JOURNAL OF COMPUTATIONAL PHYSICS **143,** 519–543 (1998)
ARTICLE NO. CP985963

Fully Threaded Tree Algorithms for Adaptive Refinement Fluid Dynamics Simulations

A. M. Khokhlov

Laboratory for Computational Physics and Fluid Dynamics, Code 6404, Naval Research Laboratory, Washington, DC 20375
E-mail: ajk@lcp.nrl.navy.mil

Received January 27, 1997; revised February 26, 1998

A fully threaded tree (FTT) for adaptive mesh refinement (AMR) of regular meshes is described. By using a tree threaded at all levels, tree traversals for finding nearest neighbors are avoided. All operations on a tree including tree modifications are $\mathcal{O}(N)$, where N is a number of cells and can be performed in parallel. An implementation of the tree requires $2N$ words of memory. In this paper, FTT is applied to the integration of the Euler equations of fluid dynamics. The integration on a tree can utilize flux evaluation algorithms used for grids, but requires a different time-stepping strategy to be computationally efficient. An adaptive-mesh time-stepping algorithm is described in which different time steps are used at different levels of the tree. Time stepping and mesh refinement are interleaved to avoid extensive buffer layers of fine mesh which were otherwise required ahead of moving shocks. A filtering algorithm for removing high-frequency noise during mesh refinement is described. Test examples are presented, and the FTT performance is evaluated. © 1998 Academic Press

1. INTRODUCTION

Adaptive mesh refinement (AMR) is used to increase spatial and temporal resolution of numerical simulations beyond the limits imposed by the available hardware. In fluid dynamics, AMR is used in simulations of both steady and unsteady flows on structured and unstructured meshes. In this paper, a mesh is called *structured* if it is composed of cells of rectangular shape; otherwise it is unstructured. Integration of the equations of fluid dynamics is easier and the overhead in computer time and memory per computational cell are minimal for structured meshes. The corresponding overhead is usually high for unstructured meshes. However, an unstructured mesh has the advantage that it can conform well to a complex boundary. Whether to use a structured or unstructured approach depends on the specific problem, code availability, and general preferences. This paper deals with structured meshes and, specifically, with a structured-mesh AMR.

There are two different approaches to structured-mesh AMR. In one standard approach, cells are organized as two-dimensional or three-dimensional arrays (grids). A coarse base grid represents the entire computational domain. Finer, nested grids are layed over coarser grids where more resolution is required. A grid hierarchy may be organized as a tree [1–12]. One advantage of this approach is that any single-grid fluid flow solver can be used without modifications for AMR. However, grids themselves are inflexible structures. It is difficult to cover complex features of a flow with a few grids. Many grids are typically required, and a substantial number of computational cells can be wasted on a smooth flow. Several grids of the same level of refinment may overlap which leads to duplication of cells. Periodic rebuilding of the entire grid hierarchy is required when the flow evolves with time.

In an alternative approach, individual computational cells are organized directly in a tree [13]. Each cell can be refined or unrefined separately from the others, as needed. At every level of the tree, the mesh may have an arbitrary shape, as opposed to grids. Duplication of cells is avoided. To date, this approach has been used mainly in steady-state fluid flow simulations [13–20]. It has been applied to unsteady flows in two dimensions [21–23]. The main advantages of the tree approach are the flexibility in refinment and unrefinement and the efficiency in using cells. However, standard grid-based solvers cannot be used on a tree. The overall time stepping strategy on a tree is different from that on a grid. In addition, an access to neighboring cells is more difficult in a tree than in an array. Tree traversals to access nearest neighbors are difficult to vectorize and parallelize. The memory overhead to maintain a tree, although less than for unstructured meshes, is substantial [21–23].

This paper attempts to address the problems with the tree approach mentioned above. It describes a new structure, a *fully threaded tree* (FTT), which provides an efficient parallel access to information on a tree and reduces the memory overhead for maintaining the tree. In an ordinary tree, each cell has pointers to its children, if any. The memory for keeping these pointers is not used in leaves (cells which do not have children). In a threaded tree, this free memory is used to organize “threads” along which the tree can be quickly traversed during various search operations. A discussion of a binary threaded tree in which threads are used for left-to-right and right-to-left search can be found in [24]. A binary-, quad-, or an oct-tree is required to organize a one-, two-, or three-dimensional regular mesh, respectively. An implementation of an oct-tree would require at least eight words of memory per cell. Some of this memory can then be used to make threads through leaves in order to facilitate finding the nearest neighbors. However, this scheme cannot find neighbors of split cells and cannot be parallelized (Section 4). In the FTT described in this paper, every cell, whether a leaf or not, has an easy access to its children, neighbors, and parents. One may say that an FTT is a tree threaded in all possible directions. At the same time, the memory overhead for supporting an FTT is significantly reduced. The maintenance of an octal FTT requires two words of memory per computational cell only. An important property of an FTT is that all operations on it, including modifications of the tree structure itself, can be performed in parallel. This then allows vectorization and parallelization of tree-based AMR algorithms.

