Volume 22 (2003), Number 2 pp. 1–9

Local Exact Particle Tracing on Unstructured Grids

Peter Kipfer and Frank Reck and Günther Greiner

Computer Graphics Group, Department of Computer Science, University of Erlangen

Abstract

For analyzing and interpreting results of flow simulations, particle tracing is a well established visualization method. In addition, it is a preliminary step for more advanced techniques like line integral convolution.
For interactive exploration of large data sets, a very efficient and reliable particle tracing method is needed. For wind channel experiments or flight simulations, large unstructured computational grids have become common practice. Traditional approachs, based on numerical integration methods of ordinary differential equations however fail to deliver sufficiently accurate path calculation at the speed required for interactive use.
In this paper we extend the local exact approach of Nielson and Jung in such a way that it can be used for interactive particle tracing in large data sets of steady flow simulation experiments. This will be achieved by sophisticated preprocessing using additional memory. For further visual enhancement of the streamline we construct an implicitly defined smooth Bézier curve that is used for ray tracing. This allows us to visualize additional scalar values of the simulation as attributes to the trajectory and enables the display of high-quality smooth curves without creating any visualization geometry and providing a good impression of the spatial situation at the same time.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Line and curve generation; I.3.7 [Computer Graphics]: Raytracing; G.1.2 [Numerical Analysis]: Spline and piecewise polynomial approximation

1. Introduction

In many scientific areas as well as in technical applications, Computational Fluid Dynamics (CFD) is of central importance. The result of a fluid dynamics simulation are data sets which usually describe the flow behavior in a 3D-environment. To understand the characteristics of the simulated process, a meaningful visualization of the data is necessary. The raw data are defined on a discrete structure, a grid, which is build up out of many cells. In the nodes or vertices of the grid the velocity (a vector field) and other simulation data, e.g. pressure, density, energy (scalar values) are stored.

One of the most common methods of acquiring knowledge of flows is the use of particle tracing [9, 11, 7, 8, 4, 3]. It is also the basis of many other visualization techniques such as streak-lines, streak-ribbons, time-surfaces or line integral convolution (LIC) [1]. The particle tracing visualization method shows the trajectory of one mass-less particle in the flow. The trajectory is obtained by the integration of the ordinary differential equation corresponding to the vector field.

The integration is usually done numerically. In this paper we describe another approach for vector fields defined on a tetrahedral mesh, i.e. an unstructured grid in which all cells are tetrahedrons. It is a modification of the *exact integration method* introduced in [8]. This method traverses every cell in a single step. Our modification comprises a sophisticated preprocessing of the data, which results in a classification of the cells and provides for each cell sufficient information to perform the exact integration very fast. We determine the exact exit points of the particle for every cell that is traversed by the particle. The particle trace is the polyline of the computed points or a interpolating curve. In contrast to all the numerical integration schemes, this approach guarantees that the local errors will not accumulate, except for floating point precision errors. Hence it is globally very accurate compared to methods built on numeric integration. See [6] for a comparison of achievable accuracy for numeric integration schemes.

As the starting point of the particle trace is selected by the user, the method must be fast enough to be usable in interactive applications. At the same time it may not produce incorrect results. Once a characteristic streamline is found, the user may wish to display it most faithfully to the phys-

ical environment. Here, more expensive rendering methods are acceptable. The streamline however must stay the same, so switching numeric methods is prohibitive as it may result in particle traces shaped differently. We present a visualization strategy that offers interactive rendering as well as high-quality ray tracing output based on the same particle path building routine.

The paper is organized as follows: In the next section we review the existing methods for particle tracing in unstructured grids. At first we briefly describe the methods based on numerical integration of ordinary differential equations and then in a more detailed way the local exact method. In Section 3 we outline our modifications of the local exact method, describe the preprocessing step in detail and give pseudocode for the algorithm. The following section details our approach of high-quality rendering of the obtained streamline in a physically motivated and visually pleasant way.

2. Particle Tracing in Tetrahedral Grids

Particle tracing in a vector field $\vec{\mathbf{v}}(\vec{x})$, considered to be the velocity field, amounts to solving the initial value problem for an ordinary differential equation (ODE)

$$\frac{d\vec{x}(t)}{dt} = \vec{\mathbf{v}}(\vec{x}(t)) \qquad \vec{x}(t_0) = \vec{x}_0 \tag{1}$$

Where $\vec{x}(t)$ represents the position of the particle at the time t, starting at time t_0 at position $\vec{x}_0$.

The vector field $\vec{\mathbf{v}}$ is usually given at discrete sample points, usually at the vertices of a grid. In this paper we discuss only unstructured tetrahedral grids in Euclidean 3D-space. The flow data is the result of a numerical simulation and the mesh is the finite element grid of the simulation.

2.1. Numerical Integration Methods

The standard way to solve the ODE is numerical integration. This topic is well investigated and the different methods [10, 2] are characterized as: single-step or multi-step methods, single-stage or multi-stage methods, explicit or implicit methods with constant or variable step-size. We only outline the necessary preparations to apply such an integration scheme. The basic algorithm is:

```
cell = mesh.findCellAt(initialPosition);

while (particle.inGrid()) {
  // interpolation
  v = cell.velocityAt(currentPosition);
  // integration
  newPosition = calculateNewPosition(v,cell);
  // point location
  cell = mesh.findCellAt(newPosition);
}
```

By this procedure we get a sequence of sample points $\vec{x}_n = \vec{x}(t_n)$ of the path line, approximating the position of the particle at time t_n. The polyline connecting these points is an approximation of the path line of the particle.

