Particle tracing algorithms for 3D curvilinear grids

Report 94-80

Ari Sadarjoen
Theo van Walsum
Andrea J.S. Hin
Frits H. Post

TU Delft
Technische Universiteit Delft
Delft University of Technology

Faculteit der Technische Wiskunde en Informatica
Faculty of Technical Mathematics and Informatics

ISSN 0922-5641

Copies of these reports may be obtained from the bureau of the Faculty of Technical Mathematics and Informatics, Julianalaan 132, 2628 BL Delft, phone +3115784568.
A selection of these reports is available in PostScript form at the Faculty's anonymous ftp-site, ftp.twi.tudelft.nl. They are located in directory /pub/publications/tech-reports. They can also be accessed on the World Wide Web at:
http://www.twi.tudelft.nl/TWI/Publications/Overview.html

Particle Tracing Algorithms for 3D Curvilinear Grids

Ari Sadarjoen Theo van Walsum Andrea J.S. Hin Frits H. Post

August 1994

Abstract

This paper presents a comparison of several particle tracing algorithms on curvilinear grids. The fundamentals of particle tracing algorithms are described and used to split tracing algorithms into basic components. Based on this decomposition, two different strategies for particle tracing are described in greater detail: tracing in computational space and tracing in physical space. Accuracy and speed tests are performed for both types of algorithms. From these tests it is concluded that particle tracing algorithms in physical space generally perform better than algorithms in computational space.

Keywords: scientific visualization, vector field visualization, particle tracing, interpolation, grid transformation

1 Introduction

Computational Fluid Dynamics (CFD) is concerned with modelling complex fluid flows. Increasingly sophisticated software is being developed to simulate interesting flow phenomena. Scientists can gain insight in the resulting data by visualizing it. One method of visualizing a velocity field is particle tracing, which involves releasing particles into a flow and calculating their positions at specific times.

In general, CFD simulations provide a velocity field defined on a discrete grid. The simplest grids are block-shaped with cubical cells. Particle tracing algorithms for such Cartesian grids are investigated in [Yeung & Pope, 1988; Kontomaris & Hanratty, 1992]. In CFD practice, the grids are often boundary-fitted and therefore curvilinear, with the purpose of solving flows in complex geometries. Particle tracing algorithms for CFD grids were presented in [Buning, 1989a; Buning, 1989b]. Some algorithms transform a curvilinear grid to a Cartesian grid and perform particle tracing in the Cartesian space [Strid *et al.*, 1989]. A more detailed description of this method was recently given in [Shirayama, 1993].

In most papers on particle tracing, the scope is limited to *a single* method. If alternatives are given at all, they are not closely examined. Often, not all details of the particle tracing process are given, suggesting that some operations are straightforward. In commercial visualization packages, background information concerning the applied particle tracing algorithms is seldom given.

The present paper addresses the above issues. Detailed descriptions are given of the particle tracing process, which is split into distinct components. The strengths and weaknesses of different implementations of these components are studied and visualized. The main aspects considered are accuracy and performance.

After covering some fundamentals on particle tracing in section 2, the details of two different classes of particle tracing algorithms will be described in section 3 and 4. Considerations for the implementation of several algorithms, as well as an overview of test cases are given in section 5. The test results and a discussion are presented in section 6. Finally, section 7 derives some conclusions.

2 Fundamentals

Although many terms are in use for grid types, we will reserve the term *Cartesian grids* for grids with straight grid lines and unit cubes as cells, and *curvilinear grids* for grids having curved (or more precisely: piecewise straight) grid lines and a regular topology. We will start by explaining the principles of particle tracing in Cartesian grids.

2.1 Particle tracing in Cartesian grids

The computation of a particle path is based on a numerical integration of the ordinary differential equation

$$\frac{d\mathbf{x}}{dt} = \mathbf{v}(\mathbf{x}) \tag{1}$$

where t denotes time, $\mathbf{x}$ the position of the particle and $\mathbf{v}(\mathbf{x})$ the velocity field. The starting position $\mathbf{x}_0$ of the particle provides the initial condition:

$$\mathbf{x}(t_0) = \mathbf{x}_0 \tag{2}$$

The solution is a sequence of particle positions $(\mathbf{x}(t_0), \mathbf{x}(t_1), \ldots)$.

A particle tracing algorithm must perform the following steps. First, a search is performed for the cell which contains the initial position of the particle. To determine the velocity in this point, the velocities in the cell corners are interpolated. Then, an integration step calculates the next position of the particle. Again, a search is performed, now for the cell containing the new position. The process of interpolation, integration and point location is repeated until the particle leaves the grid. This process can be translated into pseudo-code representing the general structure of a particle tracing algorithm:

```
find cell containing initial position                (point location)
while particle in grid
      determine velocity at current position         (interpolation)
      calculate new position                         (integration)
      find cell containing new position              (point location)
endwhile
```

Point location

Determining which cell contains a specified point is called point location. The coordinates of a point can be divided into their integer and fractional parts: $\mathbf{x} = (x, y, z) = [i, j, k] + (\alpha, \beta, \gamma)$, where i, j, k are integers and $\alpha, \beta, \gamma \in [0, 1]$. In this paper, we will refer to $[i, j, k]$ as the *indices* and (α, β, γ) as the *offsets*. Point location in a Cartesian grid is as simple as truncating the coordinates to their integer parts. Here, determining the offsets is also considered to be part

of the point location operation, but sometimes we will make a strict distinction between point location and offset determination.

Interpolation

To obtain a value of the velocity field in other points than the grid nodes, it is necessary to determine an interpolated value from the nodes surrounding the point. The standard way to do this in cubical cells is trilinear interpolation. This requires the cell indices and the offsets obtained through point location. Let $I, J, K \in \{0, 1\}$ and let the base functions Ψ be defined as $\Psi_0(\alpha) = 1 - \alpha$ and $\Psi_1(\alpha) = \alpha$. Then, for a point $\mathbf{x}$ in a cell [i,j,k] with offsets (α, β, γ), the function $\mathcal{T}$ determines the interpolated value from the corner velocities $\mathbf{v}_{i,j,k} \ldots \mathbf{v}_{i+1,j+1,k+1}$.

$$\mathbf{v} = \mathcal{T}(\mathbf{v}, \alpha, \beta, \gamma) = \sum_{I,J,K=0}^{1} \mathbf{v}_{i+I,j+J,k+K} \cdot \Psi_I(\alpha)\Psi_J(\beta)\Psi_K(\gamma) \tag{3}$$

Integration

Many integration methods are known in the literature, ranging from the simple first-order Euler scheme to the fourth-order Runge-Kutta scheme or even higher-order methods, applied with fixed or variable time steps. A well-known second-order method is Heun's scheme, also known as a second-order Runge-Kutta scheme. Starting from position $\mathbf{x}_n$ at time $t = t_n$, the position $\mathbf{x}_{n+1}$ at time $t = t_{n+1}$ is calculated in two steps:

$$\mathbf{x}^*_{n+1} = \mathbf{x}_n + \Delta t \cdot \mathbf{v}(\mathbf{x}_n) \tag{4}$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \cdot \frac{1}{2}\left\{\mathbf{v}(\mathbf{x}_n) + \mathbf{v}(\mathbf{x}^*_{n+1})\right\} \tag{5}$$

2.2 Particle tracing in curvilinear grids

In practice, the grids used in many CFD applications are *not* Cartesian. To handle complex geometries, curvilinear, boundary-fitted grids are used. These grids are also referred to as *structured* grids, because they have a regular topological structure, as opposed to unstructured grids as used in Finite Element Methods.

