Interpolation-Based Pathline Tracing in Particle-Based Flow Visualization

Jennifer Chandler, Harald Obermaier, and Kenneth I. Joy, Member, IEEE

Abstract—Particle tracing in time-varying flow fields is traditionally performed by numerical integration of the underlying vector field. This procedure can become computationally expensive, especially in scattered, particle-based flow fields, which complicate interpolation due to the lack of an explicit neighborhood structure. If such a particle-based flow field allows for the identification of consecutive particle positions, an alternative approach to particle tracing can be employed: we substitute repeated numerical integration of vector data by geometric interpolation in the highly dynamic particle system as defined by the particle-based simulation. To allow for efficient and accurate location and interpolation of changing particle neighborhoods, we develop a modified k-d tree representation that is capable of creating a dynamic partitioning of even highly compressible data sets with strongly varying particle densities. With this representation we are able to efficiently perform pathline computation by identifying, tracking, and updating an enclosing, dynamic particle neighborhood as particles move over time. We investigate and evaluate the complexity, accuracy, and robustness of this interpolation-based alternative approach to trajectory generation in compressible and incompressible particle systems generated by simulation techniques such as Smoothed Particle Hydrodynamics (SPH).

Index Terms—Pathlines, particle tracing, SPH, interpolation, flow visualization, time-varying flows

1 INTRODUCTION

Particle tracing of integral lines, surfaces, and volumes has proven valuable and central to virtually every field related to flow visualization and analysis. Moreover, it is the building block of a wide variety of visualization and analysis techniques and has been used for a diverse range of tasks such as flow probing in medicine or the tracking of complex flow structures in aerodynamics.

The numerical techniques used for the computation of such particle trajectories, however, show much less diversity. The common approach—numerical integration—utilizes available velocity information to advect a particle through the flow field, usually requiring frequent velocity interpolation to achieve a reasonably accurate particle trajectory. Therefore, both the number of required interpolation steps, as well as the complexity of the interpolation itself plays a major role in determining the performance of the tracing method. In this work we address particle tracing in flow fields that are modeled as a scattered set of dynamic flow particles, whose lack of a predefined, regular neighborhood structure leads to a high computational complexity in classic velocity-based flow integration techniques. We present a method for advecting particles in these types of flow fields without using velocity field information.

We propose an interpolation-based tracing technique for arbitrary particle simulations with special focus on SPH for illustration purposes. One of the main challenges for tracing pathlines in particle-based flow fields using traditional numerical integration methods is that obtaining the velocity at an arbitrary point in the flow field requires identifying particles in the neighborhood of the tracer. Locating this neighborhood can be an expensive operation with scattered data due to the dynamically changing structure and lack of any explicit connections between particles, and the problem is compounded by the fact that this neighborhood lookup must be performed multiple times in each time step to get accurate results using a numerical integration method. Higher order integration techniques such as adaptive Runge-Kutta methods can easily require many evaluations per time step.

Our contributions to pathline tracing in particle-based flow fields are as follows:

- Our method avoids the need for excessive neighborhood lookups by eliminating the integration of the underlying velocity field and directly using the position information of the simulation particles.
- We develop a modified k-d tree to provide a means for efficient spatial partitioning of the particle domain, allowing for fast identification of arbitrary particle neighborhoods.
- We evaluate this interpolation-based approach and answer questions about its accuracy and usefulness as tracing technique for scattered, particle-based flow fields.

The remainder of this paper is structured as follows. In Section 2 we discuss related work. Section 3 gives background information about particle-based flow fields. Section 4 details our method of tracing arbitrary pathlines in the flow field and gives details on the implementation. Section 5 discusses the performance and accuracy of our method. In Section 6 we summarize our contribution and give suggestions for further work in the area.
2 RELATED WORK

The majority of work performed in flow visualization relies on a velocity field representation of the flow field and uses various numerical integration methods for particle tracing [1], [2]. Since these integration methods can be very time consuming, a large portion of the research focuses on techniques to efficiently sample the flow field to reduce the number of integration steps needed. Several researchers use adaptive approaches to refinement or focus on speed-ups through parallelization on the GPU [3], [4]. Van Wijk [5] constructs stream surfaces using an implicit surface equation which is slower than other methods but allows for computation of families of stream surfaces. Garth et al. [6] construct integral surfaces more efficiently and accurately by decoupling the surface approximation from the graphical representation and by approximating and refining time lines. Hlawatsch et al. [7] use a hierarchical scheme to decrease the number of integration steps by constructing longer integral lines from previously computed partial solutions, reducing the complexity from linear to logarithmic. Krishnan et al. [8] construct time and streak surfaces from time-varying vector fields by building and adapting a surface mesh and representing tracer trajectories as a sequence of fourth order polynomials to allow for easier refinement. Obermaier et al. [9] use adaptive time surfaces to extract topologically relevant flow volumes. Agranovsky et al. [10] identify Lagrangian Coherent Structures with a sparse sampling scheme by using Moving Least Squares to reconstruct particle positions. Kruger et al. [11] advect particles on the GPU to allow for interactive visualization of steady flow on uniform grids and can visualize streamlines and stream ribbons.

