Physical System Description

We simulate fluid flow around a circular obstacle, specifically modeling the Karman vortex street phenomenon. The domain is a long rectangular area with a cylinder placed in the center. Fluid enters from the left boundary with a uniform velocity and exits from the right boundary. The top and bottom boundaries are treated as frictionless walls.

We'll assume the fluid is incompressible, inviscid, has a constant density of $\rho_M = 1$, not subject to gravity (external forces), and 2D.

We represent the fluid using the Marker-and-Cell (MAC) method, which discretizes the fluid domain into a grid and tracks the velocity components at the cell faces. The fluid will be advected using the Back-and-Forth Error Correction and Compensation (BFECC) method to improve the accuracy of the advection step and reduce numerical diffusion.

Formulation

To determine the equations of motion for our fluid simulation, we start with the Navier-Stokes equations for incompressible flow:

$$\begin{aligned}\frac{\partial \mathbf{u}}{\partial t} &= -\nabla_{\mathbf{u}}\mathbf{u} + \nu\nabla^2\mathbf{u} - \frac{1}{\rho_M}\nabla p + \mathbf{f}_{\text{ext}} \\ &= -\nabla_{\mathbf{u}}\mathbf{u} + 0 - \nabla p + 0 \\ &= -\nabla_{\mathbf{u}}\mathbf{u} - \nabla p \end{aligned} \tag{1}$$

And since it is incompressible, it is also subject to the constraint:

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

Where $\mathbf{u}$ is the velocity field, p is the pressure field, ν is the kinematic viscosity. We let ν be zero since it is inviscid, and we let $\mathbf{f}_{\text{ext}}$ be zero since there are no external forces (e.g. gravity).

Discretization to MAC

We first discretize the velocity field $\mathbf{u}$ using the MAC grid [1][2]. Let N_x and N_y represent the number of columns and rows of the grid, respectively. Our domain is $\mathcal{M}$, the set of grid cells in $\mathbb{R}^2$, indexed by their integer coordinates. Thus, $\mathcal{M}$ is defined as the set of $\mathcal{C}_{i,j}$ indexed by $1 \le i \le N_x, 1 \le j \le N_y$ where

$$\mathcal{C}_{i,j} := [i\Delta x, (i+1)\Delta x] \times [j\Delta y, (j+1)\Delta y]$$

For simplicity, we let $\Delta x = \Delta y$. Since our system is 2D, we have $\mathbf{u} : \mathcal{M} \to \mathbb{R}^2$, and $\mathbf{u}(x, y) = (u, v)$, where u, v is the velocity in the x and y direction, respectively. So if $\mathbf{u}$ is a vector field, then u, v are scalar fields in the x and y direction, respectively.

We denote pressure p as a matrix of size $N_x \times N_y$, where $p : \mathcal{M} \to \mathbb{R}$. Thus, our system of equations becomes:

$$\begin{cases} \dot{u} = -(\mathbf{u} \cdot \nabla)u - \dfrac{\partial p}{\partial x} & (3) \\ \dot{v} = -(\mathbf{u} \cdot \nabla)v - \dfrac{\partial p}{\partial y} & (4) \\ \nabla \cdot \mathbf{u} = 0 & (5) \end{cases}$$

Velocity u is defined at the vertical faces of the grid cells, and velocity v is defined at the horizontal faces of the grid cells. Pressure p is defined at the center of the grid cells. Indeed, if we are in cell $\mathcal{C}_{i,j}$, then the quantities are given by:

$$\mathcal{C}_{i,j} \sim \begin{cases} u_{i+1/2,j} & \text{velocity in the } x \text{ direction at the vertical face between } \mathcal{C}_{i,j} \text{ and } \mathcal{C}_{i,j+1} \\ v_{i,j+1/2} & \text{velocity in the } y \text{ direction at the horizontal face between } \mathcal{C}_{i,j} \text{ and } \mathcal{C}_{i+1,j} \\ p_{i,j} & \text{pressure at the center of the cell } \mathcal{C}_{i,j} \end{cases}$$

Discretization of Spatial Derivatives

We'll need to define how to compute the spatial derivatives in our equations. To enforce incompressibility, we calculate the discrete divergence at the cell centers:

$$(\nabla \cdot \mathbf{u})_{i,j} \approx \frac{u_{i+1/2,j} - u_{i-1/2,j}}{\Delta x} + \frac{v_{i,j+1/2} - v_{i,j-1/2}}{\Delta y} \tag{6}$$

