Improving the Performance of Particle Tracing in Curvilinear Grids

J.P.M. Hultquist
hultquist@nas.nasa.gov

Report RNR-94-005,
March 1994

Numerical Aerodynamic Simulation Systems Division
NASA Ames Research Center, Mail Stop T045-1
Moffett Field, CA 94035-1000

```
@inproceedings{xxx,
 author    = "J.P.M. Hultquist",
 title     = "Improving the Performance of Particle Tracing
                 in Curvilinear Grids",
 note      = "AIAA Paper 94-0324",
 booktitle = "AIAA Aerospace Sciences Meeting",
 address   = "Reno, NV",
 month     = jan,
 year      = 1994}
 note      = "(also RNR Tech Report 94-005)"}
```

Abstract

The tracing of particles through curvilinear gridded data is a common task in the visualization of computed flow fields. This is most often done using a numerical integration method over the interpolated physical-space vector samples. The speed of this calculation can be improved by exploiting *spatial coherence*; that is, the geometric proximity of the successive query points generated by the numerical integration method.

Nomenclature

ξ, η, ζ	computational-space coordinates
i, j, k	integer parts of ξ, η, ζ
α, β, γ	fractional parts of ξ, η, ζ
δ	interpolation coefficient
$\vec{x}$	physical-space position
$\vec{u}$	vector field value (typically velocity)
$\vec{p}, \vec{q}$	position or vector samples
$\mathcal{I}$	trilinear interpolating function
$\mathcal{F}$	point-finding "distance" function

1 Background

Several useful methods of vector field visualization are based on the calculation of tangent curves through an interpolated vector field. If this field is the velocity of a steady fluid flow, then each curve defines the path traveled by a massless advected particle. Such curves are called *particle traces* or *streamlines*. Most flow visualization packages allow the user to specify a set of initial points from which a family of streamlines is computed.

Buning[2] asserts that "... graphical analysis is first and foremost an interactive process." Weston[9] has added that:

> Particle traces can be challenging to display and time-consuming to set up because of the need to keep the number of lines small in order to avoid confusion, while still capturing the essence of the features of interest in the flowfield.

Interactive placement of the initial seed points, coupled with the rapid calculation of the resulting curves, is essential for the effective exploration of non-trivial three-dimensional flow fields. Unfortunately, particle tracing through curvilinear grids is among the more computationally expensive of the common visualization methods. Thus, visualizing complicated flow fields is far more difficult and time-consuming than one might hope.

1.1 Particle Tracing

Each streamline curve is the solution of the initial value problem posed by a vector field $\vec{u}(\vec{x})$ and an initial seed point $\vec{x}_0$. This curve can be depicted by computing a sequence of points $(\vec{x}_0, \vec{x}_1, \ldots \vec{x}_n)$ such that

$$\vec{x}_{i+1} = \vec{x}_i + \int_{t_i}^{t_{i+1}} \vec{u}(\vec{x}(t))dt$$

for a sequence of closely spaced values (t_i). Adjacent points are then joined by line segments to produce a piecewise-linear graphical representation of the ideal curve.

In the visualization program PLOT3D (Buning[3]), as in many such packages, this sequence of points is calculated by numerical integration through interpolated physical-space vector field samples. The field values within each cell are interpolated by a trilinear function of the computational-space coordinates of each query point. The computational-space coordinates are related to the physical-space coordinates by a trilinear *shape function*. Every time an interpolated field value is needed, this function must be inverted to find the computational-space location which maps onto the given physical-space coordinates of the query point. The grid-local coordinates are then used as the weights in the *element function* which interpolates the field values within the current cell. This conversion of each physical-space query point location into its computational-space representation greatly limits the speed of streamline construction.

1.2 Performance Improvement

A more rapid method for calculating streamlines first converts the vector field sample at each node from its original physical-space coordinates to the equivalent computational-space values (Eliasson[5], Shirayama[8]). The streamlines are then computed through the cubical cells of computational space and the resulting curves are mapped into physical-space coordinates for display. Since the interpolation and the integration are thereby expressed in the same regularly sampled space, the calculation of the traces can proceed much more rapidly; no conversion of the query-point coordinates is required. The great speed of the computational space method is attractive, but irregularities in the physical placement of the grid node points can introduce error into the field conversion. While this error is quite small in most cases, the uncertainty it raises makes this method perhaps less useful and certainly less used than the physical-space approach.

Lagrange (1736–1813) showed that the velocity field of an incompressible flow in two dimensions can be described by the scalar *stream function* (ψ). This concept was generalized to three dimensions by Yih[10] to describe a steady three-dimensional compressible flow using two coincident scalar fields. Kenwright and Mallinson[7] developed software which computes these *dual stream functions* in a numerical flow field. The software constructs streamlines by locating the intersections of iso-valued surfaces computed in these two fields. This approach is quite rapid. Its greatest limitation is that features within each cell are represented by a single line segment. Resolution of smaller structures would require a subdivision of the grid cells or perhaps the localized use of a numerical integration method.