In the area of particle-based flow fields—or more specifically SPH visualization—much of the focus has been on rendering surfaces [12] or isosurfaces [13] for incompressible data sets. In the area of compressible SPH simulations, common methods for visualization include column density, volume rendering, or slicing [14], [15], [16]. Biddiscombe et al. [17] use the GPU to visualize SPH data, and they also display trajectories of existing particles in the data set but do not have the ability to display arbitrary pathlines. Schindler et al. [18] focus on extracting vortex core lines from SPH simulation data. Their work is one of the exceptions in SPH visualization in that it extracts flow features rather than visualizing the point set directly.

Our approach goes further by unifying SPH visualization with generalized particle tracing. In contrast to traditional integration-based methods, which require a mesh containing velocity information, our method works with a highly dynamic particle set and only requires the position of each particle. Our approach has the potential to increase performance because it does not require repeated explicit integration of velocity information; we directly use the particle positions from the simulation data to compute pathline trajectories and are therefore independent of concrete velocity information.

3 BACKGROUND AND MOTIVATION

This section establishes the background and motivation for our approach to tracing pathlines in particle-based flow fields. We give an overview of flow field representations and discuss properties of particle neighborhoods as well as challenges arising from locating individual particles. Finally, we give a background on pathline tracing.

3.1 Flow Field Representations

There are two main approaches to represent flow field information: mesh-based and particle-based. The Eulerian (mesh-based) representation stores velocity information at each node of the mesh. To evaluate the velocity field, the velocity vectors at nodes surrounding the desired location must first be gathered and interpolated. Numerical integration methods may then be used to advect a tracer.

In contrast, the Lagrangian representation of a flow field stores information in a reference frame that moves with the flow, commonly in the form of a particle set. At each time step of the simulation, the particle positions are updated, and additional information such as velocity, pressure, and density is stored for each particle. During the simulation some particles may cease to exist because they leave the data set or are removed to maintain a certain particle density, while other particles may enter the data set or are born within the data set. In this paper we make the common assumption that particles remaining in the data set can be identified in consecutive time steps. Without the velocity information, this field representation is identical to a classic particle system.

Smoothed-particle hydrodynamics (SPH) is a grid-free method for the simulation of fluids, originally developed by Gingold and Monaghan [19] for use in astrophysical fluid simulations. In SPH simulations the fluid domain is discretized into a set of Lagrangian particles. These particles move along the flow and carry velocity, mass, and other fluid properties. They also directly indicate local fluid density.

In SPH simulations a smoothing kernel is used to weight the contributions of particles to reconstruct fluid properties anywhere in the simulation domain. With these kernels, hydrodynamic pressure and the discrete equations of motion (e.g., discretized Navier Stokes equations) may be evaluated for any particle in the field. Standard integration techniques—typically Euler integration with a very small step size—are used to displace particles and thus produce fluid motion from the resulting acceleration data.

SPH has gained in popularity due to several advantages over grid-based (Eulerian) simulation techniques. Major advantages include the ease of modeling free surfaces and moving obstacles and the ability to track the evolution of variables (particles). On the other hand, SPH techniques may require extremely small time steps during integration, thus increasing overall computation times in situations where highly accurate and close-to incompressible fluid behavior is required. Due to very small step sizes, results are generally only written to file after several tens to hundreds of integration steps in order to reduce the overall amount of data generated by an SPH simulation.

The smoothing kernel typically takes the form $W(r, h)$, where $r$ is the distance from $P$ to particle $P_i$, and $h$ is the smoothing length of $P_i$, which represents its region of influence. When interpolating using the SPH smoothing kernel, only particles whose region of influence contains the location of $P$ are used in the interpolation. Any quantities
associated with the particle, such as density or velocity, can be interpolated using the SPH smoothing kernel. For incompressible fluid simulations, the smoothing length will generally be uniform across all particles, but there may be regions that are refined for a better resolution. In contrast, with compressible fluid simulations the smoothing length will generally vary per particle over time and is inversely proportional to the particle’s density. Fig. 1 gives an example distribution of particles in a compressible flow field and shows which particles would be used for interpolation. Other particle-based techniques use similar smoothing kernels, such as multivariate Gaussians. We focus on SPH as an example of a typical particle-based flow simulation. However, our technique is applicable to all simulations in which particles can be identified across time steps.

