© 2016 by Mingcheng Chen. All rights reserved.
LAGRANGIAN COHERENT STRUCTURES AND TRAJECTORY SIMILARITY: 
TWO IMPORTANT TOOLS FOR SCIENTIFIC VISUALIZATION 

BY 
MINGCHENG CHEN 

DISsertation 
Submitted in partial fulfillment of the requirements 
for the degree of Doctor of Philosophy in Computer Science 
in the Graduate College of the 
University of Illinois at Urbana-Champaign, 2016 

Urbana, Illinois 

Doctoral Committee: 
Professor John C. Hart, Chair 
Professor David A. Padua 
Associate Professor Luke Olson 
Associate Professor Shawn C. Shadden, University of California, Berkeley
Abstract

This thesis studies the computation and visualization of Lagrangian coherent structures (LCS), an emerging technique for analyzing time-varying velocity fields (e.g. blood vessels and airflows), and the measure of similarity for trajectories (e.g. hurricane paths). LCS surfaces and trajectory-based techniques (e.g. trajectory clustering) are complementary to each other for visualization, while velocity fields and trajectories are two important types of scientific data, which are more and more accessible by virtue of the technology development for both data collection and numerical simulation.

A key step for LCS computation is tracing the paths of collections of particles through a flow field. When a flow field is interpolated from the nodes of an unstructured mesh, the process of advecting a particle must first find which cell in the unstructured mesh contains the particle. Since the paths of nearby particles often diverge, the parallelization of particle advection quickly leads to incoherent memory accesses of the unstructured mesh. We have developed a new block advection GPU approach that reorganizes particles into spatially coherent bundles as they follow their advection paths, which greatly improves memory coherence and thus shared-memory GPU performance. This approach works best for flows that meet the CFL criterion on unstructured meshes of uniformly sized elements, small enough to fit at least two timesteps in GPU memory.

LCS surfaces provide insight into unsteady fluid flow, but their construction has posed many challenges. These structures can be characterized as ridges of a field, but their local definition utilizes an ambiguous eigenvector direction that can point in one of two directions, and its ambiguity can lead to noise and other problems. We overcome these issues with an application of a global ridge definition, applied using the hierarchical watershed transformation. We show results on a mathematical flow model and a simulated vascular flow dataset indicating the watershed method produces less noisy structures.

Trajectory similarity has been shown to be a powerful tool for visualizing and analyzing trajectories. In this paper we propose a novel measure of trajectory similarity using both spatial and directional information. The similarity is asymmetric, bounded within $[0, 1]$, affine-invariant, and efficiently computed. Asymmetric mappings between a pair of trajectories can be derived from this similarity. Experimental results demonstrate
that the measure is better than existing measures in both similarity scores and trajectory mappings. The measure also inspires a simple similarity-based clustering method for effectively visualizing a large number of trajectories, which outperforms the state-of-the-art model-based clustering method (VFKM).
To my parents, Liqing Chen and Chunjiang Ma.
I am fortunate to have John C. Hart as my advisor. John sets a model to me in two aspects. The first is his broad and deep knowledges and inspiring ideas as a top-tier all-around researcher. The second, more importantly, is his persistence, which is his magic turning failure into success. Our first paper was firstly submitted to IEEE VIS in my second year of the PhD program, and eventually accepted by IEEE TVCG in my fifth year. I still remember our early experiments were running on a brand new NVIDIA Tesla K20, with CUDA 3 and Ubuntu 12. In the experiments for our last revision, I used CUDA 6 and Ubuntu 14, and everyone else was using Tesla K80. This odyssey would not have been a success without John’s continuous efforts, which is a great support to me. The most impressive part is his correspondence with the editor regarding the comments from a reviewer who strongly disagreed with us, from which I learned a lot about how to defend the value of our work, which later became the key to success in my job interviews.

I am also fortunate to work with Shawn C. Shadden and Tingbo Hou. Our LCS projects benefitted a lot from Shawn’s proficiency in theory and plentiful flow datasets. I am also thankful to Shawn for hosting my visit to the University of California, Berkeley in my fifth year, so that I can live with my wife, who just started working at Google. The project of trajectory similarity would not have been possible without the patient mentorship of Tingbo Hou, both when I was interning at Google Maps in the summer of 2014 and in my fifth year. I am very thankful to Tingbo for his trust on my capability. Even during the 12 weeks internship, he offered me much freedom to try all the ideas. I also want to thank Siavash Ameli for the countless hours of discussions on the LCS projects, and hosting my visit to the Illinois Institute of Technology.

I also sincerely thank my other committee members, David A. Padua and Luke Olson for offering insightful comments and suggestions.

It was a wonderful experience working as a teaching assistant for CS440 Artificial Intelligence in both fall and spring semesters of my fourth year under the guidance of Gerald DeJong and Svetlana Lazebnik respectively. I am thankful to my colleagues (sorted by the family name) Megha Ailavajhala, Jason Cho, Codruta Girlea, Chinmay Kulkarni, Sajjadur Rahman, and Yanglei Song.
It feels like a lifetime ago, but I still remember the first two years in John’s group, when our crew was the largest among the five years. When Apollo Ellis and Jiajun Lu came here in my third year, there was only me. Those lovely labmates are (sorted by the family name), Kalina Borkiewicz, Neelan Coleman, Wonjun Jang, Kevin Karsch, Mariyam Khalid, Dale T. Kim, JohnMark Lau, and Victor Lu. I want to especially thank Mariyam and Victor for helping me improve my English and adjust to the life in graduate school. I miss every moment we had lunch or dinner together, discussed the courses, or exchanged ideas about everything. Victor is like a big brother to me. He arranged our trips to conferences, and accommodated me when I visited Chicago. He also kindly shared me a lot of advices on PhD life. I was very lucky to witness how Victor overcame the obstacles in his way towards graduation, and finally became an excellent researcher. I am also thankful to Zhicheng Liao and Zicheng Yan from the computer vision group and Zhao Dong from Cornell University for countless discussions.

I greatly value having met my best friends at the University of Illinois, Urbana-Champaign, Jialu Liu and Haitong Tian, my former roommates in different years. They are hard-working PhD students, outstanding researchers, and loyal friends. I sincerely appreciate every happy moment we enjoyed together and sad moment we lived through together. Hopefully the graduation does not separate us from each other. I also want to thank all my friends at the computer science department, especially (sorted by the family name) Gong Chen, Nan Chen, Tobias Lei, Qingxi Li, Xun Li, Haitao Liu, and Xiang Ren.

I must thank all my old friends from 2007 ACM Class at Shanghai Jiao Tong University for their years of support, especially (sorted by the family name) Kailong Chen, Yue Chen, Yu Cheng, Duo Ding, Huan Feng, Ziming Feng, Xiao Jiang, Nanxi Kang, Yang Li, Yulong Luo, Chenjin Liang, Bin Liu, Bo Tang, Li Tian, Dingquan Wang, Hang Wang, Minjie Wang, Chenyang Wu, Jiapeng Zhang, Peichao Zhang, Shaodian Zhang, Weinan Zhang, Yan Zhang, Fuzhong Zheng, Zhao Zheng, Zhongyue Zheng, Yuanjian Zhou, and Yujiao Zhou. I am honored to serve the SJTU Alumni Union at UIUC as the president since my second year. The union feels like a big family to me, where I met a lot of good friends. This part would not be complete without thanking (sorted by the family name) Qian Cheng, Yunlong Gao, Zhenhuan Gao, Zixia Huang, Huize Li, Qingzhou Luo, Zhe Shen, Hao Tan, Xu Wu, Mingfeng Xue, Zichao Ye, Xiao Yu, and Yuhao Zheng.

My work was supported in part by the National Science Foundation under Grant Numbers OCI-1047963 and OCI-1047764, and NVIDIA.

Most importantly, I am more than grateful to my parents, Liqing Chen (father) and Chunjiang Ma (mother), and my wife Ying Chen, for their endless unconditional encouragement and inspiration. None of my achievements would have been possible without their support.
# Table of Contents

Chapter 1  Introduction ................................................................. 1
  1.1 Lagrangian Coherent Structures .............................................. 1
  1.2 Trajectory Similarity .............................................................. 2
  1.3 Contributions ................................................................. 3

Chapter 2  Fast Coherent Particle Advection through Time-Varying Unstructured Flow Datasets ......................................................... 5
  2.1 Introduction ................................................................. 5
  2.2 Previous Work ................................................................. 8
  2.3 Streaming Advection Kernels .................................................. 9
  2.4 Streaming Advection Algorithms ............................................. 12
    2.4.1 Synchronous Advection .................................................. 12
    2.4.2 Asynchronous Advection ................................................ 13
    2.4.3 Block Advection (BA) .................................................. 13
  2.5 Analysis and Tuning ............................................................ 16
    2.5.1 Particle Block Density ................................................ 19
    2.5.2 Thread Assignment .................................................. 22
    2.5.3 Block Size ............................................................. 23
    2.5.4 Memory Allocation ................................................ 28
    2.5.5 Exiting Particle Transfer ........................................... 30
  2.6 Results and Comparisons ..................................................... 30
    2.6.1 Register Allocation ................................................ 31
    2.6.2 Overall Performance ................................................ 33
    2.6.3 Timestep robustness ................................................ 35
  2.7 Conclusion ................................................................. 35

Chapter 3  Watershed Ridges for Visualizing Lagrangian Coherent Structures .......... 37
  3.1 Introduction ................................................................. 37
  3.2 Lagrangian Coherent Structures ............................................. 38
  3.3 Watershed Segmentation ...................................................... 40
  3.4 Polygonization ................................................................. 42
  3.5 Results ................................................................. 45
  3.6 Conclusions and Further Research ........................................... 49

Chapter 4  Trajectory Clustering Using Both Spatial and Directional Similarity .......... 51
  4.1 Introduction ................................................................. 51
  4.2 Previous Work ................................................................. 53
  4.3 Point Similarity ................................................................. 56
  4.4 Trajectory Similarity .......................................................... 58
    4.4.1 Asymmetric Trajectory Similarity .................................... 58
    4.4.2 Continuous Form .................................................. 59
    4.4.3 Oversqueezing .................................................. 61
Chapter 1

Introduction

Velocity field is a typical dataset for studying dynamic systems, e.g. blood flows or air flows. The successful visualization of a complex velocity field relies on the ability to emphasize structure hidden within it. This is particularly true of flow datasets that in their most basic form contain a velocity vector at each point in space, essentially doubling the dimensionality of the dataset, which confounds an observer’s ability to perceive the data as a whole. Scientists are looking for analysis techniques that can simplify a flow dataset by recognizing and displaying structures representing similar flow characteristics. The recent and compelling method of Lagrangian coherent structures (LCS) is able to reveal the boundaries of regions of shared characteristics for unsteady fluid flow, called the skeleton of fluid flow.

Trajectory datasets are used to study any moving objects. In most cases, without the prior knowledge of certain underlying dynamic systems or generative models, the similarity between trajectories is the only way of analysis. Despite various domain-specific information, the most common feature of a trajectory is its shape and location, which together determine whether a pair of trajectories look similar to each other. Many definitions of similarity have been proposed in order to predict the visual similarity perceived by human eyes, but they all have their own drawbacks. A good measure of trajectory similarity can benefit quite a few applications, e.g. trajectory clustering.

This introductory chapter is organized as follows. Section 1.1 and Section 1.2 review the relevant background of Lagrangian coherent structures and trajectory similarity respectively. Section 1.3 states the contributions of this thesis.

1.1 Lagrangian Coherent Structures

LCS can be defined in a variety of different ways, but have been commonly visualized as the ridges of a scalar field called finite-time Lyapunov exponent (FTLE), indicating the divergence of neighboring pathlines in a time-varying flow. The computation of FTLE is based on the flow map \( \Phi(x, t_0, T) \), which takes \( x \) to its new position at time \( t_0 + T \) by integrating the velocity to trace the point along its trajectory (pathline),
thus is typically performed by integrating a dense grid of massless particles through the velocities of a computational fluid dynamics (CFD) or measured flow dataset. The integration is very time-consuming in practice. For example, a recent LCS analysis of revolving door heat loss [81] discussed the computation of their flow map, a map of start and end points of particle traces. In their case, their simulation data was a 1.65M cell unstructured grid over 200 time frames. The tracing of 1.34M particles over 280 adaptive fourth-order Runge Kutta integration steps (over a linearly interpolated time slice of their data) needed to compute their LCS flow map took 37 hours to compute on a pair of quad-core CPU’s. They stated that “the computation of the flow map is by far the most expensive part for any FTLE-based visualization,” and later “As future work, looking for ways to speed up the flow map computation is an obvious goal as it is computationally by far the most expensive part.” The time-varying unstructured grid is the most common type of data, where the integration is the most memory-bounded due to the frequent cell locations for each particle. This thesis investigates fast parallel many-core (GPU) algorithms for tracing particles through an unstructured mesh of time-varying flow velocities, which is targeting unstructured velocity data.

A current commonly used approach to extract ridges from datasets are local, based on a marching cubes fit of the local ridge configuration in a cell from the FTLE data at the vertices. The local definition of a ridge is often based on a matrix eigenvector, which only indicates the orientation of a line, but the ambiguity created by the fact that \(e\) and \(-e\) are both equally valid eigenvectors can lead to an orientation ambiguity when detecting the ridge surface numerically. This ambiguity can manifest as spurious false positives and other noise in the ridge surface extracted by local methods such as marching ridges often used for LCS extraction [82]. Watershed methods provide a global approach to extract topological structures from datasets. Sahner et al. [79] describe both a non-global “continuous” watershed approach that traces ridges as separatrices in the Morse structure, as well as a global “discrete” watershed transformation, and use the global watershed transformation to extract vortex and strain skeletal surfaces. We similarly propose and demonstrate watershed separatrix surface extraction for the visualization of flow structure, but for LCS instead of vortex/strain skeletal surfaces, and using a hierarchical watershed to filter out spurious details, to more clearly define the boundaries between neighboring watersheds as a global sea level rises.

1.2 Trajectory Similarity

Trajectories are typically represented as polylines, where points may have domain-specific quantities. For visualization purpose, however, trajectory similarity only depends on the spatial and directional information, which is our focus. Sung et al. [92] pointed out that the distance (or similarity) between trajectories is not as
clear as the distance between points or lines, and the majority of existing measures fail to capture the human intuition of similarity. We think it is because the spatial and directional information have not been used to full capacity. Most existing similarity measures consider only spatial information [93, 80, 95, 96], while a few consider only directional information [71, 6]. Some works on trajectory similarity approximation [3, 56] compute global statistics of the directional information separately from the spatial information. A good combination of both information is therefore necessary and urgent in the design of trajectory similarity, as it is the central problem in trajectory analysis [80].

Given a pair of trajectories $A = \{a_i\}$ and $B = \{b_j\}$, the similarity or the distance is defined by the optimal matching $M = \{(m^1_A, m^1_B), \ldots, (m^k_A, m^k_B)\}$ with non-decreasing sequences of $\{m^1_A\}$ and $\{m^1_B\}$. The four most common measures are Fréchet distance (FD), dynamic time warping (DTW), longest common subsequence (LCSS), and edit distance (ED) [93]. FD and DTW require the matching to cover all the points on both trajectories, thus suffer from the influence by distant portions on each trajectory [80]. LCSS and ED solve this problem by omitting distant pairs with a user specified threshold, however, cannot assign a point to multiple pairs.

FD indicates the shortest longest distance, while DTW indicates the shortest average distance. They have a common problem that a pair of short and neighboring trajectories always have short distance although their shapes may be very different, while a pair of long trajectories are likely to have a long distance although they may look very similar. As a consequence, it is impossible to pick a global threshold for “being similar” if trajectories are of various sizes. LCSS and ED require the users to specify the distance threshold, thus have a similar problem. ED is also a distance measure, indicating the minimum number of edits to make the two trajectories similar. The distances given by FD, DTW and ED are all poorly interpretable in the sense that the users cannot easily translate them into similarity. An ideal similarity measure is a number within $[0, 1]$, which is widely used for the analysis of human genetic similarity [100]. LCSS is the only measure directly indicating the similarity, after normalization whose output is able to fall within $[0, 1]$.

### 1.3 Contributions

Chapter 2 focuses on solving the most time-consuming part of computing FTLE fields on GPU, tracking a dense grid of massless particles through an unstructured mesh of time-varying fluid flow. Our approach reorganizes particles into spatially coherent bundles as they follow their advection paths, which greatly improves memory coherence and thus shared-memory GPU performance.

Chapter 3 focuses on a robust ridge extraction approach on FTLE fields using watershed method. We
apply a region merging criteria similar to topological persistence that ranks ridges based on their configuration relative to neighboring ridges and valleys. This new filtering enabled by a global approach yields improved LCS extraction from scalar field data and better visualization of unsteady flow structure.

Chapter 4 focuses on a novel measure of trajectory similarity using both spatial and directional information. Our similarity is asymmetric, bounded within $[0,1]$, affine-invariant, and efficiently computed. It also provides asymmetric mappings between a pair of trajectories. We show that our measure is better than existing measures in both similarity scores and trajectory mappings. Our measure even enables a simple similarity-based clustering method to outperform the state-of-the-art model-based clustering method.
Chapter 2

Fast Coherent Particle Advection through Time-Varying Unstructured Flow Datasets

Tracing the paths of collections of particles through a flow field is a key step for many flow visualization and analysis methods. When a flow field is interpolated from the nodes of an unstructured mesh, the process of advecting a particle must first find which cell in the unstructured mesh contains the particle. Since the paths of nearby particles often diverge, the parallelization of particle advection quickly leads to incoherent memory accesses of the unstructured mesh. We have developed a new block advection GPU approach that reorganizes particles into spatially coherent bundles as they follow their advection paths, which greatly improves memory coherence and thus shared-memory GPU performance. This approach works best for flows that meet the CFL criterion on unstructured meshes of uniformly sized elements, small enough to fit at least two timesteps in GPU memory.