While this allows for a large variety of geometries, numerical procedures are much more difficult in curved grids. This is the reason why in CFD flow solvers, the curvilinear grid in the physical domain is often internally transformed to a Cartesian grid in a new domain. The physical domain will be called *P-space* ($\mathcal{P}$) and the new domain will be called computational space or *C-space* ($\mathcal{C}$). Many calculations are performed much more efficiently in the simple Cartesian grid in C-space.

In particle tracing in a curvilinear grid similar problems arise. Especially point location and interpolation become more complex. This suggests the use of a similar procedure in determining particle paths for visualization purposes as for flow solving. A transformation is defined from physical space to computational space such that the curvilinear grid becomes a Cartesian grid (see figure 1).

In a Cartesian grid, point location and interpolation can be carried out as described in the previous subsection. We must transform the positions of the complete path back to the physical domain to be able to visualize it, but the use of the convenient computational domain may increase the efficiency of the algorithm. The details will be discussed in section 3.

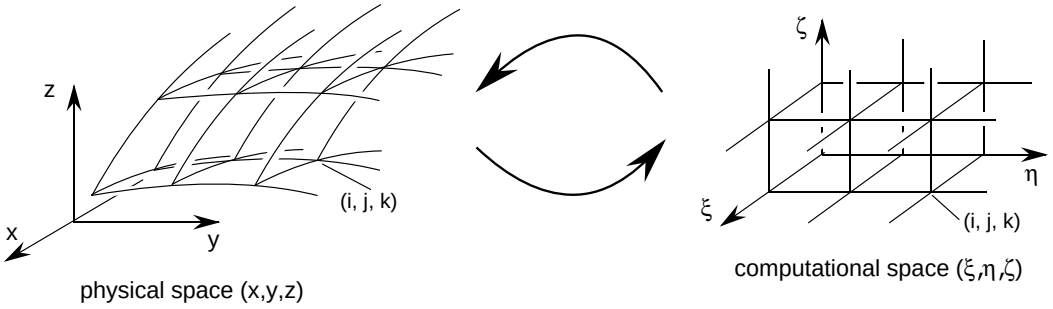

Figure 1: Transformation between $\mathcal{P}$ and $\mathcal{C}$

An alternative is to calculate the particle path directly in P-space ($\mathcal{P}$). This would involve more complex point location and interpolation, but would make grid transformation obsolete. This alternative will be covered in section 4.

2.3 Restrictions

We will restrict our attention to stepwise integration methods. Also, we will only use the second-order Runge-Kutta method with fixed time steps. For reasons of simplicity, we consider the flow field to be time-independent.

3 C-space algorithms

Particle tracing in computational space proceeds in almost the same way as described in section 2.1, except that the physical velocities or fluxes $\mathbf{v}$ must be transformed to $\mathbf{u}$ in $\mathcal{C}$ and instead of equation (1), the following differential equation is solved:

$$\frac{d\boldsymbol{\xi}}{dt} = \mathbf{u}(\boldsymbol{\xi}) \tag{6}$$

where $\boldsymbol{\xi}$ denotes a position in $\mathcal{C}$. The solution is now a sequence of positions $(\boldsymbol{\xi}(t_0), \boldsymbol{\xi}(t_1), \ldots)$ in computational coordinates. As a consequence, the particle path calculated in $\mathcal{C}$ must be transformed to physical space for visualization purposes. The general form of the algorithm then becomes:

```
find cell containing initial position          (point location)
while particle in grid
      transform corner velocities from P to C  (transform velocities)
      determine C-velocity at current position (interpolation)
      calculate new position in C-space        (integration)
      transform C-position to P                (transform position)
      find cell containing new C-position      (point location)
endwhile
```

The point location and interpolation parts in this algorithm are straightforward, because the computational grid is Cartesian. However, from the pseudo-code it can be seen that two

transformation operations are introduced. First, the physical velocities must be transformed to C-space. Second, the calculated particle positions must be transformed back to P-space. The former operation is not necessary if it is possible to use the velocities or fluxes in computational space that are used during the flow solving process. However, we also assume that the results from the flow solver are physical velocities. It should be emphasized, that in the above algorithm the velocity field is not transformed and stored in its entirety, but only the velocities in the current cell are transformed on-the-fly. They are *local* transformations, since in most of the cases no global grid transformation from $\mathcal{P}$ to $\mathcal{C}$ exists.

3.1 Transformation of positions from $\mathcal{C}$ to $\mathcal{P}$

The transformation of a position from $\mathcal{C}$ to $\mathcal{P}$ is relatively straightforward. What is necessary, is a transformation which maps the corner nodes of a cubic cell in computational space to the corner nodes of a cell in physical space. The edges between nodes are assumed to be straight in either space. This leads to a transformation which is equivalent to a trilinear interpolation between the cell corner P-space positions, as in equation (3). The function $\mathcal{T}$ transforms a C-space position $\boldsymbol{\xi} = (\xi, \eta, \zeta)$ consisting of integer parts $[i, j, k]$ and fractional parts (α, β, γ) to a P-space position $\mathbf{x}$ as

$$\mathbf{x} = \mathcal{T}(\mathbf{x}, \alpha, \beta, \gamma) \tag{7}$$

3.2 Transformation of velocities from $\mathcal{P}$ to $\mathcal{C}$.

A velocity $\mathbf{u}$ in $\mathcal{C}$ is transformed to $\mathbf{v}$ in $\mathcal{P}$ according to:

$$\mathbf{v} = \mathbf{J} \cdot \mathbf{u} \tag{8}$$

and similarly, a velocity $\mathbf{v}$ in $\mathcal{P}$ can be transformed to $\mathbf{u}$ in $\mathcal{C}$ with

$$\mathbf{u} = \mathbf{J}^{-1} \cdot \mathbf{v} \tag{9}$$

The matrix $\mathbf{J}$ is called the *Jacobian* and contains the partial derivatives:

$$\mathbf{J} = \begin{pmatrix} x_\xi & x_\eta & x_\zeta \\ y_\xi & y_\eta & y_\zeta \\ z_\xi & z_\eta & z_\zeta \end{pmatrix} \tag{10}$$

where x_ξ is short for $\frac{\partial x}{\partial \xi}$ etc.