3.2 The Particle Neighborhood

Because the Lagrangian representation of a flow simulation does not give any connectivity information about the particles, it is necessary to construct a spatial representation of the data to quickly find the relevant neighbors of any given location. Since we target compressible flow fields as well, which generally do not have a uniform distribution of particles, we require a non-uniform, balanced data structure for efficient lookup and storage. Storage and memory efficiency requirements are primarily due to the possibility of large numbers of particles being present in the data set. To fulfill these requirements we develop a modified version of a k-d tree. We construct such a k-d tree at each time step because the flow field is highly dynamic and some particles may enter or leave the simulation. In Section 4 we discuss the implementation of this k-d tree and how we use it to locate particle neighborhoods.

3.3 Pathline Tracing

Pathlines are a type of integral line defined in a time-varying flow field. A pathline $P$ is the trajectory of a particle traced through a flow field $v$ which changes over time, given by the following equation:

$$v : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}^3$$

$$p(t) = p_0 + \int_0^t v(p(\tau), \tau) \, d\tau. \tag{1}$$

Evaluation of Eq. (1) requires numerical integration, which can be an expensive process, especially in a particle-based flow field due to repeated scattered data interpolation and neighborhood lookups. This quickly becomes costly when it must be performed multiple times in each time step, as is required for higher order or adaptive integration methods. Instead, we make the observation that a tracer particle’s position can be directly approximated using the trajectories of its surrounding particles. See Fig. 2a for an illustration of this phenomenon. By directly using the spatial information given by the particle set, we do not require velocity information and can avoid costly numerical integration methods.

4 INTERPOLATION-BASED TRACING

In this section we describe our method for advancing tracers in a particle-based flow field. We begin with an overview of the process and then go into the details of tree construction and interpolation using a particle neighborhood.

Given a particle-based flow field description

$$p : \mathbb{N} \times \mathbb{R} \to \mathbb{R}^3, \tag{2}$$

that maps scattered particles $P_i, i \in \mathbb{N}$ to positions in three-space for arbitrary positions in time $t \in \mathbb{R}$, our approach facilitates the tracing of an arbitrary new particle $P$, that is inserted into the flow field at position $p \in \mathbb{R}^3$ and time $t \in \mathbb{R}$.

4.1 Method Outline

Tracing a particle $P$ starting at $(p, t_j)$ takes the following steps:

1) Load particle data of time step $t_j$.
2) Construct the modified k-d tree from the particle set.
3) Locate particle neighborhood for $P$.
4) Load particle data for time step $t_{j+1}$.
5) Construct the modified k-d tree from the particle set.
6) Represent $(p, t_j)$ as a weighted combination of neighbors $(p_i, t_j)$ where $P_i$ is also present in $t_{j+1}$.
7) Reconstruct $(p, t_{j+1})$ based on transformed neighbor positions in $t_{j+1}$.
8) Update the particle neighborhood for $P$ in time step $t_{j+1}$.
9) Repeat from step 4.

In essence, tracers are advected in a two step process. First we determine which of the tracer’s neighbors are still present in the data set. Then we calculate weights of the neighbor particles based on distance from the tracer in the previous time step and use those weights to interpolate the current particle positions to determine the updated position of the tracer in the current time step. In the
following we detail these two relevant steps of the particle tracing procedure.

4.2 Modified k-d Tree

In this section we discuss the use of a modified k-d tree to perform neighborhood queries on the data sets. In order to understand our choice of data structure, we first give a background on k-d trees and other similar spatial data structures. Subsequently, we explain the process of constructing the modified k-d tree.

4.2.1 Tree Requirements and Properties

As discussed in Section 3.2, due to the scattered distribution of particles in the flow field, especially in compressible data sets, using a uniform partitioning of space will not be efficient for performing neighborhood lookups in our data sets. Instead we use a modified version of a k-d tree. Bentley and Friedman [20] discuss various spatial data structures and present the k-d tree as a data structure well suited to range queries, which is very similar to finding all particles in the neighborhood of a given location. Similarly, k-d trees are often used as an acceleration structure in real-time ray tracing because they decrease the number of objects that must be checked for intersection with a ray. Much work has been done in this area to determine the best heuristics for efficiently constructing a k-d tree that gives the fastest lookup times [21], [22]. One of the most popular heuristics for constructing the tree is the surface area heuristic (SAH), which determines the best split by minimizing the cost of the surface areas of the two sides multiplied by the number of children in each side [23]. We use this heuristic to construct our tree because, like ray tracing, we have a large set of irregularly distributed objects that we need to check for intersections. Another type of tree that inspires our modified k-d tree is Garth and Joy’s cell tree [24]. The cell tree is a spatial data structure used to efficiently find cells in an unstructured grid, and has many similarities to interval trees and k-d trees. In our version of the k-d tree we depart from the traditional binary tree and use a third (middle) section for any particle that overlaps a split plane. After sorting all the particles we iterate through each potential split and calculate the cost of the split. The cost function is a modified version of the surface area heuristic [23] to take into account the middle section of the tree:

\[ C(\text{split}) = S A_l \cdot N_l + S A_m \cdot N_m + S A_r \cdot N_r \]  