2.1 Introduction

The analysis and visualization of flow datasets often rely on tracing the paths of collections of particles through the flow. This process is key to the production of standard visualization tools for steady and unsteady flow datasets including streamlines, streaklines, pathlines and rakes.

While particle advection has many applications, our primary motivation is due to it being a critical roadblock in the efficient computation of Lagrangian Coherent Structures (LCS) [68, 33, 86]. For unsteady flows, analytical topological features of the vector field have proven to be less useful for identifying the important flow features. The computation of LCS identifies topological features from quantities derived from particle paths. This approach analyzes how flow unsteadiness affects the existence and shape flow features, but often requires the tracing of massive numbers of particle paths. Hence the utility of LCS for analyzing and visualizing unsteady flow creates further demand for fast particle advection schemes.

The computation of LCS is typically performed by integrating a dense grid of particles through the velocities of a computational fluid dynamics (CFD) or measured flow dataset, with application to visualization as well as visual effects [104]. Several definitions of LCS exist, as do methods to reduce its computational
complexity. However all methods generically require well-resolved particle trajectory data in order to subsequently locate coherent sets that have trajectories with appropriate distinguishing properties. For example, LCS are commonly identified as ridge surfaces of finite-time strain, often by computing the Cauchy-Green (CG) strain tensor at particle path origins. Notably, the finite-time Lyapunov exponent (FTLE) is defined by the largest eigenvalue of this CG tensor, and its spatial distribution has commonly been used to visualize LCS \[25, 39, 87, 54\]. Moreover, LCS that extremize normal strain \[40\] or shear strain \[41\] are similarly identified from the eigenvectors of the CG strain tensor. Further alternative methods define LCS by tracking how particles are mapped over the fluid domain, as eigenfunctions of the resulting transfer-operator \[32\], or through appropriate averages along trajectories \[63, 14\]. In this study we do not focus on any particular method of LCS computation, but rather their shared bottleneck of dense particle path integration.

For example, a recent LCS analysis of revolving door heat loss \[81\] discussed the computation of their flow map, a map of start and end points of particle traces. In their case, their simulation data was a 1.65M cell unstructured grid over 200 time frames. The tracing of 1.34M particles over 280 adaptive fourth-order Runge Kutta integration steps (over a linearly interpolated time slice of their data) needed to compute their LCS flow map took 37 hours to compute on a pair of quad-core CPU’s. They stated that “the computation of the flow map is by far the most expensive part for any FTLE-based visualization,” and later “As future
work, looking for ways to speed up the flow map computation is an obvious goal as it is computationally by far the most expensive part.” (This has motivated others as well, e.g. to consider an adaptive approach to computing FTLE [10].)

In this paper, we investigate fast parallel many-core (GPU) algorithms for tracing particles through an unstructured mesh of time-varying flow velocities. We target unstructured velocity data, which is growing in prevalence in CFD for solving increasingly complex applications, and those that consist of roughly uniform-sized tetrahedral elements. We propose a new particle asynchronous approach that yields better throughput than a recently published particle synchronous approach, and outperforms previous GPU advection methods when particle advection adheres to the CFL condition. This approach is further improved by a multistage Runge-Kutta integration kernel designed to reduce register utilization. Moreover, a new GPU-tuned block advection system localizes memory access to overcome the cache misses that degrade the performance of other methods as particles disperse across the flow domain, though this approach is currently restricted to flow datasets that can fit at least two time steps worth of velocity data into GPU memory.

Together, these contributions result in a fast memory coherent particle advection engine for tracing particles through time-varying unstructured mesh flow datasets. This new approach can trace 8M particles over 10K (4-stage) integration steps\(^1\) in less than four minutes — the same computation time that the revolving door LCS paper [81] reported for tracing 1.34M particles for one single integration step.

Section 2.2 reviews recent work on high performance algorithms for integration and particle advection through flow fields, for both steady and unsteady flow, and for regular grids and unstructured meshes. Section 2.3 summarizes the basic kernel for unstructured mesh particle advection, including a multistage Runge-Kutta algorithm that reduces register utilization for faster GPU computing. Section 2.4 describes synchronous and asynchronous algorithms for advecting particles, and introduces the block advection method for asynchronous particle tracing. Section 2.5 analyzes the performance of block advection and provides specific heuristics for setting its constants. For example it finds a performance sweet spot when the block width is about five times the average cell width. Section 2.6 measures the performance of block advection and shows it outperforms recently published synchronous and asynchronous GPU particle advection methods. Section 2.7 concludes with a discussion of limitations and directions for further research.

\(^1\)We use the term *step* to refer to a full multistage integration over \(\Delta t\), whereas we use the term *stage* to denote a single stage of a multistage integration step.
2.2 Previous Work

Kruger et al. [50] showed the GPU could be used as an effective processing platform for particle advection in a full visualization system. This and related work [49, 50] focused on advecting particles through regular rectilinear grids. Garth, et al., [36] and Hlawatsch, et al. [46] leveraged GPU processing for particle advection in the computation of LCS, though these papers did not discuss details of their implementation or GPU optimization since this was not a focus of either paper. The recent paper by Conti, et al. [24] described FTLE computation on the GPU for a specialized application of remeshed vortex methods for bluff body flows that involved mesh-particle interpolations previously tailored for GPU implementation.

Burger et al., [15] used a vertex shader to trace dense particles through a uniform grid to create streak surfaces, controlling the particle density as the streak surface particles converged and diverged. Ferstl et al. [31] combined these GPU streak surfaces with GPU ridge extractions of the FTLE to compute LCS-like structures over a regular grid.

Data management and scheduling are key to scalable parallel implementation, especially when dealing with the large datasets used for unsteady flow fields. Burger et al. [16] used the CPU to page in new time frames of the entire spatial extent of regular-grid unsteady flow data to support GPU’s particle integration. We similarly work with a paged time frame of unsteady data, linearly interpolating velocities from the two velocity samples loaded in memory. We assume massless non-interacting particles. Our approach would not be suitable for interacting GPU particle simulations, e.g. SPH [45]. Our approach also requires two time samples of the entire velocity dataset to fit into GPU memory, which sufficed for our examples but would require additional work to extend to larger out-of-core datasets as shown in Camp et al. [19].

Irregular mesh, e.g. tetrahedral, data poses an additional challenge, as each particle must locate the mesh cell containing it to properly interpolate its advecting velocity. For particle advection, we rely on the common requirement that the time step should be sufficiently small that particles do not traverse farther than one or two cells between interpolation steps. In such cases, a cell walking strategy [48] is highly efficient for cell location. Our cell location uses this strategy, and we use it for the design and tuning of the block overlap of our GPU memory allocation. Macpherson et al. [58] has even extended this cell walking to handle non-manifold and high genus tet meshes with small overlaps and holes, though such meshes are relatively uncommon.

Often a k-D tree or other global location is used to initially assign cells to particles, followed by successive local neighborhood searches that find the cells into which particles have advected. Schirski et al. [84] combined GPU advection with CPU global cell location, but required costly CPU-GPU communication. Bußler et al. [17] implemented a GPU cell location algorithms for neighborhood search [84] as well as GPU
algorithms for exact and fast-approximate traversal of a k-D tree precomputed on the CPU. They showed that an approximate but divergence-free single pass through the k-D tree reduced the neighborhood search enough to justify its use for global location. Martin et al. [60] examined optimizations to cell walking in unusual cases where integration frequently moved particles farther than one cell away, but such cases exceed common stability of accuracy criteria.

In fluid dynamics, the stability and fidelity of a solution rely on the CFL (Courant-Friedrichs-Lewy) condition, which ensures that no information (e.g. wave, fluid particle) is allowed to propagate more than one element of the mesh at a time. The CFL condition improves CFD stability by bounding energy growth. In our case it improves the accuracy of particle advection by ensuring all of the velocity data encountered along a particle trajectory is properly sampled and integrated.

Garth & Joy [37] describe the CellTree, which accelerates global particle location on the GPU using a bounding interval hierarchy. Their results outperformed other global location techniques, by factors ranging from about 1.75× to over 3.5× faster. They also outpaced the local particle location search [52] of FAnToM [98] on streamline integration, ranging from no speedup to 1.35×. Their paper does not report the number of streamlines or integration steps. As our cell walking approach is similar to the one used in FAnToM, we would expect the CellTree to similarly improve our results on initial global point location, but not the local neighborhood location following the one-element CFL condition used for integrating and tracing our particle trajectories.

The GPU can also be applied to the general integration of ordinary differential equations in addition to particle tracing. Murray [65] implements reduced-memory versions of multistage adaptive integration methods including DOPRI5 and RK4(3)5[2R+1]C, using an asynchronous approach for better load balancing. Our multi-stage Runge-Kutta integration is similar to this work, though Murray sought to minimize the storage of all constants whereas we focus specifically on reducing register usage.

### 2.3 Streaming Advection Kernels

We consider unsteady flow data stored as time-sequences of velocity vectors at the nodes of an unstructured mesh. The mesh data structure is cell-based with connectivity information that permits easy access to neighboring cells. We work in 3-D, and the 3-cells (which we simply refer to as “cells”) are typically tetrahedra, though our approach can handle any convex polyhedral cell shapes.

We process as input the initial positions (typically organized in a regular rectilinear lattice) of particles to advect through the time-varying velocity field interpolated in space and time from the unstructured mesh.
nodes.

The advection of particles through an unsteady (time-varying) flow measured at the nodes of an unstructured mesh follows in general a repeating pattern of computation consisting of particle location, velocity interpolation and particle integration shown in Alg. 1. Note that “Integration” represents one stage of a multi-stage integration step (e.g. fourth-order Runge-Kutta), since each stage requires its own location and interpolation of the unstructured mesh data.

**Algorithm 1: Unstructured Particle Advection Kernel**

```plaintext
repeat
  Location ;
  Interpolation ;
  Integration ;
until done;
```

The “Location” step determines which cell contains each particle. We use the “walking” method [48] to perform a local, directed search to find the cell containing a particle after it has been advected from its original cell. A (convex) cell is formed by the intersection of planar half-spaces, and a particle lies in this cell if it sits in the “interior” side of all of these planes. If a particle has left the current cell, it will lie on the “exterior” side of one or more of these planes, and we pick one of these planes that the particle has moved farthest away from. We test particle for membership in the cell that shares that plane’s face with the current cell, and continue the search from that cell recursively until we find a cell containing the particle.

The “Interpolation” step linearly interpolates the velocity from the surrounding time step values stored at the cell’s vertices in the flow dataset. For tetrahedral meshes, a particle lying on a face, edge or vertex shared by multiple neighboring cells can be arbitrarily assigned to any one of these cells since the barycentric interpolation of its vertex values yields the same result regardless of cell choice.

The walking cell location method requires that the cells be organized into an adjacency structure, such as are produced by common mesh generation methods such as TetGen [91]. Meshes with t-intersections, disjoint coincident cell faces and other non-manifold artifacts would require global location method such as the CellTree [37]. Such global approaches might also be appropriate if particle integration step across many cells, but the stability and fidelity of such a flow simulation usually relies on setting the time resolution for integration consistent with the spatial resolution of the mesh, so we expect particles infrequently move more than one cell away.

A global location method is also needed to determine the initial cell location of seed particles. In practice, and specifically for LCS computation, the choice of initial layout organization can be arbitrary, and particles are typically seeded in a regular axis-aligned (AA) grid arrangement. Alg. 2 uses this regularity, streamed
over parallel buckets of particles, to enumerate the particles in each cell’s bounding box, outputting only those particle positions in the current cell. This results in a $O(M + P)$ time complexity if we assume the number of overlapping cell bounding boxes is constant. (In the worst case the overlap can be proportional to $M$ since smaller long-skinny cells can incur more overlap, but this is not a realistic property for a typical unstructured mesh.) We found the performance of this particle-grid initialization algorithm benefits well from orderly distributed access to both the particles and the flow dataset.

<table>
<thead>
<tr>
<th>Algorithm 2: Initial Cell Location for AA Grid Particles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong>: $P$ seed particles, $M$ cells and $N$ threads, where $P \gg N$.</td>
</tr>
<tr>
<td><strong>foreach</strong> cell $C_j$ in parallel <strong>do</strong></td>
</tr>
<tr>
<td>Let $B$ be the axis-aligned bounding box of $C_j$ ;</td>
</tr>
<tr>
<td><strong>foreach</strong> grid particle position $x \in B$ <strong>do</strong></td>
</tr>
<tr>
<td>test $x_i \in C_j$ ;</td>
</tr>
</tbody>
</table>

We also tried a similar parallel initial cell location algorithm that streamed over spatially coherent buckets of particles in any layout. It used global positioning (e.g. a parallel k-D tree [106, 23] or CellTree [37]) to locate the first particle in each bucket, and local positioning (cell walking) to locate the bucket’s remaining particles. We found that the global positioning structure construction was quite expensive, and that overhead outweighed the benefit of the later faster local positioning steps. We also found that the algorithm critically depended on the property that the first particle in each bucket must lie in the flow mesh domain, otherwise a conditional was needed that impeded GPU data parallelism. This first particle issue was not a problem when the initial particle positions lied inside the dataset, but could not be easily guaranteed for arbitrary cases.

For the “Integration” step, we implemented fourth-order Runge-Kutta as shown in Algorithm 3, designed to reduce register usage. Hence, the RK4 integration of a particle required four iterations through the location-interpolation-integration kernel, such that the intermediate Runge-Kutta points can be located and their velocities interpolated.

The previous work on GPU advection of particles through unsteady flow datasets similarly report results using non-adaptive (third-order) Runge-Kutta integration [17, 37]. This multi-stage approach could be applied to the GPU implementation of adaptive integration methods, e.g. DOPRI5 and RK4(3)5[2R+1C] [65], though such methods were not included in our experiments. For such adaptive cases, the “stage” parameter would be advanced conditionally, which could lead to increased thread divergence. Adaptive integration may also introduce further irregularities that could degrade the proposed block advection strategy.
Algorithm 3: Streaming Multistage RK4

**Data:** \((x_0, x, t, v, \text{stage})\) where

- \(x_0, x_1\) maintain the original and accumulated particle positions,
- \(x, t\) are the current integration sample position and time,
- \(v\) is the velocity interpolated at position \(x\) and time \(t\), and
- “stage” is the number of the current integration stage.

**Result:** Updated stage and new integration position \(x\) and time \(t\).

```plaintext
switch(stage)

  case 1:  \(x_0 = x,\)  \(x_1 = x + \frac{\Delta t}{6} v,\)  \(x = x_0 + \frac{\Delta t}{2} v,\)  \(t += \frac{\Delta t}{2};\)

  case 2:  \(x_1 += \frac{\Delta t}{3} v,\)  \(x = x_0 + \frac{\Delta t}{2} v;\)

  case 3:  \(x_1 += \frac{\Delta t}{3} v,\)  \(x = x_0 + \Delta t v,\)  \(t += \frac{\Delta t}{2};\)

  case 4:  \(x = x_1 + \frac{\Delta t}{6} v;\)

stage = (stage mod 4) + 1 ;
```

2.4 Streaming Advection Algorithms

We target the modern GPU as the computational platform for high-performance unstructured mesh particle advection. In general, the GPU is a many-core processor that distributes parallel threads onto multiple MIMD warps of typically 32 SIMD parallel processors, and multiple warps are collected into SM processing units that share local memory and draw from the same pool of registers. Register contention commonly hinders GPU performance by limiting the number of threads in flight, so we strive to reduce register utilization.

2.4.1 Synchronous Advection

Ameli et al. [4] implemented synchronous advection, which keeps all particles synchronized as they advect through a flow field, as shown in Alg. 4.

Algorithm 4: Synchronous Advection [4]

```plaintext
foreach integration stage of each timestep \(\Delta t\) do
  foreach particle \(i\) in parallel do
    locate \(i\) ;
  endforeach
  foreach particle \(i\) in parallel do
    interpolate \(i\) ;
  endforeach
  foreach particle \(i\) in parallel do
    integrate (one stage) \(i\) ;
  endforeach
```

Particle synchronous advection has several advantages. The location-interpolation-integration kernel are separated into three individual kernels, each run on all particles before moving to the next (e.g. all particles
find their containing cell before any interpolate their velocity). This improves performance by reducing register usage since some kernels use less registers that others. Synchronization also reduces code divergence since particles all run the same subkernel, and all particles will share the same integration stage state in the integration subkernel. Furthermore particles can exchange information in a synchronized manner which can support particle coupling.

The main disadvantages of synchronous particle tracing are load balancing and global memory access. Each locate kernel (cell walking) runs for a small but unknown number of iterations, so some instantly located particles may need to wait for others to find the right neighboring cell that contains them. Because synchronous advection uses three individual kernels, when a new kernel is loaded, the previous kernel needs to write its results to global memory, and the new kernel needs to read these results back in for further processing. This global memory overhead can be avoided by the asynchronous advection mentioned next.

