Simulation of 2D Point Vortex Dynamics with Passive Tracers in a Circular Domain

Param Somane
CSE 291 (Spring 2025) Final Project

Abstract

This report details the design and implementation of a numerical simulation for two-dimensional point vortex dynamics within a bounded circular domain. The system models the motion of N_v point vortices, regularized using the Lamb-Oseen model to avoid singularities and provide a finite core structure. Interactions between vortices and their influence on N_t passive tracer particles are simulated. The no-penetration boundary condition at the circular domain wall is enforced using the classical method of images. The equations of motion, comprising a system of ordinary differential equations for vortex and tracer positions, are integrated numerically using an explicit fourth-order Runge-Kutta (RK4) scheme. The implementation supports both CPU-based computation (leveraging NumPy and Numba for JIT compilation) and GPU acceleration (using CuPy), offering flexibility and performance. We describe the governing equations, numerical algorithms, initialization procedures, and visualization techniques. The simulation's fidelity is assessed by monitoring conserved quantities like linear and angular impulse. The project culminates in an animation visualizing the complex, often chaotic, advection of tracers by the vortex flow.

1 Introduction and Motivation

The study of two-dimensional ideal fluid flow provides fundamental insights into a wide range of fluid phenomena. Point vortices, which are idealized singularities of vorticity, serve as elementary building blocks for constructing and understanding complex 2D flows [1]. In this model, the vorticity is concentrated at discrete points, and these points move under the influence of the velocity field they collectively generate. While classical point vortices are singular, leading to infinite self-induced velocities, regularized models such as the Lamb-Oseen vortex [3] introduces a finite core size, smearing out the vorticity over a small region and yielding more physically realistic interactions, especially at close range.

Confining vortices within a bounded domain introduces interactions with boundaries. For a circular domain, the method of images provides an elegant way to satisfy the no-penetration boundary condition by introducing a system of image vortices outside the domain [5]. The resulting dynamics can be rich and complex, exhibiting phenomena like vortex pairing, clustering, and chaotic advection.

Passive tracer particles, which are advected by the fluid flow without influencing it, are invaluable tools for visualizing flow patterns. By tracking the trajectories of a large number of tracers, one can gain qualitative and quantitative understanding of mixing, transport, and the overall structure of the velocity field generated by the vortices.

This project implements a simulation of N_v Lamb-Oseen vortices and N_t passive tracers within a circular domain of radius R_D. The primary goals are:

- To accurately model the interactions between vortices, including their images.
- To simulate the advection of passive tracers by the vortex-induced flow.
- To implement an efficient numerical solver using the RK4 method, capable of running on both CPU (with Numba acceleration) and GPU (with CuPy).
- To visualize the resulting dynamics and monitor conserved quantities to assess simulation accuracy.

The simulation provides a flexible platform for exploring 2D vortex dynamics and serves as an example of a continuum mechanical system simulation as per the project guidelines.

2 System Model and Governing Equations

2.1 Domain and Particles

The system is defined in a two-dimensional circular domain of radius R_D. It contains two types of particles:

- **Vortices:** N_v point vortices, indexed $k = 1, \ldots, N_v$. Each vortex k is characterized by its time-dependent position $\boldsymbol{x}_k(t) = (x_k(t), y_k(t))$ and its constant circulation strength Γ_k.
- **Tracers:** N_t passive tracer particles, indexed $j = 1, \ldots, N_t$. Each tracer j has a time-dependent position $\boldsymbol{p}_j(t) = (p_{jx}(t), p_{jy}(t))$. Tracers are advected by the flow but do not affect it.

The state of the system at any time t is described by the collection of all vortex and tracer positions.

2.2 Velocity Field of a Lamb-Oseen Vortex

A single Lamb-Oseen vortex of strength Γ with a squared core radius a^2, located at the origin, induces a velocity field $\boldsymbol{u}(\boldsymbol{r}; \Gamma, a^2)$ at a point $\boldsymbol{r} = (x, y)$ given by:

$$\boldsymbol{u}(x, y; \Gamma, a^2) = \frac{\Gamma}{2\pi} \frac{1 - e^{-(x^2+y^2)/a^2}}{x^2 + y^2} \begin{pmatrix} -y \\ x \end{pmatrix}. \tag{1}$$

This model regularizes the singularity of a classical point vortex.

- As $r = \|\boldsymbol{r}\| \to 0$, the velocity approaches that of a solid body rotation: $\boldsymbol{u} \to \frac{\Gamma}{2\pi a^2} \begin{pmatrix} -y \\ x \end{pmatrix}$.
- As $r \to \infty$, the velocity approaches that of a potential vortex: $\boldsymbol{u} \to \frac{\Gamma}{2\pi r^2} \begin{pmatrix} -y \\ x \end{pmatrix}$.

