pyFTLE: A Python Package for Computing Finite-Time Lyapunov Exponents

Renato Fuzaro Miotto [1*¶], **Lucas Feitosa de Souza** [1*], **and William Roberto Wolf** [1*]

1 School of Mechanical Engineering, University of Campinas, Brazil ¶ Corresponding author * These authors contributed equally.

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor:

Submitted: 27 January 2026
Published: unpublished

Summary

The Finite-Time Lyapunov Exponent (FTLE) field is a fundamental quantity for analysis of dynamical systems, with particular applications in fluid mechanics and transport phenomena. It enables the identification of Lagrangian Coherent Structures (LCSs) (Haller, 2015) by characterizing the trajectories of attracting, repelling and shearing material surfaces in manifolds. Hence, the FTLE plays a crucial role in elucidating transport mechanisms and identifying separatrices in fluid flows (Brunton & Rowley, 2010). This work presents a Python-based solver for computing FTLE fields from velocity data, featuring optimized integration techniques and parallel computations.

This work presents pyFTLE, a Python-based solver for computing FTLE fields from velocity data, featuring optimized particle integration, flexible interpolation strategies, and parallel execution. The software provides both a command-line interface (CLI) for large-scale, file-based workflows and a lightweight in-memory API suitable for interactive use in Jupyter notebooks.

The implementation emphasizes performance and reproducibility, combining Numba-accelerated Python kernels with a SIMD-optimized C++/Eigen backend for efficient 2D and 3D interpolation on structured grids. The package is distributed via PyPI, archived on Zenodo, and can be executed natively or within Docker containers, facilitating transparent deployment across platforms.

Statement of Need

Understanding transport barriers and coherent structures in unsteady flows is crucial in various scientific and engineering applications such as oceanography, meteorology, and aerodynamics. Despite the importance of FTLE computations, existing implementations are often specialized, computationally expensive, or lack extensibility.

pyFTLE addresses these limitations by providing a modular, high-performance package that:

- Supports both 2D and 3D velocity fields (structured or unstructured);
- Enables parallelized FTLE computations for improved performance;
- Offers flexible particle integration strategies (RK4, Euler, AB2);
- Provides multiple velocity interpolation methods for particle positions;
- Includes a SIMD-optimized C++ backend for efficient 2D and 3D interpolations on regular grids;
- Features an extensible design supporting multiple file formats;
- Offers a modular, well-structured codebase for easy customization and extension.

In addition to numerical performance, modern FTLE workflows require reproducible execution, flexible data ingestion, and scalable deployment. pyFTLE addresses these needs by offering a configuration-driven CLI, standardized MATLAB-based I/O, containerized execution via Docker, and continuous integration testing. This design allows researchers to apply FTLE analysis consistently across experimental, numerical, and analytical datasets, while remaining extensible for method development and benchmarking.

Implementation

Initially, a grid of particles $X_0 \subset \mathbb{R}^2$ is established across the domain of interest. These particles are integrated in the velocity field from the initial time 0 to the final time T, yielding a time-T particle flow map denoted as Φ_0^T defined as follows:

$$\Phi_0^T : \mathbb{R}^2 \to \mathbb{R}^2; \mathbf{x}(0) \mapsto \mathbf{x}(0) + \int_o^T \mathbf{u}(\mathbf{x}(\tau), \tau) d\tau \ .$$

Here, $\mathbf{u}(\mathbf{x}(\tau), \tau)$ denotes the time-dependent velocity field over the particle trajectory $\mathbf{x}(\tau)$ at a time τ. The flow map Jacobian $\mathbf{D}\Phi_0^T$ is then computed by a central finite difference scheme using the neighbouring particles in a Cartesian mesh such as:

$$\mathbf{D}\Phi_0^T = \begin{bmatrix} \frac{\Delta x(T)}{\Delta x(0)} & \frac{\Delta x(T)}{\Delta y(0)} \\ \frac{\Delta y(T)}{\Delta x(0)} & \frac{\Delta y(T)}{\Delta y(0)} \end{bmatrix} = \begin{bmatrix} \frac{x_{i+1,j}(T) - x_{i-1,j}(T)}{x_{i+1,j}(0) - x_{i-1,j}(0)} & \frac{x_{i,j+1}(T) - x_{i,j-1}(T)}{y_{i,j+1}(0) - y_{i,j-1}(0)} \\ \frac{y_{i+1,j}(T) - y_{i-1,j}(T)}{x_{i+1,j}(0) - x_{i-1,j}(0)} & \frac{y_{i,j+1}(T) - y_{i,j-1}(T)}{y_{i,j+1}(0) - y_{i,j-1}(0)} \end{bmatrix} ,$$