The matrix $\mathbf{J}$ can be thought of as consisting of the columns $(\mathbf{j_1} \mid \mathbf{j_2} \mid \mathbf{j_3})$. These are in fact the partial derivatives $\frac{\partial \mathbf{x}}{\partial \xi}, \frac{\partial \mathbf{x}}{\partial \eta}, \frac{\partial \mathbf{x}}{\partial \zeta}$. As we are dealing with (discrete) grids, the Jacobian must be calculated with finite differences. There are several methods for doing this. Another aspect is the *number* of Jacobians used for each cell.

Jacobian approximations

To approximate a Jacobian in a grid point, several types of differencing are available. Given the coordinates of the grid nodes $\mathbf{x}_{i,j,k}$ where $[i, j, k]$ lie within the grid boundaries, there are at least three possibilities:

- forward differences: $\mathbf{j}_1 = \frac{\Delta \mathbf{x}}{\Delta \xi} = \mathbf{x}_{i+1,j,k} - \mathbf{x}_{i,j,k}$

- backward differences: $\mathbf{j}_1 = \frac{\Delta \mathbf{x}}{\Delta \xi} = \mathbf{x}_{i,j,k} - \mathbf{x}_{i-1,j,k}$
- central differences : $\mathbf{j}_1 = \frac{\Delta \mathbf{x}}{\Delta \xi} = (\mathbf{x}_{i+1,j,k} - \mathbf{x}_{i-1,j,k})/2$

Calculating the derivatives $\mathbf{j}_2 = \frac{\Delta \mathbf{x}}{\Delta \eta}$ and $\mathbf{j}_3 = \frac{\Delta \mathbf{x}}{\Delta \zeta}$ proceeds in a similar way.

In a 2D cell, these differences can be combined to the cases in figure 2. In an arbitrary grid node (figure 2a) either forward differences (figure 2b), backward differences (figure 2c) or central differences (figure 2d) can be calculated. Alternatively, mixed differences, such as forward for $\mathbf{j}_1$ and backward for $\mathbf{j}_2$ (figure 2e), or vice versa (figure 2f) can be used.

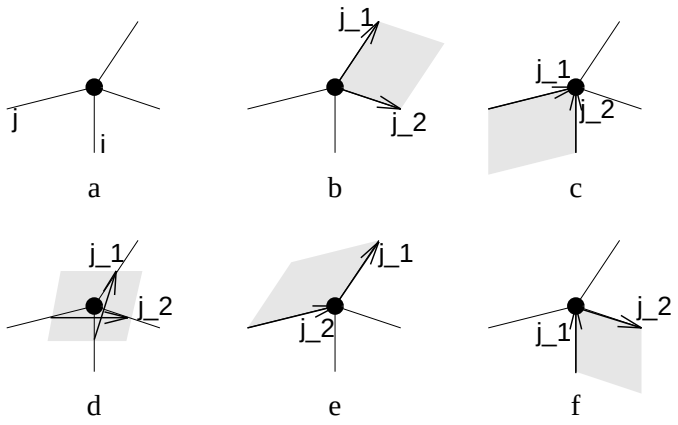

Figure 2: Several types of differences for Jacobian calculation

Number of Jacobians

The other aspect is the *number* of Jacobians calculated in a cell. Basically, there are three options:

- one for each cell (see figure 3a)
- one for each node (see figure 3b)
- eight for each node (see figure 3c)

The simplest and fastest method is to calculate only one Jacobian per cell [Strid *et al.*, 1989]. By computing forward, backward or central differences in one node, for example in the lower left node, it is assumed that this Jacobian reasonably represents the deformation throughout the entire cell (see figure 3a).

Another approach is to calculate separate Jacobians *in each grid node* [Shirayama, 1993]. Now, in a 3D cell eight different Jacobians have to be calculated, or four in a 2D cell (see figure 3b). However, to save time it is possible to transform the entire velocity field in a pre-processing stage before tracing the particles.

Using mixed types of differences leads to one Jacobian per node per cell. The Jacobian for a node is calculated according to which cell the particle is in. We shall call this *forward/backward differences* (see figure 3c). In this case, transforming the vector field in a pre-processing

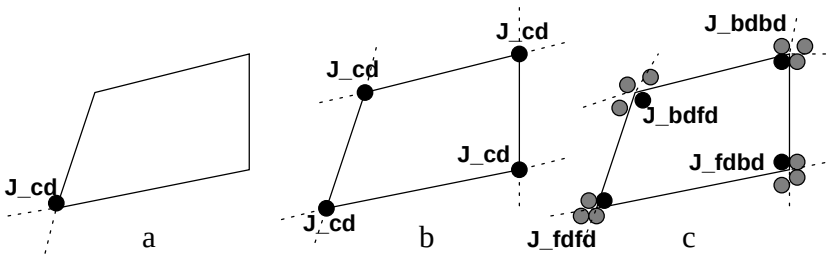

Figure 3: Number of Jacobians in a 2D cell

stage is not attractive. Because each node can have eight Jacobians, storing the transformed velocities would multiply the required storage space by a factor eight. But this method of calculating Jacobians is mathematically correct when trilinear interpolation is used for point transformation. We will elaborate on this in section 6.4.

4 P-space algorithms

The alternative to particle tracing in a simple Cartesian grid in C-space, is particle tracing in a complex curvilinear grid in P-space. This has a considerable impact on the complexity of the algorithm presented in section 2. In particular, point location and interpolation are not as straightforward anymore. There is no longer a simple relation between a physical position in space and the grid cell that contains it. Moreover, as the cells in a curvilinear grid are not cubes, it is also not as easy to perform trilinear interpolation, because the relative position (fractional offsets) of a position in a cell is hard to determine. The next two subsections will discuss alternative point location and interpolation methods, respectively.

4.1 Point location methods

We can distinguish between *global* and *incremental* point location. In global point location, a given point in a grid must be found with no previous known cell. In a curvilinear grid this is not an easy task. As with all search algorithms it is possible to use a simple brute-force algorithm which searches all grid cells one-by-one. Naturally, this is expensive. Auxiliary data structures can be used to perform a smart search [Neeman, 1990; Williams, 1992].

Fortunately, particle tracing is a step-by-step process, in which, most of the time, global point location is only necessary to find the cell containing the initial position of a particle. After this, there is always a current starting position and a current starting cell, from which the new position is to be found. This occurs at every integration step that starts from the current particle position in some known cell and calculates a new position. The following paragraphs will discuss only incremental point location.

• Tetrahedrization

To use tetrahedrization, the curvilinear hexahedral cells are decomposed into tetrahedra. The reasons for this are:

1. Tetrahedra are convex, which facilitates testing if a specified point is inside.

2. Tetrahedra have planar faces, which facilitates intersection calculation with a line.

Incremental point location can now be performed by drawing a line from the previous known position to the next position [Garrity, 1990]. Intersections with the faces of the tetrahedron and containment tests in neighbouring cells are used to locate the new point. Tetrahedrization is only performed in the cells along the path of the line and the results do not have to be stored.