To compute the pressure gradient, we calculate the discrete gradient at the cell faces:

$$(\nabla p)^x_{i+1/2,j} \approx \frac{p_{i+1,j} - p_{i,j}}{\Delta x} \tag{7}$$

$$(\nabla p)^y_{i,j+1/2} \approx \frac{p_{i,j+1} - p_{i,j}}{\Delta y} \tag{8}$$

The fluid cannot penetrate the top and bottom boundaries, nor does it experience friction along the walls. So, for all $1 \leq i \leq N_x$, we have the following boundary conditions for the normal velocity v:

$$v_{i,0} = 0 \tag{9}$$

$$v_{i,N_y} = 0 \tag{10}$$

To show absence of friction exerted on the fluid (from the top and bottom boundaries), we'll derive the boundary condition for the tangential velocity u via vorticity. Let ω represent vorticity. We have the following relationship between the tangential boundary vorticity and the interface vortex sheet:

$$(1 - \alpha)\gamma = \alpha(\omega_{\parallel} - \omega_r) \tag{11}$$

from [4]. Since our wall is "free-slip", we have $\alpha = 1$ and so our equation reduces to

$$\omega_{\parallel} = \omega_r \tag{12}$$

In particular, ω_r is twice the angular velocity of the unit normal vector to the fluid on the boundary [4]; this quantity is zero since the wall is stationary and thus the normal vector does not rotate. The tangential vorticity at the boundary is represented by $\omega_{\parallel}$. Expanding (12), and the definition of vorticity, we have

$$\begin{aligned} \omega_{\parallel} &= \omega_r \\ \frac{\partial u}{\partial y} - \frac{\partial v}{\partial x} &= 0 \\ \frac{\partial u}{\partial y} &= \frac{\partial v}{\partial x} \end{aligned}$$

Since v is zero at the boundary, we have

$$\frac{\partial u}{\partial y} = 0 \tag{13}$$

Discretizing (13) at the top and bottom boundaries, we get the following discretized final boundary conditions for all $1 \leq i \leq N_x$:

$$u_{i,0} = u_{i,1} \tag{14}$$
$$u_{i,N_y} = u_{i,N_y-1} \tag{15}$$

For inflow, we set the velocity at the left boundary $u_{0,j}$ to a constant u_{inflow} for all $1 \leq j \leq N_y$. For outflow, we want to ensure zero gradient, so we set $u_{N_x,j} = u_{N_x-1,j}$ for all $1 \leq j \leq N_y$.

$$u_{0,j} = u_{\text{inflow}} \tag{16}$$
$$u_{N_x,j} = u_{N_x-1,j} \tag{17}$$

Lastly, to ensure the pressure field does not flow out of the top and bottom boundaries, we give it a zero gradient on the boundary as well. Thus,

$$p_{0,j} = p_{1,j} \tag{18}$$
$$p_{i,N_y} = p_{i,N_y-1} \tag{19}$$
$$p_{i,0} = p_{i,1} \tag{20}$$
$$p_{N_x,j} = 0 \tag{21}$$

This sets the left, top, and bottom boundaries to have zero gradient, and the right boundary to have a fixed pressure of zero to allow fluid to freely exit.

Discretization of Obstacles

Typical demonstrations of the Karman vortex street phenomenon involve a circle placed near the fluid entry to obstruct the fluid flow. We can represent this circular obstacle in our grid by marking the cells that are occupied by the circle as solid cells, and setting the velocity at the faces of those cells to zero.

Let a, b represent the coordinates of the center of the circle, and let r represent the radius of the circle. Then, we mark cell $\mathcal{C}_{i,j}$ as a solid cell if the distance from the center of the cell to the center of the circle is less than or equal to r.

$$(i\Delta x - a)^2 + (j\Delta y - b)^2 \leq r^2 \tag{22}$$

Time Splitting and Simulation Loop

We now need to discretize time into our equations. We advance the simulation from time $t^{(n)}$ to $t^{(n+1)} = t^{(n)} + \Delta t$ using Strang Splitting and use BFECC to perform advection. The simulation loop is as follows:

1. Solve for advection using BFECC to get an intermediate velocity $\mathbf{u}^*$ for half a timestep.
2. Solve for pressure using the intermediate velocity $\mathbf{u}^*$ to get the pressure field. Then project and reflect.
3. Solve for advection using BFECC again to get the final velocity $\mathbf{u}^{(n+1)}$ for half a timestep with the pressure field from step 2.
4. Pressure project the final velocity $\mathbf{u}^{(n+1)}$ to enforce incompressibility.

At each step, we will also enforce boundary conditions and obstacle interactions as described in the previous section.

Semi-Lagrangian Advection

To solve the advection equation $\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = 0$, we'll use the Semi-Lagrangian scheme to ensure unconditional stability by tracing the flow of the fluid backwards in time and interpolating the velocity with the four surrounding grid points.