where x and y denote the particle coordinates and subscripts i and j denote their indices in the computational domain. Finally, the Cauchy-Green deformation tensor is computed as:

$$\mathbf{\Delta} = (\mathbf{D}\Phi_0^T)^* \mathbf{D}\Phi_0^T \ .$$

Here, $*$ denotes the transpose and the largest eigenvalue (λ_{max}) from this tensor is computed to form the FTLE field,

$$\sigma(\mathbf{D}\Phi_0^T; \mathbf{x_0}) = \frac{1}{|T|} \log \sqrt{\lambda_{max}(\mathbf{\Delta}(\mathbf{x_0}))} \ .$$

The FTLEs are computed using an auxiliary grid in which the flow properties are interpolated on. Figure 1 presents an example of the spatial discretization approach used in the present study. A reference mesh (indicated by the red dots in Figure 1) is first placed on top of the flow grid and the overlapping points on the airfoil solid surface are removed. The auxiliary grid (represented by the black dots) is then constructed with the maximum distance allowed from the reference points to the airfoil surface ensuring that all points remain outside the solid body.

From a software architecture perspective, pyFTLE distinguishes between structured and unstructured velocity data. When grid topology is provided, the solver exploits rectilinear grid assumptions to enable fast bi- and trilinear interpolation using a custom C++/Eigen backend. For unstructured datasets, interpolation relies on Delaunay triangulations, preserving flexibility at the cost of higher computational overhead. Parallelism is implemented at the snapshot level, where independent FTLE fields are computed concurrently using Python's multiprocessing framework.

To facilitate efficient I/O for time-resolved data, the implementation expects a set of plain text (`.txt`) files listing paths to the data: one list for velocity files, one for coordinate files, and one for particle files. The **velocity files** contain scalar components (e.g., `velocity_x`, `velocity_y`), while **coordinate files** specify the measurement locations (`coordinate_x`, `coordinate_y`).

Crucially, the **particle files** define groups of neighboring particles used to calculate the flow map Jacobian. Each row in the particle file contains a set of coordinates (tuples of `[float, float]` for 2D) defining the `left`, `right`, `top`, and `bottom` (and `front` and `bach` for 3D) neighbors surrounding a central location. This structure ensures that spatial relationships required for tensor deformation are clearly organized.

Auxiliary grid used to compute the flow map Jacobian. The properties at the reference point marked in red are computed by integrating then performing the central finite difference of the auxiliary grid points in black.

Figure 1: Auxiliary grid used to compute the flow map Jacobian. The properties at the reference point marked in red are computed by integrating then performing the central finite difference of the auxiliary grid points in black.

The FTLE fields can be computed by integrating the particles in forward or backward time. This choice yields different interpretations of the LCSs providing analogs for stable (forward time integration) and unstable manifolds (backward time integration) from dynamical systems (Brunton & Rowley, 2010; Haller, 2015). In the present work we employ the backward time integration of the particles as it enables a direct measure of material transport in forward time, mimicking experimental flow visualization by tracers.

Comparison of the vorticity field, the FTLE field, and the FTLE field shaded by vorticity sign for a moving airfoil.

Figure 2: Comparison of the vorticity field, the FTLE field, and the FTLE field shaded by vorticity sign for a moving airfoil.

Example Usage

The solver can be executed via the command-line interface (CLI) using the `pyftle` command. It requires several parameters, which can be passed as arguments or through a configuration file.

