JOURNAL OF COMPUTATIONAL PHYSICS 118, 380–387 (1995)

Robust, Vectorized Search Algorithms for Interpolation on Unstructured Grids

RAINALD LÖHNER

GMU/CSI, The George Mason University, Fairfax, Virginia 22030-4444

Received October 19, 1993; revised November 21, 1994

Several search algorithms for the interpolation of data associated with unstructured grids are reviewed and compared. Particular emphasis is placed on the pitfalls these algorithms may experience for grids commonly encountered and on ways to improve their performance. It is shown how the most CPU-intensive portions of the search process may be vectorized. A technique for the proper interpolation of volumetric regions separated by thin surfaces is included. Timings for several problems show that speedups in excess of 1:5 can be obtained if due care is used when designing interpolation algorithms.

1. INTRODUCTION

The need to interpolate quickly the fields of unknowns from one mesh to another is common to many areas of computational mechanics and computational physics. The following classes of problems require fast interpolation algorithms:

(a) *Simulations where the grid changes as the solution proceeds.* Examples of this kind are adaptive remeshing for steady-state and transient simulations [1–3], as well as remeshing for problems where grid distortion due to movement becomes too severe [4, 5].

(b) *Loose coupling of different codes for multi-disciplinary applications.* In this case, if any of the codes in question are allowed to perform adaptive mesh refinement, the worst case scenario requires a new interpolation problem at every timestep.

(c) *Interpolation of discrete data for the initialization or continuous update of boundary conditions.* Common examples are meteorological simulations, as well as climatological and geotechnical data for seepage and surface flooding problems.

(d) *Visualization.* This large class of problems makes extensive use of interpolation algorithms, in particular for the comparison of different data sets on similar problems.

The main reason that prompted us to revisit the search and interpolation problem was the second class of applications. We are currently developing a series of loosely coupled multidisciplinary codes. We have found that for these classes of problems, interpolation can take a non-negligible portion of total CPU-time, especially for large applications running on multiprocessor vector-computers.

In the following, we will concentrate on the fast interpolation between different unstructured grids that are composed of the same type of elements. In particular, we will consider linear triangles and tetrahedra. The ideas developed are general and can be applied to any type of element and grid. On the other hand, other types of grids (e.g., cartesian structured grids) will lend themselves to specialized algorithms that may be more efficient and easier to implement.

The remainder of the paper is organized as follows. Section 2 describes the basic algorithm used to decide if a point of the unknown grid is inside an element of the known grid. Sections 3–5 consider the fastest possible algorithms, given the amount of information available; brute force if only one point needs to be interpolated (Section 3), octree search for groups of points (Section 4), and the fastest known vicinity algorithm (Section 5). These algorithms are combined in Section 6, yielding the fastest grid-to-grid algorithm, an advancing front vicinity algorithm. We then focus on the main innovations of the present paper: ways of improving robustness and speed by minimizing brute-force searches at corners and edges, vectorization of the interpolation procedure, and techniques to interpolate properly volumetric data separated by thin surfaces. Section 9 presents some timings, showing the considerable speedups obtained by the proposed approach. Finally, some conclusions are drawn.

2. THE BASIC ALGORITHM

Consider an unstructured finite element or finite volume mesh, as well as a point p with coordinates $\mathbf{x}_p$. A straightforward way to determine if the point p is inside a given element el is to determine the shape-function values of p with respect to the coordinates of the points belonging to el:

$$\mathbf{x}_p = \sum_i N^i \mathbf{x}_i. \tag{1}$$

0021-9991/95 $6.00

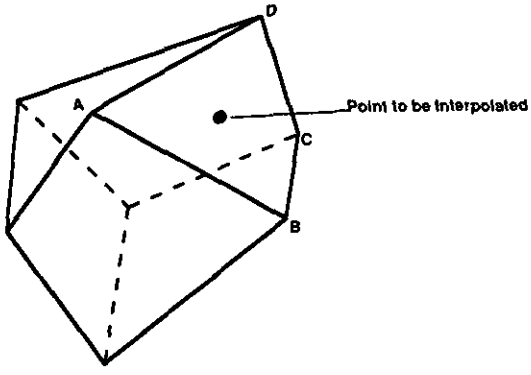

FIG. 1. Possible non-uniqueness for interpolation on bricks.