### 2.4.2 Asynchronous Advection

Particle asynchronous advection allows each thread to trace its particles independently, until additional flow data is needed, as shown in Alg. 5. Particle asynchronous advection avoids repeated per-integration-stage reads and writes to global memory. When particles leave the flow domain, the synchronous approach must either waste computation on them, or perform an expensive compaction to remove these particles, whereas the asynchronous approach can simply kill the thread, leading to improved load balance. Particles traced in the asynchronous approach benefit from better memory coherence when referencing the unstructured mesh, since the particle position will remain in the same cell or move to a nearby cell.

<table>
<thead>
<tr>
<th>Algorithm 5: Asynchronous Advection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>foreach</strong> dataset time frame do</td>
</tr>
<tr>
<td><strong>foreach</strong> particle i in parallel do</td>
</tr>
<tr>
<td><strong>repeat</strong></td>
</tr>
<tr>
<td>locate-interpolate-integrate particle i ;</td>
</tr>
<tr>
<td><strong>until</strong> particle i reaches end of current time frame (or exits flow domain);</td>
</tr>
</tbody>
</table>

Particle asynchronous advection allows thread warps to execute asynchronously, and these threads persistently focus on their current particles, reducing global memory writes of particle state. Our experiments verify that for these reasons, the asynchronous approach generally outperforms the synchronous approach.

### 2.4.3 Block Advection (BA)

The asynchronous approach assigns a particle to a thread, and lets that thread trace the particle from its initial position until it reaches the space (outflow) or time frame boundary of the flow data, or desired
Figure 2.2: A block (a) caches any unstructured flow cells (gray) that have a portion within the outer (solid) block boundary. Any particles in the solid outer-block boundary are traced until they exit. The region between the outer-block solid boundary and the dashed inner-block boundary overlap neighboring inner blocks (and their outer blocks overlap the dashed inner-block). The overlapping outer blocks of nine adjacent blocks are shown in (b), with the outer block of the center block in bold, and their inner blocks shown dashed. A particle is not added to a block’s processing queue until it leaves its neighboring block’s outer-block region.

maximum integration time. Such approaches can benefit from spatial coherence if neighboring particles are assigned to processors that share the same cached portion of the velocity data. However, particles that are initially proximate generally diverge in most flows, and any benefits from the cache coherence of their initial positions soon degrade.

We propose to preserve this cache coherence by dynamically re-clustering particles as they advect through the unstructured mesh, such that these clusters can share the same cached chunks of the unstructured mesh flow data. Rather than clustering particles, e.g. by k-means, we organize the flow domain into overlapping rectilinear block, such that particles within a block can utilize the same cached unstructured mesh flow data. (Some blocks may contain too many particles, which require multiple SM thread blocks each with its own copy of the block’s mesh data.) These blocks are organized on a uniform structured spatial grid. Blocks that do not intersect the unstructured mesh are ignored.

Particles within a block are traced until they leave the block as shown in Fig. 2.2, and their advection
Figure 2.3: Blocks overlap spatially but not temporally. A single particle trace is computed in seven segments as it crosses various block domains.

performance benefits from the cached flow mesh data. When a particle exits a block’s boundary, it must be removed from the current block and added to a neighboring block for processing. In order to avoid alternating block processing for trajectories that straddle box boundaries (Fig. 2.3), we allow the blocks to overlap. Hence each block consists of an inner block and an outer block, where the outer block overlaps its neighboring block’s outer-block and inner-block regions.

We use the inner blocks to initially allocate particles to each block’s processing thread. During advection, we only use the (overlapping) outer blocks, transferring the particle to a new thread only when it leaves it’s current block’s outer block region. The particle would then be assigned to the unique block whose inner region contains the particle.

The blocks also extend in time, from a flow data sample time $t_0$ to sample time $t_1$. This (global) block time frame\(^2\) is dictated by the amount of velocity data that can be loaded into GPU memory at a time. The blocks do not overlap in time, except that the previous end time $t_1$ becomes the new start time $t_0$ when the next set of velocity data is loaded. For our examples, we simply set the block time frame to load two velocity time samples per node. We note that these shorter time frames help avoid the load imbalance that occurs when some particles remain in their spatial blocks while others leave them quickly.

\(^{2}\)We use the term *time frame* to refer to both the number of time samples in an unsteady flow dataset, the duration between these time samples and the time duration of our blocks. Since we recommend each block uses two time samples, we trust there is no confusion with this use of terminology.
cency structure of the cells that lie within its outer boundary. The cells that intersect the outer boundaries and overlapping regions of blocks will thus be duplicated in a neighboring block’s local adjacency structure. As block size decreases and as the overlap margin increases the memory wasted by this duplication increases.

<table>
<thead>
<tr>
<th>Algorithm 6: Block Advection (BA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>foreach space-time block in shared-memory parallel do</td>
</tr>
<tr>
<td>load block’s cells into shared memory ;</td>
</tr>
<tr>
<td>foreach particle i in a parallel thread do</td>
</tr>
<tr>
<td>repeat</td>
</tr>
<tr>
<td>locate-interpolate-integrate particle i ;</td>
</tr>
<tr>
<td>until particle i exits current space-time block;</td>
</tr>
<tr>
<td>transfer exiting particles to neighboring blocks ;</td>
</tr>
</tbody>
</table>

This block advection algorithm is described in Alg. 6. It maps a given number (one or more) of particles to each thread in a thread block. The threads in the thread block trace the particles until all have escaped the block’s space or time bounds, or after a preset maximum thread computation time has been reached. Particles that escape the block are labeled “inactive” and even though they may continue to be processed by their threads, they remain frozen at the place and time of their escape.

When a particle crosses the (outer) boundary of its block, it will need to then be processed by a neighboring block using the neighboring block’s local cell adjacency structure. To facilitate this re-assignment of a particle’s cell location to the index used in the new block’s local cell adjacency structure, we maintain a global hash table indexed by each cell’s ID in the global adjacency structure that returns the list of blocks whose outer-boundary overlaps the cell, and the ID of the cell in the local adjacency structure of each of those blocks.

This blocked approach introduces additional code divergence beyond that of the asynchronous algorithm it is based on. If a particle leaves its flow block in a middle stage of a multi-stage integration, it will start in that stage in its neighboring flow block even though other particles may be at different stages of integration. To reduce this divergence, we sort particles by integration stage (as used in Alg. 3) before we begin processing each flow block. Experiments show that RK stage sorting improves tracing performance by about 13% even after adding the extra expense of the sort. While we use a fixed stage integration, the threads of an adaptive integration approach could be similarly coalesced.

### 2.5 Analysis and Tuning

We implemented our algorithms in double precision on an NVIDIA Tesla K20 GPU, with 13 SMX multiprocessors each with 192 cores for a total of 2,496 cores (though for double precision the number of cores per
SMX is reduced to 64). The SMX multiprocessors can each execute up to 64 warps sharing 64K registers, reaching a peak 1.17/3.52 T(dp/sp)FLOPS. The K20 has 5GB of GDDR5 memory accessed at 208 GB/s, with 64KB available per SMX multiprocessor, organized as 48KB shared memory and 16KB L1 cache, or vice versa.

We tested our algorithm on various data sets, which included three internal flow datasets and two external flow datasets. The internal flow examples were cardiovascular datasets used to analyze the diagnosis of adverse blood flow conditions, and are typical for biological and health simulations [88, 102, 20]. These flows reside in complex geometries, are highly unsteady, and exhibit both laminar and transiently turbulent flow regimes. The aortic model (Fig. 2.1) presents an abdominal aortic aneurysm, with complex flow structures in the aneurysmal region. The total cavopulmonary connection (TCPC) model (Fig. 2.4) contains multiple inlets and outlets, along with boundary layer meshing. The upper/cerebral vasculature (UV) model (Fig. 2.5) is useful because it contains a wide variety of flow regimes, and its divergent branching separates flows spatially.
Figure 2.5: The cerebral vasculature dataset, consisting of 798,251 tetrahedra, 165,547 nodes and 50 time frames per cycle. The displayed flow was computed on 40K particles starting at the entry for one cycle. In our performance experiment, 8,120,601 particles are traced for 18,000 integration steps by 454 executions of the block advection GPU kernel.

Figure 2.6: Airflow over an angled gray wing, which use using an anisotropic adaptive velocity grid with a $10^3$-cell refined central region. This dataset has 10 velocity time frames per cycle. The displayed flow was computed on 40K particles starting behind the wing for two cycles. Particles are advected from an initial plane, and their color is set to their altitudes in this plane. In our performance experiment, 8,120,601 particles are traced for 950 integration steps by 137 executions of the block advection GPU kernel.
We also analyzed two external flow examples. The first (Fig. 2.6) is a low aspect ratio wing (AR=2) from Kunihiko Taira at Florida State University, which demonstrates the kind of external flow example common in both aerospace and swimming/flying applications. Its irregular space mesh is highly anisotropic, with very small boundary layer elements surrounding the airfoil, and increasingly larger elements near the surrounding free stream. The second external flow example is a periodically-forced Rayleigh-Bénard type convection cell [54], which represents large scale unsteady vortices similar to those observed in geophysical flows.

We primarily analyze and tune the blocked advection algorithm on the abdominal aorta dataset in Fig. 2.1, which consists of 1,004,926 tetrahedra, 193,840 nodes and 25 time frames per cycle. The displayed flow was computed on 40K particles starting at the top entry for one cycle. In our tuning analysis experiments, 8,120,601 particles are traced for 10,000 integration steps by 344 executions of the block advection GPU kernel.

This analysis focuses on the advection of particles initiated at an inflow location, and the characteristics of the results are influenced by the dispersion of particles across the domain. Most practical applications are open flows whereby flux occurs on boundaries of the computational domain. In such applications, FTLE is likewise computed in an interior sub-volume to sufficiently advect particles before they exit, and thus we may expect similar trends as reported here. For closed flows, particles may be released over the entire domain with particle density remaining mostly constant, which is more indicative of later iterations of the inflow analysis. Nevertheless, results on the use of block advection to compute FTLE for both open and closed flow examples are provided in Figs. 2.16 and 2.17, demonstrating that both applications significantly benefit from the block advection strategy since coherence between particle data and velocity data is rapidly lost in either scenario.

### 2.5.1 Particle Block Density

Particles are processed in bundles that lie in the same flow block. The characteristics of the distribution of particles in blocks evolves through the tracing procedure. At the beginning of the advection, the particles may lie in a few flow blocks, but continued advection will distribute the particles across many more. Hence the particles in a flow block are processed by one or more thread blocks to better balance computational load as the number of particles in each blocks begins to vary significantly.

Fig. 2.7 plots the number of active particles measured at the beginning of each of the 344 executions of Block Advection (BA, Alg. 6) for the abdominal aorta dataset. Recall that this block advection algorithm is repeatedly run until there are no more blocks with particles that have not reached the end of the current time
Figure 2.7: The total active particles plotted over the 25 time frames of the abdominal aorta dataset. Within each time frame, the number of active particles is plotted per execution of Block Advection, where each BA execution advects particles until they leave their current block or reach the end of the time frame. The number of BA executions required for each frame is displayed below each iteration plot.

frame of loaded velocity data. Then the blocked flow data from the next time frame is used to block-advect the particles.

The number of particles decreases with each new time frame due to outflow. Each BA execution advects particles until they either exit their current block or reach the end of the current time frame. Each of these iterations takes an amount of time proportional to the number of active particles times the number of integration steps, and this is not indicated along the horizontal axes. In the first few time frames, most of the particles remain in a few initial blocks. The iterations at later time frames reveal that most particles reach the end of the time frame, leaving a short tail of iterations needed for particle trajectories that have visited many different blocks.

Figure 2.8 plots the number of blocks that contain at least one active particle, again over the multiple BA executions within each of the 25 flow data time frames. The number of active blocks grows with each new time frame as the particles disperse across the flow domain. Within each time frame, the number of active blocks quickly drops to a short tail of a few blocks needed to process particle trajectories that have visited many different blocks.

We can measure the distribution as the average number of particles per block as

\[ A = \frac{\text{total active particles}}{\text{total active blocks}}. \]  

Fig. 2.9 plots \( A \) over the 344 BA executions within the 25 time frames of the aorta dataset. The particle block density drops as particles disperse across the flow domain. Initially particle block density is high since fewer blocks contain more particles, but later particle block densities fall to below 1,000 particles per block.
Figure 2.8: The total active blocks plotted over the 25 time frame of the abdominal aorta dataset. Within each time frame, the number of blocks still containing particles that have not exited reduces to zero. The BA execution counts are the same as in Fig. 2.7.

Figure 2.9: The average number of active particles per active blocks of the abdominal aorta dataset. At early time frames, few blocks contain many particles, whereas at later time frames many blocks contain few particles. Within a time frame, blocks also become less densely populated as block advection drives particles from their current block. The BA execution counts are the same as in Fig. 2.7.
by the 15th time frame.

### 2.5.2 Thread Assignment

An ideal implementation of the block advection algorithm would store the flow mesh data of a block in the local memory (shared memory or cache) of one streaming multiprocessor (SM) unit, and assign to it all of the particles in that block. However, current GPU architectures (including the K20) do not allow the user to assign a thread block to a specific SM. We compromise by mapping one or more thread block to advect particles in each spatial block, and letting the GPU scheduler assign the SM.

We set a threshold $A^*$ that we use to distinguish two kinds of blocked advection workloads. When $A \geq A^*$, we assume we have few flow blocks with many particles. We set the maximum thread block size to $A^*$ threads, and allow each thread to trace a maximum of

$$PT = \min([A/A^*], 16).$$  \hspace{1cm} (2.2)$$

particles. The upper bound $PT$ on the number of particles per thread is clamped to 16 to prevent uneven thread load, but varying this parameter within its order of magnitude did not significantly affect performance.

When $A < A^*$, we assume many flow blocks contain a few particles. We set the maximum thread block size to $[A]$ threads, and allow each thread to trace only a single particle, $PT = 1$. Hence, if $A < A^*$ then a flow block of $N$ particles is processed by $[N/A]$ thread blocks of single-particle threads. Otherwise $A \geq A^*$ and the flow block’s $N$ particles are processed by $P$ particles per thread organized into $[N/(PA^*)]$ thread blocks.

Figure 2.10 replots Fig. 2.9 on a logarithmic scale, revealing that low particle density occurs often at the
end of each time frame to flush out straggling particles that likely straddle the overlaps between flow blocks. For the aorta dataset, 71 of the 344 AB executions (about 20%) run with an average particles per block less than the $A^* = 256$ threshold.

Since $A^*$ is an upper bound of the number of threads per thread block, it should be both a multiple of 32 (the SM warp size) and evenly divide 512 (the maximum number of threads allowed based on our kernel’s register usage). Hence $A^*$ can be 32, 64, 128, 256 or 512, which subdivides the 48K of SM shared memory into sixteen 3K, eight 6K, four 12K, two 24K or one 48K piece. Since the 24K memory size was sufficient to hold the block velocity data for the lower aorta dataset, we selected $A^* = 256$.

### 2.5.3 Block Size

The size of blocks, especially their overlap widths, are set to localize computation while avoiding thrashing between neighboring blocks. As shown in Fig. 2.3, a few particles can still criss-cross the overlap region multiple times, but the minimum thread count setting from the previous discussion reduces the impact of such pathological cases. We have not observed them in practice on our sample datasets, so we focus our block size and overlap tuning on full thread count cases.

Figure 2.11 measures running times for various block widths, plotted over their overlap settings as a percentage of block width. The block widths are measured in centimeters for a $11.4 \times 8.24 \times 26.6$ cm flow dataset. The minimum occurs for an overlap of 30% (on each side of the block) of an inner block width of 0.15 cm (outer width of the block is thus 0.24 cm).

Figure 2.12 shows that the running times for the aorta data are almost completely dependent on the outer block width. Table 2.1 shows that nearly any combination of inner width and overlap percentage seem to optimize performance on the aorta dataset so long as the outer width is about 0.25 cm on the aorta dataset, and our experiments verify similar dependence on outer width for other datasets. This dependence on outer width suggests performance is dependent on the memory footprint of the block, since the amount of local flow data cells depends on the outer width. Table 2.2 compares the average cell volume with the optimal outer block volume for three datasets (aorta, TCPC, UV) and finds that the optimal outer block size hold a fairly consistent number of cells, roughly 125. Hence we conclude that the outer block width should be set to roughly the width of five average cells.

The convection cell analysis in Fig. 2.13 reveals a similar block overlap behavior as did the aorta data, except for its three smallest block sizes 0.03, 0.04 and 0.05. In this case, we simply use the inner block width of 0.03 with no overlap. The performance for inner block widths 0.04 and 0.05 drop rapidly for two reasons. First, the increase in overlap margin leads to a cubic increase in the number of cells in blocks, and second,
Figure 2.11: Running time (s) on the aorta dataset for various (inner) block widths (cm) plotted over various block overlap portions, as a percentage of block width, computed on the abdominal aorta dataset, with an average cell diameter of about 0.05 cm.
Figure 2.12: Running time (s) on the aorta dataset (with an average cell diameter of about 0.05 cm) for various (inner) block widths (cm) plotted over outer block width. Percentage overlaps from 0% to 60% in 5% increments plotted from left to right on each curve.
Figure 2.13: Running time (s) on the convection cell dataset for various (inner) block widths (cm) plotted over outer block width on the regular-grid gyre dataset, whose elements have a diameter of about 0.04 cm.
increasing the overlap margin reduces the number of integration stages a typical particle spends in a given block.