The implementation uses a factor $f(r^2, a^2) = \frac{1-e^{-r^2/a^2}}{r^2}$ (with a limit value $1/a^2$ as $r^2 \to 0$ for numerical stability), such that the velocity magnitude is $\frac{|\Gamma|}{2\pi r}(1 - e^{-r^2/a^2})$. The Python code defines two distinct squared core radii:

- a_v^2 = `VORTEX_CORE_A_SQ`: Used when calculating the velocity induced by a vortex on another vortex.
- a_t^2 = `TRACER_CORE_A_SQ`: Used when calculating the velocity induced by a vortex on a tracer particle.

This means the regularization parameter a^2 in Eq. (1) effectively depends on the type of particle whose velocity is being computed.

2.3 Method of Images for Circular Boundary

To satisfy the no-penetration boundary condition ($\boldsymbol{u} \cdot \boldsymbol{n} = 0$) on the circular domain wall $r = R_D$, the method of images is employed. For each real vortex k at position $\boldsymbol{x}_k$ with strength Γ_k, an image vortex is introduced at position $\boldsymbol{x}'_k$ with strength Γ'_k:

$$\boldsymbol{x}'_k = \frac{R_D^2}{\|\boldsymbol{x}_k\|^2}\boldsymbol{x}_k \tag{2}$$

$$\Gamma'_k = -\Gamma_k \tag{3}$$

A schematic is shown in Figure 1.

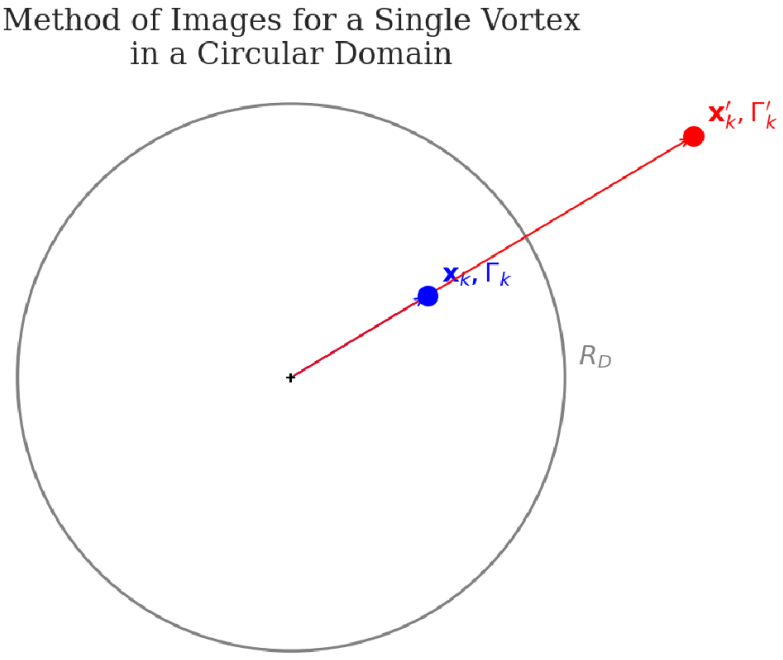

Figure 1: Schematic of a vortex Γ_k at position $\boldsymbol{x}_k$ within the circular domain of radius R_D, and its corresponding image vortex Γ'_k at $\boldsymbol{x}'_k$ outside the domain. The illustrative real vortex is placed at $(0.5R_D, 0.3R_D)$.

Additionally, if the total circulation of real vortices $\Gamma_{\text{total}} = \sum_{k=1}^{N_v} \Gamma_k$ is non-zero, a background rotational flow $\boldsymbol{u}_{\text{bg}}$ must be added to satisfy the boundary condition:

$$\boldsymbol{u}_{\text{bg}}(\boldsymbol{r}; \Gamma_{\text{total}}, R_D) = \frac{\Gamma_{\text{total}}}{2\pi R_D^2} \begin{pmatrix} -y \\ x \end{pmatrix}, \tag{4}$$

where $\boldsymbol{r} = (x, y)$ is the position where the velocity is evaluated.

2.4 Equations of Motion

The velocity of each particle is the sum of velocities induced by all other real vortices, all image vortices, and the background flow.

For vortex i at position $\boldsymbol{x}_i$:

$$\frac{d\boldsymbol{x}_i}{dt} = \sum_{\substack{j=1 \\ j\neq i}}^{N_v} \boldsymbol{u}(\boldsymbol{x}_i - \boldsymbol{x}_j; \Gamma_j, a_v^2) + \sum_{j=1}^{N_v} \boldsymbol{u}(\boldsymbol{x}_i - \boldsymbol{x}'_j; \Gamma'_j, a_v^2) + \boldsymbol{u}_{\text{bg}}(\boldsymbol{x}_i; \Gamma_{\text{total}}, R_D). \tag{5}$$

The term $j \neq i$ in the first sum indicates that a vortex does not induce a velocity on itself via its regularized real component (the Lamb-Oseen form is regular at $r = 0$, but the standard formulation sums influences from $j \neq i$). The self-image interaction is included in the second sum.

