FTK: A High-Dimensional Simplicial Meshing Framework for Robust and Scalable Feature Tracking

Hanqi Guo, Member, IEEE, David Lenz, Jiaxi Xu, Xin Liang, Wenbin He, Iulian R. Grindeanu, Han-Wei Shen, Member, IEEE, Tom Peterka, Member, IEEE, Todd Munson, and Ian Foster Fellow, IEEE

Abstract—We present the Feature Tracking Kit (FTK), a framework that simplifies, scales, and delivers various feature-tracking algorithms for scientific data. The key of FTK is our high-dimensional simplicial meshing scheme that generalizes both regular and unstructured spatial meshes to spacetime while tessellating spacetime mesh elements into simplices. The benefits of using simplicial spacetime meshes include (1) reducing ambiguity cases for feature extraction and tracking, (2) simplifying the handling of degeneracies using symbolic perturbations, and (3) enabling scalable and parallel processing. The use of simplicial spacetime meshing simplifies and improves the implementation of several feature-tracking algorithms for critical points, quantum vortices, and isosurfaces. As a software framework, FTK provides end users with VTK/ParaView filters, Python bindings, a command line interface, and programming interfaces for feature-tracking applications. We demonstrate use cases as well as scalability studies through both synthetic data and scientific applications including Tokamak, fluid dynamics, and superconductivity simulations. We also conduct end-to-end performance studies on the Summit supercomputer. FTK is open-sourced under the MIT license: https://github.com/hguo/ftk.

Index Terms—Feature tracking, high-dimensional meshing, distributed and parallel processing, critical points, isosurfaces, vortices.

1 INTRODUCTION

Feature tracking is a core topic in scientific visualization for understanding dynamic behaviors in time-varying simulation and experimental data. By tracking features such as extrema, vortex cores, and boundary surfaces, one can highlight key regions in visualization, reduce data to store, and enable further analysis based on the dynamics of features in scientific data.

This paper introduces a general framework that delivers a collection of feature-tracking tools to end users, scales feature-tracking algorithms in distributed and parallel environments, and simplifies the development of new feature-tracking algorithms. The motivations for developing this framework are threefold. First, although feature-tracking capabilities appear sporadically in today’s data analysis and visualization tools, a general-purpose toolset is lacking that would enable users to track and analyze features in scientific workflows. In community tools such as VTK [1], VTK-m [2], ParaView [3], VisIt [4], and TTK [5], most algorithms focus on single-timestep data, and only a few filters are provided for tracking features over time. Object tracking for videos is available in computer vision libraries such as OpenCV [6], but scientific data differ significantly from natural videos in their feature definitions and data representation. Second, few existing feature-tracking algorithms are designed for scalability and parallel processing. The advent of exascale computing means that data produced by supercomputers need to be efficiently handled by the same scale of computing resources. In both in situ and post hoc scenarios, the sheer data size and high complexity of tracking algorithms necessitate distributing data to many computing nodes and using GPU accelerators when available. Third, no developer framework exists for eliminating redundant efforts to implement application-specific feature-tracking algorithms.

Designing data structures and algorithms for feature tracking from scratch can be daunting; the management of time-varying data, the handling of degenerate cases, and the tracking of graphs to record merges and splits are needed in many applications; such features do not exist in publicly available software packages.
available libraries.

To these ends, we identify the common ground—high-dimensional meshing—among many tracking algorithms for isosurfaces, critical points, and vortex cores. By extruding the spatial mesh into the time dimension, a spacetime mesh connects the spatial cells in the original mesh over adjacent timesteps. For example, in 3D isosurface tracking, marching cubes [7] are generalized to higher dimensions [8, 9] by iterating and classifying 4D spacetime cells with lookup tables. In critical point tracking [10,11], the movement of critical points can be captured by checking whether spatiotemporal cells contain critical points. Likewise, in tracking quantum vortices in complex-valued scalar fields [12, 13], the moving trajectories of vortex core lines can be reconstructed in spacetime meshes.

We present the Feature Tracking Kit (FTK), which introduces high-dimensional simplicial meshing for robust and scalable feature tracking. Compared with previous spacetime meshes, the key difference of our method is that all mesh elements are simplices; each k-simplex—triangle, tetrahedron, pentachoron, or beyond—is a polytope consisting of k + 1 affinely independent points in k-dimensional space. With previous methods, 2D triangles are extruded into 3D prisms [10,13], and 3D cubes are extruded into 4D cubes [8,12]; but neither a prism nor a cube in the extruded mesh is simplicial.

Simplicial meshes offer three benefits for feature tracking: specificity, stability, and scalability. First, simplicial meshes eliminate ambiguities in feature tracking, similar to how marching tetrahedra [14] eliminates isosurface ambiguity. In nonsimplicial cells such as cubes, multiple features intersect the same cell, causing ambiguities that require careful attention. We show that with the spacetime piecewise linearity (PL) assumption, no disambiguation is needed for tracking critical points and isosurfaces in simplicial meshes. Second, simplicial meshes ease the handling of degeneracies, enabling robust feature tracking. Degeneracies, such as a critical point on an edge or isosurface intersecting a vertex, may lead to loss or duplication in the detection results due to numerical instabilities [15]. Simplicial meshes enable the use of Simulation of Simplicity (SoS) [16]—a mature programming technique to simplify the handling of degeneracies in computational geometry—to generate robust, combinatorial, and consistent tracking results regardless of numerical instabilities. Third, simplicial meshes make it straightforward to accelerate feature-tracking algorithms with both GPU parallelism and distributed parallelism. In cases when the feature detection is independent in each cell, we can distribute the tasks to different computing resources for concurrent and scalable processing.

In this study, we design and implement the simplicial subdivision of two types of high-dimensional meshes—(n + 1)-D simplicial prism and (n + 1)-D regular meshes—to enable robust and scalable feature tracking in both unstructured and regular meshes in $\mathbb{R}^n$, in order to support the tracking of critical points (0D features in 2D/3D), quantum vortices (1D features in 3D), and isosurfaces (2D features in 3D) for a wide range of applications. The n-dimensional space consists of both 2D/3D space and time, and all mesh elements are simplices. Enabled by high-dimensional simplicial meshing, each individual tracking algorithm has novelties in disambiguation, degeneracy handling, and scalability. We also enable efficient mesh element traversal over time. Considering that time-varying data are large and streamed from simulations in situ; one can iterate spacetime mesh elements within a sliding window of a few timesteps for out-of-core and streaming data access. As a software framework, FTK provides ParaView plugins, Python bindings, command line interfaces, and programming interfaces for end users to track a variety of features both in situ and post hoc. We demonstrate the use of FTK for fluid dynamics, fusion, and superconductivity simulations as well as high-speed imaging experiments. In summary, the novelty of this paper is in its combination of several individual technical contributions:

- A high-dimensional simplicial meshing scheme that generalizes both regular and unstructured spatial meshes to spacetime while tessellating spacetime mesh elements into simplices (Section 3)
- A robust and scalable critical point tracking algorithm that avoids ambiguities and handles degeneracies in a consistent manner (Section 4)
- A scalable implementation of quantum vortex tracking with distributed parallelism (Section 5)
- A robust and scalable isosurface tracking algorithm that avoids ambiguities and handles degeneracies in a consistent manner (Section 6)
- A software framework for users to track features with distributed and parallel environments both in situ and post hoc (Sections 7 and 8)
- Comprehensive performance studies of FTK algorithms on both supercomputers and commodity hardware (Section 9)

2 Related Work

This section reviews related work in the extraction and tracking of critical points, quantum vortices, and isosurfaces. We also briefly review high-dimensional meshing. In the following, we refer to the process of independently detecting features in individual timesteps as feature extraction, as opposed to feature tracking, which is the process of reconstructing trajectories of features through multiple consecutive timesteps. Comprehensive reviews of feature extraction and tracking can be found in [17]; reviews on topology-based methods for visualization are available in [18].

2.1 Critical point extraction and tracking

In general, a critical point is defined as the location where a vector field vanishes. Critical points are also defined in the gradient fields of scalar functions. Below, we review critical point extraction and tracking algorithms.

2.1.1 Critical point extraction

Numerical methods have been used to locate critical points where all vector components are zero simultaneously, assuming the vector field can be interpolated based on discrete representations. While finding such zero-crossings has been studied for bilinear and trilinear schemes [19], the piecewise linear (PL) interpolation of vector fields is more widely used...
in various applications because of its simplicity [20–22]. Tricoche et al. characterized higher-order critical points in 2D PL vector fields by partitioning neighboring regions based on different flow behavior [21]. That approach was further generalized to 3D vector fields [23]. Besides PL, extraction of critical points in piecewise-constant vector fields can be achieved by discrete Hodge decomposition [24].

A major issue with numerical methods is their sensitivity to numerical instabilities. As illustrated in Figure 1(a) and (b), a critical point may be identified multiple times if the critical point resides on the boundary of cells. To this end, Bhatia et al. [15] introduced the use of symbolic perturbation [30] in testing if a simplicial cell contains critical points, leading to a robust critical point detection in a combinatorial manner, illustrated in Figure 1(c). Our study further generalizes the use of symbolic perturbation to ensure that the tracking of critical points is robust and combinatorial, as demonstrated in following sections.

![Critical Point Extraction](image)

**Fig. 1.** Nonrobust (a and b) versus robust (c) critical point extraction. With numerical methods, when a critical point resides on an edge, the critical point may or may not be detected by all triangles that share the same edge; in this case, the number of detected critical points could range from zero to six because of numerical instabilities. With robust critical point extraction and tracking, the single critical point will be detected and associated with one of the triangles in a combinatorial manner.

**Topology methods** include the use of the Poincaré index theorem, (discrete) Morse theories, and contour trees/Reeb graphs. For example, Poincaré’s index theorem can be used to test if critical points exist in 3D regions [25,26] or PL surfaces [27]. With Morse decomposition, Chen et al. [28] proposed a vector field topology representation of 2D PL vector fields with graphs, such that critical points can be identified as part of the vector field topology. For scalar fields, critical points are the key constituents of the scalar field topology, including Reeb graphs [29] and contour trees [30] extracted with well-established algorithms.

### 2.1.2 Critical point tracking

**Spacetime meshing methods** assume the interpolation over time to track critical points. Tricoche et al. [10] extruded 2D triangular cells into 3D spacetime prisms, detected entries and exits of singularities on prism faces, and then identified paths of critical points. Garth et al. [11] generalized this approach to 3D by extruding from tetrahedra to 4D tetrahedral prisms. For both methods, the vector field is assumed linear over both space (barycentric interpolation) and time (linear interpolation). Note that similar to bilinear and trilinear interpolations, such spacetime interpolation in prisms is not PL in spacetime, as explained in the next section.