More conventional approaches to performance enhancement include the use of adaptive stepsizing and more efficient integration formulae (such as the multistep Adams-Bashforth method in lieu of the more common Runge-Kutta schemes).

1.3 An Application Testbed

I have built an interactive flow visualization package called "Flora." This application can read data and solution files in the PLOT3D format; Flora also accepts PLOT3D command files.

Many of the commands have not yet been implemented, but the software does parse the user input, and then calls the appropriate stub routine. Flora currently supports the commands for reading data files, extracting gridplanes, and placing the rakes for streamlines. (The other functions will be implemented in future versions of the code.) Once the scientist has finished specifying the input data and the visualization parameters, the software creates a scene in which the computed visualization models are displayed.

Flora adds two new features which are not provided in PLOT3D. It can compute and display stream surfaces which originate from a line segment (Hultquist [6]), and it supports the interactive repositioning of the originating point and segment *rakes*. While a rake is in motion, the stream surface or the family of streamlines is repeatedly computed and displayed. Users can move a rake in one scene at high magnification while viewing the resulting model from another angle in a second window. This approach allows the precise placement of several rakes in a few minutes, a vast improvement over what has been previously possible with indirect control and less rapid system response.

Flora computes streamlines using a physical-space second-order Runga-Kutta method with adaptive stepsizing. At this level, the code is similar to that used in PLOT3D and FAST[1]. This quite ordinary integration method is supported by various data management techniques which increase the speed of streamline construction. This paper presents some of these enhancements.

2 Cell Caching

Most visualization software uses a cell-local interpolating function to define the field values within each cell. Under such a mapping, an interpolated field value depends only upon the samples recorded at the vertices of the cell which contains the query point. Rather than indexing directly into a large arrays of grid position and solution data, Flora copies the node position and vector field samples for the eight corners of the current cell into two small buffers of (8×3) words each. This copying isolates the interpolation software from the format of the data files, it reduces the time required to interpolate field values at successive query points in the same cell, and it supports the on-the-fly calculation of derived flow field quantities.

2.1 Cell Indices

Each block in a composite grid implicitly defines a new coordinate space over its portion of the flow domain. The coordinates (ξ, η, ζ) of a point in this *computational space* can be divided into their integer and fractional parts:

$$\begin{bmatrix} \xi \\ \eta \\ \zeta \end{bmatrix} = \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}$$

which specify the index (i, j, k) of the lower corner of the the cell containing the point and the fractional displacements $(\alpha, \beta, \gamma) \in [0...1]$ within that cell. Each node point has an integer value for each of its computational-space coordinates; these values are also the index for each node point in the arrays that store the grid and field values for that block.

2.2 Implementing the Caching

Access to the flow field data is mediated in Flora by an instance of a `particle` data structure, which can specify a location in both the physical and the computational coordinate space defined by the current block. The particle structure contains pointers to the large arrays of grid and solution data. This structure also caches a copy of the position and vector field samples for the eight corner nodes of the current cell.

As a particle is advected through the flow domain, the computational-space coordinates of each new query point are split into their integer and fractional parts. The integers form the index of the node at the lowest-indexed corner of the containing cell. If this index is unchanged from that of the previous query point, then the interpolation can proceed with the previously cached samples. This avoids the repeated fetching of these values from scattered locations throughout the grid and solution data arrays.

If the cell index has changed, then the vertex positions and field samples for the new cell must be loaded into the particle structure. The three index values are multiplied by previously computed offsets for each block dimension, and the products are summed to form the total addressing offset for the lowest-indexed node. The additional offsets for the other seven corners of the cell depend only upon the dimensions of the current block. The position and vector field samples are loaded into contiguous memory words in the particle structure, starting from the lowest-indexed vertex and continuing in column-major (FORTRAN) order to the sample data for the diagonally opposite vertex.

```
offset = ((i*di) + (j*dj) + (k*dk));
for each component {
  src = field_data + offset;
  dst = particle_cache;
  dst[0] = src[ 0 ];
  dst[1] = src[ di ];
  dst[2] = src[ dj ];
  dst[3] = src[ di + dj ];
  dst[4] = src[ dk ];
  dst[5] = src[ di + dk ];
  dst[6] = src[ dj + dk ];
  dst[7] = src[ di + dj + dk ];
}
```

This caching of the vertex positions and field samples isolates the interpolation code from the format of the field data, which sometimes interleaves the components of vector samples and sometimes stores the values for each physical-space dimension in separate arrays.