The calculation of the new position is performed by an integration step. The simplest integration scheme is Euler's method:

$$\vec{x}_{n+1} = \vec{x}_n + (t_{n+1} - t_n) \cdot \vec{\mathbf{v}}(\vec{x}_n)$$

Runge-Kutta methods ($\mathbf{RK}_p$) are known to be more accurate. The order p of a Runge-Kutta method denotes the number of internal interpolations and therefore also the order of the local approximation error. Higher order means greater exactness, but also more processing time. Another factor which influences the correctness and time behavior is the step size. The shorter the step size in a particle trajectory calculation, the higher the number of steps to be executed and the longer the duration of the process. At the same time, the error will be smaller. An approach to overcome this is the use of adaptive methods, which influence their step size according to a user given error threshold. These methods aren't free from problems like measuring the error, specifying meanigful error thresholds and adequate step sizes.

Another time consuming factor in this approach is the point location, which is necessary after each integration step. This can be done easily in rectilinear grids, but in irregular grids a probably non-local cell search is needed for every evaluation. The method described in the next section does not suffer from these problems.

2.2. Local Exact Integration

This method has been introduced by Nielson and Jung [8]. The basic observation is the following: We know the vector field only at the vertices of the tetrahedral mesh. For the values in-between we have to interpolate the values at the vertices. The simplest and most efficient way is linear interpolation. That is we consider the vector field as piecewise linear mapping. Since for linear mappings $\vec{\mathbf{v}}$ the ODE equation (1) can be solved analytically for each tetrahedral cell, we can determine the exact path-line for the linearly interpolated vector field within the tetrahedron.

In fact, we can interpolate the vector field using a linear system (see [7])

$$\vec{\mathbf{v}}(\vec{x}) = A\vec{x} + \vec{o}$$

with a square matrix A and a translation vector $\vec{o}$. The analytic solution of equation (1) according to [8] is

$$\vec{x}(t) = e^{(t-t_0)A}\vec{x}_0 + (e^{(t-t_0)A} - Id)A^{-1}\vec{o} \tag{2}$$

where the exponentials of the matrices are given by the usual series expansion:

$$e^{(t-t_0)A} = \sum_{k=0}^{\infty} \frac{(t-t_0)^k}{k!} A^k$$

This can be calculated much faster if the normal form of the matrices is known, e.g. if A has three (different) real eigenvalues $\lambda_1, \lambda_2, \lambda_3$. With the transformation matrix formed by the eigenvectors $\vec{b}_1, \vec{b}_2, \vec{b}_3$, we have

$$A = \underbrace{\begin{pmatrix} | & | & | \\ \vec{b}_1 & \vec{b}_2 & \vec{b}_3 \\ | & | & | \end{pmatrix}}_{S} \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix} \underbrace{\begin{pmatrix} | & | & | \\ \vec{b}_1 & \vec{b}_2 & \vec{b}_3 \\ | & | & | \end{pmatrix}^{-1}}_{S^{-1}} \quad (3)$$

In the case of complex eigenvalues, this representation is also valid. However, since calculations with real numbers are simpler we rather consider a "real normal form" in this case. Assuming that the real matrix A has complex eigenvalues, then these appear as a conjugate pair: $\lambda_1 = \sigma + i\tau$, $\lambda_2 = \sigma - i\tau$, $(\tau > 0)$ and a real eigenvalue λ_3. The transformation to the "real normal form" of A is given by

$$A = \underbrace{\begin{pmatrix} | & | & | \\ \vec{b}_1 & \vec{b}_2 & \vec{b}_3 \\ | & | & | \end{pmatrix}}_{S} \begin{pmatrix} \sigma & \tau & 0 \\ -\tau & \sigma & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix} \underbrace{\begin{pmatrix} - & \vec{b}_2 \times \vec{b}_3 & - \\ - & \vec{b}_3 \times \vec{b}_1 & - \\ - & \vec{b}_1 \times \vec{b}_2 & - \end{pmatrix}}_{S^{-1}} \quad (4)$$

In this case $\vec{b}_1$ and $\vec{b}_2$ are the real and the imaginary part of the eigenvector corresponding to $\lambda_{1/2} = \sigma \pm i\tau$, and $\vec{b}_3$ is the eigenvector corresponding to λ_3.

Using the normal forms of equation (3) and equation (4) respectively, the exponentials of A can be determined as follows:

$$e^{\tilde{t}A} = S \begin{pmatrix} e^{\lambda_1 \tilde{t}} & 0 & 0 \\ 0 & e^{\lambda_2 \tilde{t}} & 0 \\ 0 & 0 & e^{\lambda_3 \tilde{t}} \end{pmatrix} S^{-1}$$

and

$$e^{\tilde{t}A} = S \begin{pmatrix} e^{\sigma \tilde{t}} \cos(\tau \tilde{t}) & e^{\sigma \tilde{t}} \sin(\tau \tilde{t}) & 0 \\ -e^{\sigma \tilde{t}} \sin(\tau \tilde{t}) & e^{\sigma \tilde{t}} \cos(\tau \tilde{t}) & 0 \\ 0 & 0 & e^{\lambda_3 (\tilde{t})} \end{pmatrix} S^{-1}$$

where $\tilde{t}$ stands for $t - t_0$. Then the analytic solution of equation (2) is in the real case