• Stencil Walk and Newton-Raphson

Let $\mathbf{P}$ be a point given in physical space that must be found in computational space. In the so-called *stencil walk* method [Buning, 1989a], first an initial point $\boldsymbol{\xi}$ in computational space is chosen. This point is transformed to physical space using the transformation $\mathbf{x} = \mathcal{T}(\mathbf{x}, \alpha, \beta, \gamma)$. The difference between the transformed and the target point $\mathbf{P}$ is calculated as $\Delta\mathbf{x} = \mathbf{x} - \mathbf{P}$. This difference vector in physical space is transformed to computational space using $\Delta\boldsymbol{\xi} = \mathbf{J}^{-1}\Delta\mathbf{x}$ and added to the previous point, resulting in a new guess. If one of the elements of $\Delta\boldsymbol{\xi}$ is outside the range $[0, 1]$, the centre of the corresponding neighbouring cell is the new guess. The iterative process continues until the right cell has been found. Once the correct cell has been found, one can iterate until the value of $\Delta\boldsymbol{\xi}$ is small enough.

Another approach is to use the well-known Newton-Raphson iteration [Gerritsen, 1988; Mooiman, 1993] that finds the root to the equation $\mathcal{F}(\boldsymbol{\xi}) = \mathbf{0}$ by the following relation:

$$\boldsymbol{\xi}_{n+1} = \boldsymbol{\xi}_n - \frac{\mathcal{F}(\boldsymbol{\xi})}{\mathcal{F}'(\boldsymbol{\xi})} \tag{11}$$

Here, $\mathcal{F}$ is defined as $\mathcal{F}(\boldsymbol{\xi}) = \mathbf{P} - \mathcal{T}(\boldsymbol{\xi})$, $\mathbf{P}$ is again the point in P-space to be found in C-space, and $\mathcal{T}$ is the function that transforms a point from C-space to P-space. Experiments have shown that this method converges rapidly.

It can be shown that offset calculation using a Newton-Raphson process is identical to offset calculation using a stencil walk process, if the stencil walk algorithm uses interpolated forward/backward Jacobians for the transformation of $\Delta\mathbf{x}$. By substituting the above definition of $\mathcal{F}$ into equation 11, taking into account that $\mathcal{F}' = \mathbf{J}$, the equivalence relation can be easily derived.

4.2 Interpolation methods

• Trilinear interpolation

If the offsets in the cell (α, β, γ) are available, an interpolated velocity can be determined from the data values in the cell corners, as in equation (3) [Mooiman, 1993]. One way to determine (α, β, γ) in a curvilinear grid is by using the Stencil Walk/Newton-Raphson iteration. The following two methods provide alternative ways of interpolation that do not require the offsets.

• Inverse Distance Weighting

Let $\mathbf{x}_0, \ldots, \mathbf{x}_7$ be the coordinates of the eight corner nodes of a hexahedral cell, and let $\mathbf{v}_0, \ldots, \mathbf{v}_7$ be the velocities in those nodes. Then, the interpolated value $\mathbf{v}$ in point $\mathbf{X}$ is

calculated as a weighted average of the corner values [Wilhelms *et al.*, 1990; Watson, 1992]:

$$\mathbf{v} = w_0\mathbf{v}_0 + w_1\mathbf{v}_1 + \cdots + w_7\mathbf{v}_7 \tag{12}$$

The weight of each data point is calculated as a function of the Euclidian distance between $\mathbf{x}_i$ and $\mathbf{X}$:

$$d_i = \|\mathbf{x}_i - \mathbf{X}\| \tag{13}$$

$$w_i = \frac{\frac{1}{(d_i)^2}}{\sum_{j=0}^{7} \frac{1}{(d_j)^2}} \tag{14}$$

In a 2D cell consisting of four nodes, the same principle can be applied (see figure 4).

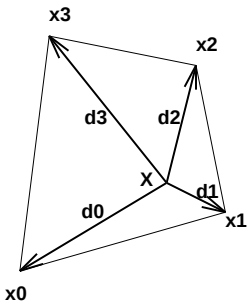

Figure 4: Inverse Distance Weighting

Should $\mathbf{X}$ be close to a corner node, the distance d_i would be nearly zero and the value w_i would be unpredictable. This is handled in the algorithm by testing whether the distance d_i is close to zero, in which case the weight of that corner is set to 1, and the weights of all other nodes are set to zero.

• Volume Weighting

In volume weighting, interpolation is performed within a tetrahedron, based on volume weights. Let $\mathbf{X}$ be a point in a tetrahedron consisting of nodes $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2$ and $\mathbf{x}_3$. Then, the tetrahedron can be subdivided into four tetrahedra, in all of which $\mathbf{X}$ is a corner node. The weight for each node of the main tetrahedron is the ratio of the volume of the subtetrahedron to the volume of the main tetrahedron [Buning, 1989a].

$$w_0 = \frac{V_{123X}}{V_{0123}} \quad w_1 = \frac{V_{023X}}{V_{0123}} \quad w_2 = \frac{V_{013X}}{V_{0123}} \quad w_3 = \frac{V_{012X}}{V_{0123}} \tag{15}$$

The volume of an arbitrary tetrahedron ABCD is calculated as

$$V_{ABCD} = \frac{1}{6}|\vec{AB} \cdot (\vec{AC} \times \vec{AD})| \tag{16}$$

This interpolation method can take advantage of the information created in decomposing cells into tetrahedra, if tetrahedrization is used as the point location method.

The principle is demonstrated in 2D in figure 5, where the equivalent of tetrahedral volumes are triangular areas. The weight of node $\mathbf{x}_0$ is the ratio of the area A_0 of the opposing subtriangle to the area of the main triangle. Volume weighting is continuous over the cell faces.

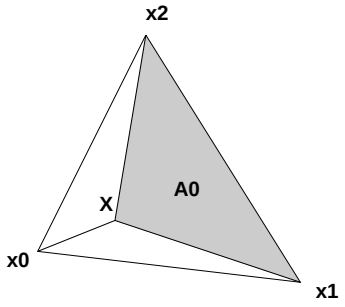

Figure 5: Area weighting

Note that in 3D only four nodes of the cell are used for the interpolation, since only one of the subtetrahedra forming the cell is used, whereas in IDW or trilinear interpolation eight nodes are used.

5 Implementation and test flows

Based on the descriptions in section 3, the algorithms listed in table 1 and 2 have been implemented.

Algorithm	Transformation
`C-FD1`	1 forward diff. per cell
`C-FD8`	8 forward diff. per cell
`C-CD8`	8 central diff. per cell
`C-FDBD`	forward/backward diff.