(3)

where \( N \) is the number of unique particles in each child node, computed using the counts from the buckets as follows:

\[ N'_l = \sum_{i=0}^{s-1} \text{count}(\text{bucket}_l[i]), \]

\[ N'_r = \sum_{i=s}^{n} \text{count}(\text{bucket}_r[i]), \]

\[ N_m = N'_l + N'_r - P_{total}, \]

\[ N_l = N'_l - N_m, \]

\[ N_r = N'_r - N_m. \]

We calculate the surface area of the middle using the previously recorded left and right boundaries for each splitting plane. Once the minimum cost split has been determined the particle IDs are then partitioned into left, middle, and right lists to continue splitting the tree. See Fig. 3 for an illustration of the splitting process. This process is repeated on the child nodes until there are too few particles in a particular branch remaining. The maximum number of particles in a leaf of the tree is a parameter that can be adjusted for optimal performance. The number of buckets is also a parameter to the tree construction algorithm. Having a large number of buckets can help find a better split, but it will also slow down tree construction because more splitting planes will be evaluated. Through experimentation we have found that a maximum of 64 particles and eight buckets is optimal.

Fig. 3. If the solid line is chosen as the optimal split then the white spheres become the left children, the light gray spheres become the right, and the dark gray spheres become the middle children with the dashed lines as the left and right boundaries of the split. The table at the bottom left shows the four buckets for the left and right boundaries. The diagram on the right shows how the tree corresponding to these particles is constructed. We use 2 bits to represent the splitting dimension with 0-2 to represent dimensions and 3 to indicate a leaf. Split is the location of the split plane, and left and right are the locations of the middle node boundaries.

We calculate the surface area of the middle using the previously recorded left and right boundaries for each splitting plane. Once the minimum cost split has been determined the particle IDs are then partitioned into left, middle, and right lists to continue splitting the tree. See Fig. 3 for an illustration of the splitting process. This process is repeated on the child nodes until there are too few particles in a particular branch remaining. The maximum number of particles in a leaf of the tree is a parameter that can be adjusted for optimal performance. The number of buckets is also a parameter to the tree construction algorithm. Having a large number of buckets can help find a better split, but it will also slow down tree construction because more splitting planes will be evaluated. Through experimentation we have found that a maximum of 64 particles and eight buckets is optimal.
The tree is stored in a very compact memory-efficient representation. Only the particle IDs are stored in the tree; the actual particle data is stored separately when it is loaded from file to avoid excessive copying and duplication of data. The diagram on the right of Fig. 3 details the storage layout of our tree. For a compact representation, we store pointers to all of the nodes in the tree in a list and only store the index of the left child in the parent node. All children of a particular node are stored contiguously in this list. Similarly, we do not store the particle indices in the leaves of the tree. Instead we store a sorted list of the particle IDs, and in the leaf we store the index of the first particle and the total number of particles for that leaf. The layout of a node is as follows:

```c
struct Node{
    unsigned int left_child : 30;
    unsigned int dimension : 2;
    union{
        Leaf l;
        InternalNode n;
    }
};
```

The internal node stores the value of the split plane and the left and right boundaries of the middle child. The leaf node stores the index of its first particle and the number of particles. To query for all particles influencing a certain position in space, the tree is traversed in the following way: At each node we compare the location to the split plane and the left and right boundaries of the middle section. If the location is greater than the split plane, traverse the right child, else traverse the left child. If the location is between the left and right boundaries of the middle, also traverse the middle child. Continue in this manner until a leaf is reached, then check the particles in the leaf for intersection. This traversal is more localized than traversal in binary interval trees, where often both branches have to be traversed completely due to large overlaps between nodes.

### 4.3 Neighborhood Interpolation and Reconstruction

The particle tracing technique is a spatial interpolation and reconstruction problem. Given a set of known particle neighbors \( P \), with positions \( p_i \) and a current tracer position \( p \), the goal of the reconstruction process is to approximate \( p \) in the next time step, taking a deformation of the particle neighborhood into account. Our interpolation approach uses weighted averaging with SPH weighting kernels to reconstruct particle positions. However, other scattered data interpolation methods such as Moving Least Squares [26] can be used as well, depending on the data set or simulation requirements.