For tracer l at position $\boldsymbol{p}_l$:

$$\frac{d\boldsymbol{p}_l}{dt} = \sum_{j=1}^{N_v} \boldsymbol{u}(\boldsymbol{p}_l - \boldsymbol{x}_j; \Gamma_j, a_t^2) + \sum_{j=1}^{N_v} \boldsymbol{u}(\boldsymbol{p}_l - \boldsymbol{x}'_j; \Gamma'_j, a_t^2) + \boldsymbol{u}_{\text{bg}}(\boldsymbol{p}_l; \Gamma_{\text{total}}, R_D). \tag{6}$$

Note the use of a_v^2 in Eq. (5) and a_t^2 in Eq. (6) for the squared core radius parameter in the Lamb-Oseen velocity function $\boldsymbol{u}(\cdot)$.

2.5 Conserved Quantities

For an unbounded, inviscid 2D flow of point vortices, certain quantities are conserved. In our bounded domain, these may not be strictly conserved due to interactions with the image system and the boundary enforcement. However, monitoring them provides a measure of simulation quality. The primary quantities are:

- **Linear Impulse**: $P_x = \sum_{k=1}^{N_v} \Gamma_k y_k$ and $P_y = -\sum_{k=1}^{N_v} \Gamma_k x_k$.
- **Angular Impulse**: $L_z = \sum_{k=1}^{N_v} \Gamma_k \|\boldsymbol{x}_k\|^2 = \sum_{k=1}^{N_v} \Gamma_k (x_k^2 + y_k^2)$.

The total circulation $\Gamma_{\text{total}} = \sum \Gamma_k$ is inherently conserved as individual Γ_k are constant. L_z is expected to change if $\Gamma_{\text{total}} \neq 0$ due to the background flow term.

3 Numerical Implementation

The system of first-order ordinary differential equations (ODEs) described by Eqs. (5) and (6) is solved numerically. Let $\boldsymbol{Y}(t)$ be the state vector containing all $2(N_v + N_t)$ coordinates of vortices and tracers. The system is $\dot{\boldsymbol{Y}} = \boldsymbol{F}(\boldsymbol{Y}, t)$.

3.1 Time Integration

We employ an explicit fourth-order Runge-Kutta (RK4) method to integrate the ODEs. Given the state $\boldsymbol{Y}^{(n)}$ at time t_n, the state $\boldsymbol{Y}^{(n+1)}$ at $t_{n+1} = t_n + \Delta t$ is computed as:

$$
\begin{aligned}
\boldsymbol{k}_1 &= \boldsymbol{F}\big(\boldsymbol{Y}^{(n)}, t_n\big), \\
\boldsymbol{k}_2 &= \boldsymbol{F}\Big(\boldsymbol{Y}^{(n)} + \tfrac{1}{2}\,\Delta t\,\boldsymbol{k}_1, t_n + \tfrac{1}{2}\Delta t\Big), \\
\boldsymbol{k}_3 &= \boldsymbol{F}\Big(\boldsymbol{Y}^{(n)} + \tfrac{1}{2}\,\Delta t\,\boldsymbol{k}_2, t_n + \tfrac{1}{2}\Delta t\Big), \\
\boldsymbol{k}_4 &= \boldsymbol{F}\Big(\boldsymbol{Y}^{(n)} + \Delta t\,\boldsymbol{k}_3, t_n + \Delta t\Big), \\
\boldsymbol{Y}^{(n+1)} &= \boldsymbol{Y}^{(n)} + \tfrac{\Delta t}{6}\Big(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4\Big).
\end{aligned}
$$

The time step Δt is specified by the `DT` parameter in the `SimConfig`.

3.2 Boundary Enforcement

After each RK4 step, particle positions are checked against the domain boundary R_D. If a particle at position $\boldsymbol{r}_{\text{old}}$ is found outside the domain (i.e., $\|\boldsymbol{r}_{\text{old}}\| > R_D$), its position is projected back just inside the boundary:

$$
\boldsymbol{r}_{\text{new}} = \boldsymbol{r}_{\text{old}} \frac{R_D(1 - \epsilon_{\text{margin}})}{\|\boldsymbol{r}_{\text{old}}\|}, \tag{7}
$$

where ϵ_{margin} is a small factor (e.g., 0.001 or 0.0001 in the code) to prevent particles from sitting exactly on the boundary. This is a simple but effective way to keep particles within the domain.

3.3 Computational Backend (CPU/GPU)

The implementation is designed to run on different computational backends for flexibility and performance, managed by the `SimConfig` object.