The paper further describes a time stepping algorithm for advancing the solution of the Euler equations in time with different time steps at different levels of the tree. In unsteady flow simulations with many levels of refinement, buffer layers of finely refined cells must be created in advance and ahead of shocks to prevent them from running out of the fine mesh. Let the maximum and minimum levels of mesh refinement be l_{max} and l_{min}, respectively, and let the cell refinement ratio be a factor of 2. If refinement is done between coarse time steps [21–23], the thickness of the buffer layers should be at least $2^{l_{\mathrm{max}}-l_{\mathrm{min}}}$ finest cells.

The number of required cells then increases exponentially when $l_{\max}$–$l_{\min}$ increases, and it becomes prohibitively large, especially in three dimensions, if $l_{\max}$–$l_{\min}$ is large. In the time-stepping algorithm presented in this paper, the integration in time and tree refinement have been coupled together, and time stepping at different levels of the tree and tree refinements of these levels have been interleaved. As a result, excessive buffer layers of refinement ahead of moving shocks are no longer needed. The time stepping described can utilize standard flux evaluation algorithms used in grid-based solvers.

The following sections describe the equations solved (Section 2), tree structure and discretization of the equations on the tree (Section 3), tree implementation (Section 4), integration of the Euler equations (Section 5), and refinement and unrefinement (Section 6). One-, two-, and three-dimensional tests simulations are presented (Section 7) and used to evaluate adaptive mesh refinement aspects of the FTT-based algorithms (Section 8).

2. EQUATIONS

In this paper, the FTT is used to solve the Euler equations for an inviscid flow,

$$\begin{aligned} \frac{\partial\rho}{\partial t} + \nabla\cdot(\rho\mathbf{U}) &= 0, \\ \frac{\partial\rho\mathbf{U}}{\partial t} + \nabla\cdot(\rho\mathbf{U}\mathbf{U}) + \nabla P &= \rho\mathbf{g}, \\ \frac{\partial E}{\partial t} + \nabla\cdot((E+P)\mathbf{U}) &= \rho\mathbf{U}\cdot\mathbf{g}, \end{aligned} \tag{1}$$

where ρ is the mass density, $\mathbf{U}$ is the fluid velocity, E is the total energy density, $P = P(E_i, \rho)$ is the pressure, $E_i = E - \rho U^2/2$ is the internal energy density, and $\mathbf{g}$ is an external acceleration.

Numerical integration of (1) requires evaluating the numerical fluxes of mass, momenta, and energy at cell interfaces. We use a second-order accurate Godunov-type method based on the solution of a Riemann problem to evaluate the fluxes [26–28]. This is one of the monotone methods known to provide good results for both flows with strong shocks and for moderately subsonic and turbulent flows [32–34]. However, the algorithms described in this paper are general and can be used in conjunction with other high-order monotone methods such as the flux-corrected-transport [35].

3. FTT STRUCTURE AND DISCRETIZATION

A computational domain is a cube of size L. It is subdivided to a number of cubic cells of various sizes $1/2, 1/4, 1/8, \ldots$ of L. Cells are logically organized in an oct-tree with the entire computational domain being the root. For the integration and refinement algorithms to be effective, the following information must be easily accessible for every cell i:

$$\begin{aligned} iLv(i) &\text{ — level of the cell in the tree} \\ iKy(i) &\text{ — TRUE/FALSE if cell is split/unsplit} \\ iPr(i) &\text{ — pointer to a parent cell} \\ iCh(i,j) &\text{ — pointers to children, } \ j = 1,\ldots,8 \\ iNb(i,j) &\text{ — pointers to neighbors, } \ j = 1,\ldots,6. \end{aligned}$$

The cell size is related to $iLv(i)$ by $\Delta_i = 2^{-iLv(i)}L$.

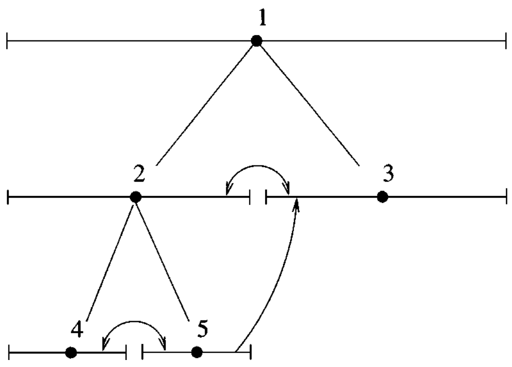


FIG. 1. Logical relationship between cells for a one-dimensional, binary, fully threaded tree. Each cell is represented by a horizontal barred line. The length of a line corresponds to the geometrical size of a cell. Cell 1 is the root of the tree, representing the entire computational domain. Cells 1 and 2 are split cells. Cells 3, 4, and 5 are leaves. Pointers to children and parents are indicated by straight lines without arrows. Pointers to neighbors are indicated by arrows.