For triangles in 2D and tetrahedra in 3D, we have, respectively, two equations for three shape-functions and three equations for four shape-functions. The sum-property of shape-functions,

$$\sum_i N^i = 1, \tag{2}$$

yields the missing equation, making it possible to evaluate the shape-functions from the following system of equations:

$$\begin{Bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{Bmatrix} = \begin{Bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \\ 1 & 1 & 1 \end{Bmatrix} \cdot \begin{Bmatrix} N^1 \\ N^2 \\ N^3 \\ N^4 \end{Bmatrix}, \tag{3}$$

or, in concise matrix notation,

$$\mathbf{x}_p = \mathbf{XN} \rightarrow \mathbf{N} = \mathbf{X}^{-1}\mathbf{x}_p. \tag{4}$$

Then, the point p is in element el iff

$$\min(N^i, 1 - N^i) \geq 0, \quad \forall i. \tag{5}$$

For other types of elements more nodes than equations are encountered. The easiest way to determine if a point is inside an element is to split the element into triangles or tetrahedra and evaluate each of these sub-elements in turn. If the point happens to be in any of them, it is inside the element. This procedure may not be unique for highly deformed bricks, as shown in Fig. 1. Depending on how the diagonals are taken for the face A-B-C-D, the point to be interpolated may or may not be inside the element. Therefore, subsequent iterations may be required for bricks or higher-order elements with curved boundaries. Other ways to determine if a point is inside a bilinear element may be found in [6].

In the following, we will use the algorithm outlined above for triangles and tetrahedra as the starting point for improvements in performance. These improvements depend on the assumptions one can make with respect to the grids employed and the information available.

3. FASTEST 1-TIME ALGORITHM: BRUTE FORCE

Suppose we only have a given grid and a single point p with coordinates $\mathbf{x}_p$. The simplest way to find the element into which point p falls is to perform a loop over all the elements, evaluating their shape-functions with respect to $\mathbf{x}_p$:

```
—DO: Loop over all the elements
       — Evaluate N^i from Eq. (4);
       — IF: Criterion (5) is satisfied:
              Exit
         ENDIF
  ENDDO
```

Because the central loop over all the elements can readily be vectorized this algorithm is extremely fast. We will use it in more refined algorithms both as a start-up procedure, as well as a fall-back position.

4. FASTEST N-TIME START ALGORITHM: OCTREE SEARCH

Suppose that, as before, we only have a given grid, but, instead of just one point p, a considerable number of points has to be interpolated. In this case, the brute-force algorithm described before will possibly require a complete loop over the elements for each point to be interpolated, and, on average, a loop over half the elements. A significant improvement in speed may be realized by only checking the elements that cover the immediate neighbourhood of the point to be interpolated. A number of ways can be devised to determine the neighbourhood (see Fig. 2):

—Bins, i.e., the superposition of a cartesian mesh [7, 8],

—Octrees, i.e., the superposition of an adaptively refined cartesian mesh [9, 10], and

—Alternate digital trees [11].

We consider octrees here, as bins perform poorly for problems where the nearest-neighbour distances vary by more than two orders of magnitude in the domain. One may form an octree with the element centroids or points. In the present case, we chose the latter option, as for tetrahedral grids the number of points is significantly less than the number of elements. The octree search algorithm then proceeds as follows:

— Form the octree for the points of the given mesh;
— Form the list of elements surrounding points for the given mesh;

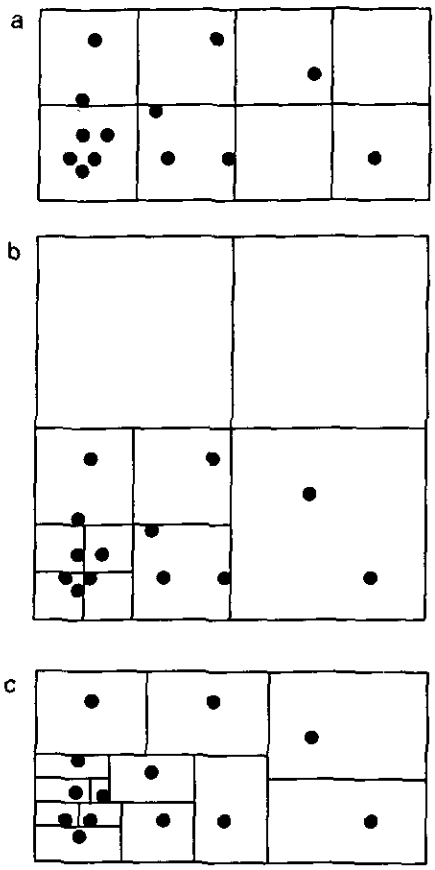

FIG. 2. Possible ways of subdividing space: (a) bins; (b) quadtree (octtree); (c) alternate digital tree.