$$\vec{x}(t) = S\Lambda(t)S^{-1}\vec{x}_0 + S\Delta(t)S^{-1}\vec{o} \quad (5)$$

with the diagonal matrices $\Lambda(t)$ and $\Delta(t)$ given in the real case by

$$\Lambda(t) = \begin{pmatrix} e^{\lambda_1 \tilde{t}} & 0 & 0 \\ 0 & e^{\lambda_2 \tilde{t}} & 0 \\ 0 & 0 & e^{\lambda_3 \tilde{t}} \end{pmatrix}$$

$$\Delta(t) = \begin{pmatrix} \frac{e^{\lambda_1 \tilde{t}} - 1}{\lambda_1} & 0 & 0 \\ 0 & \frac{e^{\lambda_2 \tilde{t}} - 1}{\lambda_2} & 0 \\ 0 & 0 & \frac{e^{\lambda_3 \tilde{t}} - 1}{\lambda_3} \end{pmatrix}$$

and in the complex case by a slightly more complicated form.

So far we have assumed that all eigenvalues are different from each other and non-zero (otherwise A^{-1} does not exist). Concerning the first restriction, we point out that this is the generic case. When choosing randomly any tetrahedron and assigning randomly vectors to its vertices, the probability of obtaining identical eigenvalues will be zero. And our examples coming from a simulation also show, that this situation rarely occurs. Actually, the second restriction (non-zero eigenvalues) was only needed to derive the result more easily. The final formulas make sense and are also correct for zero eigenvalues. In this case in $\Delta(t)$ the term $\frac{e^{\lambda(t-t_0)}-1}{\lambda}$ has to be replaced by $t - t_0$.

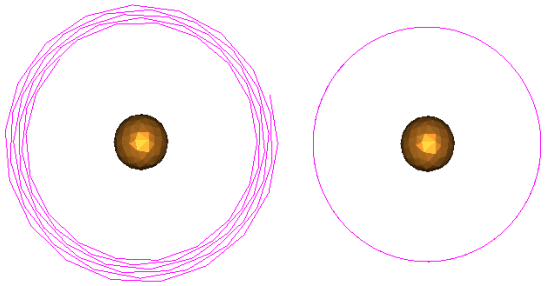


Figure 1: *The Runge-Kutta method of order 2 compared to the local exact method, for a circular flow field.*

Nielson carefully analyzes seven other cases: zero eigenvalues, identical eigenvalues, multiple eigenvectors, and identical eigenvalues of a single eigenvector. He displays the path by sampling the exact solution with suitable step size.

Comparison of integration methods Figure 1 shows a visualization of the achievable accuracy of the $\mathbf{RK}_2$ integration method and the exact integration. The $\mathbf{RK}_2$ integration was run using the mean step size of the exact method. For achieving the same accuracy, the $\mathbf{RK}_2$ method would need much more and smaller steps. The differences can be briefly summarized:

- The local exact method gives the exact solution, provided that the numerical errors are neglected, and one assumes that the underlying linear interpolation of the vector field is a valid method.
- For the local exact method no point location is necessary. In fact, knowing the neighbors of a tetrahedron, the next tetrahedron to be processed is given by the location of the exit point.

- Using the local exact method one does not have to specify the step size, nor does one have to take care of (adaptively) modifying it. The adaptation is built-in: For simulation, typically in numerically critical areas or areas of special importance, one has a fine mesh of tetrahedrons. In these areas, the path segments produced by the local exact method are then small as well.
- The local exact method does need some additional memory. However the memory requirement can be scaled from storing all preprocessing data, to computing the preprocessing data on-the-fly for each tetrahedron visited. The data may be cached for reuse or discarded immediately.

3. Our Approach

Our approach is a modification of the method of Nielson and Jung. It differs in four ways:

- A **different classification** of the cells is used.
- An eigenvalue analysis is performed as **preprocessing** and the results are stored.
- The **Newton method** is employed to find the exit point of a cell of the path line.
- The **curve tangents** are reused to build a smooth spline curve for the rendering process.

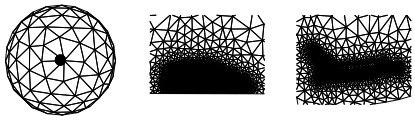
When the path line enters a cell, we determine the intersection point of the exact solution and the exit face of the tetrahedron. The exit point is the entry point to the following cell, where the method is repeated. The final particle path is the concatenation of the intersection points.

3.1. Classification of Each Cell

In contrast to [8], we do not pay much attention to all possible special cases: We classify the tetrahedra as normal cells, parallel cells or extraordinary cells.

- **Parallel cells** are cells whose vector field does not change its direction, i.e. all four velocity vectors point in the same direction. Typically ten percent of all cells belong to this group.
- **Normal cells** have three different eigenvalues as the linear part of the (linearly interpolated) vector field. In addition the critical point of the vector field is located outside the tetrahedron.
- **Extraordinary cells** are all the remaining ones. Typically less than 0.5 % of all cells are of this type. Table 1 compares three randomly selected CFD data sets.

When the path line enters a normal cell we determine the exit point as the intersection of the exact solution given by equation (2) with the exit face. In contrast to Nielson and Jung who use an incremental method for finding the exit point, we use Newton's iteration method. It exhibits quadratic convergence for finding roots. We need only 4 interations on average to achieve reasonable accuracy. Additionally, we can avoid the cell search, because our mesh handling subsystem can return the neighboring cell according to the exit face in constant time. The exit point is the next entrance point for the neighboring cell.