**Feature flow field (FFF) methods** [31] use a derived FFF vector field to characterize feature movements, such that feature trajectories can be computed as the streamlines in the FFF. For critical point tracking, one needs to find an appropriate set of critical points in spacetime as the seeds, compute streamlines from the seeds by numerical integration, and then slice streamlines back into individual timesteps. To address instabilities in the numerical integration, Weinkauf et al. [32] proposed a method to improve convergence. Klein and Ertl generalized FFF to track critical points in scale space [33]. Reininghaus et al. [34] proposed a combinatorial version of FFF based on the discrete Morse theory to track critical points in 2D time-varying scalar fields.

**Nearest-neighbor and region-overlapping approaches** are heuristics to track critical points. For example, Wang et al. [35] reconstruct trajectories of critical points by joining proximal and same-type critical points in adjacent timesteps. Skraba and Wang [36] used the closeness of robustness, the minimum necessary amount of perturbation to cancel features, to correspond critical points in adjacent timesteps.

### 2.2 Quantum vortex extraction and tracking

In this paper, we use quantum vortices as an example of tracking 1D features. Quantum vortices, or simply vortices, are topological defects in superconductivity [37], superfluidity [38], and Bose-Einstein condensates. Simulations produce 3D complex-valued fields that combine both amplitudes and phase angles. Singularities in phase fields are closed 1D curves embedded in 3D Euclid spaces. By definition, a vortex is the locus of points that

\[ -\oint_C \nabla \theta(x) \cdot dl = 2k\pi, k \neq 0, \]

where \( \theta(x) \) is the phase field; \( C \) is an infinitesimal contour that encircles the vortex curve; \( dl \) is the infinitesimal arc on \( C \); and \( k \) is a nonzero integer usually equal to ±1.

**Vortex extraction.** Based on the definition, a straightforward approach to extract vortices in 3D meshes is to first check whether the contour integral is nonzero for each face boundary and then trace singularity curves along faces [12,37]. Guo et al. [13] proved that a triangulated mesh cell intersects up to one singularity line and thus that simplicial mesh subdivision leads to combinatorial and consistent extraction results.

**Vortex tracking.** A spacetime meshing approach was proposed to correspond vortex curves in adjacent timesteps [12,39] by examining faces in both space and time. As a result, mesh faces testing positive for a singularity form graphs that characterize the movement of singularities as surfaces. Guo et al. [13] used triangular/tetrahedral prisms as the spacetime cells to extract and track singularities. However, ambiguities still exist because spacetime prisms are nonsimplicial. In Section 5, we demonstrate that a high-dimensional simplicial mesh eliminates ambiguities in a consistent manner and allows parallel vortex curve tracking in distributed environments.

Note that quantum vortices are fundamentally different from vortices in fluid flows [40], which are swirling centers of flows and have been defined by level sets or extremum lines of \( \lambda_2 \) eigenvalues [41] and vorticity magnitude [42].
Depending on definitions, tracking of fluid flow vortices may be achieved by connected component labeling [43] or FFF [31].

2.3 IsoSurface Extraction and Tracking

IsoSurface extraction is fundamental to scientific visualization, reconstructing polygon surfaces with the given iso-value in 3D scalar fields. The marching cubes [7] algorithm extracts isosurfaces in regular grid data based on lookup tables, and disambiguating how surfaces are connected inside a cube was the key research problem for a decade. A cubic cell has $2^8 = 256$ possible ways to intersect an isosurface, which boil down to 15 unique configurations. Ambiguities exist when vertex values have alternating signs on any faces. Marching tetrahedra [14] is a promising method to eliminate ambiguities by tessellating inputs into simplicial cells; there are only two unambiguous cases of intersections for each tetrahedron.

IsoSurface tracking. Two distinct ways exist for tracking isosurfaces: volume tracking [43,44] and higher-dimensional isosurfacings [8]. The former extracts regions of interest independently in each timestep and then associates regions across adjacent timesteps based on overlaps. The latter generalizes marching cubes to 4D spacetime; the outputs are 3D objects embedded in 4D and can be sliced back into 2D surfaces in individual timesteps for visualization. Similar to marching cubes in 3D, ambiguities exist in 4D spacetime, and researchers have shown that ione can disambiguate 4D cases by triangulation [9]. We demonstrate a simplified implementation of 4D isoSurface tracking based on our high-dimensional simplicial meshes in Section 6.

2.4 High-Dimensional Meshing

High-dimensional meshing has focused mostly on 4D because of the dimensionality of the physics, while many algorithms such as Delaunay triangulation work in arbitrary dimensions.

High-dimensional meshing for computational sciences. Recently, scientists have started to explore the use of 4D meshes [45,46] to numerically solve time-dependent partial differential equations (PDEs) in spacetime as opposed to traditional stepping approaches. We believe that our method could be directly applied to spacetime mesh data; but because of challenges of increased complexity, memory footprint, and cost to converge, the majority of scientific data today is still stored and represented in discrete timesteps.

High-dimensional meshing for scientific visualization. Spacetime meshing approaches, which are limited mostly to prisms to date, have been successfully used to track singularities in vector fields [10,11] and phase fields [12,13,37,39]. Prisms are a straightforward choice but challenges exist in handling ambiguities, as discussed in the rest of this paper.

3 High-Dimensional Simplicial Mesh

We design and implement the simplicial subdivision of $(n + 1)$-D simplicial prism and $(n + 1)$-D regular meshes, respectively, in order to enable robust and scalable feature tracking in unstructured and regular grid meshes; the dimensionality $n$ is 2 or 3 for the dimensionality of spatial domain. The simplicial subdivision of $(n + 1)$-D regular meshes, which is a special case of the subdivision of $(n + 1)$-D prism meshes, is implemented separately for the efficient handling of images, volumes, and curvilinear grids.

For example, in the case of $n = 2$, the input is a triangular mesh (illustrated in Figure 2(a)). One can extrude the mesh into 3D by replicating and elevating vertices in the new dimension, forming 3D triangular prisms (Figure 2(b)). The output 3D mesh is a subdivision of triangular prisms, and each mesh element in the output mesh is simplicial (Figure 2(c)). We also categorize and index simplices in all dimensions for efficient traversal (Figure 2(d)). In the rest of this section, we formalize definitions (Section 3.1), describe the subdivision of $(n + 1)$-D simplicial prism meshes (Section 3.2), and introduce the subdivision of $(n + 1)$-D regular meshes as a special case of subdividing prism meshes (Section 3.3).

3.1 Definitions

Formally, an $n$-simplex is the convex hull of $n + 1$ points $a_0,a_1,\ldots,a_n \in \mathbb{R}^n$. An $n$-simplicial complex is the set of $k$-simplices ($k = 0, 1, \ldots, n$); the intersection of any two $k$-simplices ($k > 1$) is either a common ($k - 1$)-simplicial face (such as triangles, edges, and vertices) or the empty set. For this study, all simplices are nondegenerate; that is, each $k$-simplex bounds a $k$-dimensional volume.

We define the $(n + 1)$-D simplicial prism as the extrusion of an $n$-simplex $a_0a_1\ldots a_n$ to one dimension higher. Denoting the $\mathbb{R}^{n+1}$ coordinates of each point $a_i$ as $\mathbf{x}_i = (x_{i,0}, x_{i,1}, \ldots, x_{i,n})^\top$ with the identical last component $x_{i,n}$ for all $i$, the extruded prism includes another simplex $b_0b_1\ldots b_n$; the coordinates of each point $b_i$ are $(x_{i,0}, x_{i,1}, \ldots, x'_{i,n})^\top$, $x_{i,n} < x'_{i,n}$. Note that $a_i$ and $b_i$ share the same first $n$ coordinates and that the last coordinate is different. In addition to the two simplices, the prism includes $n$ edges $a_0b_1, a_1b_2, \ldots, a_nb_n$. Note that an $(n+1)$-D simplicial prism is nonsimplicial.

We further define the $(n + 1)$-D simplicial prism mesh as a collection of $(n + 1)$-D simplicial prisms obtained by the extrusion of a conforming simplicial mesh into one dimension higher. Our goal is to tessellate the $(n + 1)$-D simplicial prism mesh into a simplicial complex without adding new vertices. The tessellation must be conformal, such that any two $k$-simplices share a common face or nothing. Examples of conformal and non-conformal subdivisions are shown in Figure 3.

3.2 Simplicial Subdivision of $(n + 1)$-D Simplicial Prism Meshes

We first review the concept of staircase triangulation [47] and then generalize the staircase triangulation to the conformal subdivision of prism meshes. Without loss of generality, we describe the case of $n = 3$, the extrusion from an unstructured 2D triangular mesh to a 3D triangular prism mesh, followed by its subdivision into a 3D tetrahedral mesh. Assuming the input is given by a list of triangles (2-simplices), our algorithm extrudes each triangle into a
Fig. 2. Extrusion of simplicial mesh: (a) the input 2D simplicial mesh, (b) the extruded 3D simplicial prism mesh, (c) the output 3D simplicial mesh as the conformal subdivision of the 3D simplicial prism mesh, (d) and (e) subdivision of a 3D triangular prism with the staircase method. Unique types of edges, faces, and tetrahedra in the extruded mesh are illustrated in d and e.

Fig. 2. Extrusion of simplicial mesh: (a) the input 2D simplicial mesh, (b) the extruded 3D simplicial prism mesh, (c) the output 3D simplicial mesh as the conformal subdivision of the 3D simplicial prism mesh, (d) and (e) subdivision of a 3D triangular prism with the staircase method. Unique types of edges, faces, and tetrahedra in the extruded mesh are illustrated in d and e.

The prism in a new dimension; each triangular prism is further subdivided into three tetrahedra in a conformal manner.

Staircase triangulation of a 3D triangular prism. As illustrated in Figure 2(d), a triangular prism consisting of two sets of three vertices, each lying in a plane, may be subdivided into three tetrahedra. For example, consider one trio of points lying in the plane $z = 0$ and the other lying in the plane $z = 1$. Denoting the “lower” vertices as $a_0a_1a_2$ and the “upper” vertices as $b_0b_1b_2$, we may subdivide this triangular prism into three tetrahedra: $a_0a_1a_2b_2$, $a_0a_1b_1b_2$, and $a_0b_0b_1b_2$. As documented by DeLoera et al. [47], the vertex list of each tetrahedra corresponds to a monotone staircase beginning with $a_0$ and ending with $b_2$ in Figure 4; each vertex is immediately above or to the right of the previous vertex in the grid.

Staircase triangulation of an $(n+1)$-D simplicial prism. The staircase triangulation can be generalized to higher dimensions, and an $(n+1)$-D simplicial prism may be subdivided into $n+1$ $(n+1)$-simplices without the introduction of new vertices. First, we impose an ordering on the $n+1$ vertices in the lower and upper hyperplanes (and use the same ordering in both hyperplanes). Second, we identify the $2(n+1)$ points of the prism with the grid $\{0, 1, 2, \ldots, n\} \times \{0, 1\}$. Third, we compute all monotone paths on the grid. The staircase triangulation of 2-, 3-, and 4-simplicial prisms are listed in Table 1.