- **Array Module (`xp`):** A common interface `xp` is used, which points to either `numpy` (for CPU) or `cupy` (for GPU).
- **Floating Point Precision:** Calculations primarily use 32-bit floating-point numbers (`float32`) by default, configurable via `USE_FLOAT32_CPU` for the CPU path.

3.3.1 CPU Path

When `GPU_ENABLED` is false:

- **NumPy:** Standard NumPy array operations are used for vectorization where possible.
- **Numba:** For performance-critical loops, such as the summation of velocities and the Lamb-Oseen factor calculation, Numba's `@njit` (Just-In-Time compilation) decorator is used. Functions like `_get_velocities_induced_by_vortices_cpu_numba_impl`, `get_vortex_velocities_cpu_numba_impl`, and `_lamb_oseen_factor_cpu_numba_impl` utilize explicit loops, often with `prange` for potential parallel execution on multi-core CPUs.

3.3.2 GPU Path

When `GPU_ENABLED` is true (and CuPy is available and functional):

- **CuPy:** CuPy provides a NumPy-compatible API for GPU-accelerated computing on NVIDIA GPUs.
- **Custom Kernels:** The Lamb-Oseen factor calculation benefits from a custom CuPy `ElementwiseKernel` (`_lamb_oseen_kernel`), which applies the regularized formula efficiently element-wise on the GPU.

```
_lamb_oseen_kernel = ElementwiseKernel(
    'float32 r_sq, float32 core_a_sq', 'float32 out',
    r'''
    const float eps = 1e-7f; // Epsilon for numerical stability
    float r_safe = (r_sq < eps) ? eps : r_sq;
    float val = (1.0f - expf(-r_safe / core_a_sq)) / r_safe;
    // Handle limit r_sq -> 0 where factor -> 1/core_a_sq
    out = (r_sq < eps * 10.0f) ? (1.0f / core_a_sq) : val;
    ''',
    'lamb_oseen'
)
```

Listing 1: CuPy ElementwiseKernel for Lamb-Oseen factor (simplified).

- **Vectorized Operations:** Velocity summations (e.g., in `_get_velocities_induced_by_vortices_xp` and `get_vortex_velocities_xp`) are expressed using CuPy's vectorized array operations, which are executed in parallel on the GPU. This involves broadcasting and sum reductions across appropriate axes.

3.4 Initialization Details

The initial state of vortices and tracers is set up by dedicated functions:

- **Vortices (`initialize_vortices`):** If $N_v \geq 4$, the first four vortices are placed deterministically (e.g., corners of a square) with alternating strengths. Additional vortices are placed randomly within a certain radius, with strengths chosen randomly from a predefined set. If $N_v < 4$, all vortices are placed randomly.
- **Tracers (`initialize_tracers`):** The placement and initial scalar values of tracers depend on `TRACER_COLORING_MODE`:
  - `group`: Tracers are arranged in `NUM_TRACER_GROUPS` distinct patches, typically circular, distributed around the domain center. Tracers in each group i are assigned a scalar value (e.g., $(i + 0.5)/$`NUM_TRACER_GROUPS`) for coloring. An example initial state for this mode is shown in Figure 2.
  - `scalar`: Tracers are distributed randomly (e.g., within a radius $0.7R_D$), and their scalar value is set based on their initial radial position (e.g., $1 - r/(0.7R_D)$), creating a radial gradient.
  - `speed`: Tracers are placed similarly to 'scalar' mode, but their scalar value (used for coloring) is determined by their instantaneous speed during the simulation.

  A global random seed (`RANDOM_SEED`) can be set for reproducible initial conditions.