# of cells	Sphere	Car	Shuttle
total	8,193	457,874	1,058,785
extraord.	15	1,247	3,232
preproc. time	5.4 sec	112.8 sec	219.2 sec

Table 1: *Number of extraordinary cells.*

When entering a parallel cell we compute the exit point by a classic ray-plane intersection calculation with all planes of the tetrahedron. In the case of an extraordinary cell we switch to a Runge-Kutta integration scheme, until the cell search routine detects a new cell. We like to point out that this case occurs very seldom as mentioned above.

Note that with our approach, only extraordinary cells need to traversed using multiple steps. Therefore the vast majority of cells is traversed in a single step using highly accurate integration.

3.2. Preprocessing for Normal Cells

In a first step we have to determine the eigenvalues and eigenvectors of the tetrahedron that build its eigenspace. For doing the eigenvalue analysis, we have to determine the linear interpolation $\vec{\mathbf{v}}(\vec{x}) = A\vec{x} + \vec{o}$ of the vector field. The 3×3 matrix A and the translation vector $\vec{o}$ are uniquely determined by the four vector equations $AV_i + \vec{o} = \vec{v}_i$, $(i = 0,1,2,3)$, where V_i are the vertices of the tetrahedron and $\vec{v}_i$ are the attached velocities. The eigenvalues of A are different from each other. We assert this condition by classifying the tetrahedron by first checking the velocity vectors $\vec{v}_0$, $\vec{v}_1$, $\vec{v}_2$ and $\vec{v}_3$ at the vertices. If they all point in the same direction (with respect to some threshold), the cell gets labeled *parallel* and we store this direction. Parallel cells then are treated differently when building the particle path line.

For the non-parallel cells, we perform an eigenvalue and eigenvector analysis of A. First the eigenvalues are computed using Cardano's formula, which allows the analytical calculation of the roots of the polynomial, which is of order three in our case. Now we label the tetrahedron as real or complex, according to whether all its eigenvalues are real or two of them are complex conjugates to be *real* or *complex* cells.

Finally we check whether the eigenvalues are the same (with respect to some threshold) and we test whether the critical point $\vec{x}_{crit}$ of the linearly interpolated vector field lies within the tetrahedron:

$$A\vec{x}_{crit} + \vec{o} = 0 \quad \rightarrow \quad \vec{x}_{crit} = -A^{-1}\vec{o}$$

If so, we label the cell to be *extraordinary*. Here is a pseudocode:

```
// check velocity field
if (cell.getVelocities().computeDeviation() <
    EPSILON)
  cell.mark(PARALLEL);
else {
  Matrix A = LinSolve(cell.getVertices(),
                      cell.getVelocities());
  Real lambda[3] = A.getEigenvalues();

  // check for singularities
  if (lambda.same)
    cell.mark(EXTRAORDINARY);
  else if (lambda.real)
    cell.mark(REAL);
  else
    cell.mark(COMPLEX);

  cell.setEigenvectors(A.getEigenvectors());

  // check location of critical point
  Vector crit = (-A).inverse() * o;
  if (crit.isInside(cell))
    cell.mark(EXTRAORDINARY);
}
```

If the tetrahedron is labeled *real* we determine the eigenvectors $\vec{b}_1$, $\vec{b}_2$ and $\vec{b}_3$ corresponding to the three eigenvalues λ_1, λ_2 and λ_3. By changing the length of the eigenvectors we can achieve that $\det(\vec{b}_1,\vec{b}_2,\vec{b}_3) = 1$. With the eigenvalues the transformation matrix $S = (\vec{b}_1,\vec{b}_2,\vec{b}_3)$ and its inverse $S^{-1} = (\vec{b}_1,\vec{b}_2,\vec{b}_3)^{-1}$ is determined. Finally we store $S^{-1}\vec{o}$.

If the tetrahedron is labeled *complex* we have eigenvalues $\lambda_1 = \sigma + i\tau$, $\lambda_2 = \sigma - i\tau$ and λ_3, with $\lambda_{1,2}$ being conjugate complex numbers, σ, τ, λ_3 being real numbers and $\tau > 0$. $\vec{b}_1$ will be the real part and $\vec{b}_2$ will be the complex part of the complex eigenvector corresponding to $\lambda_{1,2}$. $\vec{b}_3$ is the eigenvector to λ_3. Like in the previous case we can achieve that $\det(\vec{b}_1,\vec{b}_2,\vec{b}_3) = 1$ by appropriate scaling. In this case $S^{-1} = (\vec{b}_2 \times \vec{b}_3, \vec{b}_3 \times \vec{b}_1, \vec{b}_1 \times \vec{b}_2)^t$.

Note that the above calculations are independent of each other for each cell, offering easy parallelization by simple multi-threading. This is especially useful when precomputing all eigenspaces of all tetrahedra for interactive particle trace extraction as described in section 5. Given a memory requirement of 8 pointers (4 neighbours and 4 vertices) for a simple tetrahedron implementation, the preprocessing step adds 9 floating point values for the eigenvectors, 3 floating point values for the eigenvalues, 3 floating point values for the critical point and a type information flag for the classification of the tetrahedron. This represents a memory growth factor of 4.875, given a pointer size of 32 Bit and 64 Bit floating point numbers. Note that this is the worst factor in case of precomputing all eigenspaces. In practice, the tetrahedron structure will carry additional data and the vertices also contribute to the overall memory requirement.