Let $\mathbf{u}_{i,j}^{(n)}$ be the velocity evaluated at grid location (i, j) at time n. Let $\mathscr{A}^{\text{sL}}_{\mathbf{u},\Delta t}$ denote the Semi-Lagrangian advection operator, which transports an arbitrary field through the velocity field $\mathbf{u}$ over a time step Δt.

To compute $\hat{\mathbf{u}}_{i,j}^{(n+1/2)} = \mathscr{A}^{\text{sL}}_{\mathbf{u},\Delta t}(\mathbf{u}^{(n)})_{i,j}$, we first trace the flow backwards in time along a straight line trajectory to find the departure point $\mathbf{d} = (x_\mathbf{d}, y_\mathbf{d})$. Note that the hat $\hat{}$ here indicates that is only a half step before we solve for pressure.

$$\mathbf{d} = \begin{bmatrix} i\Delta x \\ j\Delta y \end{bmatrix} - \Delta t \mathbf{u}_{i,j}^{(n)} \tag{23}$$

Because it is highly unlikely that $\mathbf{d}$ will land exactly on a discrete grid node, we perform interpolation using the four grid points that form the bounding box immediately surrounding $\mathbf{d}$. Let (i', j') be the integer indices of the bottom left node of this bounding box. It is calculated via

$$i' = \left\lfloor \frac{x_\mathbf{d}}{\Delta x} \right\rfloor \qquad j' = \left\lfloor \frac{y_\mathbf{d}}{\Delta y} \right\rfloor$$

Since $\mathbf{d}$ is also unlikely to land exactly between all four grid points, we must have some weight between all four grid points. Let

$$\alpha_x = \frac{x_\mathbf{d} - i'\Delta x}{\Delta x} \tag{24}$$

$$\alpha_y = \frac{y_\mathbf{d} - j'\Delta y}{\Delta y} \tag{25}$$

The updated velocity is therefore computed as:

$$\begin{aligned} \hat{\mathbf{u}}_{i,j}^{(n+1/2)} &= \mathscr{A}^{\text{sL}}_{\mathbf{u},\Delta t}(\mathbf{u}^{(n)})_{i,j} \\ &= (1-\alpha_x)(1-\alpha_y)\mathbf{u}_{i',j'} + \alpha_x(1-\alpha_y)\mathbf{u}_{i'+1,j'} + (1-\alpha_x)\alpha_y\mathbf{u}_{i',j'+1} + \alpha_x\alpha_y\mathbf{u}_{i'+1,j'+1} \end{aligned} \tag{26}$$

as illustrated in the figure below.

BFECC Advection

We implement BFECC as defined in class:

$$\mathscr{A}^{\text{BFECC}}_{\mathbf{u},\Delta t} := \mathscr{A}^{\text{sL}}_{\mathbf{u},\Delta t} - \left(\mathscr{A}^{\text{sL}}_{\mathbf{u},\Delta t}\right)\left(\frac{1}{2}\left(\mathscr{A}^{\text{sL}}_{-\mathbf{u},\Delta t} \circ \mathscr{A}^{\text{sL}}_{\mathbf{u},\Delta t} - 1\right)\right) \tag{27}$$

We fix the notation to match the original definition of BFECC [3] by adding an extra set of parantheses on the second semi-Lagrangian operator.

One thing to note is that for the forward component $\mathscr{A}^{\text{sL}}_{-\mathbf{u},\Delta t}$, we travel forward in time (by reversing the velocity field), so like (23), we compute the reverse point $\mathbf{d}'$ as

$$\mathbf{d}' = \begin{bmatrix} i\Delta x \\ j\Delta y \end{bmatrix} + \Delta t \mathbf{u}_{i,j}^{(n)} \tag{28}$$

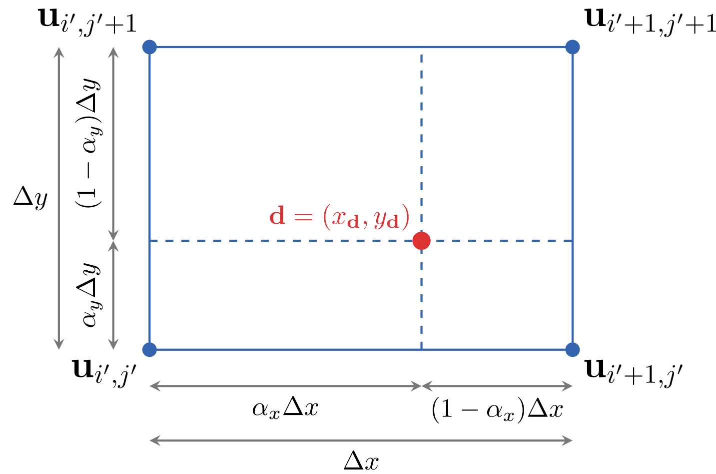

Figure 1: MAC Cell with Semi-Lagrangian Interpolation. The bounding box represents the four grid points used for interpolation, and the dashed lines indicate the relative distances $\alpha_x \Delta x$ and $\alpha_y \Delta y$ from the departure point $\mathbf{d}$ to the grid nodes.

and then perform interpolation using the four grid points that form the bounding box immediately surrounding $\mathbf{d}'$.