Table 1

| Type | 1D | 2D | 3D | 4D | 5D |
|------|----|----|----|----|----|
| 1D   | $a_0b_0$ | $a_0a_1b_1$ | $a_0a_1a_2b_2$ | $a_0b_0b_1b_2b_3$ | $a_0a_1a_2b_1b_2b_3b_4$ |
| 2D   | $a_0a_2b_0b_2$ | $a_0a_1a_2b_2b_3$ | $a_0a_1b_1b_2b_3b_4$ | $a_0a_1a_2b_0b_1b_2b_3b_4$ | $a_0a_1a_2a_0a_3a_4b_4$ |

Staircase triangulation of $(n+1)$-D simplicial prism mesh. One can prove that given a global vertex ordering on a simplicial prism mesh, the staircase subdivision method produces conformal subdivisions along prism boundaries. In a 3D case in Figure 2(a-c), we assign a global order of each vertex and then subdivide each prism with staircase triangulation. In practice, it is equivalent to subdivide each quadrilateral in 3D triangular prism meshes. For example, the quadrilateral $a_0a_1a_2b_1$ is subdivided into two triangles $a_1a_2b_1$ and $a_1b_1b_2$ along the monotonous edge $a_1b_1$.

Mesh element indexing. Each k-simplex in the subdivided $(n+1)$-D simplicial prism mesh can be one to one mapped to a tuple of integer ID, type, and timestep for traversing and compact storage. Considering the extrusion along the new dimension for multiple layers of vertices (e.g., multiple timesteps), we use the same triangulation scheme for each layer and design an efficient indexing of simplices in all dimensions in the new mesh. For $k = 3$, there are three types of 3-simplices: bottom, center, and upper tetrahedra (or type-I, II, and III tetrahedra), such that one can index each tetrahedron with a tuple of original triangle ID, type, and timestep. The original triangle ID is the integer index of the triangle in the original mesh. For $k = 2$, to uniquely index 2-simplices, we identify five unique types of faces: prism base, prism lower, prism higher, edge lower, and edge upper. For example, the "top" triangle of a prism can be indexed by the "bottom" triangle of the same prism in

1. Monotone here means that both alphabets and subscripts are ascending. For example, an edge like $b_0a_1$ or $a_1a_0$ cannot appear.
the next timestep; triangles on quadrilaterals can also be indexed by neighboring prisms in the same layer. As such, each 2-simplex can be uniquely indexed by the tuple of original triangle/edge ID, type, and timestep. Likewise, for \( k = 1 \), we identify three unique types of edges; each can be indexed by the original vertex/edge ID, type, and timestep.

**Mesh element queries.** We provide functions for querying a mesh element in the extruded mesh, including \texttt{vertices()}, \texttt{sides()}, and \texttt{side_of()}, which are used in feature tracking algorithms. The \texttt{vertices()} function returns the list of vertices of the given mesh element ID. The \texttt{sides()} function provides a list of \((k−1)\)-simplicial sides of the given \( k \)-simplex, such as the triangular faces of a tetrahedron. In contrast, the \texttt{side_of()} function gives a list of \((k+1)\)-simplices that contains the given \( k \)-simplex. For example, a triangular face in 3D simplicial meshes is usually contained by two tetrahedra unless the face is on the boundary of the domain; likewise, a tetrahedron in a 4D simplicial mesh is usually shared by two pentachora (4-simplices).

**Ordinal and interval mesh elements.** For ease of feature tracking, we further categorize \( k \)-simplices into ordinal and interval types. A simplex is an ordinal type if each of its vertices resides in the same timestep in the extruded mesh; otherwise it is an interval type. For example, the lower edge triangle (type-V face in Figure 2(d)) has two vertices in the lower layer and one vertex in the upper layer and is thus an interval type. The type-I edge is ordinal because both vertices are in the same layer.

There are two reasons to distinguish ordinal and interval types. First, this distinction allows feature-tracking algorithms to consume data in a streaming and out-of-core manner for both in situ and post hoc processing, as discussed in the next paragraph. Second, the distinction allows efficient slicing of output trajectories. If one needs results only in individual timesteps instead of intervals, it is straightforward to select features identified in ordinal mesh elements.

In a streaming pipeline, assuming data of each timestep \( 0, 1, \ldots, n_t−1 \) are available in the ascending order, \( n_t \) being the number of timestep, we show that one can traverse all \( k \)-simplices while keeping a sliding window of two timesteps of data. For each \( i \)th timestep, we first traverse ordinal types and then traverse interval types if \( i < n_t−1 \). Because each interval type consists of vertices from adjacent timesteps, both \( i \)th and \((i+1)\)th timesteps must be available in memory. As a result, it streams timesteps and traverses all \( k \)-simplices without having spacetime data of all timesteps in memory simultaneously.

**Complexity.** The space complexity of maintaining a conformal subdivision of an \((n+1)\)-simplicial prism mesh is the order of the number of simplices in all dimensions in the original \( n \)-simplicial mesh. For example, in the case of \( n = 2 \), we need to maintain lists of all triangles, edges, and vertices of the original triangular mesh. We also maintain lists of sides and parents for all simplices in the original mesh, in order to accelerate the query of sides and parents in the extruded mesh. The time complexity of querying a simplex and getting the vertex list of the simplex is constant.

**3.3 Simplicial subdivision of \((n+1)\)-D regular mesh**

Subdividing a regular mesh is a special case of that in the preceding subsection but does not require maintaining a mesh data structure (e.g., lists of vertices and triangles). We define the \((n+1)\)-D regular simplicial mesh as a simplicial subdivision of the \((n+1)\)-D regular mesh without introducing additional vertices.

**Recursive subdivision of \((n+1)\)-D regular mesh.** Conceptually, one can recursively subdivide an \((n+1)\)-D regular mesh based on the simplicial extrusion of an \( n \)-D regular simplicial mesh. For \( n = 0 \), the extruded mesh (1D regular grid) is already simplicial. For \( n ≥ 1 \), one can extrude cells in a \( n \)-D regular mesh into prisms and follow Table 1 to triangulate the prisms. As a result, each \( n \)-D cube is subdivided in the same way into \( n! \) congruent and disjoint \( n \)-simplices, and the subdivision is conformal.

**Precomputation of the subdivision for \( n \)-D unit cube.** In practice, we precompute the subdivision of the \( n \)-D unit cube for any \( n \), which enables direct access to an \( n \)-D simplicial mesh without recursive subdivision. For example, the unit 2-cube can be subdivided into two 2-simplices:

\[
\begin{align*}
(0, 0, 1), (1, 1), \\
(0, 1, 0), (1, 1)
\end{align*}
\]

where each 2-digit is the coordinate and ID of the vertex and each line is a 2-simplex. By extruding the simplices, the conformal simplicial subdivision of the unit 3-cube contains six tetrahedra:

\[
\begin{align*}
(0, 0, 0, 0), (0, 0, 1, 1), (1, 1), \\
(0, 0, 0, 1), (0, 1, 1, 1), (1, 1), \\
(0, 0, 1, 0), (1, 0, 1, 1), (1, 1), \\
(0, 1, 0, 0), (1, 1, 1, 1), (1, 1)
\end{align*}
\]
Fig. 6. Indexing and querying mesh elements in regular simplicial mesh: (a) a 2D regular mesh; (b) indexing simplicial elements with \( k : (\text{corner})/\text{type} \), where \( k \) is the dimensionality of the simplex; \( (\text{corner}) \) is the corner coordinates of cube that contains the simplex, and \( \text{type} \) is the unique simplex type ID; (c) \( \text{sides}() \) and \( \text{side_of}() \) of a simplicial element.

as is also illustrated in Figure 5(a). Likewise, the unit 4-cube can be subdivided into twenty-four \( (4! = 24) \) pentachora, as illustrated in Figure 5(b).

\[
\begin{align*}
0000, & \quad 0001, \quad 0011, \quad 0111, \quad 1111 \\
0000, & \quad 0010, \quad 0011, \quad 0111, \quad 1111 \\
0000, & \quad 0101, \quad 0111, \quad 1111 \\
& \quad \ldots \\
0000, & \quad 1000, \quad 1100, \quad 1110, \quad 1111
\end{align*}
\]

After the precomputation, each cube in the \( n \)-D regular mesh is subdivided in the same way.

**Mesh element indexing.** We use the tuple of simplicial dimension \( k \), corner coordinates, and the unique type ID to index a \( k \)-simplex in the \( n \)-D simplicial mesh. The corner coordinates encode the location of the \( n \)-cube that contains the simplex. The unique type ID is designed to encode a simplex within the cube; one cannot use other cubes to index the same simplex. In a 2D case in Figure 6(a), although there are five edges (1-simplices) in each 2-cube, we have only three unique types of edges, because horizontal and vertical edges are always shared between neighboring cubes. For example, to index the top edge of cube \((1,1)\), we can use the cube \((1,2)\) to find the same edge. Figure 6(b) enumerates all unique simplex types in 2-cubes. Table 2 enumerates the number of unique types up to 5 dimensions. If the domain is bounded, mesh element IDs can be one to one mapped to integers for iterating and traversing.

**Mesh element queries.** We also provide \( \text{vertices}() \), \( \text{sides}() \), and \( \text{side_of}() \) functions, defined in the previous subsection, for \( n \)-D regular simplicial meshes. The results of each function are precomputed for each unique type. As illustrated in the 2D mesh in Figure 6(c), the sides of Type-II 1-simplices (diagonal edges) include two vertices; two triangular cells contain the same Type-I 1-simplices (horizontal edges).

**Complexity.** The space complexity of maintaining an \( n \)-D regular simplicial mesh is \( O(n!) \), but \( n \) does not exceed 4 for tracking features for 3D data. Note that the space complexity does not grow with the size of the regular grid, because precomputed unit-cube subdivisions are stored instead of an explicit list of mesh elements. Such implicit mesh data structure allows queries of a simplex in constant time.

## 4 Example of tracking 0D features: critical points

This section describes the use of high-dimensional simplicial meshes for tracking critical points in 2D and 3D vector fields.

### 4.1 Assumptions and definitions

We assume that the input \( n \)-dimensional time-varying vector field \( \mathbf{v} : \mathbb{R}^{n+1} \rightarrow \mathbb{R}^n \) is piecewise linear (PL). The PL assumption implies that the vector field is defined on a simplicial spacetime mesh; each cell is an \((n+1)\)-simplex. For example, each cell in a 2D or 3D time-varying vector field is a tetrahedron (3-simplex) or a pentachoron (4-simplex), respectively. The \((n+1)\)-simplicial spacetime mesh can be constructed based on an existing \( n \)-dimensional mesh, detailed in Section 3. As a result, the time-varying vector field \( \mathbf{v} \) is \( C^1 \) continuous along any combination of spatial and temporal directions; that is, there exist \( \mathbf{A} \in \mathbb{R}^{n \times (n+1)} \) and \( \mathbf{b} \in \mathbb{R}^n \) for each \((n+1)\)-simplex \( S \) such that \( \mathbf{v} = \mathbf{A} \mathbf{x} + \mathbf{b}, \mathbf{x} \in S \).