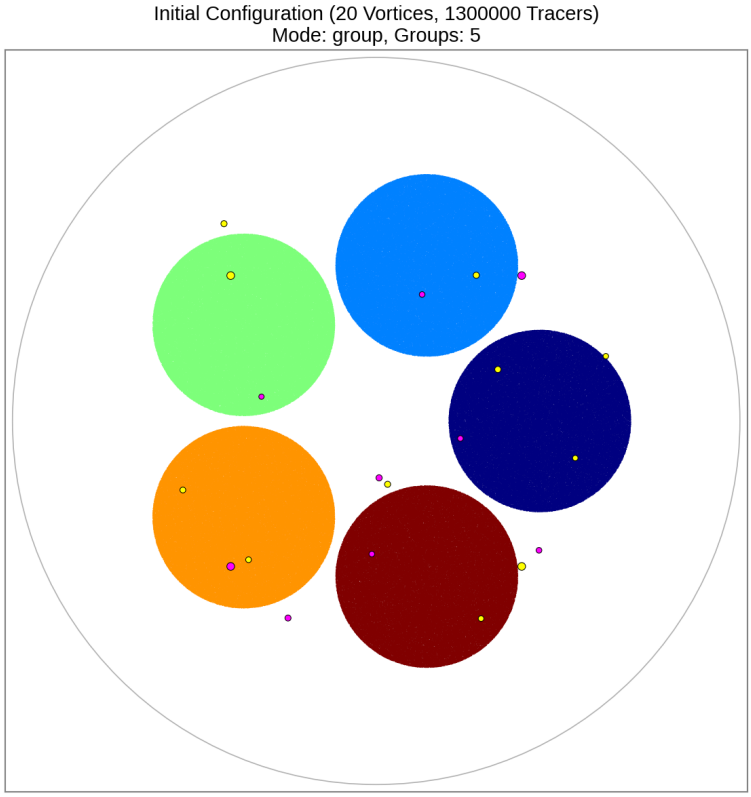

Figure 2: Example initial configuration of vortices and tracers, generated with $N_v = 20$ vortices, $N_t = 1,300,000$ tracers, domain radius $R_D = 1.0$, random seed 42, and `TRACER_COLORING_MODE='group'` with 5 tracer groups. Each group is initialized in a distinct patch and assigned a different scalar value for colormap visualization.

4 Simulation and Visualization

The diffusion, drift, and eventual decay of a viscous counter-rotating vortex pair have been analysed in detail by van Dommelen and Shankar [4], providing asymptotic limits that complement the present numerical results. The simulation evolves the system over a specified `SIMULATION_TIME` with time step `DT`. Data is typically saved at intervals defined by `PLOT_INTERVAL` for subsequent animation.

4.1 Simulation Parameters

The behavior of the simulation is controlled by parameters in the `SimConfig` dataclass. Key physical and numerical parameters include:

- `N_VORTICES`, `N_TRACERS`: Number of vortices and tracers.
- `DOMAIN_RADIUS` (R_D): Radius of the circular computational domain.
- `SIMULATION_TIME`, `DT`: Total simulation duration and integration time step.
- `VORTEX_CORE_A_SQ` (a_v^2), `TRACER_CORE_A_SQ` (a_t^2): Squared Lamb-Oseen core radii for vortex-vortex and vortex-tracer interactions, respectively.

Performance and visualization parameters like `ANIM_TRACERS_MAX` (limits number of tracers in animation for performance) and `TRACER_GLOW_LAYERS` also affect the final output.

4.2 Visualization

The simulation results are visualized as an MP4 video generated using Matplotlib's animation capabilities and FFmpeg.

- **Main View:** Shows vortices and tracers within the circular domain.
  - Tracers are rendered as small scatter points. Their color is determined by `TRACER_COLORING_MODE` (using `TRACER_CMAP`):
    - `group`/`scalar`: Color based on pre-assigned scalar values.
    - `speed`: Color based on instantaneous speed, normalized by a global maximum speed observed during the simulation (`global_max_speed_for_anim`).
  - Optional glow effects (`TRACER_GLOW_LAYERS`) can enhance tracer visibility.
  - Vortices are displayed as larger markers, with color indicating the sign of their strength Γ_k (e.g., yellow for positive, magenta for negative) and size scaled by $|\Gamma_k|$.
- **Diagnostic Plots:** The animation typically includes subplots showing the evolution of:
  - Angular Impulse $L_z(t)$, often as relative deviation $(L_z(t) - L_{z0})/L_{z0}$ if $L_{z0} \neq 0$.
  - Linear Impulses $P_x(t)$ and $P_y(t)$.
- **Output File:** Specified by `OUTPUT_FILENAME`, encoded using `FFMPEG_CODEC`.

4.3 Example Simulation Results

The dynamics of point vortices can be highly complex, leading to chaotic advection of tracers. Depending on initial conditions and vortex strengths, one might observe:

- Formation of stable vortex pairs (dipoles) or trios.
- Orbiting of weaker vortices around stronger ones.
- Merging-like behavior of same-sign vortices (if cores overlap significantly, though point vortices don't truly merge).
- Stretching and folding of tracer patches, indicative of chaotic mixing.

Example simulations, such as 'pv_group_1.3M.mp4' (illustrating grouped tracer advection with the 'jet' colormap) and 'pv_scalar_plume_1.3M.mp4' (showcasing scalar field evolution with the custom 'plume' colormap and glow effects), demonstrate these phenomena. These videos, along with the source code, are available at https://github.com/ChestnutKurisu/CSE291D_Final_Project_SP25. Figure 3 shows a representative snapshot from a simulation similar to 'pv_scalar_plume_1.3M.mp4'.