To prevent spatial oscillations (numerical dispersion), we will apply a limiter to clamp the BFECC output. During $\mathscr{A}^{\text{sL}}_{\mathbf{u},\Delta t}$, we'll identify the convex hull of the neighborhood values that are used for interpolation. In particular, we compute

$$\mathbf{u}_{\min} = \min\{\mathbf{u}_{i',j'}, \mathbf{u}_{i'+1,j'}, \mathbf{u}_{i',j'+1}, \mathbf{u}_{i'+1,j'+1}\} \tag{29}$$

$$\mathbf{u}_{\max} = \max\{\mathbf{u}_{i',j'}, \mathbf{u}_{i'+1,j'}, \mathbf{u}_{i',j'+1}, \mathbf{u}_{i'+1,j'+1}\} \tag{30}$$

If the BFECC value lies outside the convex hull, then it is an overshoot (or undershoot) and will be clamped. Let $\mathbf{u}^{\text{BFECC}}_{i,j} = \mathscr{A}^{\text{BFECC}}_{\mathbf{v},\Delta t}(\mathbf{u}^{(n)})_{i,j}$ be the BFECC output. We compute the final velocity update as follows:

$$\hat{\mathbf{u}}^{(n+1)}_{i,j} = \begin{cases} \mathbf{u}^*_{i,j} & \left(\mathbf{u}^{\text{BFECC}}_{i,j} < \mathbf{u}_{\min}\right) \vee \left(\mathbf{u}^{\text{BFECC}}_{i,j} > \mathbf{u}_{\max}\right) \\ \mathbf{u}^{\text{BFECC}}_{i,j} & \text{otherwise} \end{cases} \tag{31}$$

where $\mathbf{u}^*_{i,j}$ is the clamped minimum or maximum value, depending on whether it is an undershoot or overshoot respectively.

Pressure Projection and Reflection

From class, we learned that we can solve for pressure by projection (32) and then reflection (33).

$$\mathbf{u}^{(n+1/2)*}_{i,j} = \hat{\mathbf{u}}^{(n+1/2)}_{i,j} - \nabla \Delta^{-1} \nabla \cdot \hat{\mathbf{u}}^{(n+1/2)}_{i,j} \tag{32}$$

$$\mathbf{u}^{(n+1/2)}_{i,j} \leftarrow 2\mathbf{u}^{(n+1/2)*}_{i,j} - \mathbf{u}^{(n)}_{i,j} \tag{33}$$

where Δ^{-1} denotes the inverse discrete Laplacian operator. The divergence of the intermediate velocity field $\hat{\mathbf{u}}^{(n+1/2)}_{i,j}$ is computed via the discrete operator defined in (6). Evaluating (32) requires an iterative solver since we need to solve for the pressure field $\Delta p = \left(\nabla \cdot \hat{\mathbf{u}}^{(n+1/2)}_{i,j}\right)$ in order to

compute the gradient ∇p. Because the discrete Laplacian Δ is a computationally intractable, sparse matrix, explicitly computing the inverse would be infeasible. Instead, we can reformulate this operation as a linear system $A\mathbf{x} = \mathbf{b}$ and efficiently approximate the pressure field p using an iterative solver, such as Conjugate Gradient or Jacobi iteration.

Once the pressure field is iteratively resolved, the intermediate velocity is subtracted by the discrete gradient of the pressure field to enforce incompressibility. This is done by evaluating the discrete gradient at the cell faces using (7) and (8). Finally, the reflection step (33) is computed explicitly via simple, elementwise operations.

Boundary Conditions and Obstacles

We must handle boundary interactions during advection and pressure projection.

During the semi-Lagrangian advection step, the backward trace may yield a departure point $\mathbf{d}$ (or reverse point $\mathbf{d}'$) that leaves the fluid region. We handle these boundary intersections depending on the type of boundary:

- If $\mathbf{d}$ falls within the solid circular region defined in (22), the interpolated velocity at that point is set to zero.
- If $\mathbf{d}$ falls outside the outer rectangular simulation boundaries, the coordinates are clamped to the domain edges.