The pitching airfoil block size analysis in Fig. 2.14 exhibits a very different pattern, with execution time nearly linear with inner block size, outer block size and overlap ratio. Here the flow is mostly laminar with little turbulence, and like the previous convection cell case, larger overlaps tend to just deposit particles further inside their neighboring blocks. But the previous laminar case exhibited optimal runtime valleys that are absent in this airfoil case. Here the global memory limit does not allow block sizes small enough to reach an optimal size for this dataset. Furthermore, the mesh is highly anisotropic with edge lengths varying from 0.025 cm to 1.2 cm and a single constant block size does not perform uniformly well over these varying cell sizes. In such cases we can simply set the block size to the minimum, 0.15 in this case, with no overlap.
Table 2.1: Optimal overlap percentages (run on aorta dataset) for various block inner widths yield the same outer width

<table>
<thead>
<tr>
<th>inner width (cm)</th>
<th>overlap %</th>
<th>outer width (cm)</th>
<th>running time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.10</td>
<td>60%</td>
<td>0.22</td>
<td>249</td>
</tr>
<tr>
<td>0.15</td>
<td>30%</td>
<td>0.24</td>
<td>233</td>
</tr>
<tr>
<td>0.20</td>
<td>15%</td>
<td>0.26</td>
<td>244</td>
</tr>
<tr>
<td>0.25</td>
<td>0%</td>
<td>0.25</td>
<td>264</td>
</tr>
</tbody>
</table>

Table 2.2: Comparison of optimal block width settings on various datasets leads to a fairly consistent cells/block value.

2.5.4 Memory Allocation

The global flow data consists of geometric node data that record a spatial position and a time-varying sequence of flow velocities, and the topological data that connect nodes into a tetrahedral mesh. We order this data in memory clustered by blocks to facilitate cache coherence and coalesced memory access. A redundant copy of the node and cell data shared in the overlap region of neighboring blocks is stored in each of the overlapping blocks, which inflates the memory footprint of this representation.

We use the shared memory to locally store the cells rather than the cache, which might be overwritten if the GPU assigned two different flow block’s threads to the same SM. The Kepler architecture provides 64KB per SM that can be divided between shared memory and L1 cache. We assign each SM 48KB of shared memory and 16KB of L1 cache. We explicitly assign the flow data used by any flow block processed by the SM that fits in its shared memory limit. We have experimentally validated that the overhead of cache flow data degrades performance when we assign e.g. 48KB of L1 cache.

Figure 2.15 shows that our blocked tracing approach using the shared memory for memory coherence outperforms the GPU’s default use of the L1 cache to accelerate access to local flow blocks. Our explicit shared memory use is about 2\( \frac{1}{4} \) times faster than letting the GPU use it as L1 cache.

We estimate the amount of shared memory that each thread block can use as

\[
M = \text{total shared memory} \times \min(A, A^*) / 512
\]  

(2.3)

where 512 is the upper bound of number of threads per SM. The memory size \( M \) can be used to determine
Figure 2.15: Tracing time (s) for 8M particles over 10K RK4 integration steps divided among the aorta dataset’s 25 time frames, using 48KB of SM processing memory as explicitly-loaded shared memory vs. as an L1 cache for block advection.
if there is enough space to allocate shared memory, and if not, the thread block accesses the copy of its local mesh data held in global memory. This is less efficient but at least benefits from the global memory L1 cache. For reasonable particle block densities \( A \geq A^* \) we divide the shared memory between \( 512/A^* \) flow blocks. When \( A \ll A^* \), we have low particle densities and the benefits of a local copy of the unstructured mesh cells is not worth the memory copying expense. Equation 2.3 avoids this memory copy expense by setting its available memory for each of many flow blocks sharing an SM to a value likely lower than needed to contain the block’s flow data. In the worst case when particles are badly scattered, block advection effectively reverts to asynchronous advection.

### 2.5.5 Exiting Particle Transfer

The last step of Block Advection (Alg. 6) transfers any particles that have exited the current block to a neighboring block. This step requires finding the appropriate neighboring block. While the inner blocks are mutually exclusive, we have found particles can lie very near to block boundaries, and when cell faces also lie near inner block boundaries, numerical issues can interfere with finding the correct cell in the correct block. Hence we check to ensure the particle’s enclosing cell is properly included in the block’s list of cells. If not, we examine the neighboring blocks (up to seven in corner cases) and use the first block that includes the block’s containing cell.

When a particle is transferred from one block to a neighboring block, its enclosing cell will need a new local cell index relevant to the new flow block. We implement a GPU hash table algorithm to quickly return the local block addresses of a global cell.

At this stage (end of BA execution), to improve performance we sort the particles in order of active/nonactive status, then by new block number, then by integration stage. The active status sort key results in a compaction so we can limit computation to active particles. The block assignment key coalesces particle data by block for better cached memory coherence. The integration state reduces thread divergence in the integration kernel for any particles that switched blocks in the intermediate stages of Runge-Kutta integration. We implement this sort using a prefix-sum scan to collect active particles, followed by a radix sort on block ID and integration stage.

### 2.6 Results and Comparisons

We compare GPU implementations of synchronous advection, asynchronous advection and blocked advection. The synchronous advection implementation is CudaVC [4], a well optimized particle tracer implemented
to accelerate the investigation of Lagrangian coherent structures. Our asynchronous algorithm is a simple research prototype, though with the improved Runge-Kutta integration. The block advection implementation is analyzed and tuned as described in the previous section. Our asynchronous and blocked advection implementations are publicly available via github.com/linyufly/CUDATracer.

Our main goal for fast GPU particle advection through unstructured unsteady flows was to accelerate the computation of FTLE and thus the investigation of Lagrangian coherent structures. Figs. 2.16 and 2.17 demonstrate the use of blocked advection to accelerate the computation of FTLE and LCS. For Fig. 2.17, a portion of the aorta is seeded with 11M particles for 400ms at 100ms intervals across 51 pulsatile time frames that in total represent 952ms to generate the displayed FTLE sequence. Our code for these FTLE computations is also available online at github.com/linyufly/FastFTLE.

2.6.1 Register Allocation

Table 2.3 compares the register utilization between algorithm implementations. The stack frame used in asynchronous and block advection allows us to perform function calls that when inlined degraded performance. Our multi-stage Runge-Kutta implementation only needs a few registers. When compared to a full Runge-Kutta GPU implementation, the reduced register version sped up our blocked particle tracer by nearly a factor to two.
Figure 2.17: FTLE fields computed by advecting 11,127,296 particles over 4K timesteps on a different aorta dataset (4.4M tets, 784K nodes). FTLE computation time ranged from 159s to 189s for each of these ten cases via block advection, whereas asynchronous FTLE ranged from 664s to 768s and synchronous FTLE ranged from 946s to 1,112s.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Registers</th>
<th>Stack Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>73</td>
<td>0</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>83</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Block</td>
<td>113</td>
<td>32 bytes</td>
</tr>
</tbody>
</table>

Table 2.3: Comparison of register utilization between streaming advection algorithm implementations.
Figure 2.18: Comparison of synchronous, asynchronous and block advection running times (s) for tracing 8M particles over the aorta (400 integration steps/time frame), TCPC (500 integration steps/time frame) and UV (total 360 integration steps/timeframe).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>steps</th>
<th>Synch</th>
<th>Asynch</th>
<th>BA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aorta</td>
<td>10K</td>
<td>2,146</td>
<td>1,130</td>
<td>233</td>
</tr>
<tr>
<td>TCPC</td>
<td>50K</td>
<td>7,558</td>
<td>3,516</td>
<td>1,098</td>
</tr>
<tr>
<td>UV</td>
<td>18K</td>
<td>5,056</td>
<td>2,167</td>
<td>642</td>
</tr>
<tr>
<td>AR2</td>
<td>950</td>
<td>337</td>
<td>243</td>
<td>82.3</td>
</tr>
<tr>
<td>Gyre</td>
<td>10K</td>
<td>2,461</td>
<td>1,700</td>
<td>234</td>
</tr>
</tbody>
</table>

Table 2.4: Execution times (s) of synchronous, asynchronous and block advection for 8M particles over datasets.

### 2.6.2 Overall Performance

Figure 2.18 shows the execution time over each time frame of our three datasets of the synchronous, asynchronous and block advection implementations, and the total runtimes are compared in Table 2.4.

Figure 2.18 shows the execution time for the three advection approaches on the aorta, TCPC and UV datasets. As expected, the synchronous and asynchronous approaches drop in performance from their initial stages as particles disperse through the flow dataset, due to the associated loss in memory coherence. The block advection performance does not show such a decrease because the tuned blocks achieve the goal of maintaining local coherent memory access. All of these algorithms perform better toward the end due to outflow.

The performance on smoother exterior air flows is similarly improved by asynchronous blocked advection, but with less interesting characteristics. Performance of all three approaches on both the AR2 pitched wing flow dataset (block size 0.15, no margin) and the convection cell dataset (block size 0.30, no margin) remain constant when measured over the particles that have not yet exited the domain.

We can compare this performance to Garth & Joy [37] on the task of particle advection for tracing pathlines. Garth & Joy report a GPU version of their CellTree global location algorithm used to support real-time streamline advection to visualize their 23.6M element hexahedral “Fishtank” flow dataset. They advected 250K particles at about 35Hz, and 1M particles at about 10Hz, yielding advection rates ranging from 8.75M to 10M particles per second. The block advection rates shown in Table 2.4 range from 224M
(UV) to 367M (TCPC) particles per second (ignoring the lower number of time steps used in the AR2 example). Garth & Joy measured CellTree on a 1.5GHz 240-core NVIDIA 285GTX whereas our K20’s 2,496 cores run at 706 MHz, suggesting that the K20 runs five times faster than the 285 GTX (assuming full resource utilization). Hence the 300 M/s blocked advection rate on the K20 represents a 6× algorithm speedup over the 10M/s CellTree rate on the 285 GTX. (This does not include the additional factors that we used RK4 and double precision whereas they used RK3 and single precision.) Since CellTree is a global location method and the CFL condition advects particles about one element away, the local cell walking approach used by block advection aids such pathline tracing.

We can also compare to the performance of Bußler et al. [17], which traced particles interactively through time-varying flows (with time-varying meshes). They traced particles interactively and their Fig. 3 reported a runtime v. integration step, so their runtimes measure the time to move the entire particle population one integration step (\(\Delta t\)). Their “Gyro” dataset (unavailable to us) consists of 1.1M cells and so is comparable to the size of our aorta dataset. They report (visually in Fig. 5) a running time of about 23 ms for 1M particles for one integration step. Block advection traces 8M particles over 10k integration steps in 233 s which averages to just less than 3 ms per 1M particles per integration step. Their GPU (NVIDIA GTX 480) with 480 total cores running at 1.5GHz whereas our K20’s 2,496 cores run at 706 MHz, suggesting that the K20’s fivefold processors running at half speed should represent a GPU $2^{\frac{1}{2}} \times$ performance increase. (This does not include our use of rk4 and their use of rk3 integration.) We conclude that block advection on double precision data represents over a $4.2 \times$ performance improvement over Bußler et al. [17] on single precision data.

We can further compare our GPU performance to parallel CPU clusters, such as the Blue Gene/P (BG/P) Intrepid system at the Argonne Leadership Computing Facility used to compute FTLE for rectilinear time-varying flow fields using a similar but mutually exclusive (non-overlapping) domain decomposition [67]. This system consists of 1,024 850 MHz Powerpc-450 cores, which is 41% of the cores each clocked 20% faster than the K20, leading to a very coarsely approximate performance differential of $2 \times$ that ignores many details including precision and SIMD. Their best reported advection rate (using single-precision RK4 over four rectilinear datasets that did not require particle location ) was 13.5M particle time steps per second for the 8M element “plume” dataset (186M particles over 29 time steps in 400 seconds). Our best blocked advection rates range from 340M to 360M particle time steps per second, which includes unstructured mesh particle location but is limited to smaller domain sizes. Given these factors, blocked advection runs about $13 \times$ as does the non-overlapping blocking approach.
Table 2.5: Block advection (BA) v. asynchronous execution times (s) for different integration time steps for one second of simulation, including number of block advection calls and the average number of integration steps per ba call, measured on the aorta dataset.

<table>
<thead>
<tr>
<th>$\delta t$ (ms)</th>
<th>Steps</th>
<th>Ba</th>
<th>Asynch.</th>
<th>Speedup</th>
<th>BA Ex.</th>
<th>Steps/BA Ex.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1,000</td>
<td>36.7</td>
<td>142</td>
<td>3.86</td>
<td>319</td>
<td>3.13</td>
</tr>
<tr>
<td>0.9</td>
<td>1,112</td>
<td>39.6</td>
<td>156</td>
<td>3.95</td>
<td>320</td>
<td>3.47</td>
</tr>
<tr>
<td>0.8</td>
<td>1,250</td>
<td>42.9</td>
<td>174</td>
<td>4.06</td>
<td>323</td>
<td>3.87</td>
</tr>
<tr>
<td>0.7</td>
<td>1,429</td>
<td>47.4</td>
<td>197</td>
<td>4.16</td>
<td>328</td>
<td>4.36</td>
</tr>
<tr>
<td>0.6</td>
<td>1,667</td>
<td>53.2</td>
<td>228</td>
<td>4.28</td>
<td>331</td>
<td>5.04</td>
</tr>
<tr>
<td>0.5</td>
<td>2,000</td>
<td>61.3</td>
<td>271</td>
<td>4.42</td>
<td>331</td>
<td>6.04</td>
</tr>
<tr>
<td>0.4</td>
<td>2,500</td>
<td>73.4</td>
<td>335</td>
<td>4.57</td>
<td>335</td>
<td>7.46</td>
</tr>
<tr>
<td>0.3</td>
<td>3,334</td>
<td>93.2</td>
<td>442</td>
<td>4.74</td>
<td>338</td>
<td>9.86</td>
</tr>
<tr>
<td>0.2</td>
<td>5,000</td>
<td>133</td>
<td>655</td>
<td>4.94</td>
<td>344</td>
<td>14.5</td>
</tr>
<tr>
<td>0.1</td>
<td>10,000</td>
<td>248</td>
<td>1,289</td>
<td>5.19</td>
<td>349</td>
<td>28.7</td>
</tr>
</tbody>
</table>

Table 2.5 measures the performance of block tracing verses ordinary asynchronous over changes in the integration time step $\delta t$. While the running times of both BA and asynchronous appear to scale linearly with the number of steps (proportional to $1/\delta t$), the speedup of BA over asynchronous grows from just under 4x to over 5x as the time steps decrease. The number of BA executions grows slightly (10% as the time steps decrease to a tenth), the number of integration steps per ba execution grows linearly (at a 90% rate) with the number of integration steps. Hence the added costs of the blocked advection approach, including the shared memory loading of cells and the transfer of exiting particles, do not increase drastically as the integration time step decreases, suggesting that block advection can transfer GPU scalability directly to improved simulation precision.

2.7 Conclusion

We have established a new performance mark for tracing particles through unsteady unstructured flow datasets, in particular those with uniformly sized elements, and maintaining $CFL \approx 1$ as commonly employed for particle advection. We have shown that the performance of GPU particle tracing algorithms suffer as particles disperse across the flow domain, but these effects can be countered by block advection. We showed for example that the complex flows of most interest to LCS analysis should set the outer block width to span about five cells in each dimension, and the inner block width to about half the outer width. For more laminar flows, the benefit of the hysteresis provided by inner/outer block boundaries is not necessary and block size can be safely minimized to the smallest value available based on thread utilization.

We have also arefully analyzed block advection performance and provide advice on its tuning. For
example, we showed that the number of threads to assign in a GPU thread block should not be maximized as is usually recommended, because particles can be spread too thinly across the domain and there may not be enough particles in an advection block to fill the available threads in a thread processing block. We found the best performance on current hardware came from mapping two advection blocks to a single thread block, but this depends on a rather complex analysis of shared memory, register utilization and particle density as shown in Sec. 2.5.2.

We rely entirely on cell walking for particle location, which works well in most cases since the $CFL \approx 1$ condition prevents particles from traveling more than one or two cells per integration stage. In cases where larger time steps are used, methods such as the CellTree would perform better than our walking approach.

The algorithm works best on unstructured meshes whose elements are roughly uniformly sized. Adaptive meshes with wide variations in cell size, such as our airfoil example, can lead to blocks whose uniform size is too small, dominated by memory constraints needed to hold dense velocity nodes at the corners of small cells. Further research on this approach by varying its overlapping block size would better handle such adaptive meshes, but would require further analysis for its parameter tuning.

The algorithms require at least two velocity time-samples of the entire spatial unstructured mesh to be available in the GPU global memory. A clear direction for further research would add a level to the block memory hierarchy to include larger overlapping GPU-processing blocks that are paged in from the CPU. The overlapping regions would prevent virtual memory thrashing in the same way we already showed they prevent thread block thrashing, but would need to be similarly tuned to determine the best block and overlap size parameters.
Chapter 3

Watershed Ridges for Visualizing Lagrangian Coherent Structures

Lagrangian coherent structures provide insight into unsteady fluid flow, but their construction has posed many challenges. These structures can be characterized as ridges of a field, but their local definition utilizes an ambiguous eigenvector direction that can point in one of two directions, and its ambiguity can lead to noise and other problems. We overcome these issues with an application of a global ridge definition, applied using the hierarchical watershed transformation. We show results on a mathematical flow model and a simulated vascular flow dataset indicating the watershed method produces less noisy structures.