3.3. Calculation of the Exit Points

The creation of the particle path is started by identifying the tetrahedron wherein the start point lies by performing a full cell search. This is an expensive operation but it is only necessary once. Compare this with the pseudocode in section 2.1, where cell searches occur after each line segment that is generated.

When we have found the cell, we know the eigenvalues and the transformation matrix $S = (\vec{b}_1,\vec{b}_2,\vec{b}_3)$ and $S^{-1}\vec{o}$ that we have built as shown in previous section. Now we can compute the inverse S^{-1}. Then the pathline $\vec{x}(t)$ is given by equation (5). We determine the intersection of $\vec{x}(t)$ with the four face planes of the tetrahedron and use as exit point the one with the smallest positive t-value.

The intersection point of $\vec{x}(t)$ with the plane corresponding to face $F_0 = \Delta(V_1,V_2,V_3)$ is obtained by solving

$$\langle \vec{x}(t) | (V_1 - V_3) \times (V_2 - V_3) \rangle = \langle V_3 | (V_1 - V_3) \times (V_2 - V_3) \rangle$$

with $\langle \vec{a} | \vec{b} \rangle$ denoting the scalar product of $\vec{a}$ and $\vec{b}$. Solving for t gives the parameter value of the intersection point. Using the equation (5) we can write

$$\begin{aligned} & \langle \Lambda(t) S^{-1} x_0 | S^{transp}((V_1 - V_3) \times (V_2 - V_3)) \rangle \\ & + \langle \Delta(t) S^{-1} \vec{o} | S^{transp}((V_1 - V_3) \times (V_2 - V_3)) \rangle \\ = \; & \langle V_3 | (V_1 - V_3) \times (V_2 - V_3) \rangle \end{aligned}$$

We use Newton's method to solve this equation:

```
proc NewtonIter(Pin,cell)
{
  A = cell.S.inverse() * Pin;
  B = cell.S.transpose()*
      ((cell.V1 - cell.V3) *
       (cell.V2 - cell.V3));
  // C = cell.S.inverse() * o is precalculated
  a = cell.V3.dot( (cell.V1-cell.V3) *
                   (cell.V2-cell.V3) );
  t = 0;
  f = sum_i[0-2]{ A[i]*B[i] };
  f' = sum_i[0-2]{ cell.lambda[i]*A[i]*B[i] } +
       sum_i[0-2]{ C[i]*B[i] };

  do {

    t = t + (a-f)/f'
    f = sum_i[0-2]{ exp(cell.lambda[i]*t) *
                    A[i]*B[i] } +
        sum_i[0-2]{ ((exp(cell.lambda[i]*t)
                     -1) / cell.lambda[i]) *
                    C[i]*B[i] };
    f'= sum_i[0-2]{ cell.lambda[i] *
                    exp(cell.lambda[i]*t) *
                    A[i]*B[i] } +
        sum_i[0-2]{ exp(cell.lambda[i]*t) *
                    C[i]*B[i] };

  } until(converged);
}
```

The formulas in the pseudocode above are for the case of real eigenvalues. When the cell in question is marked *complex*, f and f' must be replaced by the appropriate form from section 2.2.

Here is the pseudocode for the whole process:

```
switch (cell.type) {
  case PARALLEL:
    direction = cell.velocityAt(Pin);
    Real t = INFINITY;
    for (i=cell.allPlanes) {
      Real tmp = cell.plane[i].distanceTo(Pin);
      if (tmp < t) {
        t = tmp;
        Int id = i;
      }
    }
    Pout = Pin + t * direction;
    break;
  case REAL:
  case COMPLEX:
    Pout = INFINITY;
    for (i=cell.allPlanes) {
      Vector tmp = NewtonIter(Pin,cell);
      if (tmp < Pout) {
        Pout = tmp;
        Int id = i;
      }
    }
    break;
  case EXTRAORDINARY:
    Pout = RungeKutta(Pin,cell);
    while(Pout.isInside(cell))
      Pout = RungeKutta(Pout,cell);
    Int id = mesh.findCellAt(Pout);
    break;
}
nextcell = cell.getNeighbour(id);
```

4. Rendering the Particle Trace

The local exact intersection points of the particle path with the grid cell facets provides an exact sampling. For drawing the particle line on screen, the simplest approach is to draw a straight line between two intersection points. Because the 3D world coordinates of the intersection points are known, interactive rendering based on OpenGL is possible. For many visualization tasks, especially in CFD simulation, this is a viable approach, because the size of the grid cells typically is very small in regions of interest. This can be due to adaptive grid refinement that has been initiated by the flow solver or simply by construction. If the user moves into the grid in order to closely examine some detail structure, the piecewise linear nature of the streamlines becomes visible again.

4.1. Building a Smooth Curve

In order to make the streamlines look more pleasant, we represent the segment within one grid element by a cubic curve. From the Newton iteration step, we have the exact exit point $\vec{x}(t_{out})$ of the tetrahedron and the exit time t_{out}. The entry point of the tetrahedron $\vec{x}(t_{in})$ is the exit point of the previously processed cell. So we can define the cell time $t_{cell} = t_{out} - t_{in}$. The corresponding velocity vectors $\vec{v}_{in}$ and $\vec{v}_{out}$ are obtained by linear interpolation (see section 3). Then the unique Bézier curve interpolating position and velocity (= derivative) can be calculated as follows.