— DO: Loop over the points to be interpolated
  — Obtain close points of given mesh from the octree;
  — Obtain the elements surrounding the close points;
  — DO: Loop over the close elements:
    Evaluate N^i from Eq. (4);
    IF: Criterion (5) is satisfied:
      Exit
    ENDIF
  ENDDO
— IF: We have failed to find the host element:
  Use brute-force over the elements
  ENDDO

Several improvements are possible for this algorithm. One may, in a first pass, evaluate the closest point of the given mesh to $\mathbf{x}_p$ and only consider the elements surrounding that point. Should this pass, which in general is successful, fail, the elements surrounding all the close points are considered in a second pass. Should this second pass also fail (see Fig. 3 for some pathological cases), one may either enlarge the search region, or use the brute-force algorithm described above in Section 2. The octree search algorithm is scalar for the first (integer) phase

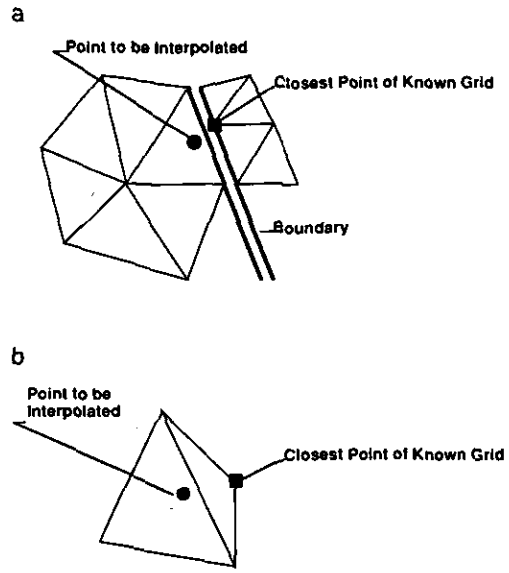

FIG. 3. Possible problems with closest point algorithm: (a) boundary gap; (b) distorted elements.

(obtaining the close points and elements), but all other stages may be vectorized. The vector lengths obtained for 3D grids are generally between 12 and 50, i.e., sufficiently long for good performance.

5. FASTEST KNOWN VICINITY ALGORITHM: NEIGHBOUR-TO-NEIGHBOUR

Suppose that, as before, we only have a given grid and a considerable number of points need to be interpolated. Moreover, assume that for any given point to be interpolated, an element of the known grid that is in the vicinity is known. In this case, it may be faster to jump from neighbour to neighbour in the known grid, evaluating the shape-function criterion [12] (see Fig. 4). If the element into which $\mathbf{x}$ falls can be found in a few attempts (<10), this procedure, although scalar, will

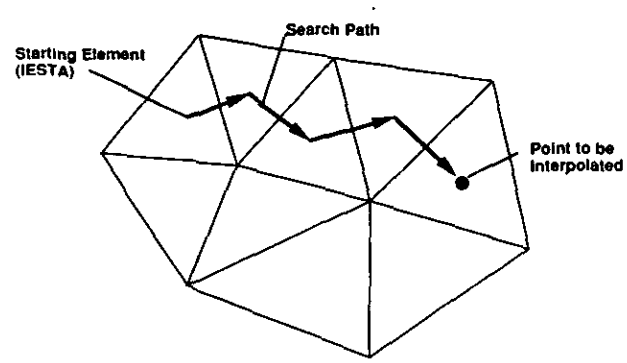

FIG. 4. Nearest neighbour jump algorithm.

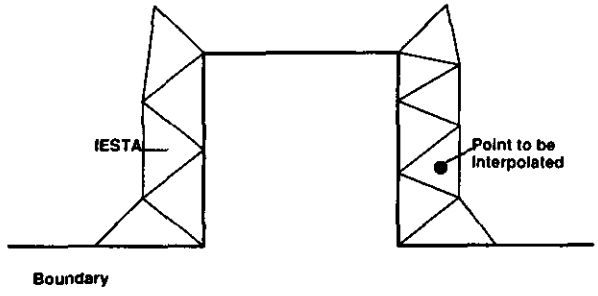

FIG. 5. Failure of nearest neighbour search algorithm.

outperform all other ones. The neighbour-to-neighbour search algorithm may be summarized as follows:

N.0. Form the List of Elements Adjacent to Elements for The Given Mesh;

N.1. DO: Loop over the points to be interpolated
N.2. Obtain good starting element START_ELEMENT;
N.3. For START_ELEMENT: Evaluate N^i from Eq. (4);
N.4. IF: Criterion (5) is satisfied THEN
Exit
ELSE
Set: START_ELEMENT to neighbour associated with min(N^i);
GOTO N.3
ENDIF
ENDDO

The neighbour-to-neighbour algorithm performs very well in the domain, but it can have problems on the boundary. Whereas the brute-force and octree search algorithms can "jump" over internal or external boundaries, the neighbour-to-neighbour algorithm can stop there (see Fig. 5). Its performance depends heavily on how good a guess the starting element START_ELEMENT is; it can be provided by bins, octrees, or alternate digital trees. On the other hand, due to its scalar nature, such an algorithm will not be able to compete with the octree search algorithm described in Section 3. Its main use is for point-to-grid or grid-to-grid transfer, where a very good guess for START_ELEMENT may be provided. This fastest grid-to-grid interpolation technique is described in the next section.