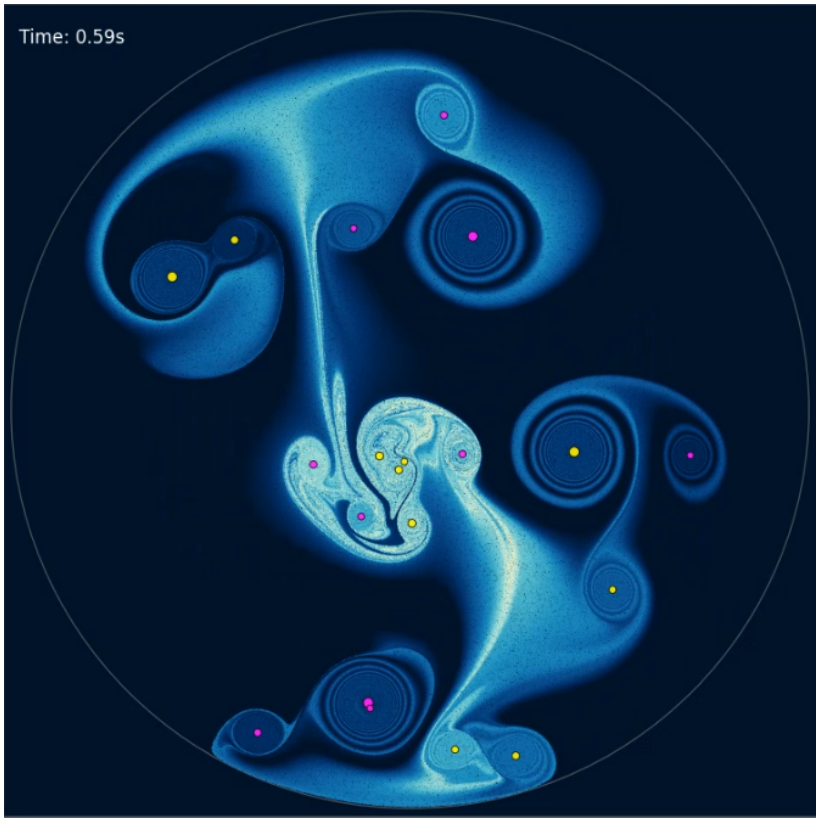

Figure 3: A snapshot from a simulation run (parameters similar to those for 'pv_scalar_plume_1.3M.mp4': $N_v = 20, N_t = 1.3 \times 10^6$, 'scalar' mode, $t \approx 5.0\,\text{s}$), illustrating the advection of tracer particles (shown as small colored dots) by the velocity field generated by the vortices (larger markers). Complex patterns often emerge due to the interactions.

The conservation of linear and angular impulses serves as a key diagnostic. The plots in Figures 4 and 5 show the evolution of these quantities from a representative simulation run detailed in Table 1. Ideally, for a perfectly symmetric system with $\Gamma_{\text{total}} = 0$ and perfectly reflecting boundaries, L_z and components of $\boldsymbol{P}$ would be well-conserved. In practice, numerical errors from RK4 integration, the approximate boundary enforcement (particle projection), and the influence of the background flow (if $\Gamma_{\text{total}} \neq 0$) can lead to small drifts or oscillations. The observed deviations are generally small, indicating good simulation fidelity.

Table 1: Parameters for the simulation run that generated the impulse-evolution plots (Figures 4 and 5).

Parameter	**Value**
Simulation Time	10 s
Time Step (`DT`)	0.002 s
Number of Vortices (`N_VORTICES`)	20
Squared Vortex Core Radius (a_v^2)	0.001
Domain Radius (R_D)	1
Random Seed	42
Initial L_z	0.75522
Final Rel. $\Delta L_z/L_{z0}$	-4.19×10^{-4}
Initial P_x	0.76063
Initial P_y	-0.11508
Final P_x	-1.266
Final P_y	0.28035

Note: Impulse conservation depends on the specific vortex configuration (e.g., total strength) and numerical factors. P_x, P_y can change more significantly if there's net drift or rearrangement relative to the origin, especially with image interactions. L_z is generally better conserved if $\Gamma_{\text{total}} \approx 0$.