We further assume that the time-varying vector field is *generic*. This means vector values on each vertex \( i \) are nonzero \((v_i \neq 0)\), and vectors at vertices of any \( k \)-simplex \((k = 0, 1, \ldots, n+1)\) are affinely independent. Thus, critical points in generic vector fields may be found in the interior of \( n \)-simplices instead of on cell boundaries. Because vector fields in scientific applications may not be generic, in the end of this subsection we discuss the relaxation of the generic assumption by using the simulation of simplicity [16] technique.

A (spacetime) critical point \( \mathbf{x}_c \in \mathbb{R}^{n+1} \) is the location where the vector value \( \mathbf{v}(\mathbf{x}_c) \) is zero. In this study, we focus only on critical points that are nondegenerate; meaning that the (spatial) Jacobian \( \mathbf{J}_v \) at \( \mathbf{x}_c \) is nondegenerate. Based on the eigensystem of \( \mathbf{J}_v \), the critical point \( \mathbf{x}_c \) can be further categorized into various types such as sources, sinks, and saddles. In the case that \( \mathbf{v} \) is the gradient field of a scalar field, the critical point types are maxima, minima, and saddles.

A *critical point trajectory* (or simply trajectory) is a locus of critical points in space and time, which are 1D curves embedded in \( \mathbb{R}^{n+1} \). Because \( \mathbf{v} \) is PL, critical point trajectories are PL parametric curves, and the intersection with each cell is a line segment. Critical point trajectories can be *sliced* into a set of critical points at an arbitrary time \( t_0 \) by intersecting the hyperplane \( t = t_0 \). In the following sections we discuss methods for reconstructing critical point trajectories.

### 4.2 Two-pass critical point trajectory reconstruction

As illustrated in Figure 7, we use a two-pass algorithm to directly reconstruct critical point trajectories from \( \mathbf{v} \). Without loss of generality, we describe this algorithm with 2D time-varying vector fields \((\mathbf{v} : \mathbb{R}^3 \rightarrow \mathbb{R}^2)\). In the first pass, we iterate each triangular face (2-simplex) to determine whether
Algorithm 1 Two-pass algorithm of tracking 0-, 1-, and 2-features—critical points, quantum vortices, and isosurfaces, respectively—with high-dimensional simplicial mesh. $S$ is the set of simplices that test positive, $UF$ being union-find.

$$S \leftarrow \emptyset, UF \leftarrow \emptyset$$

**for each** tet $\in$ simplices(2) **do**

**if** test(tet) **then** $\triangleright$ Eq.5

$S \leftarrow S \cup$ tet

**end if**

**end for**

**for each** tri $\in$ simplices(2) **do**

**if** test(tri) **then** $\triangleright$ Eq.1

$S \leftarrow S \cup$ tri

**end if**

**end for**

**for each** tet $\in$ simplices(3) **do**

$T \leftarrow S \cap$ tet.sides(2)

UF.unite($T$)

**end for**

$$S \leftarrow \emptyset, UF \leftarrow \emptyset$$

**for each** tet $\in$ simplices(2) **do**

**if** test(tet) **then** $\triangleright$ Eq.5

$S \leftarrow S \cup$ tet

**end if**

**end for**

**for each** penta $\in$ simplices(5) **do**

$T \leftarrow S \cap$ penta.sides(2)

UF.unite($T$)

**end for**

$$S \leftarrow \emptyset, UF \leftarrow \emptyset$$

**for each** edge $\in$ simplices(1) **do**

**if** test(edge) **then** $\triangleright$ Eq.11

$S \leftarrow S \cup$ edge

**end if**

**end for**

**for each** penta $\in$ simplices(5) **do**

$T \leftarrow S \cap$ penta.sides(1)

UF.unite($T$)

**end for**

Fig. 7. Two-pass critical point trajectory reconstruction for 2D and 3D vector fields with 3D (a) and 4D (b) spacetime simplicial meshes, respectively. The first pass tests whether triangular/tetrahedral sides intersects the trajectory, and then the second pass associates every pair of intersected sides if they share the same tetrahedron/pentachoron.

A trajectory intersects the face by solving the inverse linear interpolation problem:

$$
\begin{pmatrix}
u_0 & u_1 & u_2 \\
v_0 & v_1 & v_2 \\
1 & 1 & 1
\end{pmatrix}
\begin{pmatrix}
\mu_0 \\
\mu_1 \\
\mu_2
\end{pmatrix} =
\begin{pmatrix}
0 \\
0 \\
1
\end{pmatrix},
\tag{5}
$$

where $(\mu_0, \mu_1, \mu_2)^T$ are the normalized barycentric coordinates of the trajectory intersection with the face; $(u_0, v_0)^T$, $(u_1, v_1)^T$, and $(u_2, v_2)^T$ are the vector values on the three vertices of the 2-simplex. If $\mu_j \in [0, 1]$ for all $j \in \{0, 1, 2\}$, the triangular face is punctured by a trajectory, and the spacetime coordinates of the critical point $x_c$ can be interpolated as

$$
x_c = \begin{pmatrix}
x_c \\
y_c \end{pmatrix} = \begin{pmatrix}
x_0 & x_1 & x_2 \\
y_0 & y_1 & y_2 \\
1 & 1 & 1
\end{pmatrix}
\begin{pmatrix}
\mu_0 \\
\mu_1 \\
\mu_2
\end{pmatrix},
\tag{6}
$$

where $(x_c, y_c, t_c)^T$ are the spacetime coordinates of the critical point and $(x_j, y_j, t_j)^T$ are the spacetime coordinates of each vertex $(j \in \{0, 1, 2\})$. In the second pass, we iterate over each tetrahedral cell (3-simplex) to associate its sides that are punctured by trajectories, because one can prove that each tetrahedron intersects up to one trajectory. Complete trajectories can be constructed by pairing every two punctured triangular faces of the same tetrahedron.

In general, for arbitrary dimensionality $n$, the two-pass algorithm to reconstruct critical point trajectories in $v : \mathbb{R}^{n+1} \rightarrow \mathbb{R}^n$ is follows. The first pass iterates over each $n$-simplex to determine whether the simplex intersects a trajectory based on Equation 5. The second pass iterates each $(n+1)$-simplex and pairs its sides $(n$-simplices) that intersect a trajectory. The two-pass algorithm can be easily parallelized with both distributed and GPU parallelism, as discussed in the following sections.

The output reconstructed critical point trajectories are closed curves in spacetime; they either end on domain boundaries or are loops. Within each curve, the critical point type may alternate, and there may be multiple monotone intervals with respect to time. For example, as illustrated in Figure 8(a), each loop characterizes a maximum-saddle pair in the gradient field; the maximum-saddle pair establishes and annihilates simultaneously. In Figure 8(c), we see the birth of a saddle-sink pair; the saddle further merges with another sink soon after the birth. One can further simplify and filter trajectories based on their attributes, as discussed in the following sections.

4.3 Robustness

The above two-pass algorithm assumes that PL vector fields are generic, an assumption that often does not hold for real-world data. For example, gradients of an integer-valued image may be exactly zero at vertices; gradients based on central-differences, which are rational numbers, may be affinely dependent, causing nongeneric situations. In fluid flows, nonslip conditions lead to zero velocities on boundaries. With nongeneric cases, however, there is no guarantee that each $(n+1)$-simplex has up to one pair of intersected sides. A critical point may reside on the boundary of the cell, causing numerical instabilities during the test of $\mu_k \in [0, 1]$ in Equation 5. As a result, the critical point may or may not be detected in the current $n$-simplex; chances are that the same critical point be detected by neighboring cells, causing nonrobust and noncombinatorial tracking results, as illustrated in Figure 1.

We use the concept of simulation of simplicity [16] (SoS) to robustly and combinatorially compute critical point trajectories in vector fields that are not generic. As proved by Bhatia et al. [15] with the Brouwer degree theory, a critical point exists in the $n$-simplex $\{x_0, x_1, \ldots, x_n\}$ if and only if
0 lies interior of the convex hull of \( \{v_0, v_1, \ldots, v_n\} \), where \( v_j \) (\( j = 0, 1, \ldots, n \)) is the vector value on each vertex \( x_i \).

As a result, the critical point test is reduced to the point-in-simplex predicate, which can be determined by the sign of determinant of the matrix \( (v_0, v_1, \ldots, v_n, 1) \). In nongeneric cases, for example, a critical point on the boundary of the simplex, the SoS prevents the determinant from becoming zero by adding a symbolic perturbation such that the critical point is associated with one of the simplices in a consistent manner.

### 4.4 Critical point trajectory filtering, simplification, and smoothing

We provide three postprocessing approaches to help users filter, smooth, and simplify trajectories that result from tracking critical points.

**Filtering.** One can filter results based on the attributes—time duration, topology, persistence, scalar value, if applicable—of trajectories. Figure 8(a) is an example of filtering loops in the gradient of a scalar field. Typically, a loop exists when a small transient bump appears, introducing a saddle-extremum pair. Such a loop may be filtered out based on the time duration or persistence of the loop. Likewise, trajectories can be filtered based on other attributes, as demonstrated in the following sections.

**Simplification.** One can also simplify trajectories that change directions frequently in time based on a threshold of persistence in time. As illustrated in Figure 8(c), for example, a saddle-sink pair is born right before the saddle merges with another sink. Because the saddle may be caused by noise, we provide the simplification function to eliminate the saddle and merge the trajectory as a consistent sink type.

**Smoothing.** Although our trajectory reconstruction algorithm is robust, the evaluation of Jacobians \( J_v \) may subject to numerical instabilities, causing inconsistent critical point types along trajectories. One can smooth the types based on a window-shifting approach. We iterate each point in the trajectory and check whether the current critical point type is consistent with both the precedent and antecedent. If an inconsistency is identified, we mark the inconsistency location and modify the type after the iterations, as illustrated in Figure 8(b). The window size depends on the application and analysis needs. For example, we set the half-window size to be two consecutive timesteps in our experiments.

### 4.5 Evaluation and verification with synthetic data

We validate the effectiveness and evaluate the robustness of our critical point tracking approach with synthetic data.

**3D moving minimum.** We use the following scalar function to synthesize 3D time-varying scalar field data with the known position of the single minimum to evaluate the numeric robustness of our method:

\[
f(x,t) = \|x - (x_0 + d \cdot t)\|^2,
\]

where \((x,t) \in \mathbb{R}^{n+1}\) are the spatiotemporal coordinates, and \(x_0\) and \(d\) are arbitrary vectors in \(\mathbb{R}^n\). As a result, the trajectory of the single minimum \(x_c(t)\) in the data is

\[
x_c(t) = x_0 + d \cdot t.
\]

In Figure 9, we synthesized 20 different instances with the same \(x_0 = (10, 10, 10)^T\) but with different \(d\). The scalar function \(f\) is discretized into a \(21 \times 21 \times 21\) grid, which represents a \((0, 0, 0) \times (20, 20, 20)\) domain. Each component of the moving direction \(d\) is a random rational number such that the trajectory must intersect at least one grid point including \(x_0\), causing degenerate cases similar to that of Figure 1. In Figure 9(a), because the grid point of \(x_0\) is shared by multiple pentachora in the spacetime mesh, the number of tetrahedra that numerically test positive for containing a critical point ranges from zero to tens. Because the degenerate cases cause ambiguity in tracing, trajectories in the figure appear dashed and isolated. In Figure 9(b), our robust detection approach guarantees that each critical point is exclusively associated with a tetrahedron such that trajectories can be robustly tracked without any ambiguity.