6. FASTEST GRID-TO-GRID ALGORITHM: VECTORIZED ADVANCING-FRONT VICINITY

The crucial new assumption made here, as opposed to all the other interpolation algorithms described so far, is that the points to be interpolated belong to a grid and that the grid connectivity (e.g., the points belonging to each element) is given as input. In this case, whenever the element END_ELEMENT of the known grid into which a point of the unknown grid falls is found, all the surrounding points of the unknown grid that have not yet been interpolated are given as a starting guess END_ELEMENT and stored in a list of "front" points LIST_FRONT_POINTS. The next point to be interpolated is then drawn from this list, and the procedure is repeated until all points have been interpolated. The procedure is sketched in Fig. 6, where the notion of "front" becomes apparent. The complete algorithm may be summarized as follows:

A.1. Form the list of elements adjacent to elements for the given mesh;
A.2. Form the list of points surrounding points for the unknown grid;
A.3. Mark points of the unknown grid as untouched
A.4. Initialize list of front points LIST_FRONT_POINTS for unknown grid
A.5. DO: For every non-interpolated point NON_INTERP_POINT
A.5. From LIST_FRONT_POINTS:
A.6. Obtain starting element START_ELEMENT in known grid
A.7. Attempt nearest neighbour search for NTRY attempts;
— IF unsuccessful: use brute force

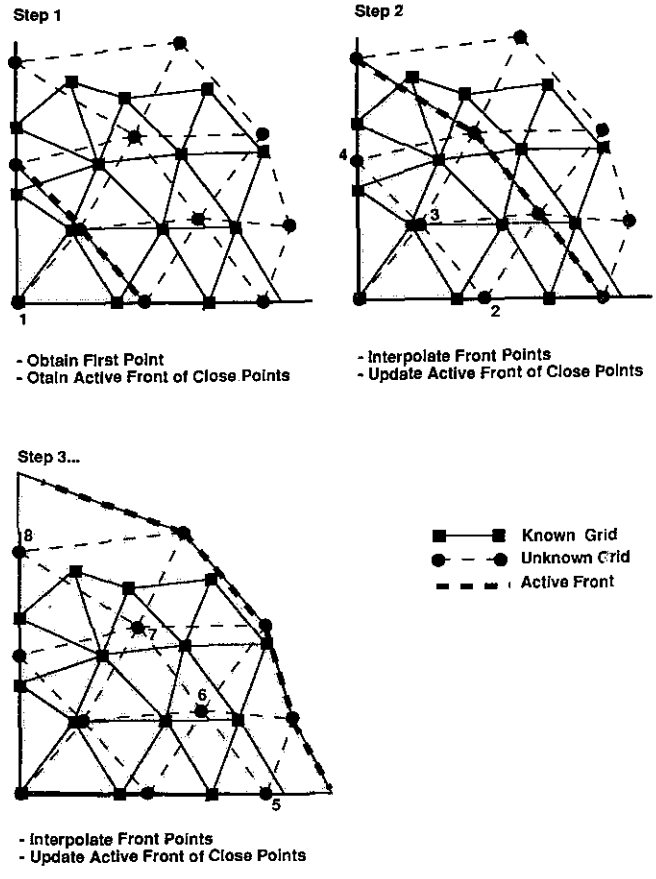

FIG. 6. Advancing front vicinity algorithm.

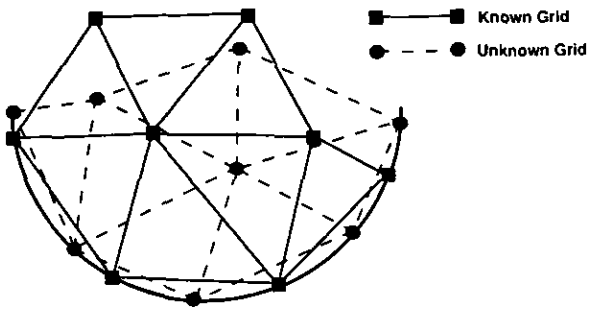

FIG. 7. Problems at concave boundaries.

— IF unsuccessful: stop or skip
⇒ END_ELEMENT

A.8. Store shape-functions and host elements

A.9. Loop over points surrounding NON_INTERP_POINT:
— IF: point has not been marked:
— Store END_ELEMENT as starting element for this point;
— Include this point in front LIST_FRONT_POINTS;
ENDIF

A.10. Mark point NON_INTERP_POINT as interpolated
ENDDO

A.11. IF: LIST_FRONT_POINTS not empty: GOTO A.5

Several possible improvements for this algorithm, layering of brute-force searches, inside-out interpolation, and vectorization, are detailed in the following.