Table 1: C-space algorithms

Algorithm	Interpolation
`P-4-IDW`	Inv. Dist. Weighting
`P-4-VOL`	Volume Weighting
`P-4-TRI`	NR + trilin.
`P-SW-TRI`	Stencil Walk + trilin.

Table 2: P-space algorithms

In all C-space algorithms, point location and interpolation are straightforward as described in section 2. Only the transformation methods vary. In `C-FD1`, one Jacobian is calculated per cell using forward differences in one node. In `C-FD8` and `C-CD8`, eight Jacobians are calculated per cell using forward and central differences, respectively. In `C-FDBD`, eight Jacobians are calculated per cell using mixed differences.

In the P-space (`P-`) algorithms, tetrahedrization (`-4-`) and the Stencil Walk (`-SW-`) have been used for point location. For interpolation, inverse distance weighting (`-IDW-`), volume weighting (`-VOL-`) and trilinear interpolation (`-TRI-`) have been implemented.

All implemented algorithms employ a second-order Runge-Kutta method as described in section 2. First-order methods (such as Euler) were found to be inadequate by several

researchers [Buning, 1989b; Shirayama, 1993]. Others use higher-order methods as well. But since the integration method was not our main point of interest, we did not implement these.

Test flows

Ideally, to test the algorithms, flows should be used for which it is possible to analytically calculate the path travelled by a particle. Given a particle starting position, it should be possible to calculate the particle position $\mathbf{x}(t)$ for any given time t. This enables a comparison of the particle paths computed by the algorithms to the theoretical paths.

Although both test flows are defined on a 3D grid, they are essentially two-dimensional. The reason for this is that in 3D flows it is seldom possible to determine the analytical solution of a particle trajectory. These factors make it more difficult to verify the accuracy of 3D particle paths. Nevertheless, reasonable conclusions can be drawn from the calculated semi-3D paths.

Uniform flow in straight pipe with curved grid

One kind of flow where the theoretical particle paths are known, is a uniform flow. This test flow is defined on a 21 x 41 x 2 (semi-3D) curvilinear grid, which originates from a practical CFD application (see figure 6). The grid represents a vertical pipe with an inlet located at the bottom and an outlet at the top.

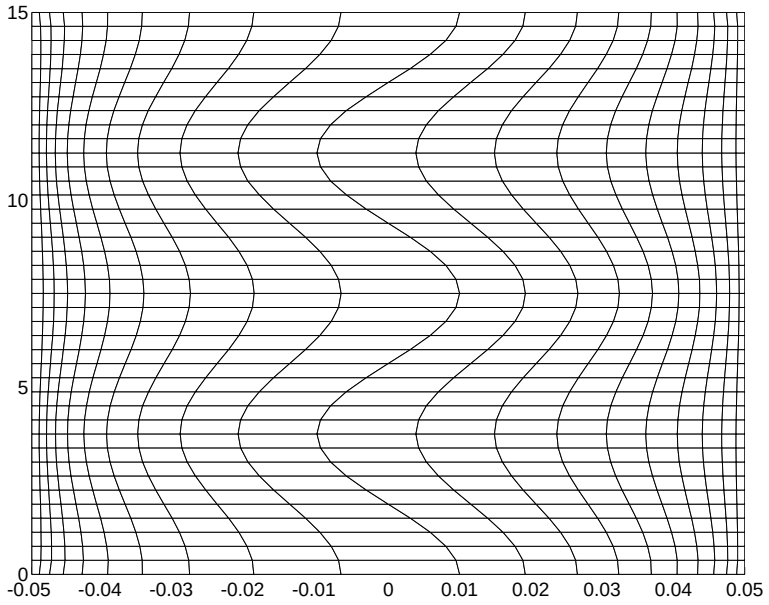

Figure 6: Curved grid

Flow in L-shaped pipe

The second test flow uses an L-shaped pipe, in which the flow has been calculated using the `ISNaS` CFD-simulation package. The grid shown in figure 7 consists of 7 x 23 x 2 nodes

(semi-3D) and contains a sharp transition. The inlet of the pipe is at the right ($x = 2.5$).

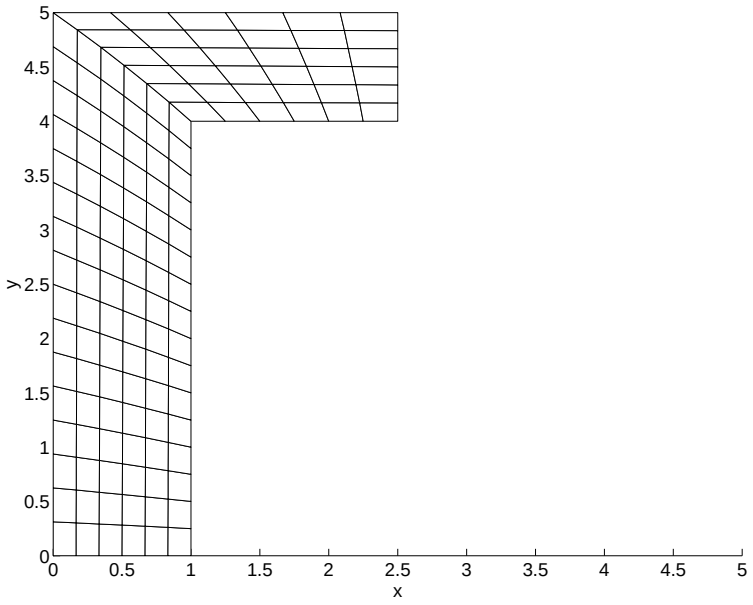

Figure 7: L-pipe grid

6 Test results

6.1 Transformation effect

The two test cases described above were used to test the effect of the transformation component in a C-space particle tracing algorithm. All C-space algorithms (`C-FD1`, `C-FD8`, `C-CD8` and `C-FDBD`) were applied to these two problems. As a reference, a P-space algorithm (`P-4-IDW`) was used.

Uniform flow in straight pipe with curved grid

As the flow field is uniform, the particle paths should be straight vertical lines. All P-space algorithms produce this exact solution. Figure 8 shows the resulting particle paths calculated by the `P-4-IDW` algorithm. Since the velocity field in P-space is constant, the P-space algorithms all give the same solution, so any other P-space algorithm could be used instead of `P-4-IDW`. The particle paths produced by `C-FD1` are shown in figure 9; they are not straight. The other C-space algorithms produce similar inaccurate results.

A closer comparison is possible when particle paths calculated by different algorithms are combined in one figure. In figure 10 the particle paths produced by five algorithms are combined and magnified to highlight the differences. The straight vertical line was produced by the P-space algorithm. Of the C-space algorithms, the greatest deviation is shown by `C-FD1` and `C-FD8`. `C-CD8` appears to be significantly more accurate, but `C-FDBD` gives the best results.