Caching of the sample values also supports the deferred or *lazy* calculation of field data, such as for computing velocity from the density and momentum samples at the vertices of each newly encountered cell. If only a few streamlines are to be calculated, this deferred calculation is much more efficient than code which first computes the value of the new field at every node point in the entire grid before computing any streamlines.

2.3 Working Set Size

In any type of caching scheme, the notion of the *working set* may be applied (Denning[4]). In a virtual memory computer system, the working set is the collection of memory segments or pages which have been accessed by a process over some specified interval of time. The number of different segments or pages in the working set is called the *working set size*. This is a measure of the *locality* of the process; that is, how many different areas of memory have been accessed by the process over that time interval. A process with a smaller working set size will tend to run more quickly, since a higher percentage of its text and data can reside in the faster hardware of the CPU cache.

We may consider the locality of an advecting particle. Clearly, the particle resides at any given moment in a single cell; however, the query points of an integration algorithm may fall within neighboring cells. Thus, performance might be further improved by caching the data for more than one cell in each particle structure. A *cache miss* occurs when the data for the enclosing cell is not already in the cell-cache. The data for one of the previously encountered cells must then be overwritten by the data for the new cell. Typically, the cell which is removed from the cache is that which was "least recently used" and presumably (but not always!) the least likely to be used soon.

To investigate the effect of cell-cache size on the frequency of cell-cache misses, I computed several streamlines through a three-dimensional flow field. The integration method was second-order Runge-Kutta in physical space, using stepsizes scaled to produce points separated by at most one-fifth of a cell-width in any grid dimension. The test was repeated several times, with a cache size ranging from zero to four cells. The CPU times were measured on a Silicon Graphics 320-VGX, using a single 33 megaHertz processor.

CACHE SIZE	QUERY POINTS	CACHE MISSES	CPU SECONDS
0	3846	3846	5.9
1	3846	435	2.8
2	3846	435	2.8
3	3846	434	2.8
4	3846	434	2.8

For a simple Runge-Kutta scheme, the locality is quite high and little improvement is gained by having space for more than one cell in the cache. Other methods have somewhat worse locality, as for a truly adaptive scheme which effectively executes two integration methods in parallel. A cache of two cells or more would allow each method to advance independently, without unfortunate interaction in the particle data cache.

Note the great difference in speed between having zero and one cell in the cache. By loading the data for the current cell, we can avoid much of the indexing arithmetic for accessing the data from the original grid and solution arrays. The cell cache becomes even more valuable as the computational effort required for loading new field data increases. In the experiment above, the momentum was divided by density to find the fluid velocity at the vertices of each newly encountered cell. More involved calculations might be used to evaluate the curl of a vector field or the gradient of some scalar flow field measure. But even in these cases, the cache need only be large enough to contain the data for one or two cells.

Additional savings could be obtained by recognizing that adjacent cells share four vertices. The already cached values for these shared nodes can be copied directly into four of the slots in the newly vacated cell-cache entry; these samples

need not be recomputed from the original flow data. This has not yet been implemented in Flora, but would be worthwhile for vector fields which are more time-consuming to compute.
Summary: The flow field values at each query point are typically defined by an interpolation of the values stored at the vertices of the enclosing cell. By caching these samples, we can reduce the cost of accessing this data for subsequent query points in this same cell.

3 Interpolation

A trilinear interpolating function produces a first-order continuous, piecewise-linear reconstruction of the ideal flow field. If the position and vector field samples for the current cell are loaded into small contiguous buffers, then a rapid and general implementation of the interpolating function becomes possible.

3.1 Trilinear Interpolation

In the cell-local trilinear interpolating function, the relative contribution of each vertex sample is a product of the fractional displacements of the query point along each computational-space dimension:

$$\begin{aligned} \alpha_0 &= (1-\alpha) & \alpha_1 &= \alpha \\ \beta_0 &= (1-\beta) & \beta_1 &= \beta \\ \gamma_0 &= (1-\gamma) & \gamma_1 &= \gamma \end{aligned}$$

$$\begin{aligned} \mathcal{I}_{ijk}(\alpha,\beta,\gamma) = \; & [\alpha_0\beta_0\gamma_0]u_{000} + \\ & [\alpha_0\beta_0\gamma_1]u_{001} + \\ & [\alpha_0\beta_1\gamma_0]u_{010} + \\ & [\alpha_0\beta_1\gamma_1]u_{011} + \\ & [\alpha_1\beta_0\gamma_0]u_{100} + \\ & [\alpha_1\beta_0\gamma_1]u_{101} + \\ & [\alpha_1\beta_1\gamma_0]u_{110} + \\ & [\alpha_1\beta_1\gamma_1]u_{111} \end{aligned}$$

The interpolation weight which is applied to each vertex sample is the computational-space volume bounded by the opposite vertex and the position of the query point. For example, if the offsets (α, β, γ) are all equal to one third, their product of $1/27$ is applied as the coefficient for the sample value (u_{111}) at the uppermost-indexed vertex of the current cell.

