity. For the velocity we use a learning rate of 2e-2 in both cases, while the viscosity learning rate is 4e-7 in the top row and 6e-7 in the bottom row. From left to right: final frame of the simulation with initial parameters, with optimized parameters, with the reference parameters, loss over the optimization iterations, lid u-velocity over the optimization iterations, viscosity over the optimization iterations. The bold dashed line shows the target value.