When assembling the linear system to solve $\Delta p = \nabla \cdot \hat{\mathbf{u}}^{(n+1/2)}$, the boundary changes how we calculate the interpolation step. We need to ensure that the fluid does not flow out of the domain or into the solid obstacle(s). This amounts to checking for (18), (19), (20), (21) for the pressure field.

We also override the velocity at the faces of the solid cells from (14), (15), (16), (17) to ensure that the fluid cannot penetrate the solid obstacle and that the inflow and outflow boundary conditions are satisfied.

Implementation Details

We implemented the simulation in Python 3.12 using Taichi for GPU acceleration and visualization. The first steps were to implement a Python class `FluidSim` using Taichi's `@ti.data_oriented` decorator to define GPU kernels for each of the operations described in the previous sections. It contains discretization parameters, physical parameters, and the state of the simulation (velocity and pressure fields). For visualization, we used Taichi's built-in GUI to render the vorticity field of the fluid and an ink map to visualize the flow patterns.

For testing, the hardware we used was an Intel i7-12700H CPU, 40 GB of RAM, and an Intel Iris Xe GPU. We used Taichi's Vulkan backend for GPU acceleration for testing and development. For the final results, we used an Intel i7-14700 CPU, 32 GB of RAM, and an NVIDIA RTX A2000 GPU. This allowed us to use the CUDA backend for GPU acceleration, which provided a large performance boost compared to the Vulkan backend.

Development was a top-down approach, implementing the high level simulation loop for Strang splitting, then moving down to implement the individual components of the simulation loop (advection, pressure projection, and reflection), and then finally implementing the boundary conditions and obstacle interactions.

Implementing advection involved reading the original BFECC paper [3] and to implement the semi-Lagrangian advection and BFECC steps. We set $\Delta x = \Delta y$ (equal to 1) for simplicity and implemented `compute_divergence`.

For the pressure projection step, we implemented the Jacobi iterative solver to solve for the pressure field. It was recommended to use the Conjugate Gradient method, but Taichi's built-in sparse linear solvers are currently only available on CPU and NVIDIA GPUs. For local testing, we used Jacobi iteration.

For reference, the functions in `fluid_sim.py` are 1:1 with the following equations:

Function(s)	Equation
`advect_sL` `advect_u_sL`	(26)
`advect_bfecc`	(27)
`pressure_projection`	(32)
`pressure_reflection`	(33)
`apply_bcs` `jacobi_step`	(9) - (21)
`compute_vorticity`	(35), (36)

Table 1: Python functions and their corresponding equations

Passive Scalar (Ink) Advection

We also added ink to the simulation for visualization of the flow patterns. The ink is treated as a passive scalar field that is advected by the velocity field, but does not affect the velocity field itself (i.e., it has no mass or momentum).

Let $C_{i,j}^{(n)}$ be the concentration of ink at cell (i, j) at time n. The evolution of the ink concentration is governed by the standard advection equation:

$$\frac{\partial C}{\partial t} + (\mathbf{u} \cdot \nabla)C = 0 \tag{34}$$

Since the ink field is defined at the cell centers (similar to pressure), we can advect it using the exact same Semi-Lagrangian and BFECC schemes we use for velocity. To make the vortex shedding visually apparent, we continuously inject ink at the inflow boundary in a vertically banded pattern, which then gets carried downstream by the fluid.

Vorticity Calculation

For the visualization of the fluid, we also compute the vorticity field $\omega = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$. Because our velocity components u and v are defined on the faces of the MAC grid, we calculate the discrete vorticity at the cell centers (i, j) using central differences.

First, we interpolate the velocities from the surrounding faces to align them with the adjacent cell

centers, and then we compute the gradients over a distance of $2\Delta x$ and $2\Delta y$:

$$\left(\frac{\partial v}{\partial x}\right)_{i,j} \approx \frac{\frac{1}{2}(v_{i+1,j-1/2} + v_{i+1,j+1/2}) - \frac{1}{2}(v_{i-1,j-1/2} + v_{i-1,j+1/2})}{2\Delta x} \tag{35}$$

$$\left(\frac{\partial u}{\partial y}\right)_{i,j} \approx \frac{\frac{1}{2}(u_{i-1/2,j+1} + u_{i+1/2,j+1}) - \frac{1}{2}(u_{i-1/2,j-1} + u_{i+1/2,j-1})}{2\Delta y} \tag{36}$$

Taking the difference between these two quantities gives us the vorticity $\omega_{i,j}$ at the center of each cell, which we then pass to our colormaps for rendering. We use a red/blue colormap to visualize the vorticity, where the color intensity corresponds to the magnitude of the vorticity, and the color hue (red vs blue) corresponds to the sign of the vorticity (clockwise vs counterclockwise). To improve visual contrast, the calculated vorticity is artificially scaled by a factor of 5 before being clamped and mapped to the colormaps.