3.1 Introduction

The successful visualization of a large complex scientific dataset often relies on the ability to emphasize structure hidden within it. This is particularly true of flow datasets that in their most basic form contain a velocity vector at each point in space, essentially doubling the dimensionality of the dataset, which confounds an observer’s ability to perceive the data as a whole. Moreover, in unsteady flow applications, instantaneous rate of change information becomes less directly relevant to visualize, as the more salient flow information is contained by Lagrangian measures that intrinsically incorporate the integrated flow behavior.

A variety of analysis techniques can simplify a flow dataset by recognizing and displaying structures representing similar flow characteristics. The recent and compelling method of Lagrangian coherent structures [89, 69] reveals the boundaries of regions of shared characteristics for unsteady fluid flow.

Lagrangian coherent structures can be defined in a variety of different ways (e.g. [42, 90, 43]), but have been commonly visualized as the ridges of a scalar field, such as the finite-time Lyapunov exponent (FTLE), indicating the divergence of neighboring pathlines in a time-varying flow. Ridges are features typically derived from the second derivatives of the field, and so for common datasets are susceptible to noise and other issues [82].

A current commonly used approach to extract ridges from datasets are local, based on a marching cubes fit of the local ridge configuration in a cell from the FTLE data at the vertices. The local definition of a
ridge is often based on a matrix eigenvector, which only indicates the orientation of a line, but the ambiguity created by the fact that \( e \) and \(-e\) are both equally valid eigenvectors can lead to an orientation ambiguity when detecting the ridge surface numerically. This ambiguity can manifest as spurious false positives and other noise in the ridge surface extracted by local methods such as marching ridges often used for LCS extraction [82].

Watershed methods provide a global approach to extract topological structures from datasets. Sahner et al. [79] describe both a non-global “continuous” watershed approach that traces ridges as separatrices in the Morse structure, as well as a global “discrete” watershed transformation, and use the global watershed transformation to extract vortex and strain skeletal surfaces. We similarly propose and demonstrate watershed separatrix surface extraction for the visualization of flow structure, but for LCS instead of vortex/strain skeletal surfaces, and using a hierarchical watershed to filter out spurious details, to more clearly define the boundaries between neighboring watersheds as a global sea level rises.

These global watersheds can miss some spurious ridge features and can also produce small disjoint ridges due to noise and small field undulation. As stated in Sahner et al. [79], every watershed boundary corresponds to a height ridge or valley, but they do not necessarily coincide and furthermore ridges and valleys might exist that lack corresponding watershed boundaries.

This paper specializes the watershed approach for extracting ridges in FTLE data. We apply a region merging criteria similar to topological persistence that ranks ridges based on their configuration relative to neighboring ridges and valleys. This new filtering enabled by a global approach yields improved LCS extraction from scalar field data and better visualization of unsteady flow structure.

### 3.2 Lagrangian Coherent Structures

Let \( \mathbf{v}(\mathbf{x}, t) \) represent a time-varying velocity function. We denote the flow map \( \Phi(\mathbf{x}, t_0, T) \), which takes \( \mathbf{x} \) to its new position at time \( t_0 + T \) by integrating the velocity to trace the point along its trajectory

\[
\Phi(\mathbf{x}, t_0, T) = \mathbf{x} + \int_{t_0}^{T} \mathbf{v}(\Phi(\mathbf{x}, t_0, t), t)dt.
\]

The Cauchy-Green strain tensor is the positive definite matrix

\[
\mathbf{C}(\mathbf{x}, t_0, T) = \left[ \frac{\partial \Phi(\mathbf{x}, t_0, T)}{\partial \mathbf{x}} \right]^T \left[ \frac{\partial \Phi(\mathbf{x}, t_0, T)}{\partial \mathbf{x}} \right],
\]

(3.2)
and measures finite-time strain of infinitesimal line elements in the fluid. The maximum separation rate is achieved when $\Delta \mathbf{x}$ is parallel to the major eigenvector of $C(\mathbf{x}, t_0, T)$. The finite-time Lyapunov exponent (FTLE) measures this maximum separation rate as

$$f(\mathbf{x}) = \frac{\ln \lambda_{\text{max}}(C(\mathbf{x}, t_0, T))}{2|T|},$$

(3.3)

for points $\mathbf{x}$ at a given time $t_0$ over a given time interval $T$. Lagrangian coherent structures are often obtained as ridges of FTLE, but their specific definition relies on the particular definition of “ridge” that is used.

The height ridges of a scalar field $f$ are defined as the points satisfying

$$\frac{\partial f}{\partial \mathbf{e}_1} = 0$$

(3.4)

$$\frac{\partial^2 f}{\partial \mathbf{e}_1^2} < 0$$

(3.5)

where $f$ is a scalar function and $\mathbf{e}_1$ is either the minimum [26] or largest magnitude [55] eigenvector of the Hessian of $f$. The C-ridges of a scalar field $f$ are defined similarly, except $\mathbf{e}_1$ is the major eigenvector of the Cauchy-Green tensor (3.2) [83] based on “normally hyperbolic” LCS [43]. In this work we do not target hyperbolic LCS per se, but the more generic FTLE ridge.

Lagrangian coherent structures can be revealed by a continuation method that tracks the surface from one or more seed points placed at FTLE local maxima [83]. The surface grows from these seed points by integrating a tangent plane orthogonal to the major eigenvector of the Cauchy-Green tensor. While the algorithm is shown to be quite efficient, it required at least one seed point on every LCS component, and multiple seeds on the same component could lead to duplicated surfaces.

LCS can also be revealed through a marching ridges technique [82]. Marching ridges [34] is a variant of the marching cubes isosurface technique [57] used when the orientation needed to define the isosurface is inconsistently specified. An eigenvector $\mathbf{e}$ represents an axis, without preference of $\mathbf{e}$ or $-\mathbf{e}$. Ridge surfaces formulated from eigenvectors often must choose one of these two directions $\mathbf{e}$ or $-\mathbf{e}$ for each eigenvector $\mathbf{e}$. Marching ridges strives to consistently choose eigenvector directions to define an orientable isosurface, but can fail especially when sorted eigenvectors change their order across a single cell.

LCS can also be extracted as a subset of raw features [70] satisfying

$$\det (H^0 \mathbf{g}, \ldots | H^{n-1} \mathbf{g}) = 0.$$  

(3.6)

The matrix $H$ is the Hessian of the scalar function $f$, but can also be interchanged with the Cauchy-Green tensor $C$. 
tensor. Since (3.6) does not rely on eigenvectors, raw features can be extracted as an ordinary isosurface, e.g. using marching cubes, except where they may contain non-manifold self intersections. These self intersections can confound the use of raw features to find LCS, as can the numerical instability of (3.6).

All of these approaches rely on a local definition of ridges and LCS, which makes them susceptible to noise and other algorithm specific issues, such as surface duplication or non-orientability. A global approach would overcome these issues by defining ridges as region boundaries by growing the regions they bound instead of tracking the boundaries between regions.

### 3.3 Watershed Segmentation

In image processing, “watershed” methods have long been used for image segmentation [77]. These techniques outline the objects depicted in an image by finding ridges in the image pixel values. For LCS and FTLE ridge extraction, such global watershed methods can reduce the false positives and non-orientability of previous local approaches.

A variety of methods can be applied to a scalar field to separate it into watershed\(^1\) ridges and regions. A region can be defined as the points that flow to the same local minimum but this can be inefficient to compute.

It is more efficient to increment a sea-level threshold value from the global minimum value to the global maximum value. When this threshold value passes a local minimum, it creates a region that grows as the threshold increases. Neighboring regions grow into each other, identifying ridges where they meet.

The Vincent-Soille (V-S) algorithm [94] runs in linear time (proportional to the number of datapoints), and classifies all of the datapoints in an dataset as either ridge or region, labeling non-ridge datapoints by the region to which they belong. The V-S algorithm first bucket sorts the datapoints, then floods each bucket of datapoints in order from least to greatest. Ridges form when a datapoint in the current bucket has neighbors belonging to two regions, but the points in the bucket often form thick regions. Hence a (linear) distance transform is applied to the buckets to compute the distance from the nearest previously defined region (using a circular queue). In the computation of this distance transform, datapoints are assigned to their closest region. If a datapoint is not connected to a closest region, then it forms a new region. If a datapoint is equidistant to multiple regions, then it is classified as a ridge.

When applied to image segmentation, the watershed method typically oversegments, yielding many small regions. When used in LCS applications this leads to a distracting number of insignificant ridges due to

---

\(^1\)The term watershed comes from hydrology, where it denotes a drainage basin region. Some texts that apply it to dataset analysis incorrectly use it to refer to the ridges separating these basins, and call the basins “catchment basins.” To avoid confusion, we will refer to ridges that separate *regions.*
noise and subtle variation in the FTLE field, as shown in Figure 3.1. Hierarchical watershed methods merge similar regions to form progressively coarser segmentations and have been useful for discerning the important features in a dataset.

One method for constructing a watershed hierarchy is the waterfall transformation [12]. It constructs a graph consisting of nodes representing each ridge segment. The node’s value is set to the difference between the median values of the two regions its corresponding ridge separates. Each pair of these nodes is connected with an edge if their corresponding ridge segments border the same region. Then the next level higher in this hierarchical watershed is the watershed of these ridge nodes, using the average ridge value for each node.

Alternatively, regions can be merged based on similarities in their level and/or the characteristics of the ridge separating them. To provide better control for the application of LCS extraction, we utilized this region merging approach to filter unnecessary FTLE ridges.

Our criteria to define a criterion for merging neighboring regions resembles the notion of topological persistence [27]. The persistence of a topological feature indicates how robust it is to perturbation. A well-chosen perturbation in the dataset could remove a ridge, merging the regions it separated into a single region. From the Morse theory viewpoint, this perturbation would merge a saddlepoint with the minimum of one of the regions. The persistence of a ridge is thus the difference between the lowest point on the ridge (its saddlepoint) and the larger of its two neighboring minima.

We set a persistence threshold and merge regions separated by ridges that do not meet this threshold. We accelerate this merging with a union-find data structure.

This approach is similar to scale-space hierarchical methods that smooth datasets before performing
the watershed transform, using e.g. Gaussian smoothing [47]. Such techniques smooth the data with increasing filter widths to produce coarser levels of the watershed hierarchy. These smoothing operations merge neighboring regions because they cancel saddlepoint-minima pairs.

### 3.4 Polygonization

The implicit function theorem shows that the isosurface of a regular isovalue of an analytic field function is a manifold. However, the ridges arising from processing FTLE are not necessarily so, and can include non-manifold junctions that require special methods for surface extraction [44, 76]. The “crease surfaces” analysis [85] for example shows that ridge surfaces consist of manifold patches that meet at non-manifold junctions where the Hessian is degenerate.

We utilize a variation of marching cubes for polygonization of the ridge surfaces. The watershed transform labels each voxel value with a region, and ridges arise in cells whose eight corner vertices (where the voxel values are evaluated) lie in two or more disjoint regions. If a cell’s vertices lie in only two regions, we use ordinary marching cubes to polygonize the cell.

For cells that straddle three or more regions, we implement a variation of multiple material marching cubes [101]. For each of the six cell faces, we add a face center vertex and insert a pair of triangles to separate any edges whose vertices lie in separate regions. We then add a vertex at the cell center to connect these triangles. Figure 3.2 demonstrates the case where all eight cell corners belong to different regions. There are
two cases where a vertex at the voxel face center is not needed, as shown for the front face of each example in Figure 3.3.

Since the cell corners indicate only the region, and not a scalar value, the vertices used to polygonize a cell are inserted at the center of edges, faces and the cell. This leads to a blocky cuberille appearance of the resulting surface as shown in Figure 3.4. We remove these distracting visual artifacts through a smoothing process. We implemented a constrained Laplacian smoothing through conjugate gradient minimization of the energy functional

$$E(\{x_i\}) = \sum_i ||x_i - \bar{x}_i||^2 + \lambda ||x_i - x'_i||^2$$

(3.7)

where $\bar{x}_i$ is the centroid of the vertices neighboring vertex $x_i$, and $x'_i$ is its original position in the cuberille polygonization. The parameter $\lambda$ indicates how much the original position is respected. When $\lambda = 0$, the optimal solution is squeezing all the points at one point; When $\lambda \to \infty$, the mesh becomes blocky. Therefore, it is easy for the users to decide the best trade-off. In our experiments, we set $\lambda = 0.01$.

The non-manifold surfaces that arise from multiple region marching cubes require special care for proper smoothing. Branch points whose neighbors may represent ridges between several different regions can confound the smoothing process, as shown in Figure 3.5 (left).

For each vertex, we find the maximum number of faces that share one of its edges. We limit that vertices neighbors to the ones whose edge is shared by that maximum number of faces. This process smooths non-manifold junctions well, as shown in the example of two intersecting spheres shown in Figure 3.5 (right).
Figure 3.4: Blocky artifacts created from multiple region marching cubes for voxels that only indicate region number.

Figure 3.5: Ordinary Laplacian smoothing of non-manifold surfaces creates unsmooth results (left) which are fixed by limiting the neighborhoods used for Laplacian averaging (right).
This Laplacian smoothing approach differs from the one used for multiple material marching cubes (M3C) [101]. Our approach smooths vertices even when they are shared by more than two surfaces, whereas M3C smoothing leaves such vertices stationary. Our approach also does not require additional information that M3C uses, such as which materials are adjacent to a given vertex.

3.5 Results

We compared the watershed approach to marching ridges on two datasets. The first is an Arnold-Beltrami-Childress (ABC) flow, shown in Figure 3.6. The ABC flow dataset yields an FTLE field over a $201^3$ voxel array with values ranging from 0.0643 to 0.512.

Figure 3.7 compares the Lagrangian coherent structures extracted from the FTLE field of the ABC Flow dataset. The marching ridges example follows the recommended noise filtering steps [78], including (1) scalar thresholding (remove ridges with FTLE less than 0.3), (2) least eigenvalue thresholding (remove “flat” ridges with eigenvalue greater than -1.0), and (3) a threshold on the size of connected components (removing disjoint components with less than 500K vertices). The displayed denoised marching ridge result consists of one connected component of 677K vertices, but even with filtering, some noise persists.

The V-S watershed approach yields 797 watershed regions at the lowest level of the watershed hierarchy, which we merge by removing low persistence ridges to 103 regions. The resulting mesh, after smoothing, consists of 1.275M faces and 622K vertices. Figure 3.7 also shows some smoothed stairstep artifacts that reveal some issues with the merging of watershed regions as discussed further at the end of the section.

The second dataset we used to compare the watershed approach to marching ridges is the abdominal aortic aneurysm (AAA) dataset, shown as an FTLE field in Fig. 3.8. The AAA dataset is constructed from a pulsatile bloodflow simulation of a lower aorta, reconstructed as a 4.4M tetrahedral mesh. The FTLE field is a $206 \times 231 \times 261$ voxel array, ranging from 0 to 5.29812, using the value -1 indicates the outside of the aorta.

Figure 3.8 compares the Lagrangian coherent structures extracted from the FTLE field of the AAA dataset. The marching ridges example filtered out ridges smaller than 3.0 (which was the highest setting that prevented holes from forming in the main connected structures), set an eigenvalue threshold of zero (negative values did not improve the result) and filtered out all but the largest connected component. This yielded a mesh of 2.27M faces and 1.29M vertices. As before, the structure is evident but noise is clearly visible.

The V-S watershed algorithm yields 663 regions, which we merge into 199 regions when the height
Figure 3.6: The ABC flow, displayed as the FTLE field data (upper left) along with the embedded ridges extracted by the watershed method (upper right) and the watershed ridges themselves (lower center).
Figure 3.7: Lagrangian coherent structures extracted from FTLE of the ABC Flow dataset using the marching ridges method v. the watershed method.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>FTLE Resolution</th>
<th>Watershed</th>
<th>Mesh Vertices</th>
<th>Smoothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC Flow</td>
<td>201 × 201 × 201</td>
<td>34.21 s</td>
<td>1,225,558</td>
<td>36.4 s</td>
</tr>
<tr>
<td>Patient 96</td>
<td>206 × 231 × 261</td>
<td>50.87 s</td>
<td>1,800,230</td>
<td>61.4 s</td>
</tr>
</tbody>
</table>

Table 3.1: Performance of the watershed method for LCS extraction.

between neighboring regions differs by 0.05 or less. The resulting smoothed mesh consists of 1.80M faces and 854K faces.

The time required for the watershed transform, polygonization and smoothing for the ABC flow and the AAA data is shown in Table 3.1. The watershed method produces a voxel region classification on the FTLE field. The AAA FTLE field is 53% larger than that of the ABC flow, and its watershed transform takes 49% longer to compute. These watershed voxel regions polygonized to produce the number of vertices listed which follow similar proportions, but the time of the polygonization was not a significant portion of the total time and so is not listed. The smoothing time represents a total of 20 smoothing iterations, but takes 68% longer for the larger AAA data, because the polygonized ridges are more complex and their vertices have a greater number of neighbors.

One of the most exciting aspects of the global watershed approach is that the regions can be used to coherently color the ridge surfaces, as shown in Figure 3.9. The choice of color can be arbitrary, but is useful to differentiate the LCS surfaces from each other as they undulate through the flow domain. Such colorings are enabled by two-sided surface shading, but are unavailable for local ridge definitions (e.g. marching ridges).
Figure 3.8: Lagrangian coherent structures extracted from FTLE (top) of the AAA dataset using the marching ridges method (lower left) v. the watershed method (lower right).
Figure 3.9: Lagrangian coherent structures for the ABC flow and Patent 96, displayed using random colors assigned by regions, as extracted using our hierarchical watershed approach.

that lack identification of these regions.