Using the cell time, we construct a simple Bézier spline with the four points

$$\begin{aligned} Q_0 &= \vec{x}(t_{in}) \\ Q_1 &= \vec{x}(t_{in}) + t_{cell}/3 \cdot \vec{v}_{in} \\ Q_2 &= \vec{x}(t_{out}) - t_{cell}/3 \cdot \vec{v}_{out} \\ Q_3 &= \vec{x}(t_{out}) \end{aligned}$$

The spline has the following four basis functions (with t being the time elapsed since t_{in})

$$\begin{aligned} \vec{\mathbf{B}}_0(t) &= \frac{(t_{cell}-t)^3}{{t_{cell}}^3} \\ \vec{\mathbf{B}}_1(t) &= 3 \cdot \frac{(t_{cell}-t)^2 t}{{t_{cell}}^3} \\ \vec{\mathbf{B}}_2(t) &= 3 \cdot \frac{(t_{cell}-t) t^2}{{t_{cell}}^3} \\ \vec{\mathbf{B}}_3(t) &= \frac{t^3}{{t_{cell}}^3} \end{aligned}$$

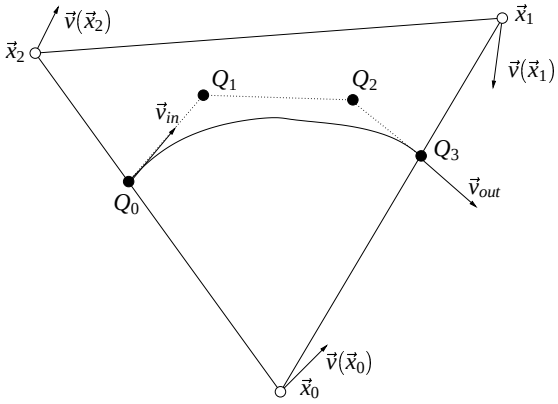


Figure 2: *Building a simple Bézier spline.*

We can now express the Bézier spline $\vec{\mathbf{c}}(t)$ that represents the particle trace by the simple t-depended vector-valued cubic polynomial

$$\vec{\mathbf{c}}(t) = \sum_{i=0}^{3} Q_i \cdot \vec{\mathbf{B}}_i(t)$$

In addition, please note the the Bézier spline (see figure 2)

being the Hermite interpolant, has the better local approximation error $O(h^4)$ compared to $O(h^2)$ for the linear interpolant.

4.2. Ray Tracing the Spline

The cubic Bézier spline between the local exact intersection points within one grid cell now gives a very natural looking trajectory that is also physically plausible. It can also be drawn directly using OpenGL NURBS. Because the cross section of a line has no extent, it can be hard for the user to get the correct depth cue if there is no neighboring surface geometry. One solution is to apply shading or to define visualization geometry like stream ribbons that provide 3D cues. Additionally, other data values can be mapped onto the geometry by manipulating shape (rotation) or appearance (color-coding) of the basic particle trace.

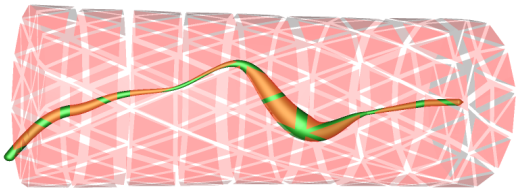


Figure 3: *A ray-traced particle line through a unstructured grid. Velocity is mapped to line thickness (slow $\mapsto$ thick).*

Our approach is to render the trajectory segment directly using ray tracing. The geometry of the curve is a tube that is obtained by sweeping a sphere along the path. For mapping additional data values on the tube, we can vary the sphere radius and the color of the surface. Note that this is equivalent to creating a stream tube visualization geometry. Our approach has the advantage that the surface of the tube is never constructed explicitly. This saves a lot of computation time and memory for the vertices of the sampled spline tube, because the sampling must otherwise be quite fine to avoid shading errors. The sampling may be coarser in areas of low curvature and/or low variation in the mapped data, but this again would introduce additional computation overhead.

The ray tracing approach automatically chooses the correct sampling rate and can be implemented very efficiently. The crucial point of our approach is to compute the ray-tube intersections. We are using a modification of a Graphics Gem introduced by Andreas Leipelt [5]. We can find all intersections of the ray $\vec{w} = \vec{a} + \theta\vec{d}$ with the sphere centered at $\vec{\mathbf{c}}(t)$ with radius $r(t)$ of the sweeping process $t \in [t_{in} \ldots t_{out}]$ by looking at the equation

$$\|\vec{\mathbf{c}}(t) - \vec{w}\|^2 = r(t)^2$$

Inserting the ray equation and substituting $p(t) = \vec{\mathbf{c}}(t) \cdot \vec{d}$ and $q(t) = \|\vec{\mathbf{c}}(t) - \vec{a}\|^2 - r(t)^2$ we get the t-depending solutions of this equation by

$$\theta(t) = p(t) \pm \sqrt{p(t)^2 - q(t)}$$

which is in general a complex-valued function. The positive minimum of $\theta(t)$ can be found by solving

$$p'(t) \pm \frac{2p(t)p'(t) - q'(t)}{2\sqrt{p(t)^2 - q(t)}} = \theta'(t) = 0$$

This equation can be slightly rewritten and by squaring it we obtain the generalized form

$$\vec{\mathbf{s}}(t) = q'(t)^2 + 4p'(t)(p'(t)q(t) - p(t)q'(t)) = 0$$

which is an equation of degree $n = 4m - 2$ where m is the degree of $\vec{\mathbf{c}}(t)$. So in the case of tracing cubic splines $n = 10$.