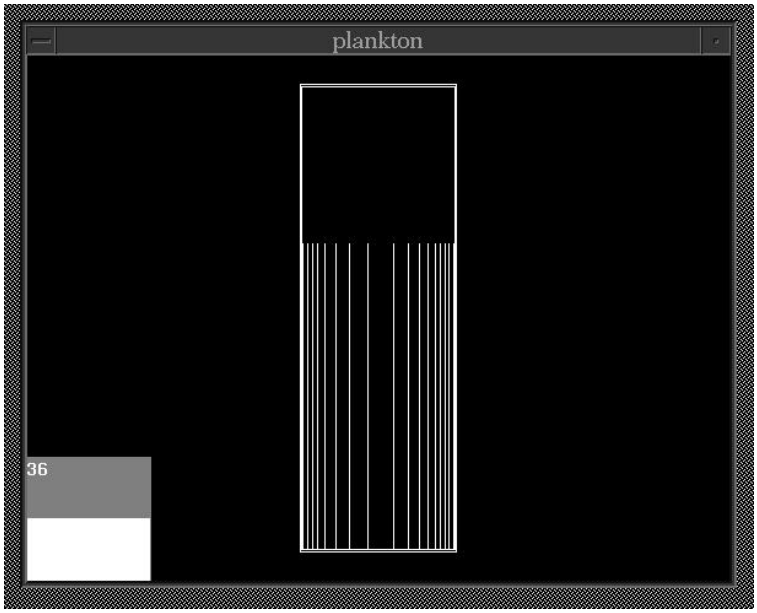

Figure 8: Flow in straight pipe with curved grid. Straight particle paths `P-4-IDW`

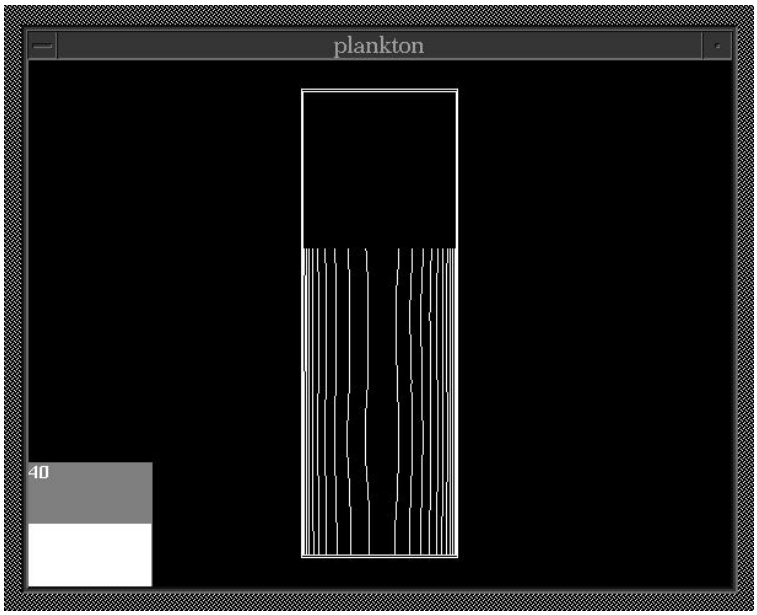

Figure 9: Flow in straight pipe with curved grid. Curved particle paths `C-FD1`

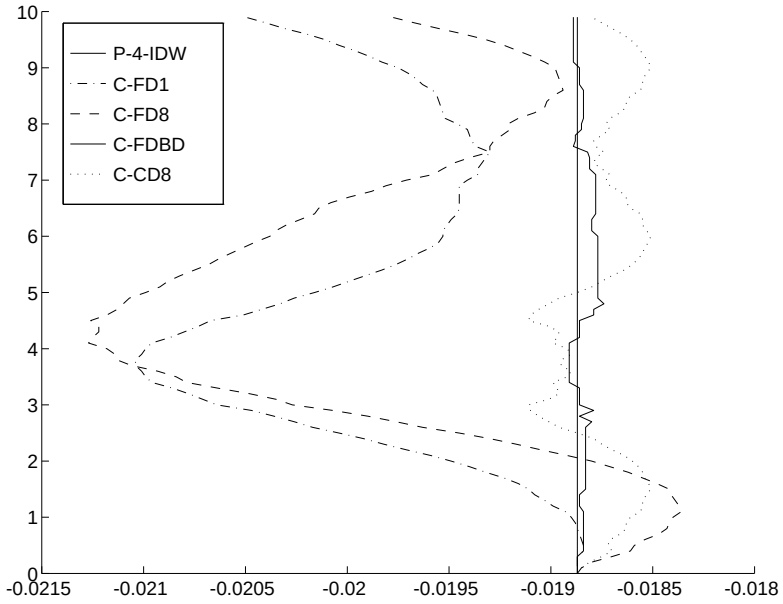

Figure 10: particle path produced by different algorithms

From these results the conclusion could be drawn that a simple transformation as described in [Strid *et al.*, 1989] (`C-FD1`) or [Shirayama, 1993] (`C-FD8`) gives inadequate results.

Flow in L-shaped pipe

In the previous test, two C-space algorithms were found to be significantly better than the others: the `C-CD8` algorithm which uses central differences and the `C-FDBD` algorithm, which uses forward/backward differences. These two algorithms were applied to the test flow in the L-shaped pipe, which features a sharp transition. The integration timestep was set to 0.01, with up to 1000 timesteps calculated. Figure 11 shows the result of the `C-CD8` algorithm, figure 12 shows the result of the `C-FDBD` algorithm.

Again, we observe that the results of `C-FDBD` are better than those of `C-CD8`, in spite of the fact that `C-CD8` is a second-order method and `C-FDBD` a first-order method. However, the difference between the two methods is that `C-CD8` transforms the physical velocity field into a continuous velocity field in C-space, while `C-FDBD` transforms it into a discontinuous velocity field. Since the grid lines are not differentiable at the grid nodes, the transformed velocity field in C-space must be discontinuous. We will return to this subject in the discussion (section 6.4).