The main drawback of the watershed approach is that the initial application of the watershed transformation (before merging) creates significant over-segmentation of the datasets, resulting in many small watersheds. These small watersheds are merged when separated by shallow ridges, but sometimes the shallowness, which we use as the persistence of the ridge, not properly eliminate some shallow ridges even though this persistence based approach works well for most spurious ridges such as are shown in Fig. 3.1.

As shown in Fig. 3.10, the ridges detected by the hierarchical watershed approach are largely at the mercy of the fluctuations of the FTLE field. In this example, such an FTLE fluctuation causes the hierarchical watershed merging to follow the wrong shorter ridge instead of keeping the correct longer ridge. This wrong shorter ridge is indeed part of the FTLE field and in fact forms a better defined ridge according to persistence than does the correct ridge, so further work beyond the persistence measure used by the hierarchical watershed method is needed to eliminate these last few pathological cases.

3.6 Conclusions and Further Research

While watershed methods commonly appear in the summaries of ridge extraction methods for Lagrangian coherent structures, they are often dismissed in favor of local methods such as marching ridges. We have
shown that their results are often much smoother and less noisy than such local approaches, and should be considered further.

One of the main shortcomings of the watershed approach is that it does not detect ridges that end at a minimum. Such an example might be a ridge descending from the rim to the bottom of a crater. We plan to address such cases with a combination of local and global combinations, using the local ridge definition evaluated on the current sea-level coastline front of the V-S watershed method. This combination of local and global ridge methods could yield the best of both worlds.

We have used a persistence measure as the criterion for hierarchical watershed method to merge watershed regions based on ridge shallowness. This criterion works well in many but not all cases, as illustrated in Figure 3.10. Further analysis and experimentation will be needed to explore new merging criteria to better preserve the important ridges for LCS visualization.

We also plan to work on high performance streaming implementations of the V-S and other watershed algorithms, updating earlier such work [77], as well as their applications to larger, out-of-core datasets.
Chapter 4

Trajectory Clustering Using Both Spatial and Directional Similarity

Trajectory similarity has been shown to be a powerful tool for visualizing and analyzing trajectories. In this paper we propose a novel measure of trajectory similarity using both spatial and directional information. The similarity is asymmetric, bounded within $[0, 1]$, affine-invariant, and efficiently computed. Asymmetric mappings between a pair of trajectories can be derived from this similarity. Experimental results demonstrate that the measure is better than existing measures in both similarity scores and trajectory mappings. The measure also inspires a simple similarity-based clustering method for effectively visualizing a large number of trajectories, which outperforms the state-of-the-art model-based clustering method (VFKM).

4.1 Introduction

Trajectories, frequently seen in geo-spatial data, are typically represented as polylines, where points may have domain-specific quantities. For visualization purpose, however, trajectory similarity only depends on the spatial and directional information, which is our focus. Sung et al. [92] pointed out that the distance (or similarity) between trajectories is not as clear as the distance between points or lines, and the majority of existing measures fail to capture the human intuition of similarity. We think it is because the spatial and directional information have not been used to full capacity. Most existing similarity measures consider only spatial information [93, 80, 95, 96], while a few consider only directional information [71, 6]. Some works on trajectory similarity approximation [3, 56] compute global statistics of the directional information separately from the spatial information. A good combination of both information is therefore necessary and urgent in the design of trajectory similarity, as it is the central problem in trajectory analysis [80].

The most common ways of measuring the similarity between two trajectories are based on finding a correspondence between their vertices. Let $A, B$ be two trajectories each consisting of a sequence of polyline vertices, e.g. $\{a_i\}, \{b_j\}$. The trajectories $A, B$ need not be the same length nor contain the same number of vertices. A total correspondence between $A$ and $B$ would be a set of vertex pairs $(a_i, b_j)$ such that every vertex in $A$ and every vertex in $B$ appears in at least one vertex pair of the correspondence set, whereas a
1,792 hurricane trajectories from the Atlantic hurricane reanalysis project (HURDAT) [51]. Each trajectory travels from orange to blue as it cools.

Figure 4.1: We present a new trajectory similarity measure that combines spatial and directional metrics to provide more meaningful clustering of trajectory data. This new similarity measure separates the trajectories of the HURDAT Atlantic hurricane database into seven meaningful categories: (a) long duration hurricanes that originate off the coast of Africa but remain in the Atlantic, (b) hurricanes that originate near the Caribbean and move up the Eastern US coast, (c) hurricanes originating off the coast of Africa that reach the North American East coast, (d) short lived hurricanes originating off the coast of Africa, (e) Gulf hurricanes that reach Mexico, (f) Gulf hurricanes that reach the US, and (g) Central American hurricanes. Other trajectory metrics yield clusters with significant overlap and less clear categories, as compared later in the paper.

**unique correspondence** requires instead that each vertex in $A$ or $B$ appear in no more than one vertex pair of the correspondence set.

These trajectory similarity measures are based on optimizing these correspondences according to some objective function. For example, the Fréchet distance (FD) finds the total correspondence that minimizes the farthest distance between corresponding vertices, and reports that smallest farthest distance as the distance between the two trajectories. Similarly, the normalized dynamic time warping (DTW) minimizes over all correspondences the average distance between corresponding vertices, and reports that smallest average distance as the distance between the two trajectories. These FD and DTW trajectory distances can be inverted to produce a similarity measure in the range $[0, 1]$.

Other similarity measures restrict this optimization to only a “proximate” subset of these total vertex-vertex correspondences, where the distance between corresponding vertices is within some threshold. For example, the longest common subsequence (LCSS) reports the largest number of proximate vertex-vertex pairs in a unique correspondence set, whereas the (uniform-cost) edit distance (ED) [93] reports the largest sum of the size of the unique correspondence set and the number of its proximate vertex pairs. Both LCSS and ED can be influenced by the number of vertices in the two trajectories, but can be normalized to reduce this influence by dividing LCSS by the number of vertices in $A$ or $B$, whichever is smaller, and by dividing
ED by twice the size of the unique correspondence set. This results in a normalized LCSS or ED similarity measure in the range $[0,1]$ which can be inverted if used as a distance metric.

A reported drawback of FD and DTW is that both can yield large distances between similar appearing trajectories if the trajectories have outliers, and the distance between these outlier vertices can dominate the FD/DTW trajectory distance [80]. Furthermore, because FD and DTW are based on spatial distances between vertices, a pair of small, overlapping but different trajectories can be reported as more similar than a pair of larger trajectories that are relatively much closer, as shown in Fig. 4.2. The LCSS and ED metrics address this issue partially with a global distance threshold, which filters out the large outlier distance effects of FD and DTW, but does not address the sliding scale relative comparisons needed to distinguish spatially large trajectories from spatially smaller trajectories.

![Figure 4.2: A pair of dissimilar trajectories A and B, and a pair of similar trajectories C and D. In the view of both FD and DTW, the distance between A and B is shorter than the distance between C and D.](image)

We propose a similarity measure between trajectories using both the spatial and directional information, with which a symmetric similarity is easily computed as an $F_1$ score [99]. Unlike distance measures, our similarity is a number within $[0,1]$, and it is affine-invariant, i.e., for any affine transformation $F$ and trajectories $A$ and $B$, the similarity between $A$ and $B$ is equal to the similarity between $F(A)$ and $F(B)$. We also provide a dynamic programming method in quadratic time.

Section 4.2 reviews previous work relevant to trajectory similarity. Section 4.3 defines the point similarity. Section 4.4 introduces the trajectory similarity. Section 4.5 evaluates the performance of our similarity measure. Section 4.5.2 shows an application on trajectory clustering using our similarity. Section 4.6 concludes with a discussion of limitations.

### 4.2 Previous Work

Let $A = (a_i)_{i=0}^{n-1}$ and $B = (b_j)_{j=0}^{m-1}$ be two trajectories. Let $M$ be the set of all total pairings of trajectory vertices $a_i, b_j$, such that every $M \in M$ represents a total pairing such that $(a_i, b_j) \in M$ for every $a_i \in A$ and independently for every $b_j \in B$. Furthermore, the total pairing satisfies an ordering condition such that
if \((a_i, b_j) \in M\) then \((a_k, b_\ell) \not\in M\) if \(k < i\) and \(\ell > j\) or if \(k > i\) and \(\ell < j\).

The most popular distance metrics on trajectories are Frechet distance (FD) [28, 38] and dynamic time warping (DTW) [11, 64], and the latter is believed to be the best [74]. FD minimizes the longest distance whereas DTW minimizes the overall distance,

\[
\begin{align*}
\text{FD}(A, B) &= \min_{M \in M} \max_{(a, b) \in M} ||a - b|| \\ 
\text{DTW}(A, B) &= \min_{M \in M} \sum_{(a, b) \in M} ||a - b||
\end{align*}
\]

The number of pairs in the matching affects DTW magnitude, so the normalized DTW reports the average distance per vertex pairing,

\[
\hat{\text{DTW}}(A, B) = \min_{M \in M} \frac{\sum_{(a, b) \in M} ||a - b||}{|M|}
\]

Both FD and DTW can be efficiently computed by dynamic programming in \(O(nm)\), and Agarwal et al. [1] even proposed a sub-quadratic algorithm for FD. The computation of \(\hat{\text{DTW}}\), however, takes \(O(nm(n + m))\), thus is often approximated by dividing FD by the number of pairs in the optimal matching [75].

Other similarity measures focus only on proximate pairs of vertices in the trajectories. Let \(M_\varepsilon\) represent the set of all proximate pairings such that each \(M_\varepsilon \in M_\varepsilon\) consists only of pairings \((a_i, b_j)\) such that \(||a_i - b_j|| \leq \varepsilon\). Furthermore \(M_\varepsilon\) satisfies the aforementioned ordering condition, but unlike total pairings, each \(M_\varepsilon\) also must satisfy a vertex uniqueness condition such that each vertex of \(A\) and \(B\) can only appear in at most one pairing in \(M_\varepsilon\).

We can now define two similarity measures: the longest common subsequence (LCSS) [96] and the edit distance (ED) [61]. The LCSS maximizes the number of proximate pairings,

\[
\text{LCSS}(A, B) = \max_{M_\varepsilon \in M_\varepsilon} |M_\varepsilon|,
\]

and can be normalized as

\[
\text{LCSS}(A, B) = \frac{\text{LCSS}(A, B)}{\min(n, m)}
\]

since the vertex uniqueness condition implies \(|M_\varepsilon| < \min(n, m)\) for any \(M_\varepsilon \in M_\varepsilon\).

The edit distance combines total pairings with proximate pairings. It has several variants, e.g. real penalty (ERP) [21] or real sequence (EDR) [22], but we focus on the most common version, the uniform-cost EDR, as

\[
\text{ED}(A, B) = \min_{M \in M} \min_{M_\varepsilon \subseteq M} (n + m - |M| - |M_\varepsilon|),
\]
and is typically normalized by the length of the total pairing \[61\],

\[
\hat{\text{ED}}(A, B) = \min_{M \in \mathcal{M}} \min_{M_e \in \mathcal{M}_e} \frac{n + m - |M| - |M_e|}{n + m - |M|}. \tag{4.7}
\]

LCSS, \(\hat{\text{LCSS}}\), and \(\hat{\text{ED}}\) can be computed by similar dynamic programming approaches in \(O(nm)\). The fastest approach for computing \(\hat{\text{ED}}\) is \(O(nm \min(n, m))\) in uniform-cost cases \([8, 9]\). In addition to the incompatibility of multiple matching, \(\hat{\text{LCSS}}\) and \(\hat{\text{ED}}\) have a common fatal drawback that they are divergent even with uniform sampling on both trajectories, as demonstrated in Fig. 4.3.

![Figure 4.3: Two parallel straight trajectories, A being roughly half the length of B, with \(\epsilon\) equal to the distance between them. When A has the same sampling rate as B, as shown in (a), \(\hat{\text{LCSS}}(A, B) = 1\) and \(\hat{\text{ED}}(A, B) \approx 0.5\). When A’s sampling rate is a large multiple of B’s sampling rate, \(\hat{\text{LCSS}}(A, B) \approx 0.5\) and \(\hat{\text{ED}}(A, B) \approx 1\).](image)

Sankararaman et al. \([80]\) decomposed the total pairing \(M\) used in FD and DTW into a pair of monotonic injective mappings \(\alpha : A \rightarrow B\) and \(\beta : B \rightarrow A\). They also eliminated distant pairings similar to LCSS and ED, but did not provide a normalized version.

Other approaches compare two trajectories without finding a fully optimized matching. Agrawal et al. \([2]\), Vlachos et al. \([95]\) and Cai et al. \([18]\) used alternative trajectory representations and indexes to estimate distances. Buchin et al. \([13]\) considered a pair of trajectories as time shifted versions of the same function, and solved a semi-optimization problem to find the time shift. Andreopoulos et al. \([6]\) rasterized the trajectories and into pixel sequences, matching trajectory parts that map to the same pixel. Pelekis et al. \([71]\) proposed a similarity measure based on the weighted sum of polygonal areas between a pair of trajectories. Elnekave et al. \([29]\) subdivided spatial-temporal trajectories into a list of minimal bounding boxes (MBBs) similar to the work of Anagnostopoulos et al. \([5]\), and then defined the similarity between trajectories as the sum of similarities between MBBs. Piciarelli et al. \([72]\) evenly subsampled every trajectory into 16 two-dimensional vertices, and then represented each trajectory as a 32-dimensional vector, using vector metrics as trajectory metrics.
For trajectory datasets whose spatial information is not important, directional similarity has been proposed to focus on the translation-invariant or rotation-invariant features of trajectories. For instance, Andreopoulos et al. [6] studied protein unfolding pathways by transforming trajectories into the derivative and curvature spaces, and Peleakis et al. [71] used the angle difference between points for computing the similarity score, in order to classify routes by their directions. For datasets whose spatial and directional information are both important, previous work used global statistics of the directional information as a complementary hint to the spatial information. For examples, Liu et al. [56] computed the displacement $s$ of trajectories (the difference vector between the origin and the destination), and added the term $\text{avg} (\|s_A\|, \|s_B\|) \cdot \cos (s_A, s_B)$ to the distance measure; Al-Serafi et al. [3] counted the number of occurrences in eight directions (N, NE, E, SE, S, SW, W, NW) forming an eight-dimensional vector for each trajectory, and used the difference between the vectors in their multi-dimensional trajectory distance measure (MTDM).

Trajectory clustering is a direct application of trajectory similarity [29, 72], equipped with well studied clustering techniques, e.g. $k$-means [59] and spectral clustering [66]. According to Ferreira et al. [30], the major drawback of the similarity-based approaches, however, is the difficulty of defining a good similarity, thus they often result in either undesired features or missing important components. A competitive alternative approach is fitting a generative model over trajectories and clustering them by model parameters. For instance, Wei et al. [97] fit polynomials by the expectation maximization (EM) algorithm to categorize large data from scientific simulations; Ferreira et al. [30] fit velocity fields by the conjugate gradient (CG) method to outperform TRACLUS [53], a partition-and-group framework based on the similarity between trajectory segments, on the Atlantic hurricane data. Their method is to our knowledge the state-of-the-art in trajectory clustering.

### 4.3 Point Similarity

We expand the representation of a trajectory as a sequence of positions and directions, e.g.

$$A = (\ldots, (a_i, \hat{a}_i), \ldots),$$

where $\hat{a}_i$ is a unit direction vector (assuming an arc length parameterization). Similarly

$$B = (\ldots, (b_j, \hat{b}_j), \ldots).$$
We can now incorporate the similarity of trajectory vertex directions in addition to positions. We define

\[\angle(\dot{a}, \dot{b}) = \frac{\dot{a} \cdot \dot{b} + 1}{2}\]  

which is one when \(\dot{a}\) and \(\dot{b}\) are parallel and zero when antiparallel, and can be computed efficiently.

For spatial similarity, we similarly convert the distance between the two points into a measure within \([0, 1]\). We use the Gaussian function

\[g(x, \sigma) = \exp(-x^2/\sigma^2),\]  

where \(x\) is the distance \(|a - b|\) and \(\sigma\) is the standard deviation, which represents a “soft” distance threshold. Unlike LCSS and ED which require a fixed distance threshold, we will set this soft distance threshold relative to the trajectory size.

We can estimate the size of a trajectory in a variety of ways, including its arc length, the area of its bounding box or convex hull, its diameter (distance between farthest two points), or many other methods. We selected the radius of its minimum enclosing circle, which is rotationally invariant and can be efficiently computed in linear time \([62]\) (which is faster than the diameter that uses the convex hull for optimal computation). Therefore, we use

\[\sigma(A) = r(A)10^\alpha,\]  

where \(r(A)\) is the radius of the minimum enclosing circle of trajectory \(A\), and \(\alpha\) provides user control over the “feature size” sensitivity of the relative spatial distance between a pair of trajectories.

Our measure of trajectory point spatial similarity becomes

\[\phi(a, b, A) = g(||a - b||, \sigma(A)),\]  

where \(a \in A\). Fig. 4.4 shows an example of this asymmetry of Eqn. 4.11.