Results

Using a CUDA backend on the NVIDIA RTX A2000, we were able to achieve some decent performance. The following table are the parameters we used for the simulation.

Parameter	**Variable**	**Value**
Grid Resolution (Columns)	`nx`	1024
Grid Resolution (Rows)	`ny`	256
Time Step	`dt`	0.05
Fluid Density	`rho`	1.0
Inflow Velocity	`u_inflow`	1.0
Pressure Solver Iterations	`pressure_iters`	30
Center X	`cx`	$0.1 \times n_x$ (102.4)
Center Y	`cy`	$0.5 \times n_y$ (128.0)
Radius	`r`	$0.1 \times n_y$ (25.6)

Table 2: Simulation and obstacle parameters

During testing, we found that the performance using Conjugate Gradient was much slower on the A2000 GPU. We also saw the same blurring effect on the vorticity field when using the CUDA backend as we did with the Vulkan backend, so we figured that our accuracy was not bottlenecked by the linear solver, but likely by the advection step. With that in mind, the images in the Visualization Gallery only use the Jacobi iterative solver for the pressure projection step.

Discussion

Overall, the fluid visualizations look very similar to the expected Karman vortex street patterns, and we can clearly see the alternating vortices being shed from the circular obstacle. The flow patterns are visually apparent in both the vorticity field and the ink field.

One thing to note is that the the vorticity “orbs” that are shed from the obstacle do not lose intensity as they are advected downstream, which matches our lack of viscosity in the simulation. We do see some blurring in the vorticity field, which is likely due to numerical diffusion from the Semi-Lagrangian advection scheme.

Addendum

We have also included an extra video of the simulation with the same parameters as above, but with over 97 minutes of runtime sped up 48x using `ffmpeg`. The video is available at https://youtu.be/1mdHIOyNERc.

Visualization Gallery

Below is a gallery showcasing the fluid simulation rendered under the different visualization filters implemented. All screenshots were captured at different time steps using the screenshot tool in Taichi and were on the same simulation with the same parameters.

Ink Advection

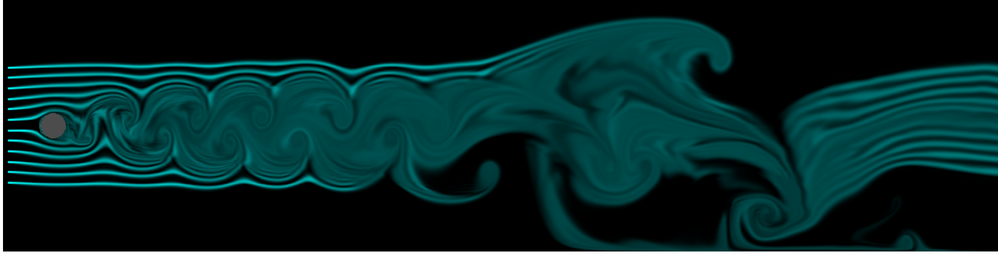

Figure 2: Ink field (Capture 1).

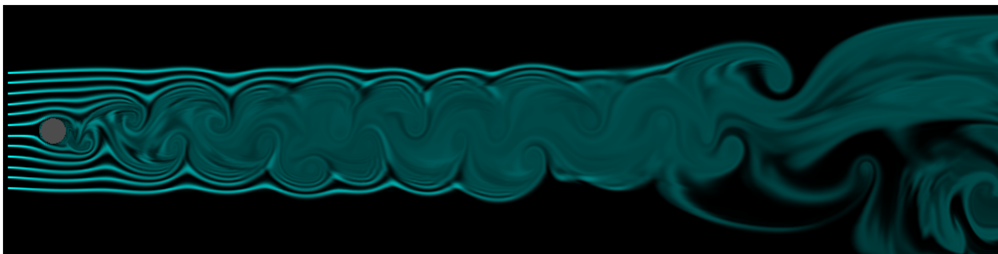

Figure 3: Ink field (Capture 2).

Figure 4: Ink field (Capture 3).

Red/Black (Thermal) Colormap

Figure 5: Thermal vorticity (Capture 1).