The following command illustrates a typical execution:

```
pyftle \
    --experiment_name "my_experiment" \
    --list_velocity_files "velocity_files.txt" \
    --list_coordinate_files "coordinate_files.txt" \
    --list_particle_files "particle_files.txt" \
    --snapshot_timestep 0.1 \
    --flow_map_period 5.0 \
    --integrator "rk4" \
    --interpolator "cubic" \
    --num_processes 4 \
    --output_format "vtk" \
    --flow_grid_shape 100,100,100 \
    --particles_grid_shape 100,100,100
```

Alternatively, a configuration file can be used:

```
python main.py -c config.yaml
```

Parameters

Parameter	Type	Description
experiment_name	str	Name of the subdirectory where the FTLE fields will be saved.
list_velocity_files	str	Path to a text file listing velocity data files.
list_coordinate_files	str	Path to a text file listing coordinate files.
list_particle_files	str	Path to a text file listing particle data files.
snapshot_timestep	float	Timestep between snapshots (positive for forward-time FTLE, negative for backward-time FTLE).
flow_map_period	float	Integration period for computing the flow map.
integrator	str	Time-stepping method (euler, ab2, rk4).
interpolator	str	Interpolation method (cubic, linear, nearest, grid).
num_processes	int	Number of workers in the multiprocessing pool. Each worker computes the FTLE of a snapshot.
output_format	str	Output format (mat, vtk).
flow_grid_shape	list[int]	(Optional) Grid shape for structured velocity measurements. It must be a comma-separated list of integers.
particles_grid_shape	list[int]	(Optional) Grid shape for structured particle points. It must be a comma-separated list of integers.

In addition to the CLI, pyFTLE exposes a lightweight Python API that allows FTLE computation directly from in-memory velocity fields, enabling compact, self-contained examples and interactive exploration in Jupyter notebooks without intermediate file I/O.

Optimized Interpolation

The solver's performance relies heavily on the definition of the grid shape. If `flow_grid_shape` is provided, the velocity field is treated as structured. This enables `pyFTLE` to utilize custom bi- and trilinear interpolators implemented in a SIMD-optimized C++/Eigen backend (by setting interpolator to `grid`), achieving up to 10x speedup compared to standard implementations. If `flow_grid_shape` is omitted, the data is treated as unstructured, and the solver defaults to Delaunay triangulation-based interpolation (cubic, linear, or nearest).

Software Quality

The software adheres to modern best practices in scientific Python development, including comprehensive automated testing with pytest, static code analysis, continuous integration, and fully automated online documentation generated with Sphinx. Versioned releases are published on PyPI and DockerHub and archived on Zenodo with a persistent DOI, ensuring long-term reproducibility, accessibility, and citability.

Recent works using pyFTLE

To date, the works that have utilized pyFTLE include: Lui & Wolf (2026), Lucas Feitosa de Souza et al. (2025b), Lucas Feitosa de Souza et al. (2025a), and Lucas F. de Souza et al. (2024)

Acknowledgements

We acknowledge Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) for supporting this work under Grants No. 2013/08293-7, 2013/07375-0, 2019/17874-0, 2021/06448-0, 2022/08567-9, 2022/09196-4, and 2024/20547-9. We also acknowledge the support of Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) under grant No. 304320/2024-2.

References

Brunton, S. L., & Rowley, C. W. (2010). Fast computation of finite-time Lyapunov exponent fields for unsteady flows. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, *20*(1), 017503. https://doi.org/10.1063/1.3270044

de Souza, Lucas Feitosa, Miotto, R. F., & Wolf, W. R. (2025a). Active flow control of vertical-axis wind turbines: Insights from large-eddy simulation and finite-time resolvent analysis. *Journal of Fluids and Structures*, *139*, 104410. https://doi.org/https://doi.org/10.1016/j.jfluidstructs.2025.104410

Haller, G. (2015). Lagrangian coherent structures. *Annual Review of Fluid Mechanics*, *47*(1), 137–162. https://doi.org/10.1146/annurev-fluid-010313-141322

Lui, H. F. da S., & Wolf, W. R. (2026). Interplay between streaks and vortices in shock-boundary layer interactions with conditional bubble events over a turbine airfoil. *Phys. Rev. Fluids*, *11*, 013401. https://doi.org/10.1103/sy5g-jz1m

Souza, Lucas F. de, Miotto, R. F., & Wolf, W. (2024). Analysis of dynamic stall for a simplified single blade vertical axis wind turbine configuration. In *AIAA AVIATION FORUM AND ASCEND 2024*. https://doi.org/10.2514/6.2024-4476

Souza, Lucas Feitosa de, Wolf, W., Safari, M., & Yeh, C.-A. (2025b). Control of deep dynamic stall by duty-cycle actuation informed by stability analysis. *AIAA Journal*, *63*(11), 4958–4969. https://doi.org/10.2514/1.J064980