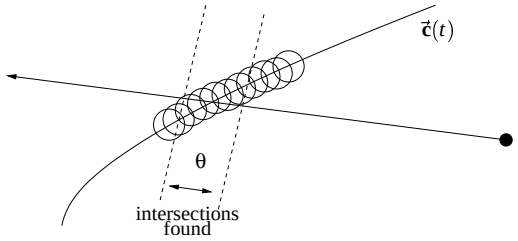


Figure 4: *Intersections of the ray with the tube.*

For actually computing the ray-tube intersection, we now determine all roots of $\vec{\mathbf{s}}(t)$ and evaluate θ at these roots to find the smallest positive real number. Figure 4 shows the intersections found by this process. Because we deliver the smallest positive θ back to the ray tracer, we are guaranteed to display the correct surface intersection point.

As the tube is constructed by sweeping a sphere, the normal at the computed ray-tube intersection point is simply the sphere normal which is calculated straightforward. The tube primitive also nicely integrates with acceleration structures for ray tracing, because we can compute the axis-aligned bounding box by looking at the three components $c_0(t)$, $c_1(t)$ and $c_2(t)$ of the polynomial $\vec{\mathbf{c}}$. We build the polynomials

$$m_i(t) = c_i(t) - r(t) \quad M_i(t) = c_i(t) + r(t) \quad i = 0 \ldots 2$$

and define min_i to be the absolute minimum of $m_i(t)$ treated by component. Note that a unique t_{min} therefore need not exist. We use an analogue approach to obtain max_i. The vectors (min_0, min_1, min_2) and (max_0, max_1, max_2) then form the best-fitting axis-aligned bounding box.

5. Results

Table 2 shows the overall time needed by distinct integration methods to compute the streamlines. The calculated paths consist of an identical number of segments with the same

overall length, so the average step size is the same. The paths can be different, because the methods offer different accuracy. In order to achieve the same numerical accuracy as the local exact method, the Runge-Kutta methods would need smaller step sizes, resulting in longer processing times. In the last two lines of the table, we compare our method with and without precomputed eigenspaces of the tetrahedra. Note that we need additional memory in order to store the intermediate variables as mentioned in section 3.2 when the tetrahedra are preprocessed. The creation of the tetrahedra's eigenspaces for the Sphere dataset from table 1 took 5.4 seconds. In the second case, the calculation of the eigenvalues and eigenvectors is delayed and performed on-the-fly when the trajectory enters a cell, demonstrating the ability of our algorithm to trade memory for speed. All time measurements are wall-clock time on a SGI Octane 300MHz R12000.

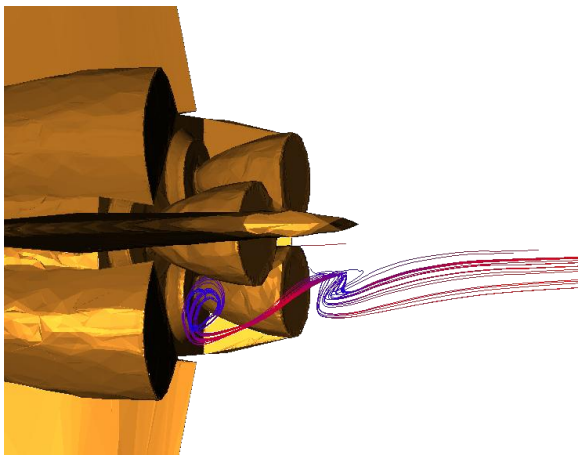


Figure 5: *The shuttle model in the wind tunnel computed with the local exact method.*

method	time (sec)
Euler	1.75
Heun	2.31
Runge-Kutta 3	3.11
Runge-Kutta 4	3.33
Exact with preprocessing	2.63
Exact on-the-fly	9.22

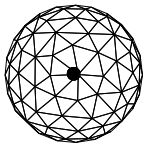


Table 2: *Timings for different integration methods for 1000 integration steps on the Sphere dataset.*

Using OpenGL to draw line primitives in interactive applications is fast. The user however often cannot clearly see the spatial position and shape of the streamline due to missing shading effects. In figure 6, we compare the OpenGL image on the left with a ray traced output. In the latter, the shape of the streamline is clearly visible. Our viewing application offers interactive exploration of the dataset by letting the user freely position and compute streamlines which are drawn as polylines immediately. In case needed, the user can render the current view using ray tracing by simply clicking a button. If the camera is moved, the viewer automatically falls back to OpenGL polyline drawing. For the dataset in figure 3, the ray tracing of one frame took three seconds at 800×400 pixel resolution.

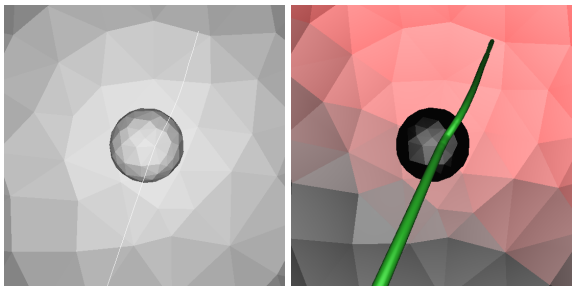


Figure 6: *High-quality visualization through ray tracing gives better depth cues.*

The local exact particle path faithfully follows the true vector field. Figure 7 shows on the left side the path computed in a synthetic three-tetrahedron dataset displayed using OpenGL line primitives. The flow vectors of the outermost vertices are perpendicular and point up, right and down when going from the left tetrahedron to the right one. Traditional integration methods need a small step size to produce an equivalent line. Our algorithm computes the particle trace in just three steps. In the ray traced image on the right, the segments of the particle curve are emphasized by different colors.