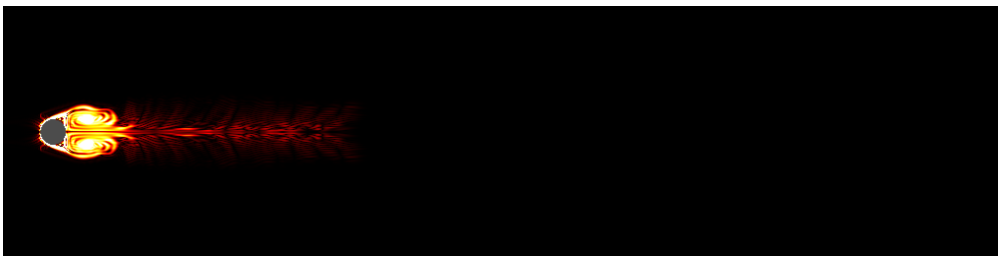

Figure 6: Thermal vorticity (Capture 2).

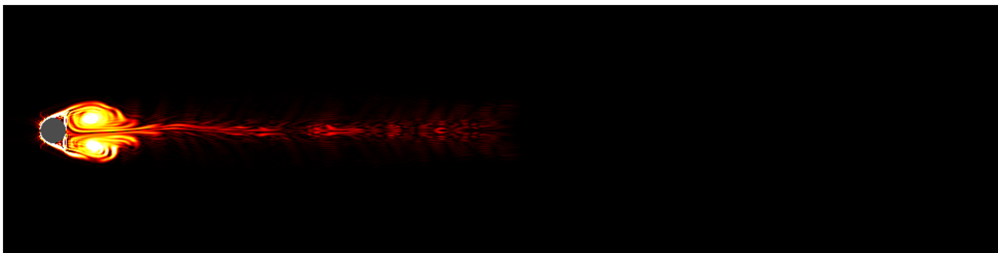

Figure 7: Thermal vorticity (Capture 3).

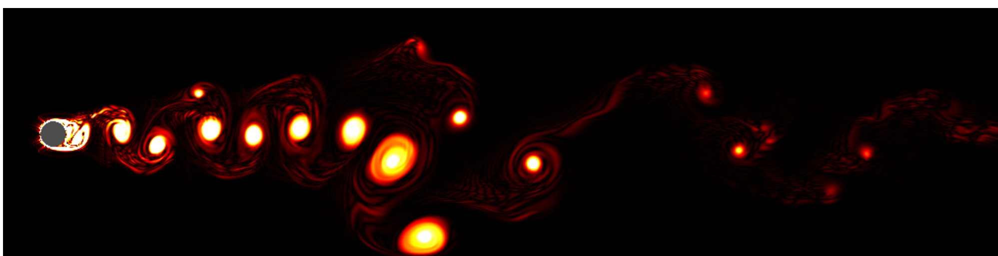

Figure 8: Thermal vorticity (Capture 4).

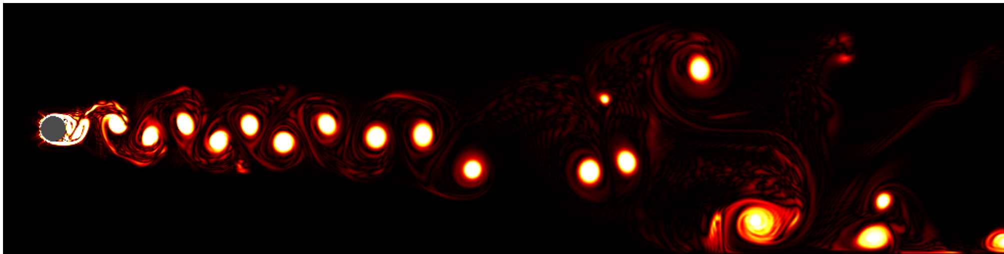

Figure 9: Thermal vorticity (Capture 5).

Ocean-to-Fire Colormap

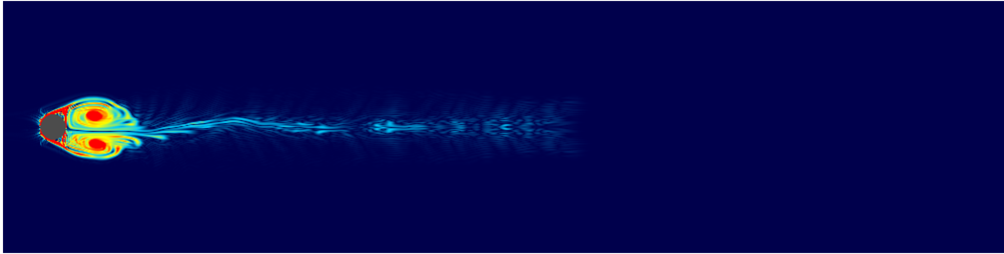

Figure 10: Ocean-to-Fire vorticity (Capture 1).