6.1. *Layering of Brute-Force Searches*

In most instances (the exception being grids with very large disparity in element size where NTRY attempts are not sufficient), the neighbour-to-neighbour search will only fail on the boundary. Therefore, whenever a brute-force search is required, it is advisable to test first the elements connected to the boundary. This will reduce the brute-force search times considerably. Note, however, that we have to know the boundary points in this case. In the present case, the elements of the known grid are renumbered in such a way that all elements with three or more nodes on the boundary in 3D and two or more nodes on the boundary in 2D appear at the top of the list. These NR BOUNDARY ELS < NELEM elements are scanned first whenever a brute-force search is required. Moreover, after a front has been formed, only these elements close to boundaries are examined whenever a brute-force search is required.

6.2. *Inside-Out Interpolation*

This improvement is directed towards complex boundary cases. We group under this category cases where the boundary has sharp concave corners or ridges, or those cases where, due to the concavity of the surface points, the boundary may be close but outside of the known grid (see Fig. 7). In this case, it is advisable to form two front lists, one for the interior points and one for the boundary points. The interpolation of all the interior points is attempted first, and only then are the boundary points interpolated. This procedure reduces drastically the number of brute-force searches required for the complex boundary cases listed above. This may be seen from Fig. 8, where the brute-force at the corner was avoided by this procedure. As before, knowledge of the boundary points is required for this improvement.

6.3. *Vectorization*

The third possible improvement is vectorization. The idea is to search for all the points on the active front at the same time. It is not difficult to see that for large 3D grids, the vector-lengths obtained by operating in this manner are considerable, leading to very good overall performance. To obtain a vectorized algorithm we must perform steps N.3, A.7 as described above in vector mode executing the same operations on as many uninterpolated points as possible. The obstacle to this approach is that not every point will satisfy criterion (5) in the same number of attempts or passes over the points to be interpolated. The solution is to reorder the points to be interpolated after each pass such that all points that have as yet not

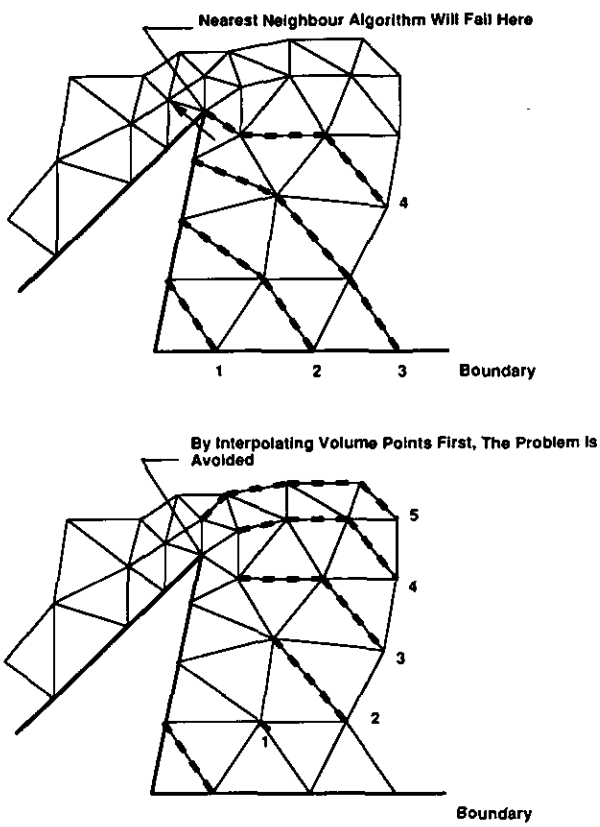

FIG. 8. Avoiding brute-force searches during interpolation.

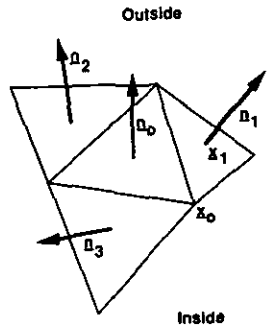

FIG. 9. Measuring surface concavity.

found their host element are at the top of the list. Such an algorithm proceeds in the following fashion:

V.0. Set the remaining number of points NR_REMAINING_POINTS=NR_FRONT_POINTS, where NR_FRONT_POINTS is the total number of points to be interpolated on the current front.

V.1. Perform steps N.3, A.7 in vector mode for all remaining points NR_REMAINING_POINTS.

V.2. Write the NR_NEXT_POINTS points that do not satisfy criterion (5) into a list LIST_OF_CURRENT_POINTS (1:NR_NEXT_POINTS). If NR_NEXT_POINTS=0: stop.

V.3. Write the NR_REMAINING_POINTS − NR_NEXT_POINTS points that do satisfy criterion (5) into LIST_OF_CURRENT_POINTS(NR_NEXT_POINTS +1 : NR_REMAINING_POINTS).