3.2 Operation Counts

The interpolating function as written above requires 10 additions and 24 multiplications for each component of the field being sampled. The interpolating of a vector in three dimensions would therefore require $3 \times (10 + 24)$ or 102 floating-point operations. Factoring of common sub-expressions and the pulling of some invariant expressions out of the loop body can bring these totals down to 3 additions and 12 multiplications in a pre-iterative setup phase, with 7 additions and 8 multiplications required for each component of the sampled field. This rearrangement brings the total down to 60 operations in the case of three-dimensional vectors.

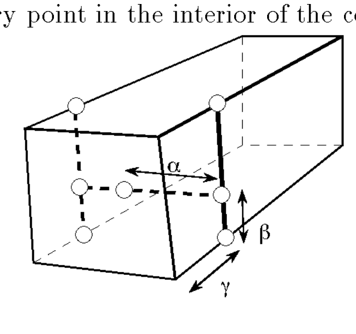
A different factoring uses a sequence of seven linear interpolations or *lerps*. Each lerp produces the weighted average of two values, using an interpolation coefficient (δ) ranging between zero and one.

$$\text{LERP}(\delta, \vec{p}, \vec{q}) = ((1 - \delta)\vec{p} + \delta\vec{q}) = (\vec{p} + \delta(\vec{q} - \vec{p}))$$

The trilinear interpolation may be computed by using the third computational-space offset γ to combine the eight original vertex samples into four interpolated values. These intermediate results are combined pairwise using β as the interpolation weight, and the final value is computed using α as the weight in a single step.

$$\begin{aligned}
u_{00\gamma} &= \text{LERP}(\gamma, u_{000}, u_{001}) \\
u_{10\gamma} &= \text{LERP}(\gamma, u_{100}, u_{101}) \\
u_{10\gamma} &= \text{LERP}(\gamma, u_{010}, u_{011}) \\
u_{11\gamma} &= \text{LERP}(\gamma, u_{110}, u_{111}) \\
u_{0\beta\gamma} &= \text{LERP}(\beta, u_{000}, u_{\alpha 10}) \\
u_{1\beta\gamma} &= \text{LERP}(\beta, u_{101}, u_{\alpha 11}) \\
u_{\alpha\beta\gamma} &= \text{LERP}(\alpha, u_{0\beta\gamma}, u_{1\beta\gamma})
\end{aligned}$$

This method is depicted graphically in the figure below. The first four steps produce the values interpolated along four edges of the cell. The next two steps compute values on opposite faces. The final evaluation yields the interpolated value at the specified query point in the interior of the cell.



This implementation requires 14 additions and 7 multiplications for each component of the sampled field, with no provision for pulling common operations outside the loop. The three different methods of implementing the interpolation function yield the following operation counts for the case of three-dimensional vector samples:

	OPERATIONS		$(n = 3)$
METHOD	add	mul	total
SIMPLE VOLUME	$10n$	$24n$	102
FACTORED VOLUME	$3 + 7n$	$12 + 8n$	60
SUCCESSIVE-LERP	$14n$	$7n$	63

The successive-lerp method and the factored version of the volume method have roughly a comparable performance of about 60 operations for vector samples in three dimensions. Each runs in about two-thirds of the time required by the straightforward and non-optimized implementation of the volume method.

3.3 A General Implementation

The caching of the vertex samples allows a very general implementation of the *successive-lerp* method for evaluating the interpolating function. This simple code can interpolate field samples with any number of components within cells having any number of dimensions. For vector samples in a three-dimensional cell, the calculation of the interpolated value consists of three iterations of a loop which repeatedly combines the upper and lower halves of the cell-cache:

```
num = cache_size;
src = cache_data;
dst = tmp_buffer;
for each offset (...γ, β, α)
  del =  next fractional offset
  num = num/2;
  lo  = src;
  hi  = src + num;
  DOTIMES(i,num) {
    dst[i] = LERP(del, lo[i], hi[i]);
  }
  src = tmp_buffer;
}
```

The first pass combines the four cached samples on half of the cell with their counterparts on the opposite cell face. These interpolations use the offset γ as the coefficient to produce four new values which are written into a temporary buffer. These intermediate values are combined pairwise using β as the interpolation weight, again merging the two halves of the array by pairwise interpolation. These results can be written back into the same temporary buffer, since the previous values are no longer needed. These two new values are combined into a single result using α as the interpolation coefficient. This iteration of pairwise interpolations could, of course, be unrolled for the frequently occurring case of three-dimensional vectors on the eight corners of a three-dimensional cell.