**2D double gyre flow.** We demonstrate critical point tracking in a 2D unstructured mesh with the double gyre function, which is widely used to study Lagrangian coherent structures.\(^2\) The double gyre function is defined in the domain of \([0, 0] \times [2, 1]\), and we generate a triangular mesh with 2,098 triangles and 1,100 vertices to demonstrate the results. Figure 10 visualizes critical trajectories in \(t \in [0, 40]\); the timespan \(\Delta t\) between adjacent timesteps is 0.1. The color of each trajectory is categorical and encodes the unique ID of the trajectory. Slices at \(t = 0\) and \(t = 2.8\) visualize flow directions with line integral convolution (LIC), and the slice at \(t = 1.2\) visualizes the 2D mesh. As shown in the figure, two critical points move back and forth along the \(x\)-axis periodically, and the saddle points repeatedly enter and exit the 2D domain. Because the double gyre function is analytical, we verified that the vector field is exactly zero at all points in the trajectories, and all critical points are identified by our method.

---

2. [https://shaddenlab.berkeley.edu/uploads/LCS-tutorial/examples.html](https://shaddenlab.berkeley.edu/uploads/LCS-tutorial/examples.html)
critical points in the publicly available turbulent vortex
blobs. The trajectories of critical points can 56,980 vertices and 112,655 triangles in an XGC particle-in-
malized electron density field, we reconstruct trajectories to a maximum/minimum in the preconditioned and nor-
future power generation. Assuming each blob corresponds
do damage the Tokamak—is critical for the reactor design and under-
blob—regions of high turbulence that could
Tokamak fusion plasma simulations (Figure 12(b)). Un-
applications, we also demonstrated the use of our tools to
track microparticle clouds in laboratory plasma experi-
4.6 Case studies with applications
We demonstrate use cases of our critical point tracking
methods in science applications.
Fluid dynamics. We track vector field critical points to
visualize the vortex streets in a 2D flow-past-cylinder sim-
ulation (Figure 12(a)). The data are available on a uniform
grid with the resolution of 400 × 50 for 1,001 timesteps. We
use the average velocity as the frame of reference and track
critical points rotate around the origin at a fixed angular
speed over time. We discretized the function into a 128² grid
and injected Gaussian noise with two different standard
deviations to the data, as shown in Figure 11. Note that
the range of the data is [−1, 1], and thus perturbation of
σ = 0.02 and σ = 0.08 introduces up to 3% and 12% relative
error, respectively, to the data in the 3-sigma limits. Because
the noise injection produces many artificial bumps in the
data, the output critical point trajectories contain artifacts.
Artifacts such as small loops can be removed through tra-
jectory filtering, as discussed in the previous subsection.

4.7 Limitations
First, in the case of critical point tracking in scalar fields, one has to use the derived
gradient fields as the input. However, the spacetime PL gradient field implies
C² continuity of the original scalar field. The outputs may be distorted because of the smoothness of differen-
tiation kernels used for gradient derivation. That being said, the output trajectories are as
accurate as the quality of the gradient field and how close the scalar field is to
C² continuity. In addition, the fidelity of critical point trajectories is related to the
spacetime resolution of the inputs. Down-
sampling the data may lead to an over-
simplified topology of the trajectories. We
leave the study of topology change due to
downsampling to future work.

Second, the determination of critical point types is subject to numerical insta-
bilities. Because Jacobians are usually es-
timated numerically, in extreme cases the
signs of eigenvalues may change when
perturbation is introduced. We therefore introduced a
smooth filter to heuristically correct classification errors, but one can also remediate the problem based on domain-
specific knowledge.

Third, the PL assumption prevents the native identifi-
cation of higher-order critical points [21, 23]. An example
of higher-order cases is illustrated in Figure 13(b), where
two local maxima periodically merge and split. However, the
result includes a persisting trajectory that characterizes
one of the maxima and a number of maxima-saddle loops
periodically. At the time of merging, ideally three critical
points (two maxima and one saddle) should merge and
then split into another three critical points. Because only
up to one trajectory is allowed to intersect each cell, our
method cannot identify the “3-in-3-out” event over time. We
leave the study of higher-order dynamics of critical points
to future work as well.

3. https://www.cs.ucdavis.edu/~ma/ITR/tvdr.html
difficult to scale in distributed and parallel environments in general can be embarrassingly parallelized, while computing streamlines streamline tracing. Second, our two-pass algorithm can be such as Runge–Kutta methods, further introduces error in require such transformation. Numerical integration in FFF, of the gradients of vector fields, our method does not comparison with FFF, which requires numerical approximations can be computed as the streamlines in FFFs. First, in com-
parison with FFF, this method consists of iterative numerical algorithms such as Newton’s method. Third, our simplicial mesh enables the robust tracking of critical points based on SoS, producing combinatorial and consistent results when nongeneric cases occur.

Compared with approaches based on feature flow fields (FFFs) [31–34], our method is numerically robust and computation scalable. FFFs are vector fields that characterize the movement of critical points such that the trajectories can be computed as the streamlines in FFFs. First, in comparison with FFF, our method requires numerical approximations of the gradients of vector fields, our method does not require such transformation. Numerical integration in FFF, such as Runge–Kutta methods, further introduces error in streamline tracing. Second, our two-pass algorithm can be embarrassingly parallelized, while computing streamlines in distributed and parallel environments in general can be difficult to scale [51, 52].

5 EXAMPLE OF TRACKING OF 1D FEATURES: QUANTUM VORTICES

We demonstrate the use of high-dimensional simplicial meshing for tracking 1D topological defects (quantum vortices) in 3D complex-valued scalar fields that are produced by superconductivity, superfluidity, and Bose–Einstein condensate simulation data. Unlike critical points in vector fields, vortices are 1D curves in individual timesteps, and the trajectories of vortices are 2D surfaces embedded in 4D spacetime.

4.8 Comparison with existing critical point tracking algorithms

Compared with existing approaches using spacetime meshing [10, 11], our approach offers three improvements. First, our method uses simplicial meshes instead of prism meshes in 4D spacetime. Because each simplicial cell intersects up to one singularity in our framework, our method consistently avoids ambiguity when multiple trajectories intersect a prism. Second, the PL assumption makes it easier to localize zero-crossings in simplicial cells than in prism cells, a task that involves iterative numerical algorithms such as Newton’s method. Third, our simplicial mesh enables the robust tracking of critical points based on SoS, producing combinatorial and consistent results when nongeneric cases occur.

Compared with approaches based on feature flow fields (FFFs) [31–34], our method is numerically robust and computationally scalable. FFFs are vector fields that characterize the movement of critical points such that the trajectories can be computed as the streamlines in FFFs. First, in comparison with FFF, which requires numerical approximations of the gradients of vector fields, our method does not require such transformation. Numerical integration in FFF, such as Runge–Kutta methods, further introduces error in streamline tracing. Second, our two-pass algorithm can be embarrassingly parallelized, while computing streamlines in distributed and parallel environments in general can be difficult to scale [51, 52].
boundaries; each tetrahedral cell has up to one pair of intersected faces [13], as illustrated in Figure 14(b). As a result, vortex curves can be reconstructed by associating intersected faces that share the same tetrahedral cells. The process involves two passes: first iterating all triangular faces and then scanning tetrahedral cells that have intersected faces.

**Vortex surface reconstruction.** The two-pass curve reconstruction can be generalized to 4D spacetime to characterize the trajectory of vortex curves as surfaces. In 4D, each pentachoron has five tetrahedral faces; each tetrahedron may intersect a vortex. Considering that tetrahedron shares triangular sides in the pentachoron, the number of triangular sides that test positive may be 3, 4, or 5 (corresponding to Case I, II, or III in Figure 14). Each intersection sits on the same 2-manifold of the vortex surface. In the reconstruction of the vortex surface, the first pass tests all triangular faces in 4D spacetime, and then the second pass joins all triangular faces that test positive and share the same pentachoron.

**Benefits.** The use of spacetime simplicial meshing simplifies and improves on previous implementations based on cubic meshes [12] and triangular prism meshes [13]. In the former study, ambiguity exists when two vertices penetrate the same cube. In the latter study, to eliminate ambiguities with cubic cells, each spatial cube is tessellated into six tetrahedra such that each tetrahedron intersects up to one vortex line. Although mesh elements in individual timesteps are simplices, simplicial prisms are used in spacetime, causing two problems: (1) two different functions are needed to test triangular and quadrilateral faces of a prism, and (2) ambiguity is still possible with prism cells. With simplicial spacetime meshes, we need only one function to test triangular faces, and there is no ambiguity.

### 5.2 Case study of a superconductivity simulation

Figure 15 demonstrates vortex tracking results of a time-dependent Ginzburg-Landau superconductivity simulation, produced by the finite-difference partial differential equation solver GLGPU [53]. In this case, the resolution of the mesh is 512 × 128 × 64, and the number of the timesteps is 200. As a result, 6.5M out of 251M spacetime triangular faces tested positive for encircling a vortex, forming a set of disjoint surfaces.

From the scientific perspective, the dynamics of vortices determine all electromagnetic properties of the superconducting materials, and recombinations of vortices are directly related to energy dissipation [54]. The surface-based visualization produced by our tools enables scientists to investigate the time-varying features with a single image. The reconstructed surface also makes it possible to derive the moving speed of each vortex when there is no topological changes; the moving speed has positive correlation with the voltage drop, which is critical to the material design.

### 6 Example of tracking of 2D features: isosurfaces

Isosurface tracking in FTK is also based on high-dimensional simplicial meshes. As demonstrated by Bhaniaramka et al. [8] and Ji et al. [9], the tracking of isosurfaces in 3D scalar fields can be achieved by extracting and slicing levelsets in \( \mathbb{R}^4 \) regular meshes. While the major complexities of previous efforts—disambiguation of isosurfaces intersecting the same hypercube—may be resolved by triangulation, our implementation with high-dimensional simplicial meshing intrinsically avoids ambiguities in a consistent manner and scales to larger computing resources with the FTK framework.