6.2 Interpolation effect

To investigate the effect of the interpolation method on the accuracy of the particle paths, the P-space algorithms (`P-4-IDW`, `P-4-VOL`, `P-4-TRI`) are examined. These algorithms differ only in their interpolation components. Again, the flow in the L-pipe was used. The same particles as in the previous test are released and their paths traced. Figure 13 shows the

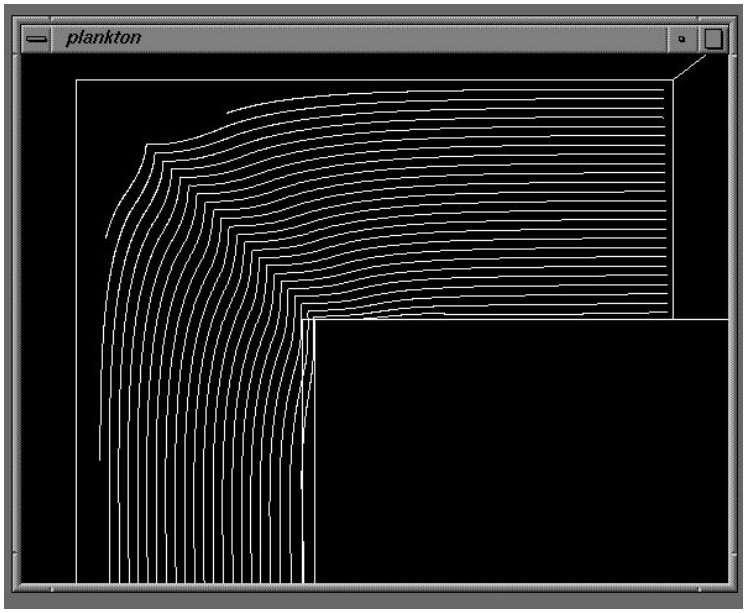

Figure 11: Particle paths in L-shaped pipe; Algorithm `C-CD8`

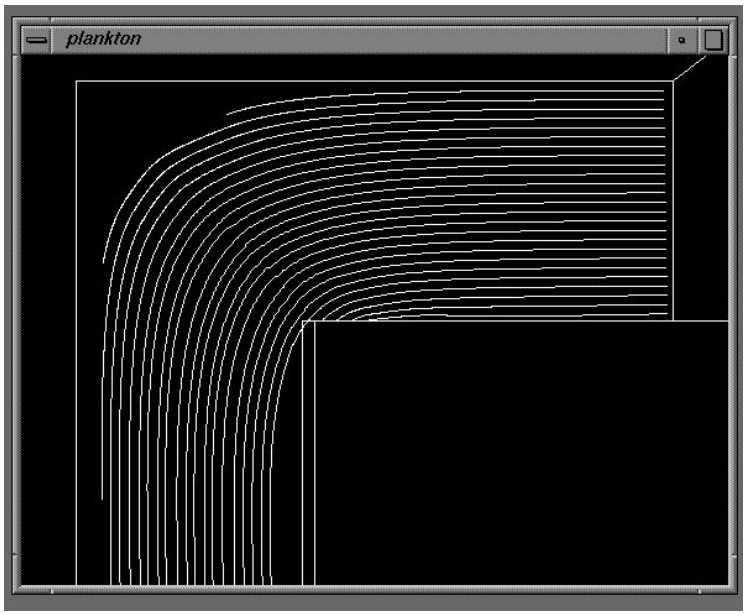

Figure 12: Particle paths in L-shaped pipe; Algorithm `C-FDBD`

results for the P-4-VOL algorithm. The results for the other P-space algorithms cannot be visually distinguished and are therefore not included.

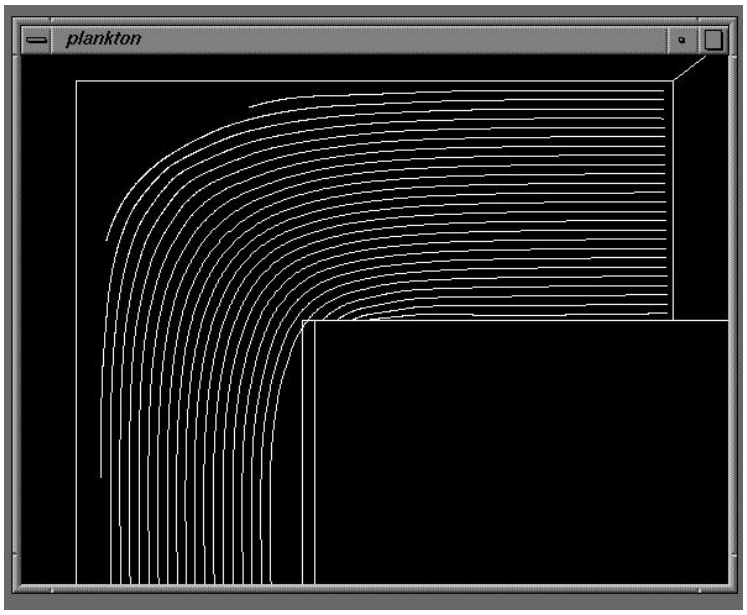

Figure 13: L-pipe particle paths; Algorithm P-4-VOL

6.3 Speed comparison

All algorithms were applied to the flow in the L-shaped pipe and total execution times were measured with timing routines. Table 3 lists the results in seconds. The platform used was a SiliconGraphics Iris 4D/310-VGX workstation.

Algorithm	P-4-IDW	P-4-VOL	P-4-TRI	P-SW-TRI
Execution time	56.2	42.4	96.5	151.6
Algorithm	C-FD1	C-FD8	C-CD8	C-FDBD
Execution time	28.3	76.2	76.0	83.1

Table 3: Timing results

6.4 Discussion

Accuracy: interpolation

In general, the differences in accuracy between the P-space algorithms caused by various interpolation methods are marginal. Both the semi-3D and 3D test cases that we have used

have shown this. Only in some cases in vector field topology analysis has the interpolation error turned out to become more important.

Accuracy: velocity field transformation

The accuracy of C-space algorithms is largely determined by the quality of the transformation. `C-FDBD` is the most accurate algorithm. This can be explained from an analysis of the transformation based on a simple test case. Consider the uniform vector field in the four curvilinear cells in figure 14.

When transforming this vector field to computational space, the vectors at the left and right borders ($x = 0$ and $x = 3$) are not a problem because the grid lines are differentiable there. The problem lies in the vectors in the middle of the grid, especially the vector in (2, 1).

Now, assume that particles are released in (1, 0) and (2, 0). In P-space this would result in straight the particle paths in $x = 1$ and $x = 2$ (see figure 15). When these particle paths are transformed to C-space, we obtain the curved paths shown in figure 16. Note that here, not vectors are transformed, but *positions* on the paths, using the Newton-Raphson iteration described earlier.