Summary: The query point lies within some cell. The position and vector field data at the corners of this cell can be copied into a small buffer. This cell-cache supports a rapid and general implementation of the interpolating function.

4 Computational-Space Extrapolation

When a particle is advected by physical-space integration, the computational-space coordinates for each new query point must be found. We have seen that these coordinates are then split into their integer and fractional parts. The integer values identify the enclosing cell and the fractional parts are used as the weights in the interpolating function. Finding these computational-space coordinates for each new query point typically involves a Newton-Raphson method which iteratively computes a sequence of computational-space positions which lie increasingly close to the desired location. The iteration is often started from the computational-space position identified by the previous invocation of the point-finding method. A closer starting point can be found by extrapolating, in computational coordinates, some short distance beyond the end of the growing streamline. This improved initial estimate can reduce the number of iterations required for the convergence of the Newton-Raphson method, and this can increase the speed of the physical-space integration of streamlines.

4.1 The Point-Finding Problem

The *point-finding* problem is to identify the real-valued computational-space coordinates which interpolate onto a specified physical-space location. This can be posed as a multi-dimensional root-finding problem on a distance function $\mathcal{F}$ that measures the error between the specified physical-space location ($\vec{x}^\star$) and the interpolated result at a given offset (α, β, γ) in the current cell:

$$\mathcal{F}(\xi, \eta, \zeta) = (\mathcal{I}_{ijk}(\alpha, \beta, \gamma) - \vec{x}^\star) = \vec{0}$$

Since the grid is bent in physical space, this field may have several non-zero minima. The zero-point is usually found using two phases: a *searching* method that finds a candidate point in the neighborhood of the proper minimum, and a subsequent *polishing* method that refines this approximate value into the final answer. The initial search is used to ensure that the subsequent refinement phase does not inadvertently fall into an erroneous local minimum.

The Newton-Raphson method is commonly used in the polishing phase. This method begins in some cell at some offset $\vec{\alpha}_0 (= [\alpha, \beta, \gamma]_0)$. This estimated position is repeatedly shifted though a sequence of new locations $\vec{\alpha}_{n+1}$ which, one hopes, lie increasingly more close to the unknown ideal location $\vec{\alpha}^{\star}$. This method is based on the recurrence relation:

$$\vec{\alpha}_{n+1} = \vec{\alpha}_n - \frac{\mathcal{F}(\vec{\alpha}_n)}{\mathcal{F}'(\vec{\alpha}_n)}$$

The denominator of this equation is the 3×3 Jacobian matrix which contains the partial derivatives of the interpolating function. This matrix is constructed from finite-differences of the node-point positions. This matrix is inverted, and the result used to map the current physical-space error measure into its computational space representation.

After a few iterations, the distance between successive points ($\vec{\alpha}_i$) should be quite small. If the distance is increasing, then the method has failed. The iteration is halted when the successive values differ by some tiny amount. If the resulting offsets lie outside the range [0...1] in any grid dimension, then the query point lies outside the current cell and outside the domain of the function $\mathcal{F}$. The cell index is shifted at most one step in one or more grid dimensions and the iteration is restarted to solve the new distance function defined by the positions of the corner nodes of this neighboring cell.

4.2 Extrapolation

In a second-order Runge-Kutta method, two query points are generated by each integration step. The first query point is placed beyond the end of the curve using one-half the current stepsize, and the second query point is advanced using an average vector sample scaled by the full stepsize. The point-finding method for each of these query locations is typically begun from the previously identified computational-space coordinates of the most recent query point.

Let $\vec{p}$ be the computational-space position of the most recently computed point on the streamline. Let $\vec{q}$ be the coordinates of its immediate predecessor on this partially computed curve. Then $(\vec{p} - \vec{q})$ is a first-order approximation of the local value of the computational-space vector field. We might expect the point-finding method to converge more quickly if the initial point ($\vec{\alpha}_0$) is first placed at a better computational-space location, shifted beyond $\vec{p}$ by either one-half of the vector $(\vec{p} - \vec{q})$ or by the full amount.

If the two most recent points on the streamline lie in different grid blocks, then this extrapolation approach cannot be used. The numerical difference between the computational-space coordinates of points in two separate blocks has no useful meaning. Similarly, the extrapolation of the curve can yield an estimate beyond the bounds of the current block. Once again, the extrapolation fails in this case. These failures should occur in only a small percentage of the integration steps; most blocks are several tens of cells in each dimension and each integration step typically advances the particle by only fraction of the current cell size.