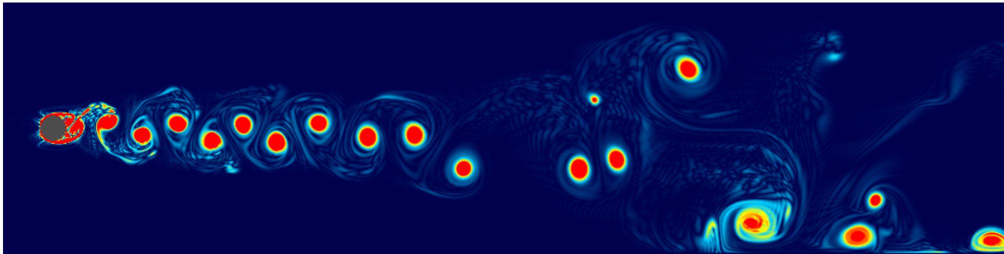

Figure 11: Ocean-to-Fire vorticity (Capture 2).

Diverging Colormap

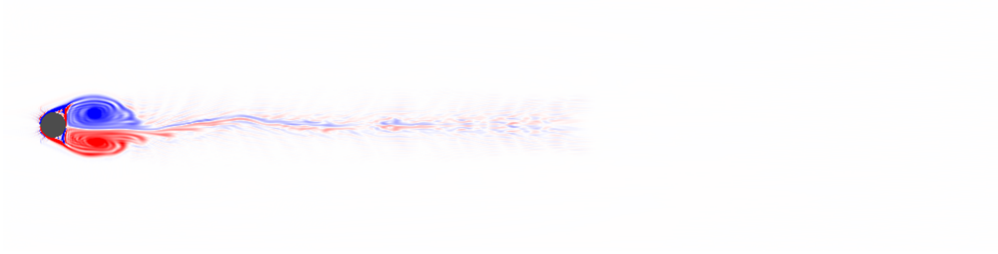

Figure 12: Diverging vorticity (Capture 1).

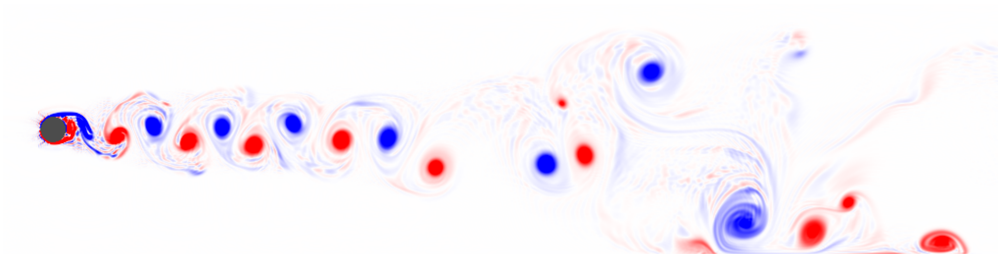

Figure 13: Diverging vorticity (Capture 2).

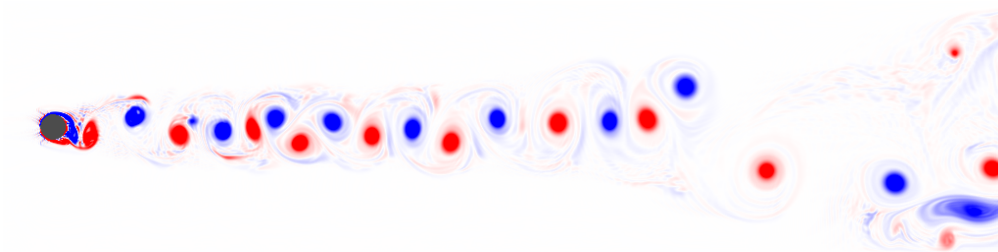

Figure 14: Diverging vorticity (Capture 3).

References

[1] Long Chen. Programming of mac scheme for stokes equations. 2022. https://www.math.uci.edu/~chenlong/226/MACcode.pdf.

[2] Long Chen. Programming of finite difference methods in matlab. 2025. https://www.math.uci.edu/~chenlong/226/FDMcode.pdf.

[3] Todd F. Dupont and Yingjie Liu. Back and forth error compensation and correction methods for removing errors induced by uneven gradients of the level set function. *Journal of Computational Physics*, 190(1):311–324, 2003. https://www.sciencedirect.com/science/article/pii/S0021999103002766.

[4] S.J. Terrington, M.C. Thompson, and K. Hourigan. Vorticity dynamics at partial-slip boundaries. *Journal of Fluid Mechanics*, 980:A58, 2024. https://doi.org/10.1017/jfm.2024.68.