Given a tracer particle \( P \) with position \( p(t_j) \) and surrounding particles \( P_i \), we assign weights \( w_i \) to each \( P_i \) based on the distance from \( p(t_j) \) to \( p_i(t_j) \) to use for interpolation. Thus, we effectively represent \( P \) as a weighted combination of neighbor positions, resulting in a set of weights \( w_i \):

\[
p(t_j) = \sum w_i p_i(t_j).
\]

In this weighted averaging approach, the position of the tracer particle \( P \) in time step \( t_{j+1} \) is simply reconstructed by using these same weights \( w_i \) with updated neighbor positions \( p_i(t_{j+1}) \) as follows:

\[
p(t_{j+1}) = \sum w_i p_i(t_{j+1}).
\]

For typical weighting functions, this updated position lies within the convex hull of its neighbors. We use SPH smoothing kernel weighting functions to obtain the weights \( w_i \) for the interpolation. This weighted averaging method most closely resembles the SPH interpolation method used in the simulation, leading to consistent results during these post-processing steps. For compressible SPH simulations, the following equation used by Price [14] assigns weights to each particle:

\[
W(r, h) = \frac{m}{\rho h^3} \begin{cases} 
1 - \frac{3}{2}q^2 + \frac{1}{4}q^3 & \text{if } 0 \leq q < 1, \\
\frac{1}{2}q & \text{if } 1 \leq q < 2, \\
0 & \text{if } q \geq 2,
\end{cases}
\]

where \( r \) is the distance from \( P \) to \( P_i \), \( m \) is the mass of \( P_i \), \( \rho \) is the density of \( P_i \), \( h \) is the smoothing length of \( P_i \), and \( q = r/h \). As shown in this SPH kernel function only particles within two smoothing lengths of \( P \) contribute to the interpolation. Since particles in compressible SPH simulations can have varying densities, this kernel uses the density as well as the distance from \( P \) to \( P_i \) to calculate the weight. For incompressible SPH simulations, Müller et al. [27] use the following equation to assign particle weights:

\[
W(r, h) = \frac{15}{\pi h^6} \begin{cases} 
(h - r)^2 & \text{if } 0 \leq r \leq h, \\
0 & \text{if } r > h,
\end{cases}
\]

where \( r \) and \( h \) are defined as in the previous kernel. In this SPH kernel function only particles within one smoothing length of \( P \) are considered in the interpolation. The data sets used throughout this paper use one of these two weighting functions.

### 4.4 Neighborhood Update

After advecting the tracer to the next time step \( t_{j+1} \) we update its neighborhood information. This is done by performing a lookup in the \( k \)-d tree constructed for \( t_{j+1} \) to find all particles whose radii overlap the tracer’s current position. To maintain an accurate representation of the surrounding flow, it is necessary to update the neighborhood at each time step because in areas of converging or diverging flow some particles may leave the neighborhood while others may enter the neighborhood. See Fig. 2b for an illustration of neighborhood change. The neighborhood may also change due to particle birth and death in a particular time step. Tracer particles only need to store a list of their current neighbors and can discard references to previous lists of neighbors.

### 4.5 Method Extensions

In this section we discuss modifications to the basic method to improve accuracy and performance.
due to our assumption in Eq. (4). This equation can be expanded to
\[ p(t_{j+1}) = \sum w_i (p_i(t_{j+1}) - p_i(t_j)) + \sum w_i p_i(t_j). \]

We then make the observation that we can replace the second term of Eq. (8) with \( p(t) \) due to our assumption in Eq. (4) that the position of the tracer \( P \) in the current time step can be represented by a weighted sum of its neighbor positions. This gives us a new representation of the updated position of the tracer \( P \) as
\[ p(t_{j+1}) = \sum w_i (p_i(t_{j+1}) - p_i(t_j)) + p(t_j). \]
which resembles a simple velocity-based advection step using the movement of the neighboring particles as a pseudo-velocity. Unlike the velocities used in traditional integration-based methods, which are derivatives over time, i.e., linearizations of particle motion that are only guaranteed to be accurate for very short time frames, these pseudo-velocities represent the average velocity of the particle over the whole time step. Thus, we effectively replace \( v_i \) as provided by the simulation and used in classic numerical integration, with \( \frac{dp_i}{dt} \approx \frac{p_i(t_{j+1}) - p_i(t_j)}{\Delta t} \). We are therefore able to use a quantity that describes actual particle displacement for the given time span. Eq. (9) is equivalent to the advection in Eq. (5) when all of the neighboring particles in time step \( t_j \) are also present in the simulation in time step \( t_{j+1} \). However, if some of the neighboring particles are not present in \( t_{j+1} \), Eq. (5) becomes inaccurate because we can no longer rely on the assumption in Eq. (4) that the previous position of the tracer can be exactly represented by the weighted combination of its neighbors. Using this new representation of the advection step gives us the ability to extrapolate the particle position in the presence of neighborhood loss because the tracer particle is no longer restricted to lie in the convex hull of its remaining neighbors.

### 4.5.1 Modified Weighted Averaging Interpolation

The weighted averaging method of interpolating particle positions works well if all of the particles in the neighborhood are present in the simulation at \( t_{j+1} \). When some particles leave the domain of the data set this method starts to break down because the assumption that the tracer position can be represented exactly by a weighted combination of its neighbors only holds true if all neighbors are present in both time steps. The current equation for the tracer position in \( t_{j+1} \) is given by Eq. (5).

\[ p(t_{j+1}) = \sum w_i (p_i(t_{j+1}) - p_i(t_j)) + \sum w_i p_i(t_j). \]

We then use linear interpolation to reconstruct the tracer position in the time steps that were skipped.