4.3 Performance Measurement

Several streamlines were calculated with a physical-space second-order Runge-Kutta method, both with and without the computational extrapolation enhancement. In each case, I counted the number of query points generated by the integrator, the number of times the Newton-Raphson method was executed, the total number of iterations of this method, and the elapsed CPU time.

	EXTRAPOLATION?	
	no	yes
QUERY POINTS	3846	3846
CELLS LOADED	435	437
NR INVOCATIONS	4281	3854
NR ITERATIONS	8143	3904
CPU SECONDS	2.8	1.9

Extrapolation prior to the invocation of the point-finding method reduces the CPU time by about one-third. This improvement can be attributed to a reduction in the total number of iterations of the Newton-Raphson method, each of which includes the construction and the inversion of a 3×3 Jacobian matrix.

The improvement is also obtained by reducing the number of times the Newton-Raphson method converges to an offset which lies outside the current cell. When this does happen, the cell index must be adjusted and the method must be restarted in the neighboring cell. By extrapolating in computational-space, the proper cell is used more frequently in the initial attempt and the number of recalculations is reduced. This is shown in the reduction in the number of invocations of the Newton-Raphson method, from 4281 to 3854, the latter being only eight invocations above the minimum possible number of once per query point.

Summary: By extrapolating the partially constructed streamline, we can estimate the computational-space location of the next query point. This reduces the number of invocations of the Newton-Raphson method and it reduces the number of iterations executed within that method.

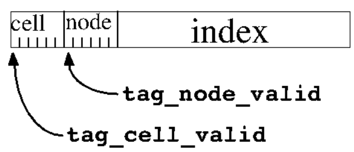
5 Cell Tags

The *iblank* field in a composite grid allocates a full thirty-two bit word to every node point. Disallowed node points, which do not carry valid field samples, are marked with an IBLANK value of 0. Usable node points in the interior of a block are marked with the value 1. The meaning encoded by the IBLANK values must be preserved by the visualization software. The interpolation of field samples at each query point location must be preceded by a check of the IBLANK values at the eight corners of the cell which contains that point. Field values may be interpolated from the samples recorded at the cell vertices only when all eight nodes are marked with non-zero IBLANK numbers.

This method of annotating the grid is rather wasteful, consuming four megabytes for a moderately sized grid of only one million nodes. A more efficient encoding provides more information in the same amount of space, simplifies the sampling of the field data, and makes the code slightly more efficient.

5.1 Valid Cells

In Flora, each node point is assigned a *tag word*, which is subdivided into several single bit flags and a small integer field.



Information for each node is stored in its tag, and comparable information for each cell is stored in the node tag of its lowest-indexed corner. The nodes on the three uppermost faces of each grid block have no associated cell and the cell bits within these tag words are ignored.

Each tag word contains a single-bit flag called `TAG_NODE_VALID`. This is set to TRUE for nodes at which the original IBLANK number was non-zero. The state of this bit indicates whether a node point carries valid sample data. A comparable flag is assigned to each cell and set to the logical-conjunction (AND) of the node flags for its eight corner nodes. A fetch of a single tag word and a test of the `TAG_CELL_VALID` bit-flag within that tag is therefore sufficient for determining the validity of interpolated values within each newly encountered cell.

5.2 Wall Cells

An IBLANK value of 2 marks any node which lies on an impermeable boundary. These node points carry valid field samples, but if all four corners of a cell face are so marked, then no fluid may pass through that face. PLOT3D handles this situation using a special constraint called *wall bouncing*. By explicitly restricting the curves away from these surfaces, the software avoids one particularly annoying artifact of numerical error: streamlines that leave the flow domain by passing through the skin of the vehicle.

Wall-bouncing is not yet implemented in Flora, but the cell tags would allow the convenient marking of these impermeable cell faces. The integration software could then test certain bit-flags in the cell tag and displace the advancing particle as appropriate in any marked cell.

5.3 Boundary Cells

A composite CFD grid contains several partially overlapping or abutting blocks of node points. The node points in the overlap regions carry a negative IBLANK number. The absolute value of this tag is the number, counting from one, of a locally overlapping block.

Two bit-flags have been allocated in each tag word. The node flag **TAG_NODE_DONOR** is true at those nodes in the overlap region between blocks, nodes which carry a negative IBLANK value. A cell flag **TAG_CELL_DONOR** is set to the logical-disjunction (OR) of the donor-flag for the corner nodes of each cell. When a query point falls outside the current block or into an invalid cell, then the donor flag of the previous cell is tested. A TRUE value indicates that one or more vertices of the cell lie in an overlap region. The neighboring block can then be searched for the position of the new query point.