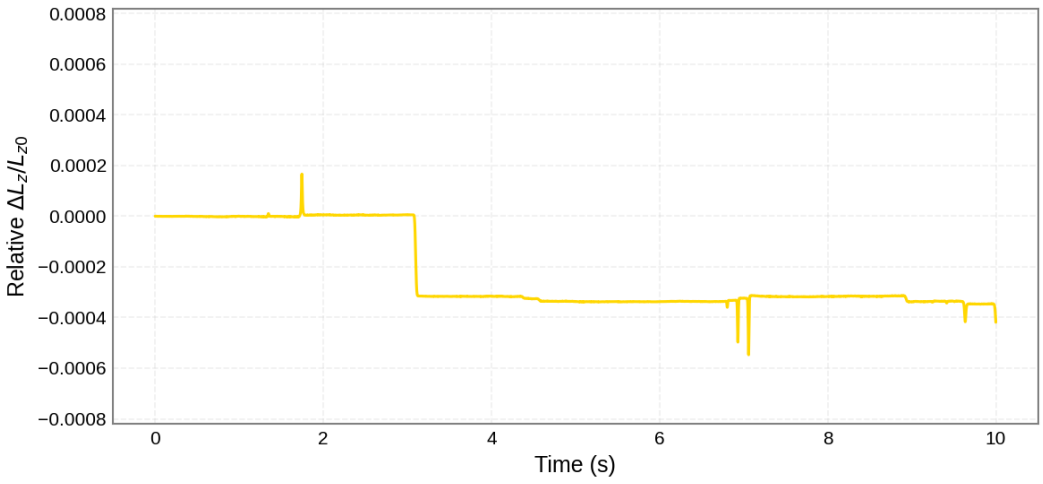

Figure 4: Evolution of the relative change in angular impulse ($\Delta L_z/L_{z0}$) over time for the simulation detailed in Table 1. Small fluctuations are typically due to numerical integration and boundary interactions.

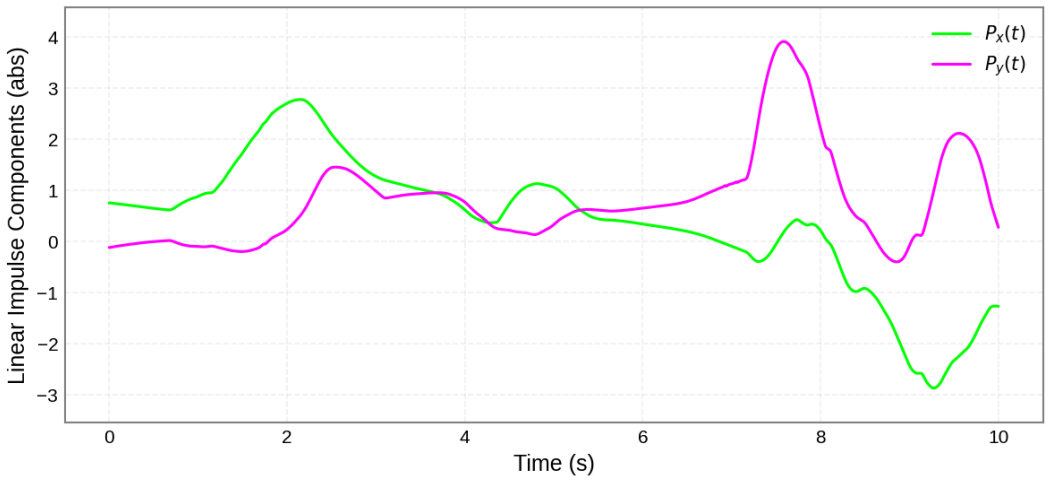

Figure 5: Evolution of linear impulses $P_x(t)$ (green) and $P_y(t)$ (magenta) over time for the simulation detailed in Table 1. These quantities are also expected to be nearly conserved, though changes can occur due to interactions with the image system representing the boundary.

5 Conclusion and Future Work

We have successfully developed and implemented a 2D point vortex dynamics simulation. The model incorporates Lamb-Oseen regularization for vortex cores, the method of images for handling a circular boundary, and passive tracer advection for flow visualization. The system's time evolution is computed using an explicit fourth-order Runge-Kutta scheme. A key feature of the implementation is its support for both CPU (accelerated by Numba) and GPU (via CuPy) computation, allowing for efficient simulation of systems with a large number of particles. The generated animations and diagnostic plots of conserved quantities demonstrate the capability of the simulator to capture the complex behavior of vortex flows.

Future Directions. Potential extensions and improvements to this work include:

- **More Complex Geometries:** Implementing the method of images or boundary element methods for more complex domain shapes (e.g., airfoils).
- **Viscous Effects:** Incorporating viscous diffusion, for example, by using random walk methods for vortex particle methods or by solving the vorticity-streamfunction formulation of the Navier-Stokes equations.
- **Performance Enhancements:** For very large N_v, exploring fast multipole methods or tree codes to accelerate the $O(N_v^2)$ velocity computations.
- **3D Vortex Dynamics:** Extending the simulation to 3D vortex filaments, which involves more complex Biot-Savart law calculations and filament reconnection models.

- **Adaptive Time Stepping:** Implementing adaptive time-stepping in the RK4 solver to improve efficiency and accuracy, especially when vortices undergo close encounters.
- **Interactive Interface:** Developing a graphical user interface for interactive setup of initial conditions and real-time visualization.

This project provides a solid foundation for further explorations in computational fluid dynamics and vortex methods.

A Details of Numba and CuPy Implementations

The computational performance for large numbers of vortices and tracers relies heavily on efficient execution of the velocity calculation routines. This appendix briefly outlines the strategies used for Numba (CPU) and CuPy (GPU).