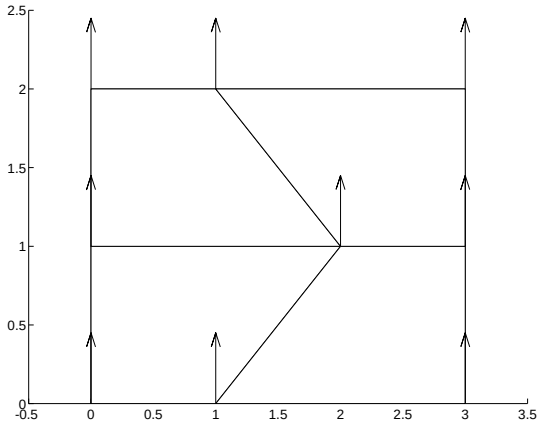

Figure 14: Uniform vector field in four cells

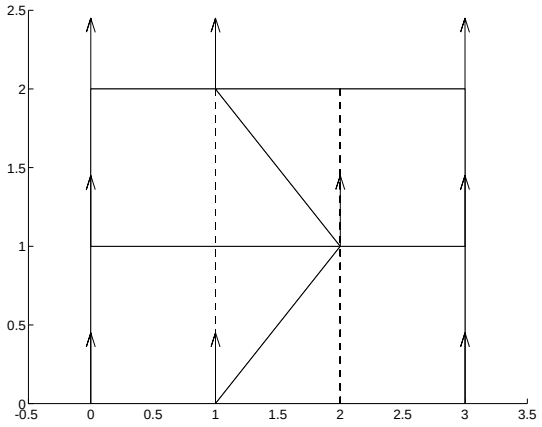

Figure 15: Straight particle paths in P-space

Consider the rightmost path, with a sharp transition at (1, 1). At the cell boundary, the direction of the path changes sharply. Since the path contains grid node (1,1) and the velocity in a grid node is defined entirely by the velocity in that node, this means that the velocity vector in (1, 1) must also change in the same way. In other words: the velocity field must be *discontinuous* over cell boundaries. Consequently, the velocity in the corresponding point in P-space (2, 1) must map to different vectors in C-space, depending on which cell the node belongs to.

The transformation that has this property is FDBD, because different Jacobians are used in a node, depending on the cell it belongs to. In other words: the continuous velocity field in P-space must be transformed to a discontinuous field in C-space. The reason for this, is that

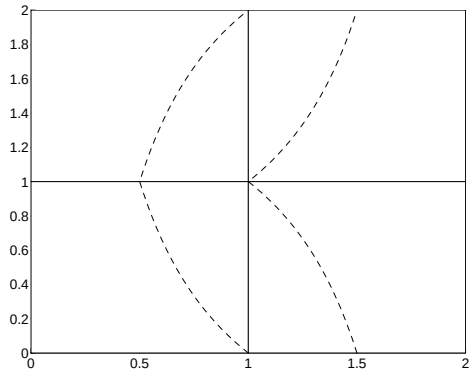

Figure 16: Curved particle paths in C-space

the grid lines are also discontinuous, since they are assumed to be straight lines connecting the grid nodes.

This example also shows that the transformations which use either one Jacobian per cell or one Jacobian per node give less accurate results, even if they are second-order, such as the central differencing in the `C-CD8` algorithm. It can also be verified analytically that the tangent vectors of the particle paths are correctly calculated, if all the vectors are transformed with FDBD.

Speed

`C-FD1`, which uses one Jacobian per cell, is the fastest algorithm. The other C-space algorithms are roughly six times as slow. This can be explained from the fact that eight Jacobians per cell are calculated rather than one.

The P-space algorithms are in general faster than the C-space algorithms that use eight Jacobians, although the efficiency of the C-space algorithms could probably be improved upon by applying some optimizations. Among the P-space algorithms, `P-4-VOL` is the fastest one. Although the calculation of the volume weights is more complex than in inverse distance weighting, this disadvantage seems to be outweighed by the fact that only four weights in each cell need to be calculated, rather than eight.

7 Conclusions

Particle tracing algorithms have been decomposed into their characteristic components. Various alternatives for these components have been implemented and compared. A distinction has been made for computational space (C-space) and physical space (P-space) algorithms.

The most important component in C-space algorithms is the transformation. Varying this component has a large impact on the accuracy of the results. Varying the interpolation component in P-space algorithms has much less effect on the results.

The accuracy of C-space algorithms depends highly on the deformation of the grid. In Cartesian grids, C-space algorithms are identical to P-space algorithms. If the transformation

is not calculated correctly, the accuracy of C-space algorithms decreases rapidly in deformed grids. The best C-space algorithm can be as good as P-space algorithms.

The reason for considering particle tracing in C-space was efficiency. However, tests have shown that most C-space algorithms are computationally more expensive than P-space algorithms, at least when the data is provided in P-space.

In general, P-space algorithms are more accurate and efficient than C-space algorithms. The effect of varying the interpolation method is small compared to the transformation effect. Therefore, the use of C-space is less useful for visualization purposes.

Acknowledgements

This work was carried out as the first author's Master's thesis project. We wish to thank his supervisor, Arthur Mynett of Delft Hydraulics, for his help and encouragement and for his comments on earlier versions of this paper. We thank Jan Mooiman of Delft Hydraulics, for providing interesting data sets and for many discussions on fluid dynamics. This work was supported by Delft Hydraulics.

References

Buning, P. 1989a. Numerical Algorithms in CFD Post-Processing. *In: Computer Graphics and Flow Visualization in Computational Fluid Dynamics*. Lecture Series 1989-07. Von Karman Institute for Fluid Dynamics, Brussels, Belgium.

Buning, P. 1989b. Sources of Error in the Graphical Analysis of CFD Results. *In: Computer Graphics and Flow Visualization in Computational Fluid Dynamics*. Lecture Series 1989-07. Von Karman Institute for Fluid Dynamics, Brussels, Belgium.

Garrity, M. 1990. Raytracing Irregular Volume Data. *Computer Graphics*, **24**(5), 35–40.

Gerritsen, M. 1988. *Geometrical modelling of 3D Aerodynamic Configurations*. M.Phil. thesis, Technische Universiteit Delft, Faculteit Lucht- en Ruimtevaart.

Kontomaris, K., & Hanratty, T.J. 1992. An Algorithm for Tracking Fluid Particles in a Spectral Simulation of Turbulent Channel Flow. *Journal of Computational Physics*, **103**, 231–242.

Mooiman, J. 1993. *Private Communications*.

Neeman, H. 1990. A Decomposition Algorithm for Visualizing Irregular Grids. *Computer Graphics*, **24**(5), 49–62.

Shirayama, S. 1993. Processing of Computed Vector Fields for Visualization. *Journal of Computational Physics*, **106**, 30–41.

Strid, T., Rizzi, A., & Oppelstrup, J. 1989. Development and Use of Some Flow Visualization Algorithms. *In: Computer Graphics and Flow Visualization in Computational Fluid Dynamics*. Lecture Series 1989-07. Von Karman Institute for Fluid Dynamics.

Watson, D.F. 1992. *Contouring: A Guide to the Analysis and Display of Spatial Data*. Computer Methods in the Geosciences, vol. 10. Pergamon Press.

Wilhelms, J., Challinger, J., Alper, N., & Ramamoorthy, S. 1990. Direct Volume Rendering of Curvilinear Volumes. *Computer Graphics*, **24**(5), 41–47.

Williams, P. 1992. *Interactive Direct Volume Rendering of Curvilinear and Unstructured Data*. Ph.D. thesis, Univeristy of Illlinois.

Yeung, P.K., & Pope, S.B. 1988. An algorithm for Tracking Fluid Particles in Numerical Simulations of Homogeneous Turbulence. *Journal of Computational Physics*, **79**, 373–416.