Summary: By allocating a tag word to each node, and subdividing these words into subfields, we can encode more information about the grid than is provided by the IBLANK number. This allows a more convenient and slightly more efficient implementation of certain data query operations.

6 Donor Points

Visualization software must be able to continue the calculation of streamlines through the boundaries which separate adjacent blocks. A negative IBLANK number indicates which other block overlaps or abuts the current block in some region of interest, but a coarse search of the new block is required before the point-finding method can be applied. Augmenting the grid with donor-point locations can eliminate the need for the coarse search and improve the speed of streamline block transitions.

6.1 Storing the Donor-Receiver Equivalences

Within the regions of overlap, the position of each node can be expressed in the computational coordinate space defined by the other block. During the solving of the flow field, every such node point *receives* the interpolated field quantities sampled at the corresponding computational-space *donor point* in the other block. This exchange of data allows for the eventual calculation of consensus values for the flow field quantities in these regions.

Receiver node points generally occur over an entire block face. The neighboring block has its own collection of receiver nodes in this shared region of overlap. Each block provides the computational-space donor points for the receiver nodes of its neighbor. The donor coordinates for these two sets of receiver nodes describe the relative position of the two blocks.

When a grid is loaded into Flora, the software counts the nodes which carry a negative IBLANK number. Storage is then allocated to hold the records for that many donor-receiver pairs. In a typical grid, only about five percent of the nodes are located in overlap regions. Because so few node points are receivers, the donor-receiver information can be recorded in a separate small array of records which accompanies the node position data of each block. This secondary array is indexed by a small integer which is stored in the tag word of each node. Nodes which are not within an overlap region carry a zero index; a non-zero index specifies the record for that receiver node. The number of bits presently allocated in Flora to store the index in the tag word is currently twenty. This supports the indexing of over one million receiver points per block while still leaving twelve bits for the storage of node and cell bit-flags.

6.2 Computing the Donor Points

These computational-space donor point coordinates are sometimes recorded in a secondary data file which is produced by the grid generation code. But the donor point information is not always available in this form. The donor-receiver equivalences can be computed from the node position data and the IBLANK numbers. In the simplest approach to this problem, the first donor-point for each block is found by a coarse search, followed by the usual Newton-Raphson point-finding method. The donor points for the receiver nodes along a single gridline are then computed by repeated invocations of the point-finding method, each time commencing from the computational-space coordinates of the previously identified donor point.

We can improve the speed of this calculation by using computational-space extrapolation. The first two receiver node points on a gridline are processed as before. Donor points for subsequent nodes on this same gridline can then be identified by first advancing by the computational-space distance between the two previously identified donor points. The Newton-Raphson method is started from this improved initial position, thus limiting number of invocations of and iterations within this method. As in streamline calculation, this enhancement can be used only if two previously located computational-space points reside in the same donor block, and only if the extrapolated position also lies within that block. Furthermore, this extrapolation is sensible only when the two previous receiver points and the current receiver node point are successors along the same gridline.

Summary: Donor-point coordinates can be used as an efficient means to continue the calculation of streamlines into neighboring blocks. Computational-space extrapolation can be used to improve the speed at which these donor-point coordinates can be computed from the node position and IBLANK data.

7 Self-Abutting Blocks

Single blocks are often wrapped around a cylindrical body so that node points on opposite block faces are coincident. Blocks are also folded around airfoils, bringing one block face to abut against itself. These two varieties of self-abutting block, the O-type and C-type topologies, exhibit a branch cut in the flow domain which is comparable to the inter-block boundaries discussed earlier. Unfortunately, negative IBLANK numbers are not often provided along the seams of these self-abutting blocks, since this juncture is handled using other mechanisms in most flow solver software.

Unfortunately, the visualization software has traditionally had access only to the IBLANK field, and not the auxiliary files which describe the presence and the form of self-abutting blocks. Many visualization packages (including PLOT3D and FAST) thereby fail to continue streamlines across these unmarked branch cuts. An small heuristic test can be used to cross these boundaries efficiently.

7.1 Using Iblank Numbers

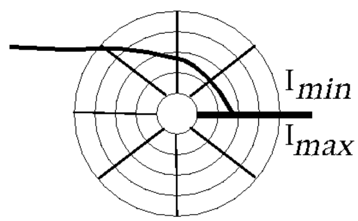
Some researchers have edited the IBLANK field to explicitly indicate the connectivity of self-abutting blocks, after the simulation and prior to the visualization of the computed fields. It would be preferable if the visualization software itself were to handle these special cases without the need for manual post-simulation editing of the IBLANK data.