### 4.5.2 Parallelism

Tree construction is the most time consuming part of this algorithm when advecting small numbers of particles. To speed up the tree generation we parallelize the construction of the three child nodes since they are independent of each other. We use OpenMP and restrict the spawning of new threads to only the top two levels of the tree to avoid exponential growth of threads. We allocate one thread to the construction of each of the child nodes. The other area of the program that gains a significant speed-up from parallelism is advection. When tracing large numbers of particles we advect them in parallel.

### 4.5.3 Super- and Sub-Sampling

The ability to identify particles over multiple time-steps allows us to perform super- and sub-sampling during advection. Because we can track particles across time steps, it is not necessary to use every time step in the simulation to trace new pathlines. If only a rough approximation of the trace is required, we can limit neighborhood lookups and updates to fewer time-steps by only using every nth time step. In this case the pathlines will not be as smooth visually, but the average error of the pathlines at the time steps used for advection is no greater than if all time steps were used. Fig. 4 illustrates the process of skipping time steps using our method compared to a velocity integration-based method. This phenomenon is evaluated in detail in the results section.

In addition to skipping entire time steps we can also perform adaptive subsampling. As a heuristic, we measure the angle between the last three positions of the tracer, and if it is within \( \epsilon \) of 180 degrees, where \( \epsilon \) is a parameter to the sub-sampling, then we consider the pathline to be linear and allow skipping of time-steps (Fig. 5a). To skip from time step \( t \) to time step \( t+k \) we use the weights \( w_i \) of the neighboring particles found in time step \( t \) and the positions of the particles in time step \( t+k \) just as in Equation (5) but with \( t+1 \) replaced by \( t+k \). We then use linear interpolation to reconstruct the tracer position in the time steps that were skipped.

Super-sampling can improve accuracy of the tracing process if the neighborhood of a tracer changes drastically between time steps. If the neighborhood is updated multiple
This data set is a predicate that evaluates to 0 if \( U \). This data set is an \( \sum w_i \cdot U(P_i) > 0.5 \), \( i \)

where \( w_i \) is the weight assigned by the SPH kernel function and \( U \) is a predicate that evaluates to 0 if \( P_i \) has left the data set or was destroyed and 1 otherwise. This inequality assigns weights to each neighbor particle based on their distance from the tracer with closer particles receiving greater weight according to the SPH weighting kernel. We sum the weights of the particles remaining in the data set and divide by the total weight to get a percentage of the neighborhood that still exists. We have chosen 0.5 as the threshold because if less than half of the most significant neighbors remain then it is highly likely that the tracer would have also left the simulation boundaries in the next time step. A simpler version of Inequality 10 would be to use the percentage of neighbors remaining without weighting them, and this gives very similar results. Fig. 7 shows the original version on the left and the version with neighborhood loss handling on the right.

5 Results

We evaluate our method using multiple SPH data sets. For each data set we present timing results for advecting various numbers of tracer particles and show images of the pathlines we traced. We also compare the accuracy of tracing pathlines with our method to tracing pathlines using adaptive Runge Kutta 4/5 integration [28]. All of the calculations were run on a MacBook Pro running Windows 7 with an Intel Core i7-2820QM processor running at 2.30 GHz with 8 GB of main memory.

Heating simulation (Fig. 8). This data set is an incompressible fluid simulation with 124,768 particles over 184 time steps. The particles remain in the data set for the duration of the simulation. This data set contains a heat source that runs across the bottom of the data set. The particles nearest the heat source move up as they become hotter and then fall back down as they cool. Particles to the sides of the heat source exhibit circular flow patterns.

Blender mixing simulation (Fig. 9). This data set is an incompressible fluid simulation with 131,072 particles over 240 time steps. All particles remain in the data set throughout the simulation. There are two mixing blades in the data set, one on the left and one on the right, that rotate in the same direction about the y-axis. The blade rotations cause the particles to form two counterclockwise spirals around the blades. Particles in between the two blades exhibit diverging flow.