A.1 Numba JIT-Compiled Kernels (CPU)

For the CPU execution path, key functions are JIT-compiled using Numba.

- `_lamb_oseen_factor_cpu_numba_impl`: This function calculates the Lamb-Oseen interaction factor $f(r^2, a^2) = (1-e^{-r^2/a^2})/r^2$. It typically involves an explicit loop (potentially parallelized with `prange` if operating on arrays) that iterates through squared distances, applies the formula, and handles the $r^2 \to 0$ limit.
- `_get_velocities_induced_by_vortices_cpu_numba_impl` (for tracers) and `get_vortex_velocities_cpu_numba_impl` (for vortices): These functions compute the total velocity for each target particle. They generally use nested loops: an outer loop over target particles (parallelized with `prange`) and an inner loop over source vortices (both real and image). Inside the inner loop, relative positions and squared distances are computed, the Lamb-Oseen factor is found (by calling its Numba version or inlining), and velocity contributions are summed up. This explicit looping, when compiled by Numba, often results in highly efficient machine code, rivaling or exceeding manually vectorized NumPy code for complex access patterns.

A.2 CuPy Kernels and Vectorization (GPU)

For the GPU execution path, CuPy is used.

- `_lamb_oseen_kernel` (CuPy `ElementwiseKernel`): As shown in Listing 1, this custom kernel applies the Lamb-Oseen factor calculation directly to CuPy arrays on the GPU. This is highly efficient as it minimizes data transfer and leverages massive parallelism.

- `_get_velocities_induced_by_vortices_xp` and `get_vortex_velocities_xp`: These functions are written using CuPy's NumPy-like API. Operations such as calculating all pairwise differences between target and source positions are achieved using broadcasting:

```
# E.g., diff_real = target_pos_exp - v_pos_exp
# target_pos_exp has shape (M, 1, 2)
# v_pos_exp has shape (1, N, 2)
# diff_real will have shape (M, N, 2)
```

  Subsequent calculations of r^2, applying the Lamb-Oseen factor (via the `ElementwiseKernel`), and summing contributions (e.g., `xp.sum(..., axis=1)`) are all performed as high-level vectorized operations executed on the GPU. This approach avoids explicit looping in Python and relies on CuPy's underlying CUDA libraries for parallel execution.

Both approaches aim to maximize parallelism and data locality appropriate for their respective hardware architectures. The choice between them is managed by the `GPU_ENABLED` flag in `SimConfig`.

References

[1] P. G. Saffman, *Vortex Dynamics*. Cambridge University Press, 1993. ISBN 9780511624063. https://doi.org/10.1017/CBO9780511624063

[2] H. Lamb, *An Introduction to Fluid Dynamics*. Cambridge University Press, 2000. https://www.cambridge.org/core/books/an-introduction-to-fluid-dynamics/18AA1576B9C579CE25621E80F9266993

[3] C. W. Oseen, "Über die Wirbelbewegung in einer reibenden Flüssigkeit," *Arkiv för Matematik, Astronomi och Fysik*, vol. 7, pp. 14–21, 1912. Available via the Swedish Royal Academy scan: https://books.google.com/books/about/%C3%9Cber_Wirbelbewegung_in_einer_reibenden.html?hl=da&id=wrCJPgAACAAJ.

[4] L. L. van Dommelen and S. Shankar, "Two counter-rotating diffusing vortices," *Physics of Fluids*, vol. 7, no. 4, pp. 808–819, 1995. https://doi.org/10.1063/1.868604

[5] L. M. Milne-Thomson, *Theoretical Hydrodynamics*, 5th ed. Macmillan, 1968. https://www.scirp.org/reference/referencespapers?referenceid=1894227

[6] G. K. Batchelor, *An Introduction to Fluid Dynamics*. Cambridge University Press, 2000 (corrected ed.). https://doi.org/10.1017/CBO9780511800955

[7] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT Compiler," in *Proc. LLVM-HPC '15*, Austin, TX, 2015. https://doi.org/10.1145/2833157.2833162

[8] Numba Development Team, *Numba v0.59.0 Documentation*. https://numba.pydata.org/ (accessed 3 Jun 2025).

[9] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations," in *Proc. Workshop on Machine Learning Systems (LearningSys) at NIPS 2017*, Long Beach, CA, 2017. http://learningsys.org/nips17/assets/papers/paper_16.pdf

[10] CuPy Development Team, *CuPy v13.0.0 Documentation*. https://cupy.dev/ (accessed 3 Jun 2025).

Acknowledgments: OpenAI's ChatGPT (model gpt-4o) was utilized to add comments and docstrings to Python functions, explain code implemented by me and ensure its correctness and alignment with equations of motion, and help debug code errors. The outputs from this AI model were modified with major changes. I actively reviewed, tested, and adjusted the generated code and explanations to reflect my own understanding.