![Figure 4.4](image-url)
Given the symmetric directional similarity and the asymmetric spatial similarity, our asymmetric point similarity between the $i$-th point of trajectory $A$ to the $j$-th point of trajectory $B$ is defined as their product

$$\text{aps}_{AB}(i, j) = \angle(\dot{a}_i, \dot{b}_j)\phi(a_i, b_j, A). \quad (4.12)$$

Eqn. 4.12 ensures that a pair of points has to be similar both directionally and spatially for a high score of the point similarity. It also bounds the similarity between any two points within $[0, 1]$, so that no pair can dominate the trajectory similarity.

### 4.4 Trajectory Similarity

#### 4.4.1 Asymmetric Trajectory Similarity

Consider two trajectories of points $A = (a_0, \ldots, a_i, \ldots, a_{n-1})$ and $B = (b_0, \ldots, b_j, \ldots, b_{m-1})$ ignoring the tangent directions of the points for the moment. We assume these trajectories have been resampled uniformly, such that the distance between neighboring trajectory vertices $||a_i - a_{i+1}||$ is approximately constant. We denote the this approximate distance between vertices of $A$ as $\Delta A$ and refer to it as the “step size.” We similarly denote the step size of a uniformly sampled $B$ as $\Delta B$, and even though both may be uniformly sampled, $\Delta B$ need not be equal to $\Delta A$.

Let $I = (0, \ldots, n-1)$ and $J = (0, \ldots, m-1)$ be the index sets of $A$ and $B$. We define a mapping $p : I \rightarrow J$ that satisfies weak monoticity, $i_1 < i_2 \implies p(i_1) \leq p(i_2)$. We can then define an objective quality function of the index map $p$,

$$\text{qual}(p) = \sum_{i \in I} \text{aps}_{AB}(i, p(i)) \frac{1}{|A|}, \quad (4.13)$$

and by definition, $\text{qual}(p) \in [0, 1]$.

We define the asymmetric trajectory similarity as the highest possible quality of a mapping from $A$ to $B$,

$$\text{ats}(A, B) = \max_p \text{qual}(p). \quad (4.14)$$

In (4.13), $|A|$ is constant, thus (4.14) can be solved by a simple greedy algorithm using dynamic programming. Let $A^i$ denote the prefix $(a_0, \ldots, a_i)$ of trajectory $A$.

We denote $\text{OPT}(i, j)$ as the optimal numerator of (4.13), replacing $A$ and $B$ with prefixes $A^i$ and $B^j$,

$$\text{ats}(A, B) = \frac{\text{OPT}(|A| - 1, |B| - 1)}{|A|}. \quad (4.15)$$
We can thus build up a definition of $\text{OPT}(i,j)$ recursively as

\[
\begin{align*}
\text{OPT}(0,0) &= \text{aps}_{AB}(0,0) \\
\text{OPT}(i,0) &= \text{aps}_{AB}(i,0) + \text{OPT}(i-1,0) \\
\text{OPT}(0,j) &= \max(\text{aps}_{AB}(0,j), \text{OPT}(0,j-1)) \\
\text{OPT}(i,j) &= \max(\text{aps}_{AB}(i,j) + \text{OPT}(i-1,j), \text{OPT}(i,j-1)) 
\end{align*}
\]

which leads to Alg. 7 with time complexity $O(nm)$.

**Algorithm 7: Dynamic Programming for Asymmetric Trajectory Similarity**

- **input**: Trajectories $A$ and $B$.
- **output**: $\text{ats}(A,B)$ and $p^*$.

create $\text{OPT}[0 \ldots |A| - 1, 0 \ldots |B| - 1]$;
create $\text{trace}[0 \ldots |A| - 1, 0 \ldots |B| - 1]$;
create $p^*[0 \ldots |A| - 1]$;

for $i$ from 0 to $|A| - 1$ do
  for $j$ from 0 to $|B| - 1$ do
    if $i = 0$ and $j = 0$ then
      $\text{OPT}[0,0] \leftarrow \text{aps}_{AB}(0,0)$;
      $\text{trace}[0,0] \leftarrow 0$;
    else
      $\text{val} \leftarrow -\infty$;
      $\text{tra} \leftarrow \text{none}$;
      if $i > 0$ then
        $\text{val} \leftarrow \text{OPT}[i-1,j] + \text{aps}_{AB}(i,j)$;
        $\text{tra} \leftarrow j$;
      if $j > 0$ and $\text{OPT}[i,j-1] > \text{val}$ then
        $\text{val} \leftarrow \text{OPT}[i,j-1]$;
        $\text{tra} \leftarrow \text{trace}[i,j-1]$;
      $\text{OPT}[i,j] \leftarrow \text{val}$;
      $\text{trace}[i,j] \leftarrow \text{tra}$;
    $j \leftarrow |B| - 1$;
  for $i$ from $|A| - 1$ downto 0 do
    $j \leftarrow \text{trace}[i,j]$;
    $p^*[i] \leftarrow j$;
  $\text{ats}(A,B) \leftarrow \frac{\text{OPT}(|A|-1,|B|-1)}{|A|/|B|}$;

**4.4.2 Continuous Form**

The continuous form of a trajectory $A$ is defined by a continuous function $a : [0,1] \to A$ such that $a(t)$ generates points on the trajectory $A$ for values of $t$ from zero to one. In continuous form, a trivial matching between two trajectories $A$ and $B$ is formed by the pairings of $a(t)$ and $b(t)$ for each value of $t \in [0,1]$. Other total matchings are defined for this continuous form through the use of a weak-monotonic surjective pairing
function \( p : [0, 1] \rightarrow [0, 1] \), which creates a pairing between \( a(t) \) and \( b(p(t)) \).

Among previous trajectory matching measures, FD is the only one supporting a continuous form [7],

\[
\tilde{FD}(A, B) = \inf_p \max_{t \in [0, 1]} ||a(t) - b(p(t))||.
\]

(4.16)

Other metrics, e.g. DTW, LCSS or ED do not support a continuous form. For example, normalized DTW (4.3) would yield as a continuous form

\[
\tilde{\text{DTW}}(A, B) = \inf_p \int_0^1 ||a(t) - b(p(t))|| \, dt.
\]

(4.17)

The problem is that the minimum integral is trivial, equal to the shortest distance between two trajectories, as is shown in Fig. 4.5. LCSS and ED have the same problem. Without a well defined continuous form, DTW is not guaranteed to converge although it seems so. LCSS and ED have been proven to be divergent in Section 4.2.

Our asymmetric trajectory similarity has a valid definition of the continuous form,

\[
\tilde{\text{ats}}(A, B) = \sup_p \int_0^1 \tilde{\text{aps}}_{AB}(t, p(t)) \, dt
\]

(4.18)

where \( \tilde{\text{aps}}_{AB}(\cdot, \cdot) \) is the continuous form of (4.12). As the number of sampled points on both trajectories
increases, \( \text{ats}(A, B) \) converges to \( \tilde{\text{ats}}(A, B) \).

### 4.4.3 Oversqueezing

The total matching used in FD and DTW allows a point on \( A \) to match to multiple points on \( B \). Similarly, \( \text{ats}(A, B) \), according to the weak monotonicity of its matching, can match multiple points on \( A \) to the same point on \( B \), which we call squeezing. Squeezing is necessary when \( A \) has more points than \( B \) on correspondent parts, as the example in Fig. 4.6(c). Another cause of squeezing is the low directional similarities between points on the two trajectories, which leads to low asymmetric point similarities, as shown in Fig. 4.6(d) with an extremal case of two opposite straight trajectories. Squeezing usually indicates low similarity, especially when the neighbours of the point on \( B \) are not mapped from \( A \) by the optimal pairing \( p \).

---

**Figure 4.6:** (a) and (b) show the same pair of trajectories \( A \) and \( B \). (a) shows the optimal mapping \( p^* \) of \( \text{ats}(A, B) \) using a proper \( \alpha \), and (b) shows the optimal mapping \( p^* \) using a too large \( \alpha \), which is undesirably squeezed. (c) shows the necessary squeezes for two arc-ish trajectories \( A \) and \( B \). (d) shows the optimal mapping \( p^* \) of two opposite straight trajectories \( A \) and \( B \). The squeezed mapping indicates their low directional similarity.
If \( \alpha \) in (4.10) is set too large, for a close pair of points, the spatial similarity is almost always one, thus the directional similarity dominates the point similarity. In the case of Fig. 4.6 (b), the direction of the central point of \( B \) is horizontal, thus all the points on \( A \) are mapped to it. We refer to this undesirable squeeze caused by setting \( \alpha \) too large as “oversqueezing.” When \( \alpha \) is set small enough that the spatial similarity makes a difference, a desirable mapping in Fig. 4.6 (a) can be achieved. On the other hand, if \( \alpha \) is set too small, the spatial similarity can dominate the point similarity, ignoring directional similarity. Ideally, the value of \( \alpha \) should be large enough to capture common directional features between two trajectories, which differentiates our approach from previous work.

We further reduce oversqueezing by imposing a penalty on the mapping \( p \). If the distance along trajectory \( B \) from \( b_{p(i)} \) to \( b_{p(i+1)} \) is shorter than \( \Delta A \), the step size of trajectory \( A \), both \( \text{aps}_{AB}(i, p(i)) \) and \( \text{aps}_{AB}(i + 1, p(i + 1)) \) are multiplied by a damping factor \( \beta \).

We update the definition of the discrete mapping quality (4.13) with a damping term

\[
\text{qual}(p) = \frac{\sum_{i \in I} \text{aps}_{AB}(i, p(i)) \text{damp}_p(i)}{|A|} \tag{4.19}
\]

where

\[
\text{damp}_p(i) = \begin{cases} 
\beta & \text{if } (p(i) - p(i-1)) \Delta_B < \Delta_A \\
\text{or } (p(i+1) - p(i)) \Delta_B < \Delta_A \\
1 & \text{otherwise} 
\end{cases} \tag{4.20}
\]

In the continuous form, this penalty is applied everywhere \( dp/dt < 1 \), leading to the updated definition

\[
\hat{\text{ats}}(A,B) = \sup_p \int_0^1 \hat{\text{aps}}_{AB}(t, p(t)) \hat{\text{damp}}_p(t) \, dt \tag{4.21}
\]

where

\[
\hat{\text{damp}}_p(t) = \begin{cases} 
\beta & \text{if } dp/dt < 1 \\
1 & \text{otherwise} 
\end{cases} \tag{4.22}
\]

Fig. 4.7 shows two hurricane trajectories \( A \) and \( B \) over the North Atlantic Ocean, which are from the Atlantic hurricane reanalysis project (HURDAT) [51]. For the computation of \( \text{ats}(A,B) \), we first set \( \alpha = 0.5 \). As is shown in Fig. 4.7 (a), the optimal mapping has several squeezes. The squeeze in the beginning of \( A \) indicates the low similarity (both spatial and directional) between the beginning of \( A \) and the beginning of \( B \), which makes the starting point of \( B \) the best match for all the points at the beginning of \( A \). All the other big squeezes, however, are undesirable oversqueezing due to the relatively large value of \( \alpha \). It can be observed that the mapped point on \( B \) of every oversqueeze is the most similar to those squeezed points.
among its neighborhood. Large $\alpha$ drives the squeezed points to favor the directional similarity more than the spatial similarity. When we set $\alpha = 0$, the importance of spatial similarity is increased, thus alleviates the oversqueezing as in Fig. 4.7 (b). Fig. 4.7 (c) shows the result when we keep $\alpha = 0.0$ and adopt the damping factor $\beta = 0.5$. The damping factor drives the points on $A$ to have more interactions with their neighbours. As a consequence, all the oversqueezings are removed, while the big squeeze from the beginning of $A$ to the point $a$ is kept. $a$ serves as a separatrix, every point after which can be nicely mapped to a unique point on $B$. Because of the damping factor in addition to the order restriction, quite a few points on $A$ cannot be mapped to the most similar point on $B$, but they achieve the best mapping as a whole.

![Figure 4.7: Two hurricane trajectories $A$ and $B$ from HURDAT. (a) shows the optimal mapping of ats ($A$, $B$) using $\alpha = 0.5$ without damping, where many undesirable oversqueezing occur. When $\alpha = 0$, the points on $A$ tend to favor the near points over the points with similar directions, thus the oversqueezing is reduced in (b). (c) shows the case where we keep $\alpha = 0$ and use the damping factor $\beta = 0.5$. All the undesirable oversqueezes are removed, and the only remaining squeeze in the beginning of $A$ reveals that the subtrajectory from $a$ to the end of $A$ can be mapped to $B$ nicely while the beginning of $A$ cannot.]

### 4.4.4 Dynamic Programming

The incorporation of damping to reduce squeezing affects the optimization of the mapping quality. We thus augment the objective function $\text{qual}()$ with an additional boolean damping parameter. We redefine

$$\text{OPT}(i, j) = \max(\text{OPT}(i, j, true), \text{OPT}(i, j, false)),$$

where the third parameter of $\text{OPT}$ indicates whether or not the damping term $\beta$ has been applied.

The computation of $\text{OPT}(0, 0, true)$ and $\text{OPT}(0, 0, false)$ is trivial. For the case of $i > 0$ and $j > 0$,
OPT\((i,j,\text{true})\) and OPT\((i,j,\text{false})\) are solved differently by

\[
\begin{align*}
\text{OPT} (i, j, \text{false}) &= \max_{k=0}^{j - \lceil \frac{\Delta A}{\Delta B} \rceil} \text{OPT} (i - 1, k) + \text{aps}_{AB} (i, j) \quad (4.24) \\
\text{OPT} (i, j, \text{true}) &= \max_{k=0}^{j - \lceil \frac{\Delta A}{\Delta B} \rceil} \text{OPT} (i - 1, k), \\
&\quad \max_{k=j-\lceil \frac{\Delta A}{\Delta B} \rceil+1}^{\max} \text{OPT} (i - 1, k, \text{true}) \\
&\quad + \text{aps}_{AB} (i, j) \cdot \beta \quad (4.25)
\end{align*}
\]

The bottleneck is the two queries

\[
\begin{align*}
\text{query}_1 (i, j) &= \max_{k=0}^{j - \lceil \frac{\Delta A}{\Delta B} \rceil} \text{OPT} (i - 1, k) \quad (4.26) \\
\text{query}_2 (i, j) &= \max_{k=j-\lceil \frac{\Delta A}{\Delta B} \rceil+1}^{\max} \text{OPT} (i - 1, k, \text{true}). \quad (4.27)
\end{align*}
\]

For every pair of \(i\) and \(j\), the enumeration of \(k\) takes \(O(|B|)\), thus the overall time complexity becomes \(O(|A||B|^2)\).

Careful observation reveals that Eqn. 4.26 can be recursively represented as

\[
\begin{align*}
\text{query}_1 (i, j) &= \max \left( \text{query}_1 (i, j - 1), \\
&\quad \text{OPT} \left( i - 1, j - \lceil \frac{\Delta A}{\Delta B} \rceil \right) \right). \quad (4.28)
\end{align*}
\]

Therefore, if we sort all the \((i, j)\) pairs by the increasing order of \(i\) and then the increasing order of \(j\), the computation of each \(\text{query}_1 (i, j)\) only takes \(O(1)\). Eqn. 4.27 is different because the range of \(k\) for \((i, j)\) does not cover the range of \(k\) for \((i, j - 1)\). Compared to \((i, j - 1)\), the range of \(k\) for \((i, j)\) has \(j\) in addition, but loses \(j - \lceil \frac{\Delta A}{\Delta B} \rceil\). Fortunately, this is a variant of the concave problem \([103, 35]\), which can be solved in linear time. The approach we take is maintaining a double-ended queue of \(k\) so that OPT\((i - 1, k, \text{true})\) is strictly decreasing. For each \((i, j)\), OPT\((i - 1, j, \text{true})\) is appended to the end of the queue, and all the \(k\) in the queue such that OPT\((i - 1, k, \text{true}) \leq \text{OPT} (i - 1, j, \text{true})\) are removed. If the first element of the queue is \(k = j - \lceil \frac{\Delta A}{\Delta B} \rceil\), it will be removed, too. After all the updates, the first remaining element of the queue is the \(k \in \left[ j - \lceil \frac{\Delta A}{\Delta B} \rceil + 1, j \right]\) with the maximum OPT\((i - 1, k, \text{true})\). Because all the queue operations are in constant time and every element can be removed at most once, the overall time complexity is \(O(|A||B|)\).