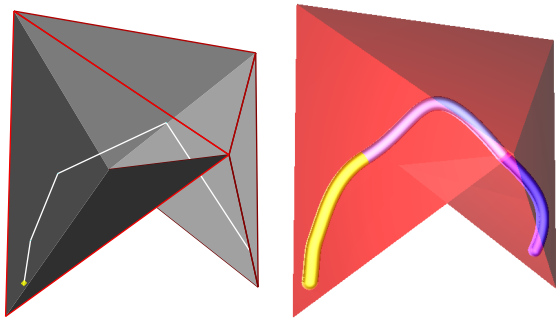


Figure 7: *The smooth curve on the right is colored differently in every segment computed.*

6. Conclusion

This paper presented the idea of creating local exact particle traces for tetrahedral grids. The method presented re-

places the Euler or Runge-Kutta integration steps that are performed traditionally with an exact computation of the exit points of the trajectory in the traversed tetrahedra. Because the method is locally exact, numerical errors do not accumulate.

The concatenation of the computed curve points builds the particle trace. It can be displayed in several ways according to the capabilities and rendering speed requirements of the client. This ranges from rendering the particle trace as polyline in interactive applications to ray tracing a smooth tube that is implicitly defined by the curve points and the curve tangents at these points.

From the point of view of integrating this visualization method in a larger system, the ability to control the memory requirement and the opportunity for parallelization is interesting. The computation of eigenvalues and eigenvectors of all tetrahedra can be performed as a parallel preprocessing step, or it can be executed on-the-fly during the creation of the particle trajectory only for the traversed cells. If the memory is strictly limited, the algorithm can therefore operate in a constant memory footprint.

Acknowledgments

This work is supported by the German Science Foundation DFG (Sonderforschungsbereich 603 “Modellbasierte Analyse und Visualisierung komplexer Szenen und Sensordaten”, Teilprojekt C7 “Adaptive Verfahren zur Berechnung und Visualisierung von mechatronischen Sensoren und Aktoren”) and the KONWIHR project “*gridlib*”.

References

1. B. Cabral and L. Leedom. Imaging Vector Fields Using Line Integral Convolution. In J. T. Kajiya, editor, *Computer Graphics Proceedings*, volume 27 of *Annual Conference Series*, pages 263–270, Los Angeles, California, August 1993. ACM SIGGRAPH, Addison-Wesley Publishing Company, Inc. 1

2. D. Darmofal and R. Haimes. An analysis of 3d particle path integration algorithms. *Journal of Computational Physics*, 123:182–195, 1996. 2

3. Thomas Frühauf. Interactive visualization of vector data in unstructured volumes. *Computers and Graphics*, 18:73–80, 1994. 1

4. David Kenwright and David Lane. Optimization of time-dependent particle tracing using tetrahedral decomposition. In G. M. Nielson and D. Silver, editors, *IEEE Visualization ’95*, Los Alamitos, CA, 1995. IEEE Computer Society Press. 1

5. Andreas Leipelt. Ray tracing a swept sphere. In *Graphics Gems V*. Academic Press, 1995. 7

6. Adriano Lopes and Ken Brodlie. Accuracy in 3D particle tracing. In Hans-Christian Hege and Konrad Polthier, editors, *Mathematical Visualization*, pages 329–341. Springer Verlag, Heidelberg, 1998. 1

7. G. M. Nielson, I.-H. Jung, N. Srinivasan, J. Sung, and J.-B. Yoon. Tools for Computing Tangent Curves and Topological Graphs for Visualizing Piecewise Linearly Varying Vector Fields over Triangulated Domains. In G. M. Nielson, H. Hagen, and H. Müller, editors, *Scientific Visualization: Overviews, Methodologies, and Techniques*, chapter 21, pages 527–562. IEEE Computer Society Press, Los Alamitos, California, 1997. 1, 2

8. Gregory M. Nielson and II-Hong Jung. Tools for computing tangent curves for lineary varying vector fields over tetrahedral domains. *IEEE Transactions on Visualization and Computer Graphics*, pages 360–372, 1999. 1, 2, 4

9. Ari Sadarjoen, Theo van Walsum, Andrea J. S. Hin, and Frits H. Post. Particle tracing algorithms for 3D curvilinear grids. Technical Report DUT-TWI-94-80, Delft University of Technology, 1994. 1

10. C. Teitzel, R. Grosso, and T. Ertl. Efficient and Reliable Integration Methods for Particle Tracing in Unsteady Flows on Discrete Meshes. In W. Lefer and M. Grave, editors, *Eighth Eurographics Workshop on Visualization in Scientific Computing*, pages 49–56, Boulogne sur Mer, France, April 1997. EG, The EuroGraphics Association. 2

11. S. Ueng, K. Sikorski, and K. Ma. Efficient streamline, streamribbon, and streamtube constructions on unstructured grids. *IEEE Transactions on Visualization and Computer Graphics*, 2:100–110, 1996. 1