V.4. Reorder all point arrays using LIST_OF_CURRENT_POINTS. In this way, all points that have not yet found their host element are at the top of their respective lists (locations 1 : NR_NEXT_POINTS).

V.5. Set NR_REMAINING_POINTS=NR_NEXT_POINTS and go to V.1.

One can reduce the additional memory requirements associated with indirect addressing by breaking up all loops over the NR_REMAINING_POINTS remaining points into subgroups. This is accomplished automatically by using scalar temporaries on register to register machines. For memory to memory machines, a user-specified maximum group vector length must be specified.

7. CONCAVE SURFACES

For concave surfaces, criterion (12.5) will not be satisfied for a large number of surface points, prompting many brute-force searches. The algorithmic complexity of the interpolation procedure could potentially degrade to $O(N_b^2)$, where N_b is the number of boundary points. A considerable reduction of brute-force searches may be attained if the concavity of the surface can be measured. Assuming the unit face-normals $\mathbf{n}$ to be directed away from the domain, a possible measure of concavity is the visibility of neighbouring faces from any given face. With the notation of Fig. 9, the concavity of a region along the boundary may be determined by measuring the normal distance between the face and the centroids of the neighbouring faces. The allowable distance from the face for points to be interpolated is then given by some fraction α of the minimum distance measured:

$$d = \alpha|\min(0, \mathbf{n}\cdot(\mathbf{x}_0 - \mathbf{x}_i))|. \tag{6}$$

Typical values for α are $0.5 < \alpha < 1.5$. If a neighbour-to-neighbour search ends with a boundary face and all other shape-functions except the minimum satisfy Eq. (5), the distance of the point to be interpolated from the face is evaluated. If this distance is smaller than the one given by Eq. (6), the point is accepted and interpolated from the current element. Otherwise, a brute force search is conducted. The application of this procedure requires some additional arrays, such as face-arrays, a distance-array to store the concavity, and the relation between element faces and the face-array.

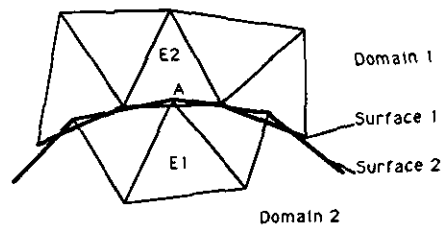

FIG. 10. Thin surface separating volumetric data.

8. VOLUMETRIC DATA SEPARATED BY THIN SURFACES

The interpolation of volumetric data for regions separated by thin surfaces is commonly encountered in computational physics. Examples for problems of this kind are flow simula-

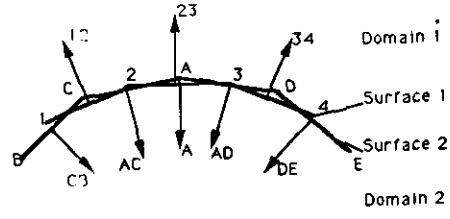

FIG. 11. Comparison of face and point normals. Note. IJ: normal of face IJ; I: normal of point I.