Collimated jet simulation (Fig. 10). This data set is a compressible astrophysical SPH simulation of the production of collimated jets and has 40 time steps. This simulation starts with 1,479,440 particles and ends with 854,829 particles. The particles in this simulation have varying densities and non-uniform smoothing lengths. Starting in the second time step there is a sink in the center of the data set. Initially the particles all start moving towards the sink, and later they reverse direction and fan out towards the edges. Particles that come in contact with the sink die and in subsequent time steps are no longer part of the active particle set. There are also some particles that exit the data set on the sides, and new particles also enter from the sides.
5.1 Performance Analysis

For particle-based flow fields, performing integration-based particle tracing methods can be costly. To advect a particle $P$ at location $p$ requires first determining the velocity at $p$. This is done by finding the neighborhood of particles $P_i$ surrounding $P$ in time step $t_j$ and $t_{j+1}$, interpolating the velocities of the $P_i$ in both time steps and then linearly interpolating between these two velocities. This process is repeated multiple times per time step depending on the complexity of the flow; for methods such as adaptive Runge-Kutta this can be hundreds of times per time step. In contrast, our method requires only one neighborhood lookup and one interpolation per time step. Fig. 11 shows a comparison of the time it takes to advect tracers using our method compared to Runge-Kutta 4/5 integration. Our method is a factor of 10 times faster when tracing large numbers of particles. Both our method and the Runge-Kutta 4/5 implementation use the modified k-d tree to find neighbors for interpolation. We show both serial and parallel tracing for comparison. For the collimated jet data set, we trace pathlines in a box formation centered around the sink. The number of particles traced varies from 9,261 particles to 531,441 particles. For the mixing simulation we trace pathlines in a box centered between the two blades. For the heat simulation we trace particles in a box near the heat source. For both the mixing simulation and the heat simulation the number of particles traced varies from 9,261 particles to 357,991 particles. In summary, our method greatly reduces the number of times the particle system needs to be accessed to perform tracer advection. In our experiments this led to an average speed up of a factor of 10 to 30 depending on the data set.
5.2 Accuracy Analysis

5.2.1 Definition of Error

From their mathematical definition, the error order of numerical integration techniques can be computed readily for any given integration step size. Adaptive Runge-Kutta 4/5 integration schemes make use of these error estimates to control integration step-size and produce highly accurate results with respect to the given velocity information. As argued throughout this paper, however, the velocity information provided by particle-based flow fields is not a faithful representation of particle transport over longer periods of time. In the following section we evaluate accuracy not by examining how well a particle trajectory follows velocity data (i.e., how well it approximated the continuous integral), but by comparing ground truth transport data as given by particle positions with solutions provided by the advection methods under investigation.

5.2.2 Ground Truth

We compare the error of tracing with our method to the error of tracing with Runge-Kutta 4/5 on the same data set. We define error as the ability of each method to correctly reproduce the trajectories of particles from the simulation when seeding tracers at those locations. We believe this to be a valid error metric because the simulation particles are the best representation of ground truth data that we have available. Due to the nature of the SPH simulation process, it is difficult to produce a simulation where ground truth data is known for any arbitrary location in the domain of the data set. Furthermore, we cannot simply remove selected particles from the data set because in an SPH simulation the smoothing kernel only gives an accurate reconstruction when all particles in the kernel radius are present. In contrast, with Eulerian representations of flow fields, a velocity field can be defined such that it is possible to analytically determine a particle trajectory for any given seeding location.

5.2.3 Evaluation

We show the results of this analysis as well as the error when skipping time steps in Fig. 12. In Fig. 13 we show the distribution of error among pathlines for each method. For many simulations, it may be infeasible to output every time step of the simulation due to storage constraints. It is also common in flow simulations to use a very small time step size during the simulation to ensure convergence to a robust solution and satisfy physical constraints such as incompressibility but then only output every nth step of the simulation. When using fewer time steps, our method is still able to accurately reproduce pathlines with no additional error. This is because we only rely on particle positions, which are still accurate even when intermediate time steps are missing. However, numerical integration methods are not able to reproduce pathlines accurately when time steps are missing because they rely on first derivatives, i.e., velocities, to approximate changes in particle position. With coarser time steps these intermediate velocities can become very inaccurate when the flow behavior is non-linear leading to a less accurate reconstruction of particle trajectories. (See Fig. 4.) The results given in these graphs support these theoretical considerations, as our method is shown to consistently outperform classic Runge-Kutta 4/5 integration with respect to the definition of error given previously. An interesting observation is that skipping time steps (sub-sampling) does not automatically lead to an increase in error when tracing particles with our method. Often, the more stable behavior of neighborhood weights \( w_i \) during reconstruction, as caused by time step skipping, even leads to a slight reduction in error for some data sets. This stands in stark contrast to the performance of integration-based tracing in these scenarios.

We also compare the error when using the proposed method extensions. For the collimated jet data set, we show both our original weighted averaging technique and the weighted averaging technique as described in Section 4.5.1 as well as with and without the neighborhood loss handling described in Section 4.5.4. In the other data sets all of the original particles exist throughout the simulation, and no particles are created during the simulation, so these two modifications to our method do not affect the results. As shown in Fig. 14, all versions of our method have smaller average error than the adaptive Runge-Kutta 4/5 advection.