**6.1 Two-pass isovolume reconstruction in spacetime scalar fields**

We formalize the isosurface tracking problem as the reconstruction of isovolumes in the time-varying scalar field \( f: \mathbb{R}^{n+1} \rightarrow \mathbb{R} \), isovolumes being the solution of \( f = c \), where \( c \) is the isovalue. In general, the isovolume is an \((n−1)\)-dimensional object embedded in \( \mathbb{R}^{n+1} \). With \( n = 3 \), the object can be further sliced into 2D surfaces with fixed time values. We assume that \( f \) is generic; that is, the scalar value \( f_i \neq c \) for each vertex \( i \), and scalars at vertices of any \( k \)-simplex \((k = 1, 2, \ldots, n + 1)\) are affinely independent. Thus, each edge (2-simplex) in the spacetime mesh intersects the level set at most once; the edge neither resides on the level set nor intersects the it at vertices. Similar to the robust critical point tracking method in Section 4, we use symbolic perturbations [16] to relax the generic assumption and to reconstruct 4D level sets in a robust and consistent manner for real application data.

As shown in Algorithm 1, the reconstruction consists of two passes for isovolume reconstruction in 4D: the edge pass and pentachoron pass. In the first pass, we check whether every edge (2-simplex) intersects a level set by solving the inverse interpolation:

$$
\mu f_0 + (1 - \mu) f_1 = c,
$$

where \( f_0 \) and \( f_1 \) are the scalar values on each vertex of the simplex, \((\mu, 1 - \mu)\) being the barycentric coordinates of the intersection point. The edge intersects a level set if and only if \( \mu \in (0, 1) \). In the second pass, we iterate every pentachoron; edges that intersect the level set are associated and labeled with the same identifier.

In the case of a 3D time-varying scalar field, the output isovolumes are 3D objects and can be represented as a tetrahedral grid, which can be further sliced into isosurfaces in individual timesteps for visualization and analysis. As illustrated in Figure 16(a), the intersection between a pentachoron and an isovolume has only two distinct cases. We use the positive and negative sign, respectively, to represent whether the scalar value on a vertex is greater or less than the threshold. In the 4D case I (+−−− or other permutations), one of the vertices has a different sign than other vertices do; in this case, the isovolume inside the pentachoron is the single tetrahedron consisting of the four intersections. In the 4D case II (++−−− or other permutations), the pentachoron has six intersections, which form a polytope and can be triangulated into three tetrahedra, as explained below.

Without loss of generality, we show that the second case +++−− leads to an 3-polytope that can be tessellated into three tetrahedra. In the five tetrahedral sides of the pentachoron, we find three tetrahedra with +++−− and the other
loop annihilation in early stages

Fig. 15. Trajectory surfaces of quantum vortices in a 3D superconductivity simulation data with 200 timesteps; color encodes the timestep. Vortices at timestep 132 are visualized with tubes. The two zoomed subfigures visualize how a pair of vortices exchanges parts before and after the recombination event.

Fig. 16. Cases of an isovolume intersecting a pentachoron (a) and a tetrahedron (b).

two tetrahedra with +−−−. As illustrated in Figure 16(b), in the 3D case I (+−−−), the intersection is a triangle; in the 3D case II (++−−), the intersection of the tetrahedron and the isovolume leads to a coplanar quadrilateral. As a result, we have two triangles and three quadrilaterals, forming a prism-like polytope that resides in the same 3D subspace. With the same staircase triangulation method described in Section 3.2, we can decompose the polytope into three non-overlapping tetrahedra in a combinatorial manner.

6.2 Robustness

The robustness of this method is guaranteed by the assumption that the function is generic; that is, the scalar value of each vertex is either greater or less than the isovalue, and Equation 11 has one solution. For real problems that usually do not observe this assumption, degenerate cases may appear. For example, should the scalar value of a vertex exactly equal the isovalue, the intersection may or may not be identified by other edges that share the same vertex, leading to unstable results. Should an edge reside on an isosurface, every point on the edge is a solution of Equation 11, leading to numerical instabilities.

In nongeneric cases, we ensure the robustness with the same SoS programming technique [16] used for robust critical point tracking (described in Section 4.3), and we regard the test in Equation 11 as a special case of critical point detection in a 1D vector field. Thus, if an intersection is on any vertex of the edge, the SoS prevents the divisor from becoming zero by adding a symbolic perturbation, such that the intersection is exclusively associated with one of the edges that share the same vertex.

6.3 Case studies

We demonstrate our isovolume reconstruction with both synthetic and simulation data in Figure 17 and below.

Synthetic data. We demonstrate the reconstruction of isovolume of $f = 0$ for the function $f(x, y, z, t) = x - \alpha t$ on a $21 \times 21 \times 21$ grid and 12 timesteps, with $\alpha = 0.9$, as shown in Figure 17(a). Since the closed form of the surface trajectory ($x = \alpha t$) is available, we can verify that the reconstructed isovolume and the isosurfaces sliced from the isovolume are correct.

Supernova simulation data. We reconstruct the isovolume of a supernova simulation dataset [55] with the isovalue of 0.8 and visualize the sliced isosurfaces of four timesteps in Figure 17(b). The resolution of the data is $432^3$, which in turn produces $\approx 6G$ spacetime edges to test intersections for the 4-timestep data. As a result, the isovolume consists of 25M intersection points and 141M tetrahedra, which can be sliced back into individual timesteps for visualization and analysis.

6.4 Comparison with existing isosurface tracking algorithms

Compared with existing methods of isosurfacing in higher dimensions [8, 9], the benefits of our method include (1) straightforward implementation, (2) no disambiguation cases, (3) guaranteed robustness, and (4) straightforward
parallelization with GPUs and distributed environments. First, the two-pass algorithm can be written in a few lines of code based on FTK’s meshing APIs; there is no need to generate large lookup tables for higher-dimensional marching cubes. Second, compared with marching cubes [7], there are no ambiguity cases with simplicial meshes. Our method can be viewed as a generalization of marching tetrahedra [14] in higher dimensions, which automatically eliminates any ambiguities. Third, the use of the simplicial mesh makes it possible to ensure robustness with symbolic perturbations, leading to consistent tracking results. Fourth, our two-pass algorithm can be easily distributed and computed with parallel computing resources.

7 FTK library design

We describe major components—meshing, numeric, union-find, and I/O—in FTK. In the FTK library design, we consider parallelization for both distributed- and shared-memory environments. We also incorporate the needs of both in situ and post hoc analyses. This section also introduces developer-level APIs for implementing feature-tracking algorithms.

7.1 High-dimensional simplicial mesh APIs

We describe three major C++11 APIs for developing feature-tracking algorithms in FTK to traverse mesh elements in all dimensions in \((n+1)\)-D spacetime simplicial meshes.

- `element_for(int k, function<void(elem_t)> f)` is the API used to iterate over all \(k\)-simplices, \(k = 0, 1, \ldots, n+1\); `elem_t` is the mesh element data type; and `f(elem_t e)` is a user-defined C++ callable—function, lambda, or bind expression—for processing the mesh element `e`. This function simplifies the traversal of mesh elements.
- `sides(elem_t e)` returns the set of \((k-1)\)`-simplicial sides for the given \(k\)-simplex `e`.
- `side_of(elem_t e)` returns the set of \((k+1)\)`-simplices, whose side contains `e`.

All three functions are frequently used in the implementation of feature-tracking algorithms in FTK. For example, in the first pass of 3D critical point tracking (Section 4), we use `element_for(3, detect_critical_point)` to detect critical points in all 3-simplicial cells in the 4D spacetime mesh. In the second pass, we use both `sides(elem_t e)` and `side_of(elem_t e)` to help determine how triangular faces should be connected.

The element type `elem_t` is specific to mesh types. For the simplicial subdivision of \((n+1)\)-D regular meshes, we use the tuple \((k, \text{corner}, \text{type})\) as the unique identifier of a mesh element, where \(k\) is the dimensionality of the element, `corner` is the lower coordinate of the subdivided \(n\)-cube, and `type` is the unique type identifier of \(k\)`-simplices in \(n\)-D regular simplex meshes. Each tuple can be bidirectionally mapped to a long integer type for compact storage and easier task partitioning for parallelism. For the simplicial subdivision of \((n+1)\)-D simplicial prism meshes, we use an integer to encode spacetime mesh elements.

7.2 Inline numerical functions

FTK implements numerical functions—small-matrix linear algebra and symbolic perturbations—for feature-tracking algorithms. The implementation is header-only and template-based, such that the numerical functions can be directly compiled with C/C++/CUDA and executed in GPU kernel functions. For example, in 3D critical point tracking, the robust critical point test relies on the sign of a determinant calculated by symbolic perturbation. If the sign is tested positive, the exact location of the critical point can be estimated by solving a \(3 \times 3\) linear system, and the type of the critical point is determined by the eigenvalues of the \(3 \times 3\) Jacobian matrix. We are aware that header-only numerical libraries are widely available, such as Eigen\(^4\), OpenGL mathematics library (GLM)\(^5\), and Boost\(^6\), but none of these libraries support all the needed numerical functions, especially symbolic perturbations, in GPU kernel functions.

7.3 Distributed union-find

Union-find is a key algorithm for partitioning a set of mesh elements into disjoint sets for feature tracking. We implement an asynchronous distributed union-find method to enable distributed feature tracking. Because the operation of merging sets is order-independent, we use an asynchronous message-passing approach to exchange interprocess requests; the final results remain unchanged even if messages are received in random order. By eliminating frequent and expensive global synchronizations, our method outperforms existing implementations based on the bulk-synchronous parallel programming model. Details of our distributed union-find algorithm can be found in [56].

7.4 I/O

Inputs. FTK libraries can be built with various I/O libraries, including VTK [1], NetCDF [57], HDF5 [58], and ADIOS2 [59], to load data in a variety of formats. Data streamed in situ can be loaded into memory with ADIOS2. In addition, NumPy data are supported through PyFTK, described in Section 8.3.

Outputs. Output formats include VTK, text, and Python objects. With VTK, trajectories and surfaces are transformed into `vtkPolyData`; isovolumes are written in `vtkUnstructuredGrid` formats. FTK can also supports human-readable text formats. Python objects can be retrieved and serialized into either JSON or pickle formats.

8 In situ and post hoc analysis with FTK

FTK provides four different utilities for end users: VTK/ParaView filters, a command line interface, Python bindings, and a C++ programming interface. The VTK/ParaView filters are designed for interactive visualization, with the possibility to couple simulations through ParaView Catalyst and other in situ frameworks. The command line interface can be used for loosely coupled in situ processing and post hoc analyses. The Python bindings

---

4. http://eigen.tuxfamily.org/
5. https://github.com/g-truc/glm
6. https://www.boost.org/
result = ftk.trackers.track_critical_points_2d_scalar(data)

ni = dataset.variables['NI_TRZ']

ni_smooth = gaussian_filter(nitrz, sigma=4.0)

data = ni_smooth.reshape((1, 704, 260, 3059))

result = ftk.trackers.track_critical_points_2d_scalar(data)

Fig. 18. Example use of FTK's (a) ParaView plugins and (b) Python bindings.

are designed for post hoc analyses and integration with data science libraries. The C++ programming interface is designed mainly for tightly coupled in situ analyses.