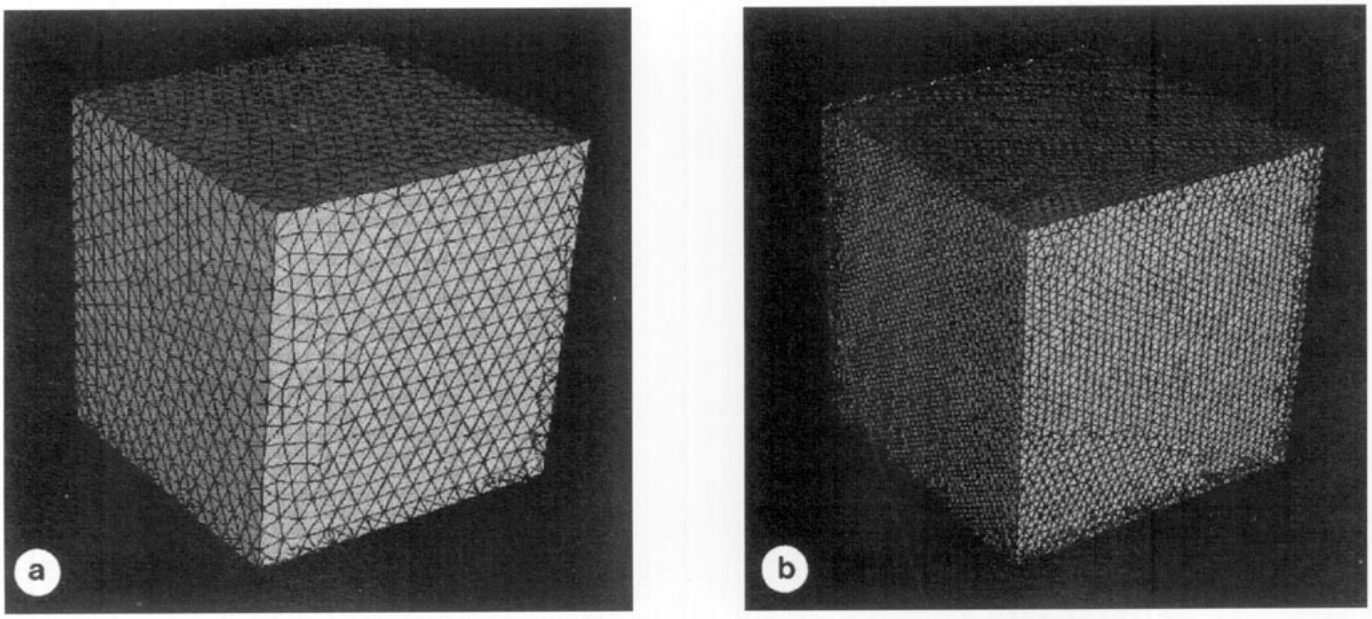

FIG. 12. Surface grids for a cube: NELEM = 34,661 (left); NELEM = 160,335 (right).

tions with thin separating sheets, such as trailing edges of wings, parasols, sails, airbags, shells, and others. In many of these cases, the surface points belonging to one of the two sides may lie inside an element that is attached to the other side. The situation is sketched in Fig. 10. Point A, although inside element E1, i.e., satisfying Criterion 5, should be interpolated from element E2. In order to avoid such an erroneous interpolation, the surface normals of the faces of the known grid are compared with the point normals of the points to be interpolated (see Fig. 11). If the scalar product of these normals falls below a preset tolerance (e.g., -0.5), the host element is rejected, and a brute search is performed. The surface point normals are obtained by averaging the normals of the faces surrounding them. While averaging, a comparison of the normals for all the surrounding faces is conducted. If these normals differ substantially, an edge or corner is detected, and the points are marked accordingly. For these points, the surface normal is considered as undefined, and no comparison of surface normals is conducted. The alignment test for surface normals just described can be

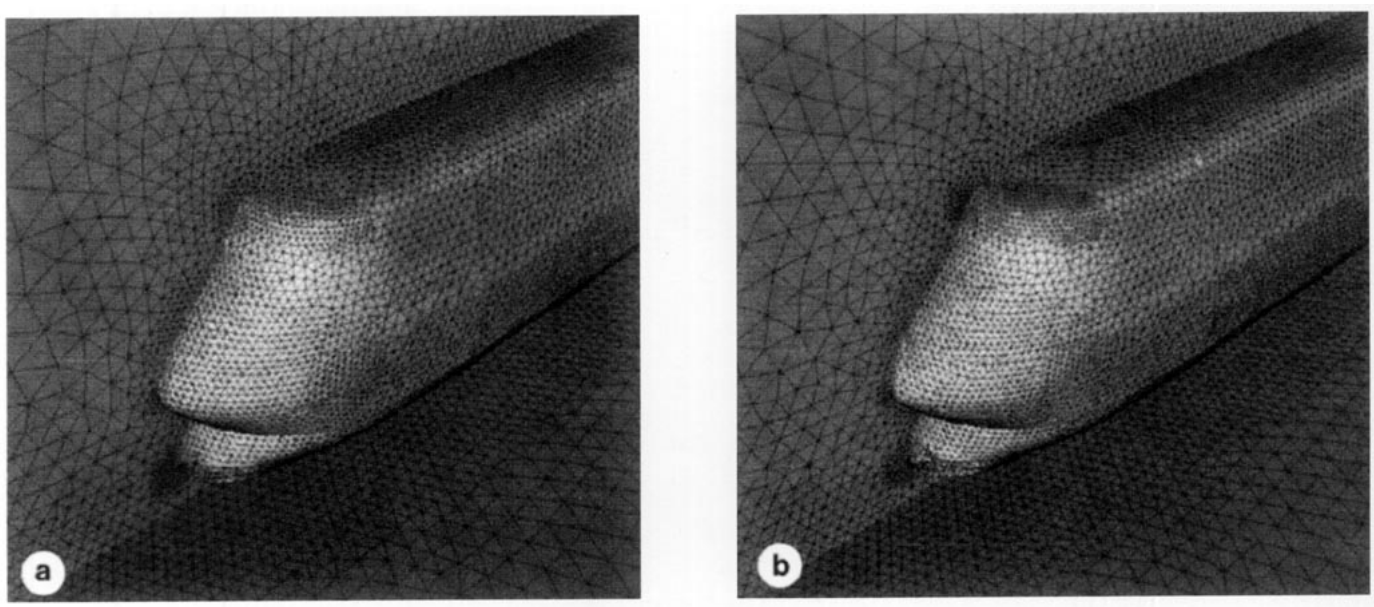

FIG. 13. Surface grids for a train: NELEM = 180,670 (left); NELEM = 243,068 (right).