For this error comparison we filter out the error due to pathlines that continue in the data set despite the reference
We feel this gives an unbiased comparison since our method has the ability to detect significant neighborhood loss and terminate those pathlines whereas adaptive Runge-Kutta 4/5 does not, which would reduce the accuracy of adaptive Runge-Kutta 4/5 in reproducing those pathlines. We also show how well our neighborhood loss handling works for detecting when a pathline should terminate. As shown in Fig. 15, when using our modified weighted averaging in combination with the neighborhood loss handling, we are able to detect a very high portion of the pathlines that terminate and have a small percentage of false positives.

The neighborhood loss handling performs better with the modified weighted averaging because the advection error is reduced so the neighborhood is closer to the true neighborhood. In Fig. 16 we show a visual comparison of tracing using our method compared to using adaptive Runge-Kutta 4/5. In the collimated jet data set, the higher error pathlines are concentrated in the flow from the center to the outside. As indicated in the chart above, the image shows that our method has fewer pathlines that continue after the reference pathline has terminated. In the heating simulation the pathlines with the highest error are...
clustered towards the center above the heating strip. The highest error pathlines for the mixing simulation occur in the spirals around the two blades.

5.3 Tree Analysis

The time it takes to construct the modified $k$-d tree is fixed for each time step no matter how many particles we trace. The construction time depends only on the number of simulation particles and their distribution. Fig. 17 shows the time it takes to construct the tree at each time step for each of our data sets.

We compute branching statistics on the modified $k$-d tree to demonstrate its advantage over a traditional $k$-d tree for particle-based flow fields. For particle advection in the collimated jet data set the middle branch is chosen a little over 50 percent of the time the lookup algorithm must choose a branch. In the modified $k$-d tree the middle branch is taken along with either the left or the right branch. However, in a traditional $k$-d tree in this situation both the left and right branch would need to be traversed which does not eliminate any comparisons at a higher level in the tree. For this matter, we use the middle branch to estimate branching complexity as compared to a traditional $k$-d tree. In Fig. 18 we show the percentage of time the middle branch is taken out of all branching decisions during neighborhood lookups. These results show that 30 to 50 percent of the time in a traditional $k$-d tree both branches of the tree would need to be traversed whereas in the modified $k$-d tree only the middle branch and one of the other two branches has to be traversed.
simulation, which is likely due to the moving holes in the flow field caused by the changing blade locations.

6 Conclusion

In this paper we have presented a new method for tracing pathlines in particle-based flow fields such as SPH simulations. This method directly uses the particle trajectories to compute pathlines seeded at arbitrary locations and works for both compressible and incompressible flow fields. Our method provides an alternative technique for the extraction of trajectories in particle-based flow fields which avoids costly numerical integration methods in favor of direct interpolation. The results show that such velocity-free methods are able to capture fluid transport accurately. In the future we would like to extend this method to be implemented on the GPU to make the tree construction and particle advection faster by taking advantage of the inherently parallel nature of these two steps of the method. Because our method allows for advection of tracers at any arbitrary location in the flow we could extend this method to other types of integral lines such as streak lines or time lines and to integral surfaces such as path surfaces.

Acknowledgments

This work was supported in part by the National Science Foundation under contracts IIS 0916289 and IIS 1018097, and NSF GRFP grant DGE-1148897, and the Office of Advanced Scientific Computing Research, Office of Science, of the US Department of Energy under Contract No. DE-SC0007443 through the Scientific Discovery through Advanced Computing (SciDAC) programs Scalable Data Management, Analysis and Visualization Center (SDAV). We thank the members of IDAV for their support. The authors thank Daniel Price for providing the compressible SPH data set.

References


Jennifer Chandler received the bachelor's degree in computer science from the University of California, San Diego, in 2011. She is currently working toward the PhD degree in the Institute for Data Analysis and Visualization (IDAV) at the University of California, Davis. Her research interests lie in scientific visualization and GPU computing related to particle-based flow fields.
Harald Obermaier received the master’s and PhD (Dr rer nat) degrees in computer science from the University of Kaiserslautern, Germany, in 2008 and 2011, respectively. He is currently a postdoctoral researcher at the University of California Davis. His research interests lie in visualization, data analysis, and computational science with applications in areas such as continuum mechanics and earth and environmental sciences.

Kenneth I. Joy received the BA and MA degrees in mathematics from UCLA in 1968 and 1972, respectively, and the PhD degree from the University of Colorado, Boulder, in 1976. He is a professor with the Department of Computer Science and the director of the Institute for Data Analysis and Visualization at University of California, Davis (UC Davis). He came to UC Davis in 1980 in the Department of Mathematics and was a founding member of the Computer Science Department in 1983. He is a faculty computer scientist at Lawrence Berkeley National Laboratory and is a participating guest researcher at Lawrence Livermore National Laboratory. His areas of interest lie in the fields of visualization, geometric modeling, and computer graphics, where he leads research efforts in multiresolution representations of large-scale data sets, visualization of multidimensional data, applications and visualization algorithms to imaging problems, and simplification of data sets resulting from terascale simulations. He is a member of the Association for Computing Machinery (ACM) and the IEEE.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.