8.1 ParaView/VTK filters

We developed VTK wrappers to use FTK interactively with ParaView. The filters are designed for interactive visualization, but they also can be used with ParaView Catalyst for in situ processing.

Synthetic data sources. We implemented synthetic data generators for users to learn FTK filters. Synthetic data include double gyre, ABC flows, spiral vorten, and merger functions (Section 4.5). Users can tune parameters controlling how data are synthesized and the spatiotemporal resolutions.

Filters. We wrapped FTK’s feature-tracking algorithms, such as vtkCriticalPointTracker and vtkLevelsetTracker. Currently, the inputs need to be image data types, and the output vtkPolyData can be directly rendered and further processed with other filters with ParaView. Figure 18(a) demonstrates a critical point tracking case: a scalar field is generated from a synthetic data source, and then critical point trajectories are extracted and transformed into tubes for visualization.

8.2 Command line interface

We provided executables for users to track critical points, quantum vortices, and iso-surfaces from files or in situ data streams. For file inputs, FTK supports inputs in multiple file formats including NetCDF, HDF5, VTK, ADIOS2, and raw binaries. Users need to explicitly specify variable names for self-described formats. For in situ data streams, users need to specify ADIOS2 stream sources, variable names, mesh information, and other needed parameters. The command line interface automatically loads and handles data in parallel if executed with MPI. Users are also provided with options to use multithreading and GPU accelerators.

8.3 Python bindings

PyFTK, which is a Python binding library for FTK, enables post hoc feature tracking and analysis in Python. PyFTK functions take NumPy arrays as inputs, allowing users to load data with Python’s NetCDF4, h5py, and other I/O libraries. PyFTK also enables easy integration with other libraries such as SciPy and Matplotlib for data science workflows. For example, in Figure 18(b), we load BOUT++ [60] fusion plasma simulation data with the NetCDF4 Python module, apply Gaussian smoothing in spacetime with SciPy, and track critical points with PyFTK.

8.4 C++ programming interface

T C++ FTK APIs enable users to couple simulation codes with FTK and/or customize feature-tracking algorithms. The former can be achieved with user-level APIs; the latter requires the use of developer-level APIs.

User-level APIs. APIs are available to users to feed time-varying data into FTK algorithms. Users can call push_field_data() functions in various tracker classes. Each timestep of the input field data must be converted into ftk::ndarray, a dynamic multidimensional array that eases the internal data transformation and access. Utility functions are provided to convert C-style arrays and vtkDataArrays into ftk::ndarray.

Developer-level APIs. Users can customize feature-tracking algorithms by using element_for(), sides(), and side_of() functions for access to high-dimensional mesh elements. One can also use linear algebra functions to help localize features on both CPUs and GPUs.

9 Performance Evaluation

We benchmarked the scalability of FTK on Summit, a 200-petaflop supercomputer in Oak Ridge National Laboratory. The system consists of 4,608 IBM AC922 computing nodes, each of which has two 22-core Power9 CPUs and six NVIDIA Tesla V100 GPUs. The clock frequency of each CPU is 3.07 GHz, and each node is equipped with 1,600 GB of high-bandwidth memory. The interconnection between nodes is 100Gbps EDR InfiniBand. As shown in Figure 19, we characterize the strong scalability by measuring the execution time of solving three feature-tracking problems with different numbers of processes, each of which uses either one GPU or one CPU core.

GPU acceleration. The magnitude of acceleration comparing GPUs and CPUs typically ranges from O(100) to O(1000). The acceleration is expected because there is no synchronization or communication between GPU threads; each thread independently tests a mesh element. The magnitude of GPU acceleration varies, possibly because of different complexities of handling mesh elements, precision
processor and memory technologies. First, we will expand meshes, incorporating scale spaces, and porting to new porting more types of features, adapting to time-varyingities show that FTK algorithms can be accelerated by both and post hoc analysis and visualization. Performance stud-
ies show that FTK algorithms can be accelerated by both GPUs and distributed parallelism with high scalability.

In the future we plan to investigate four aspects: sup-
porting more types of features, adapting to time-varying meshes, incorporating scale spaces, and porting to new processor and memory technologies. First, we will expand

FTK’s support of features including parallel vectors (1D features in 3D), ridge/valley surfaces (2D features in 3D), and interval volumes (3D features in 3D). Second, we will develop spacetime mesh generalizations for time-varying meshes such as finite volume and adaptive mesh refinement meshes for broader scientific applications. Third, we will incorporate scales as the fifth dimension for scale-space tracking with our high-dimensional meshes. Fourth, we will port FTK to emerging hardware including GPUs, FPGAs, and high-bandwidth memory in order to enable accelerated computing and data handling.

10 CONCLUSIONS AND FUTURE WORK

This paper demonstrates the use of spacetime simplicial meshes for simplifying, scaling, and delivering feature-tracking algorithms. Our contributions include high-dimensional simplicial meshing and feature-tracking algorithms over these meshes. We present methods to tessellate (n + 1)-D simplicial prisms and regular meshes in arbitrary dimensions. We track three types of features: critical points (0D features embedded in 2D/3D), quantum vortex curves (1D features embedded in 3D), and isosurfaces (2D features embedded in 3D). These contributions are implemented in a suite of feature-tracking tools called FTK. FTK provides VTK/ParaView plugins, a command line interface, Python bindings, and C++ programming interfaces for both in situ and post hoc analysis and visualization. Performance studies show that FTK algorithms can be accelerated by both GPUs and distributed parallelism with high scalability.

In the future we plan to investigate four aspects: supporting more types of features, adapting to time-varying meshes, incorporating scale spaces, and porting to new processor and memory technologies. First, we will expand

FTK’s support of features including parallel vectors (1D features in 3D), ridge/valley surfaces (2D features in 3D), and interval volumes (3D features in 3D). Second, we will develop spacetime mesh generalizations for time-varying meshes such as finite volume and adaptive mesh refinement meshes for broader scientific applications. Third, we will incorporate scales as the fifth dimension for scale-space tracking with our high-dimensional meshes. Fourth, we will port FTK to emerging hardware including GPUs, FPGAs, and high-bandwidth memory in order to enable accelerated computing and data handling.

ACKNOWLEDGMENTS

This research is supported by the Exascale Computing Project (ECP), project number 17-SC-20-SC, a collaborative effort of Department of Energy Office of Science and the National Nuclear Security Administration, as part of the Co-design center for Online Data Analysis and Reduction (CODAR) [61]. It is also supported by the U.S. Department of Energy, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program, and by Laboratory Directed Research and Development (LDRD) funding from Argonne National Laboratory, provided by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.

REFERENCES

[1] W. Schroeder, K. Martin, and B. Lorensen, The Visualization Toolkit (4th ed.). Kitware, 2006.
[2] K. Moreland, C. M. Sewell, W. Usher, L. Lo, J. S. Meredith, D. Pugmire, J. Kress, H. A. Schroots, K. Ma, H. Childs, M. Larsen, C. Chen, R. Maynard, and B. Geveci, “VTK-m: Accelerating the visualization toolkit for massively threaded architectures,” IEEE Computer Graphics and Applications, vol. 36, no. 3, pp. 48–58, 2016.
[3] U. Ayachit, The ParaView Guide: A Parallel Visualization Application. Kitware, 2015.
[4] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátili, “VisIt: An end-user tool for visualizing and analyzing very large data,” in High Performance Visualization—Enabling Extreme-Scale Scientific Insight, E. W. Bethel, H. Childs, and C. Hansen, Eds., 2012, pp. 357–372.
[5] J. Tierny, G. Faveller, J. A. Levine, C. Gueunet, and M. Michaux, “The Topology Toolkit,” IEEE Transactions on Visualization and Computer Graphics, vol. 24, no. 1, pp. 832–842, 2018.
[6] G. Bradski, “The OpenCV Library,” Dr. Dobb’s Journal of Software Tools, 2000.
[7] W. E. Lorensen and H. E. Cline, “Marching cubes: A high resolution 3D surface construction algorithm,” in Proc. SIGGRAPH ’87, 1987, pp. 163–169.
[8] P. Bhaniramka, R. Wenger, and R. Crawfis, “Isosurfacing in higher dimensions,” in Proc. IEEE Visualization ’00, 2000, pp. 267–273.
[9] G. Ji, H. Shen, and R. Wenger, “Volume tracking using higher dimensional isocountouring,” in Proc. IEEE Visualization ’03, 2003, pp. 209–216.
[10] X. Tricoche, T. Wischgoll, G. Scheuermann, and H. Hagen, “Topology tracking for the visualization of time-dependent two-dimensional flows,” Computers & Graphics, vol. 26, no. 2, pp. 249–257, 2002.
[11] C. Garth, X. Tricoche, and G. Scheuermann, “Tracking of vector field singularities in unstructured 3D time-dependent datasets,” in Proc. IEEE Visualization ’04, 2004, pp. 329–336.
[12] H. Guo, C. L. Phillips, T. Petkka, D. Karpeyev, and A. Glatz, “Extracting, tracking and visualizing magnetic flux vortices in 3D complex-valued superconductor simulation data,” IEEE Transactions on Visualization and Computer Graphics, vol. 22, no. 1, pp. 827–836, 2016.
[13] H. Guo, T. Peterka, and A. Glatz, "In situ magnetic flux vortex visualization in time-dependent Ginzburg-Landau superconductor simulations," in Proc. IEEE Pacific Visualization Symposium, 2017, pp. 71–80.

[14] A. Doi and A. Koide, “An efficient method of triangulating equilateral surfaces by using tetrahedral cells,” IEICE Transactions on Information and Systems, vol. E74-D, no. 1, pp. 214–224, 1991.

[15] H. Bhatia, A. Gyulassy, H. Wang, P. Bremer, and V. Pascucci, “Robust detection of singularities in vector fields,” in Topological Methods in Data Analysis and Visualization III, Theory, Algorithms, and Applications, P. Bremer, I. Hotz, V. Pascucci, and R. Peikert, Eds. Springer, 2014, pp. 3–18.

[16] H. Edelsbrunner and E. P. Mücke, “Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms,” ACM Transactions on Graphics, vol. 9, no. 1, pp. 66–104, 1990.

[17] F. F. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch, “In situ magnetic flux vortex visualization in time-dependent Ginzburg-Landau superconductor simulations,” in Proc. IEEE Visualization and Computer Graphics, vol. 24, no. 8, pp. 2440–2455, 2018.

[18] C. L. Phillips, H. Guo, T. Peterka, D. Karpeyev, and A. Glatz, “Tracking vortices in superconductors: Extracting singularities from a discretized complex scalar field,” Physical Review E, vol. 93, no. 023305, pp. 1–13, 2016.

[19] Y. Guo, X. Liu, C. Xiong, X. Xu, and C. Fu, “Towards high-quality visualization of superfluid vortices,” IEEE Transactions on Visualization and Computer Graphics, vol. 24, no. 8, pp. 3308–3318, 2018.