TABLE I

Interpolation Timings

Case	$NELEM_1$	$NELEM_2$	# BFS	CPU-scalar	CPU-vector	Speedup
$Cube_1$	34,661	30,801	0	0.1399	0.0283	4.94
$Cube_2$	34,661	160,355	0	0.5360	0.1104	4.86
$Train_1$	180,670	243,068	31	1.1290	0.3405	3.32
$Train_2$	180,670	243,068	0	0.9905	0.2020	4.90

incorporated into the vectorized advancing front search procedure without complications.

9. EXAMPLES

The interpolation techniques described were tested on several 3D grids that discretized the volume inside a cube or around a train configuration with varying degrees of mesh density. Figures 12 and 13 show the surface grids of some of these grids. Table I summarizes the performance recorded on the CRAY-C90 for the advancing-front vicinity algorithm. Both the scalar and vector version of the algorithm were carefully optimized for speed. We therefore regard this as a fair comparison.

One can observe speedups between 1 : 3.3 and 1 : 5.0 for the vectorized version. The interpolation speed per point per full interpolation varied between 3.7×10^{-6} s/pt and 7.4×10^{-6} s/pt. This number, as well as the speedup obtained, depends on the number of brute-force interpolations required, as well as the average number of tries required in order to find the host element for each point to be interpolated. The more tries required, the higher the speedup achieved by the vectorized version, as the transcription and rearrangement costs are amortized over more vectorized CPU-intensive operations. For the third and fourth cases, the only difference is the number of brute-force interpolations required. The train configuration, which is typical of CFD runs with moving bodies that require many re-interpolations of the solution during a run [5], had a few concave surfaces. Some of the points of the second grid were outside the first mesh, prompting a search over all elements with three or more nodes on the boundary. As one can see, this exhaustive search, which runs at 325 Mflops on the CRAY-C90, does not affect the performance of the scalar version significantly. The timings for the vectorized version, however, are deteriorated significantly for this case. We note that the number of elements with three or more nodes on the boundary only constitutes about 9% of the total number of elements. Doing an exhaustive search over the complete mesh would therefore have led to a dramatic increase in interpolation times.

10. CONCLUSIONS

Several search algorithms for the interpolation of unstructured grids were reviewed and compared. Particular emphasis was placed on the pitfalls these algorithms may experience for grids commonly encountered in practice and ways to improve their performance. It was shown how the most CPU-intensive portions of the search process may be vectorized. Timings for several problems were given, indicating that speedups of 1 : 5 can be obtained if the algorithm is properly vectorized.

Common areas of computational mechanics that will benefit from the higher speeds achieved for the vectorized interpolation process are: field simulations with moving bodies, interdisciplinary problems approached via loose coupling, initialization and boundary condition update for field solvers attached to a discrete data base, and visualization.

ACKNOWLEDGMENTS

This work was supported by DNA, as well as AFOSR, under Contract F49620-92-J-0058. Doctors Michael Giltrud and Leonidas Sakell were the technical monitors. The author thanks IBM for providing support in the form of a RISC-6000/550 workstation. Most of the debugging of the described algorithms was performed on this machine.

REFERENCES

1. R. Löhner, *Comput. Systems Engrg.* **1**(2–4), 257 (1990).
2. J. Peraire, K. Morgan, and J. Peiro, *J. Comput. Phys.* **103,** 269 (1992).
3. N. Weatherill, O. Hassan, M. Marchant, and D. Marcum, AIAA-93-3390-CP, 1993 (unpublished).
4. J. D. Baum and R. Löhner, AIAA-93-0783, 1993 (unpublished).
5. E. Mestreau, R. Löhner, and S. Aita, AIAA-93-0890, 1993 (unpublished).
6. T. Westermann, *J. Comput. Phys.* **101,** 307 (1992).
7. R. Löhner and K. Morgan, *Int. J. Numer. Methods Eng.* **24,** 101 (1987).
8. R. L. Meakin AIAA-93-3350-CP, 1993 (unpublished).
9. D. N. Knuth, *The Art of Computer Programming*, Vols. 1–3, Addison–Wesley, Reading, MA, 1973.
10. H. Samet, *Comput. Surveys* **16**(2), 187 (1984).
11. J. Bonet and J. Peraire, *Int. J. Numer. Methods Eng.* **31,** 1 (1991).
12. R. Löhner and J. Ambrosiano, *J. Comput. Phys.* **91**(1), 22 (1990).