In a three-dimensional O-type block, the vertices at the corners of a block face create four coincident pairs with the corners of the opposite face. In blocks of the C-type, two corners of one block face are coincident with the two other corner nodes on the same face. In one possible method, the visualization software could compare the relative positions of the eight corners of each block and generate the appropriate IBLANK masks for any block which is thus recognized to be self-abutting. The software would then attempt to compute donor-receiver equivalences across these abutting faces. The transition across such boundaries would then be handled in the manner described earlier, using the donor-receiver equivalences to more precisely indicate where an advected particle re-enters a self-abutting block.

7.2 Implicit Connection

Flora uses a different method for handling this transition. Whenever a streamline exits a block, the Flora software checks the tag word of the most recent valid cell to determine whether there are donor points at any of the vertices of that cell. If no donor points are found, then either the particle has exited the computational domain or it has crossed the unmarked boundary of a self-abutting block. If we assume that the node points are coincident across this boundary, then there are three possible points near which the streamline may have re-entered this same block. One of these is found in blocks of the O-type and the two other cases occur in C-type blocks. If any one of these three re-entry positions prove to be the starting position for a successful point-finding method, then the calculation can be resumed at this new location.

A streamline crossing the seam of a O-type block will exit one block face and re-enter the same block through the opposite face. For a specific example, let us assume that the streamline exits the block through the lower block face in the first grid dimension. Then, if the streamline exits the block from the cell with index (i_{min}, j, k), the point of re-entry will be near the node (i_{max}, j, k) in that same block.



A C-type block may be folded along one of two grid dimensions to bring one block face into a self-abutting state. In the example used in the previous paragraph, the streamline will re-enter the block at either $(i_{min}, j_{max} - j, k)$ or $(i_{min}, j, k_{max} - k)$, depending upon whether the block has been folded along the j or the k grid dimension.

Summary: The branch cuts in C-type and O-type blocks are rarely marked by negative IBLANK numbers. Visualization packages often fail to continue streamlines which cross these boundaries. One simple method of streamline resumption requires only the testing of three node points whenever a streamline exits a block from a cell which carries no donor records. This resumption technique requires no user intervention. It incurs no startup cost for pre-processing the grid. It requires no storage of additional donor records. Most importantly, it properly continues streamlines through self-abutting blocks which match point-to-point across the branch cut.

8 Conclusion

The successive query points generated by a numerical integration method occur close together and in a predictable pattern. This behavior can be exploited by caching the sample values for the vertices of one or more cells, and by extrapolating along partially computed streamlines to reduce the calculation required in the point-finding routine. Precomputing the donor point coordinates avoids the loss of coherence which would otherwise occur at the block boundaries. Extrapolation along lines of receiver points improves the efficiency of donor point calculation. Checking for particle re-entry in self-abutting blocks allows the resumption of streamlines across these previously troublesome boundaries.

9 Acknowledgements

I wish to thank Tom Lasinski for his patient support of this work. I also wish to acknowledge the support and advice of Pieter Buning, Fred Brooks, Jim Coggins, and Henry Fuchs.

References

[1] Gordon V. Bancroft et al. FAST: A multi-processed environment for visualization of computational fluid dynamics. In *Proceedings of Visualization '90*, pages 14–27, San Francisco, CA, October 1990.

[2] Pieter G. Buning. Sources of error in the graphical analysis of CFD results. *Journal of Scientific Computing*, 3(2):149–164, 1988.

[3] Pieter G. Buning and J.L. Steger. Graphics and flow visualization in CFD. In *AIAA 7th CFD Conference*, pages 162–170, Cincinnati, OH, July 1985. AIAA Paper 85-1507-CP.

[4] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 26(1):43–48, January 1983. (Reprinted from a 1967 conference proceedings.).

[5] P. Eliasson, J. Oppelstrup, and A. Rizzi. STREAM3D: Computer graphics program for streamline visualization. *Advances in Engineering Software*, 11(4):162–168, 1989.

[6] J.P.M. Hultquist. Constructing stream surfaces in steady 3d vector fields. In *Proceedings of Visualization '92*, pages 171–178, Boston, MA, October 1992.

[7] David Kenwright and Gordon Mallinson. A streamline tracking algorithm using dual stream functions. In *Proceedings of Visualization '92*, pages 62–68, Boston, MA, October 1992.

[8] S. Shirayama. Visualization of vector fields in flow analysis I. In *29th Aerospace Sciences Meeting*, Reno, NV, January 1991. AIAA Paper 91-0801.

[9] Robert P. Weston. Applications of color graphics to complex aerodynamics analysis. In *AIAA 25th Aerospace Sciences Meeting*, Reno, NV, January 1987. AIAA Paper 87-0273.

[10] Chia-Shun Yih. Stream functions in three-dimensional flows. In *Selected Papers*, pages 893–898. World Scientific, Teaneck, NJ, 1991.