[20] M. Jiang, R. Machiraju, and D. Thompson, “Detection and visualization of vortices,” in The Visualization Handbook, C. D. Hansen and C. R. Johnson, Eds., 2005.

[21] J. Reininghaus, J. Kasten, T. Weinkauf, and I. Hotz, “Efficient computation of combinatorial feature flow fields,” IEEE Transactions on Visualization and Computer Graphics, vol. 18, no. 9, pp. 1563–1573, 2012.

[22] B. Wang, P. Rosen, P. Skraba, H. Bhatia, and V. Pascucci, “Visualizing robustness of critical points for 2D time-varying vector fields,” Computer Graphics Forum, vol. 32, no. 3, pp. 221–230, 2013.

[23] P. Skraba and B. Wang, “Interpreting feature tracking through the lens of robustness,” in Topological Methods in Data Analysis and Visualization III, Theory, Algorithms, and Applications, P. Bremer, I. Hotz, V. Pascucci, and R. Peikert, Eds. Springer, 2014, pp. 19–37.

[24] J. Xu, H. Guo, W. Shen, M. Raj, X. Xu, and Z. Wang, “Detecting vortices in superconductors: Extracting one-dimensional topological singularities from a discretized complex scalar field,” Physical Review B, vol. 94, no. 2, pp. 020501(R), 2016.

[25] X. Tricoche, G. Scheuermann, and H. Hagen, “A topology simplification method for 2D vector fields,” in Proc. IEEE Visualization 2000, 2000, pp. 359–366.

[26] X. Liang, H. Guo, S. Di, F. Cappello, M. Raj, C. Liu, K. Ono, Z. Chen, and T. Peterka, “Toward feature-preserving 2D and 3D vector field compression,” in Proc. IEEE Pacific Visualization Symposium 2020, 2020, pp. 81–90.

[27] T. Weinkauf, H. Theisel, K. Shi, H. Hege, and H. Seidel, “Extracting higher order critical points and topological simplification of 3D vector fields,” in Proc. IEEE Visualization ‘05, 2005, pp. 559–566.

[28] G. Reeb, “Sur les points singuliers d’une forme de Pfaff complètement intègrable ou d’une fonction numérique,” Comptes Rendus Acad. Sciences, vol. 222, pp. 847–849, 1946.

[29] D. Pugmire, T. Peterka, and C. Garth, “Parallel integral curves,” in Proc. IEEE Visualization ‘05, 2005, pp. 559–566.

[30] K. Polthier and E. Preuß, H. Hege and K. Polthier, Eds. The Visualization Handbook, 2010.

[31] C. M. Greene, “Locating three-dimensional roots by a bisection method,” Journal of Computational Physics, vol. 98, no. 2, pp. 194–198, 1992.

[32] S. Mann and A. P. Rockwood, “Computing singularities of 3D vector fields with geometric algebra,” in Proc. IEEE Visualization ‘02, 2002, pp. 283–289.

[33] W. Li, B. Vallet, N. Ray, and B. Lévy, “Representing higher-order singularities in vector fields on piecewise linear surfaces,” IEEE Transactions on Visualization and Computer Graphics, vol. 12, no. 5, pp. 1315–1322, 2006.

[34] G. Chen, K. Mischakow, R. S. Laramee, and E. Zhang, “Efficient Morse decompositions of vector fields,” IEEE Transactions on Visualization and Computer Graphics, vol. 14, no. 4, pp. 848–862, 2008.

[35] G. Reeb, “Sur les points singuliers d’une forme de Pfaff complètement intègrable ou d’une fonction numérique,” Comptes Rendus Acad. Sciences, vol. 222, pp. 847–849, 1946.

[36] J. Zhang, H. Guo, F. Hong, X. Yuan, and T. Peterka, “Dynamic load balancing based on constrained K-D tree decomposition for parallel particle tracing,” IEEE Transactions on Visualization and Computer Graphics, vol. 24, no. 1, pp. 954–963, 2018.

[37] D. Pugmire, T. Peterka, and C. Garth, “Parallel integral curves,” in High Performance Visualization: Enabling Extreme Scale Scientific Insights, E. W. Bethel, H. Childs, and C. Hansen, Eds. CRC Press, 2012.

[38] I. Sadovsksiy, A. Koshelev, C. Phillips, D. Karpeyev, and A. Glatz, “Stable large-scale solver for Ginzburg-Landau equations for superconductors,” Journal of Computational Physics, vol. 294, no. C, pp. 639–654, 2015.

[39] A. Glatz, V. K. Vlasko-Vlasov, W. K. Kwok, and G. W. Crabtree, “Vortex cutting in superconductors,” Physical Review B, vol. 94, no. 084505, pp. 1–11, 2016.

[40] A. G. Evans, D. A. Raouffe, A. Burrows, and D. Vantayan, “The gravitational wave signal from core-collapse supernovae,” The Astrophysical Journal, vol. 861, no. 10, pp. 1–19, 2018.

[41] A. Tran, H. Guo, W. Shen, M. Raj, X. Xu, X. Wang, Z. Wang, and T. Peterka, “Distributed asynchronous union-find for scalable feature tracking,” arXiv:2003.02351 [cs.DC], 2020.

[42] R. Rew and G. Davis, “NetCDF: an interface for scientific data access,” IEEE Computer Graphics and Applications, vol. 10, no. 4, pp. 76–82, 1990.

[43] The HDF Group. (2000-2010) Hierarchical data format version 5. [Online]. Available: http://www.hdfgroup.org/HDF5

[44] J. F. Lofstead, F. Zheng, S. Klasky, and K. Schwan, “Adaptable, metadata rich IO methods for portable high performance IO,”
in IPDPS’09: Proc. IEEE International Symposium on Parallel and Distributed Processing, 2009, pp. 1–10.

[60] B. Dudson, M. Umansky, X. Xu, P. Snyder, and H. R. Wilson, “BOUT++: A framework for parallel plasma fluid simulations,” Computer Physics Communications, vol. 180, no. 9, pp. 1467–1480, 2009.

[61] I. Foster, M. Ainsworth, J. Bessac, F. Cappello, J. Choi, S. Di, A. M. Gok, H. Guo, K. A. Huck, C. Kelly, S. Klasky, K. Kleese van Dam, X. Liang, K. Mehta, M. Parashar, T. Peterka, L. Pouchard, T. Shu, H. van Dam, J. M. Wozniak, M. Wolf, W. Xu, I. Yakushin, S. Yoo, and T. Munson, “Online data analysis and reduction: An important co-design motif for extreme-scale computers,” International Journal of High-Performance Computing Applications, vol. in press, 2020.

Hanqi Guo is an assistant computer scientist at Argonne National Laboratory, scientist at the University of Chicago Consortium for Advanced Science and Engineering (CASE), and fellow of the Northwestern Argonne Institute for Science and Engineering (NAISE). His research interests include data analysis, visualization, and machine learning for scientific data. He has published more than 40 research papers in top visualization journals and conferences including IEEE VIS, IEEE TVCG, and IEEE TPDS. He is also the recipient of the best paper award in IEEE VIS 2019 and the winner of the 2017 Postdoctoral Performance Award in Basic Research in Argonne National Laboratory. He received his Ph.D. degree in computer science from Peking University in 2014 and his B.S. degree in mathematics and applied mathematics from Beijing University of Posts and Telecommunications in 2009.

David Lenz is a postdoctoral appointee in the Mathematics and Computer Science Division at Argonne National Laboratory.

Jiayi Xu is a Ph.D. student in computer science and engineering at the Ohio State University. He received his B.S. degree in computer science and technology from Chu Kochen Honors College at Zhejiang University in 2014. His research interests include graph visualization and scientific feature tracking.

Xin Liang is a computer/data scientist in the Workflow Systems Group at Oak Ridge National Laboratory. He received his Ph.D. degree from the University of California, Riverside, in 2019 and his bachelor’s degree from Peking University in 2014. His research interests include high-performance computing, parallel and distributed systems, data management and reduction, big data analytic, scientific visualization, and large-scale machine learning. He has published more than 20 peer-reviewed papers in prestigious HPC and Big Data conferences and journals, such as ACM ICS, HPDC, PPoPP, SC, IEEE BigData, Cluster, IPDPS, TPDS, and UHPCA, has received two best paper awards.

Iulian R. Grindeanu is a software development specialist in the Mathematics and Computer Science Division at Argonne National Laboratory. He obtained his Bachelor of Science degree in aerospace engineering at the Military Technical Academy in Romania and both his M.S. and Ph.D. degrees in mechanical engineering at the University of Iowa. He has been a member of ASME and AIAA since 1996. Grindeanu has authored several journal and conference papers. In 1986, he was awarded a gold medal at the Balkan Mathematical Olympiad and a silver medal at the International Mathematical Olympiad.

Wenbin He is a research scientist at Bosch Research. He received his Ph.D. in computer science and engineering at the Ohio State University in 2019 and his B.S. degree from the Department of Software Engineering at Beijing Institute of Technology in 2012. His research interests include analysis and visualization of large-scale scientific data, uncertainty visualization, and flow visualization.

Han-Wei Shen is a full professor at the Ohio State University. He received his B.S. degree from the Department of Computer Science and Information Engineering at National Taiwan University in 1988, his M.S. degree in computer science from the State University of New York at Stony Brook in 1992, and his Ph.D. degree in computer science from the University of Utah in 1998. From 1996 to 1999, he was a research scientist at NASA Ames Research Center in Mountain View California. His primary research interests are scientific visualization and computer graphics. He is a winner of the National Science Foundation's CAREER award and U.S. Department of Energy's Early Career Principal Investigator Award. He also won the Outstanding Teaching award twice in the Department of Computer Science and Engineering at the Ohio State University.
Tom Peterka is a computer scientist at Argonne National Laboratory, a scientist at the University of Chicago Consortium for Advanced Science and Engineering (CASE), an adjunct assistant professor at the University of Illinois at Chicago, and a fellow of the Northwestern Argonne Institute for Science and Engineering (NAISE). His research interests are in large-scale parallel in situ analysis of scientific data. Recipient of the 2017 DOE Early Career Award and four best paper awards, Peterka has published over 100 peer-reviewed papers in conferences and journals that include ACM/IEEE SC, IEEE IPDPS, IEEE VIS, IEEE TVCG, and ACM SIGGRAPH. Peterka received his Ph.D. in computer science from the University of Illinois at Chicago in 2007, and he currently leads several DOE- and NSF-funded projects.

Todd Munson is a senior computational scientist at Argonne National Laboratory, a senior scientist at the Consortium for Advanced Science and Engineering at the University of Chicago, and the Software Ecosystem and Delivery Control Account Manager for the Exascale Computing Project. He has broad interests ranging from numerical methods for nonlinear optimization and variational inequalities to workflow optimization for online data analysis and reduction.

Ian Foster is a Distinguished Fellow, senior scientist, and director of the Data Science and Learning Division at Argonne National Laboratory and a professor in the Department of Computer Science at the University of Chicago.