Unlike Alg. 7, we need to go through every OPT\((|A| - 1, j, \cdot)\) to get the optimal solution. The complete dynamic algorithm is described in Alg. 8.
Algorithm 8: Dynamic Programming for Asymmetric Trajectory Similarity with Damping

**input**: Trajectories $A$ and $B$, and the damping factor $\beta$.

**output**: $\text{ats} (A, B)$ and $p^*$.  

create $\text{OPT} \ (0 \ldots |A| - 1, 0 \ldots |B| - 1, \{\text{true, false}\})$;  

create $\text{trace} \ (0 \ldots |A| - 1, 0 \ldots |B| - 1, \{\text{true, false}\})$;  

create $p^* \ (0 \ldots |A| - 1)$;  

for $j$ from 0 to $|B| - 1$ do  

$\text{OPT} \ (0, j, \text{false}) \leftarrow \text{aps}_{AB} \ (0, j)$;  

$\text{OPT} \ (0, j, \text{true}) \leftarrow \text{aps}_{AB} \ (0, j) \cdot \beta$;  

for $i$ from 1 to $|A| - 1$ do  

$\text{query}_1 \ (i, 0) \leftarrow -\infty$;  

$k_1 \leftarrow \text{none}$;  

create double-ended queue $\text{deque}$;  

for $j$ from 0 to $|B| - 1$ do  

if $\text{deque}.\text{front}() = j - \lceil \frac{\Delta A}{\Delta B} \rceil$ then  

$\text{deque}.\text{pop}.\text{front}()$;  

while $\text{OPT} \ (i - 1, \text{deque}.\text{back}(), \text{true}) \leq \text{OPT} \ (i - 1, j, \text{true})$ do  

$\text{deque}.\text{pop}.\text{back}()$;  

$\text{deque}.\text{push}.\text{back}(j)$;  

$\text{OPT} \ (i, j, \text{true}) \leftarrow \text{OPT} \ (i - 1, \text{deque}.\text{front}(), \text{true}) + \text{aps}_{AB} \ (i, j) \cdot \beta$;  

$\text{trace} \ (i, j, \text{true}) \leftarrow \text{deque}.\text{front}()$;  

$\text{OPT} \ (i, j, \text{false}) \leftarrow \text{query}_1 \ (i, j) + \text{aps}_{AB} \ (i, j)$;  

$\text{trace} \ (i, j, \text{false}) \leftarrow k_1$;  

if $j = |B| - 1$ then  

break;  

$\text{query}_1 \ (i, j + 1) \leftarrow \text{query}_1 \ (i, j)$;  

if $j + 1 - \lceil \frac{\Delta A}{\Delta B} \rceil \geq 0$ and  

$\text{OPT} \ (i - 1, j + 1 - \lceil \frac{\Delta A}{\Delta B} \rceil) > \text{query}_1 \ (i, j + 1)$ then  

$\text{query}_1 \ (i, j + 1) \leftarrow \text{OPT} \ (i - 1, j + 1 - \lceil \frac{\Delta A}{\Delta B} \rceil)$;  

$k_1 = j + 1 - \lceil \frac{\Delta A}{\Delta B} \rceil$;  

$j, s \leftarrow \arg \max_{j, s} \text{OPT} \ (|A| - 1, j, s)$;  

$\text{ats} \ (A, B) \leftarrow \frac{\text{OPT}(|A| - 1, j, s)}{|A|}$;  

for $i$ from $|A| - 1$ downto 0 do  

$p^* \ (i) \leftarrow j$;  

$j' \leftarrow j$;  

$j \leftarrow \text{trace} \ (i, j, s)$;  

if $i = 0$ then  

break;  

if $j' - j < \lceil \frac{\Delta A}{\Delta B} \rceil$ then  

$s \leftarrow \text{true}$;  

else  

$s \leftarrow \arg \max_{s} \text{OPT} \ (i - 1, j, s)$;
4.4.5 Symmetric Trajectory Similarity

For trajectories $A$ and $B$, our method computes a pair of asymmetric trajectory similarities $ats(A, B)$ and $ats(B, A)$, both within $[0, 1]$. We then use the $F_1$-score [99] to produce a symmetric trajectory similarity

$$sts(A, B) = 2 \frac{ats(A, B) \cdot ats(B, A)}{ats(A, B) + ats(B, A)},$$

(4.29)

The $F_1$ score is typically used for evaluating the performance of binary classifiers, which depends on precision and recall. Precision is the ratio of the positive samples within the positive guesses, while recall is the ratio of the positive guesses within the positive samples. Their relationship is similar to the relationship between $ats(A, B)$ and $ats(B, A)$, because $ats(A, B)$ can be regarded as the coverage on $A$ of $B$, and $ats(B, A)$ can be regarded as the coverage on $B$ of $A$.

4.5 Results

We test our method on the trajectories from HURDAT (Fig. 4.1), which records 1,792 hurricane trajectories over the North Atlantic Ocean from 1861 to 2015, sampled at a resolution of 6 hours. The longest duration hurricane consists of 133 vertices whereas the average is 27.03. The hurricane dataset demonstrates that spatial and directional features are both important in judging the similarity between a pair of trajectories. We could also consider traffic data, e.g. GeoLife [105], but the direction data is highly quantized and does not demonstrate the combination of spatial and directional measures as well.

The precision of our similarity measure is controlled by has two parameters: $\alpha$ and $\beta$. The parameter $\alpha$ serves as a size-relative distance threshold whereas $\beta$ distributes the mapping between two trajectories more evenly, especially when $\alpha$ is set so large that finer details are ignored. Setting $\beta = 1$ allows any and all “squeezing” whereas setting $\beta = 0$ provides the strongest penalty to more evenly redistribute the mapping pairs more evenly along the two trajectories. For the HURDAT data, we set $\alpha = 0.3$ and $\beta = 0.8$.

4.5.1 Trajectory Mapping

The mapping used by trajectory similarity identifies potentially similar parts between two trajectories. We looked into a random subset of trajectory pairs. Our method either produces similar mappings as DTW or has more reasonable mappings. Due to the space limit, we only show a selected set of pairs in Fig. 4.8.

In Fig. 4.8 (a), the trajectory $A$ is consist of three segments, $a$ to $b$, $b$ to $c$, and $c$ to $d$. In Fig. 4.8 (a.1), the three segments are nicely mapped to $a'$ to $b'$, $b'$ to $c'$, and $c'$ to $d'$. The trajectory $B$ is consist of two
segments, $a'$ to $b'$, and $b'$ to $d'$. In Fig. 4.8 (a.2), they are nicely mapped to $a$ to $b$ and $b$ to $d$. $c'$ is also mapped to $c$, which is not necessary but a good coincidence, as both of them are the convex points on their trajectories. Fig. 4.8 (a.3) shows the DTW matching, which is separated by the intersection point $e$ of the two trajectories. Separation on intersections is common in DTW, impeding the matching between similar points like $c$ and $c'$. Fig. 4.8 (b) shows a simple example of the importance of the directional information, where $A$ and $B$ are both consist of two segments separated by $b$ and $b'$ respectively. Both mappings of $\text{ats}(A, B)$ and $\text{ats}(B, A)$ correctly capture the directional similarity between the two separatrices, as in Fig. 4.8 (b.1) and (b.2). The DTW matching, however, fails to build the correspondence between them, as is shown in (b.3).

The two trajectories in Fig. 4.8 (a) and (b) are similar to each other, but in Fig. 4.8 (c), the trajectory $B$ has significant extensions in both the beginning and the end compared to the trajectory $A$. The matching of DTW covers every point on both trajectories as is shown in Fig. 4.8 (c.3), with the separation on the intersection. Our method has the ability to determine which parts to be mapped. Fig. 4.8 (c.1) shows the mapping of $\text{ats}(A, B)$, where the extensions on both ends of $B$ are not mapped. We can observe that $a$ to $b$ is mapped to $a'$ to $b'$, and $c$ to $d$ is mapped to $b'$ to $d'$. The segment from $b$ to $c$ is not similar to any part on $B$, thus is squeezed and mapped around the point $b'$. In the mapping of $\text{ats}(B, A)$ shown in Fig. 4.8 (c.2), $a'$ to $b'$ and $b'$ to $d'$ are mapped back to $a$ to $b$ and $c$ to $d$. The segment $b$ to $c$ is not mapped, and the two extensions are squeezed. Fig. 4.8 (d) is another example of significant extensions, where $A$ has an extension in the beginning and $B$ has an extension in the end. The segment from $a$ to $b$ on $A$ is similar to the segment from $a'$ to $b'$ on $B$. Both asymmetric mappings correctly identify the correspondence and squeeze the extensions as in Fig. 4.8 (d.1) and (d.2). The matching of DTW gives the optimal matching assuming every point should be matched, however, fails to see those features.

Fig. 4.8 (a) through (d) may leave the wrong impression that the two asymmetric mappings always identify the same correspondences between trajectories, so we show a counterexample in Fig. 4.8 (e). According to Fig. 4.8 (e.1), $a$ to $b$ on $A$ is mapped to $a'$ to $b'$ on $B$. In (e.2), however, the segment from $d'$ to $a'$ is not squeezed. $\text{ats}(B, A)$ maps $d'$ to $e'$ to $a$ to $b$, and squeezes the segment from $e'$ to $b'$. Both asymmetric mappings successfully match $b$ and $b'$, the turning points on their trajectories, while the DTW matching fails to do so.

Fig. 4.9 shows a typical example where our scoring is more reasonable than DTW in picking the more similar trajectory regarding a given trajectory.
4.5.2 Trajectory Clustering

The state-of-the-art method of trajectory clustering is vector field \( k \)-means (VFKM) proposed by Ferreira et al. [30]. Like the framework of \( k \)-means, their method repeatedly reassigns the trajectories into a user-specified number of clusters until convergence. They assume each trajectory is generated by an underlying velocity field. Thus a discrete vector field of a user-specified resolution \( R \) is fit for each trajectory cluster by solving an optimization problem, which is a linear combination of velocity consistency and Laplacian smoothness. The vector field is then used as the representative for the cluster, and each trajectory is assigned to the cluster whose vector field is the most consistent to it.

While VFKM suffices to cluster trajectories, this approach suffers several drawbacks. First, the performance is limited by the resolution \( R \) of the vector fields, which is typically no more than 5 due to the time required for the optimization. Second, distant trajectories can be assigned to the same cluster no matter how dissimilar they are. Third, static velocity fields as used by VFKM are not compatible with self-intersecting trajectories. Fourth, there is a delicate interplay between the smoothness parameter \( \lambda \) and resolution \( R \) that is significantly less intuitive than the \( \alpha \) and \( \beta \) parameters used to tune our proposed similarity measure. We used VFKM to reproduce the seven clusters of the HURDAT dataset in Fig. 4.10, which match the results originally published for VFKM on this same dataset [30].

The typical strategy of similarity-based trajectory clustering is to represent each trajectory as a high-dimensional vector \((\text{sts}(T_1, T_i), \text{sts}(T_2, T_i), \ldots, \text{sts}(T_N, T_i))\), where \( N \) is the total number of trajectories. The vectors can be clustered by the original \( k \)-means algorithm, typically with spectral clustering. We propose a different approach by defining the cluster trajectory similarity of any cluster \( C \) and trajectory \( T \).

\[
\text{cts}(C, T) = \sum_{T_C \in C} \text{sts}(T_C, T) w(T_C, C).
\] (4.30)

In order to compute the weight \( w(T, C) \), we define the authority of a trajectory as

\[
\text{auth}(T, C) = \sum_{T_C \in C} \text{ats}(T, T_C)^2
\] (4.31)

so that trajectories with high authorities are representative of common paths. The weight is therefore defined as

\[
w(T, C) = \frac{\text{auth}(T, C)}{\sum_{T_C \in C} \text{auth}(T_C, C)}.
\] (4.32)

Our clustering approach is also an iterative method. Each trajectory \( T \) is repeatedly reassigned to the cluster \( \arg \max_C \text{cts}(C, T) \) until convergence. Because the similarity between trajectories can be precomputed, the
The overall time complexity of one iteration is $O(n^2)$.

The result of this k-means clustering approach for $k = 7$ and using our proposed trajectory similarity is shown in Fig. 4.1. For comparison, we demonstrate the results of this same k-means clustering approach using DTW in Fig. 4.11, and using LCSS in Fig. 4.12. Hence trajectory similarities based on total pairings (DTW) as well as those on proximate mappings (LCSS) fall short of the meaningful clustering produced by our proposed trajectory similarity measure as shown in Fig. 4.1.

4.6 Conclusion

We have proposed a new measure of trajectory similarity for datasets where both the location and shape are of interest, and we have provided a dynamic programming approach that efficiently computes the similarity score. We have reviewed the previous work, revealing the typical drawbacks (not necessarily suffered by all the existing measures though), (1) the lack of consideration of either the spatial or the directional information, (2) unbounded or incomparable distance or similarity score, (3) the lack of convergence with increasing number of sampled points, (4) complex and hard-to-tune parameters, (5) inefficient computation, (6) the lack of affine-invariance, (7) incapability of multiple matching on a single point, and (8) the lack of robustness against highly dissimilar pairs in the optimal matching, all of which have been addressed by our method.

To our knowledge, we are the first to propose an asymmetric trajectory similarity and combine the spatial and directional similarities on each pair of points. The simplest form of our measure suffers from a unique problem, oversqueezing, which has not been a serious issue before. We therefore impose a damping factor on squeezed mapping, and then make use of the concavity of the modified optimization problem to keep the time complexity from increasing, the consideration of which is the limitation of our modification. More sophisticated penalties for oversqueezing do exist, but so far as we have considered, they all require a higher time complexity, plus the experiments justify the effectiveness of the damping factor.

The experiment shows that our method is able to (1) provide a more reasonable mapping between a pair of trajectories, and (2) give a similarity score consistent to the human eyes. Our method enables a simple similarity-based approach to outperform the state-of-the-art trajectory clustering algorithm on a real dataset of Atlantic hurricanes. Our method only has two parameters $\alpha$ and $\beta$, which are easy for the users to understand and tune. Moreover, we also suggest a setting based on the practice. Because our measure is affine-invariant and the parameters are unitless, the users can always use the setting for all the datasets.

Our trajectory similarity is dependent on the location and direction at each sampled point, thus is
more vulnerable to noises or insufficient sampling rates than the previous similarities only dependent on
the locations. Necessary smoothing or super-sampling may be required for low quality datasets, potentially
increasing the time cost for the main process. In urban traffic datasets, a trajectory is typically comprised
of sharp turns and the path between turns is usually regular, which can be a street or bridge. Our method
is not designed for the cases where only locations are important for the users. When applied on such cases,
the term of directional similarity should be removed.
Figure 4.8: Each row shows a pair of trajectories $\mathbf{A}$ and $\mathbf{B}$ from HURDAT. The first and second column are the mapping of $\text{ats}(\mathbf{A}, \mathbf{B})$ and $\text{ats}(\mathbf{B}, \mathbf{A})$ respectively, and the last column is the matching of DTW.
Figure 4.9: In this example, our method scores the similarity between A and B as 0.74 and the similarity between A and C as 0.54, indicating that B is more similar to A than C is, which looks obvious to us, as A and B are both smooth right U-turns. However, DTW scores the distance between A and B as 9.59, while scores the distance between A and C as 9.15, indicating that C is slightly closer to A than B is. This is a typical consequence of not considering the directional information.

Figure 4.10: The seven clusters produced by VFKM on the HURDAT dataset.
Figure 4.11: The seven clusters produced by the DTW trajectory similarity measure on the HURDAT dataset.

Figure 4.12: The seven clusters produced by the LCSS trajectory similarity measure on the HURDAT dataset.
Chapter 5

Conclusion

This chapter summarizes the contributions of this thesis and indicates future work.

In Chapter 2, we have established a new performance mark for tracing particles through unsteady unstructured flow datasets, in particular those with uniformly sized elements, and maintaining $CFL \approx 1$ as commonly employed for particle advection. We have shown that the performance of GPU particle tracing algorithms suffer as particles disperse across the flow domain, but these effects can be countered by block advection. We showed for example that the complex flows of most interest to LCS analysis should set the outer block width to span about five cells in each dimension, and the inner block width to about half the outer width. For more laminar flows, the benefit of the hysteresis provided by inner/outer block boundaries is not necessary and block size can be safely minimized to the smallest value available based on thread utilization. The algorithm works best on unstructured meshes whose elements are roughly uniformly sized. Adaptive meshes with wide variations in cell size, such as our airfoil example, can lead to blocks whose uniform size is too small, dominated by memory constraints needed to hold dense velocity nodes at the corners of small cells. Further research on this approach by varying its overlapping block size would better handle such adaptive meshes, but would require further analysis for its parameter tuning.

In Chapter 3, we have proposed a robust ridge extraction methods for Lagrangian coherent structures using watershed method. While watershed methods commonly appear in the summaries of ridge extraction methods for Lagrangian coherent structures, they are often dismissed in favor of local methods such as marching ridges. We have shown that their results are often much smoother and less noisy than such local approaches, and should be considered further. One of the main shortcomings of the watershed approach is that it does not detect ridges that end at a minimum. Such an example might be a ridge descending from the rim to the bottom of a crater. We plan to address such cases with a combination of local and global combinations, using the local ridge definition evaluated on the current sea-level coastline front of the V-S watershed method. This combination of local and global ridge methods could yield the best of both worlds. We also plan to work on high performance streaming implementations of the V-S and other watershed algorithms, updating earlier such work [77], as well as their applications to larger, out-of-core datasets.
In Chapter 4, we have proposed a new measure of trajectory similarity for datasets where both the location and shape are of interest, and we have provided a dynamic programming approach that efficiently computes the similarity score. To our knowledge, we are the first to propose an asymmetric trajectory similarity and combine the spatial and directional similarities on each pair of points. The simplest form of our measure suffers from a unique problem, oversqueezing, which has not been a serious issue before. We therefore impose a damping factor on squeezed mapping, and then make use of the concavity of the modified optimization problem to keep the time complexity from increasing. The experiment shows that our method is able to (1) provide a more reasonable mapping between a pair of trajectories, and (2) give a similarity score consistent to the human eyes. Our method enables a simple similarity-based approach to outperform the state-of-the-art trajectory clustering algorithm on a real dataset of Atlantic hurricanes. Our trajectory similarity is dependent on the location and direction at each sampled point, thus is more vulnerable to noises or insufficient sampling rates than the previous similarities only dependent on the locations. Necessary smoothing or super-sampling may be required for low quality datasets, potentially increasing the time cost for the main process.
References