Cells in the tree are either split (have children) or leaves (do not have children). Logical relations between cells in the tree, and the directions of various pointers are illustrated in Fig. 1 for a one-dimensional binary tree. Relations for a quad- or an oct-tree are similar. In Fig. 1, the root (cell 1) represents the entire computational domain. It has two children (cells 2 and 3), each representing half of the domain. There will be four or eight children in 2D or 3D, respectively. Cell 2 is further subdivided on two cells (cells 4 and 5). Neighboring leaves are not allowed to differ in size by more than a factor of 2. The neighbor–neighbor relation is not reciprocal for leaves of different sizes that face each other. In Fig. 1, cell 5 has cell 3 as its neighbor, but cell 3 has cell 2 as its neighbor, and not cell 5. A jth neighbor of a cell i either has the same size as the cell itself, $\Delta_{iNb(i,j)} = \Delta_i$, or it is two times larger, $\Delta_{iNb(i,j)} = 2\Delta_i$. In the former case, the neighbor may be a leaf or a split cell. In the latter case, it can only be a leaf.

Physical state information $\mathcal{U}_i = \{\rho_i, E_i, (\rho\mathbf{U})_i\}$ is kept in both split and unsplit cells. Information kept in split cells is needed to evaluate mass, momenta, and energy fluxes across interfaces between leaves of different sizes (Section 5) and to make decisions about refinement and unrefinement (Section 6). The physical state vectors for split cells are updated by averaging over the children, when required. The memory overhead for keeping state vectors of split cells is

$$\text{Overhead} = \frac{\text{Number of splits}}{\text{Number of leaves}} = \frac{1}{2^{\text{dim}}} + \frac{1}{2^{2\text{dim}}} + \frac{1}{2^{3\text{dim}}} + \cdots \simeq \frac{1}{2^{\text{dim}} - 1},$$

where dim $= 1, 2, 3$ is a number of spatial dimensions. The overhead is $\simeq$100% in 1D, $\simeq$33% in 2D, and $\simeq$14% in 3D.

Coordinates $\mathbf{r}_i = \{x_i, y_i, z_i\}$ of cell centers are associated with every cell. It is assumed that neighbors 1 to 6 of a cell are left X, right X, left Y, right Y, left Z, and right Z neighbors, respectively, and that coordinates increase from left to right: $x_{iNb(i,1)} < x_i < x_{iNb(i,2)}$, $y_{iNb(i,3)} < y_i < y_{iNb(i,4)}$, $z_{iNb(i,5)} < z_i < z_{iNb(i,6)}$. Keeping coordinates is not necessary for finite-difference operations. They are useful, however, for evaluating long-range forces (e.g., gravity) by direct summation or multipole expansion algorithms.

4. IMPLEMENTATION OF FTT

The FTT, as described above, can be implemented by storing all pointers, coordinates, and flags in computer memory and then modifying them when the tree is refined or unrefined. However, such a direct implementation requires $17N$ words of memory, where N is the number of cells in the tree. This memory overhead is too large. In addition, such a tree cannot be modified in parallel. Because neighbors of a cell keep pointers to that cell, cell removal requires retargeting these pointers. Removing one cell thus affects others. A conflict appears if a cell and its neighbors are to be removed simultaneously. Similar problems arise if neighboring cells are created simultaneously.

An improved parallel version of the tree is based on the following three observations: (i) When a cell is split, its eight children are created simultaneously. Thus, they may be stored in memory contiguously [13], so that only one pointer is needed to find all eight children. (ii) Since all eight children (siblings) are kept in memory together, neighbor–neighbor relations between them are known automatically. There is no need for pointers between the siblings. (iii) Neighbors of any cell are either its siblings or they are children of a neighboring parent. There is a small fixed number of neighbor–neighbor relations between siblings of neighboring parents. Thus, a child's neighbor that belongs to a different parent can be determined without search by accessing its parent.

The FTT structure implemented is illustrated in Fig. 2. All cells are organized in groups called octs. Each oct contains eight cells. Each cell has a physical state vector $\mathcal{U}$ associated with it, and a pointer to an oct which contains its children, if any, or a *nil* pointer. Each oct knows its level, *OctLv*, which is equal to the level of the oct's cells. Each oct has a pointer *OctPr* to a parent cell. Each oct has six pointers *OctNb*(6) to *parent cells of neighboring octs*. The memory requirement is 16 words per oct or 2 words per cell. The memory overhead is thus significantly reduced. The price for this is the computer-time overhead associated

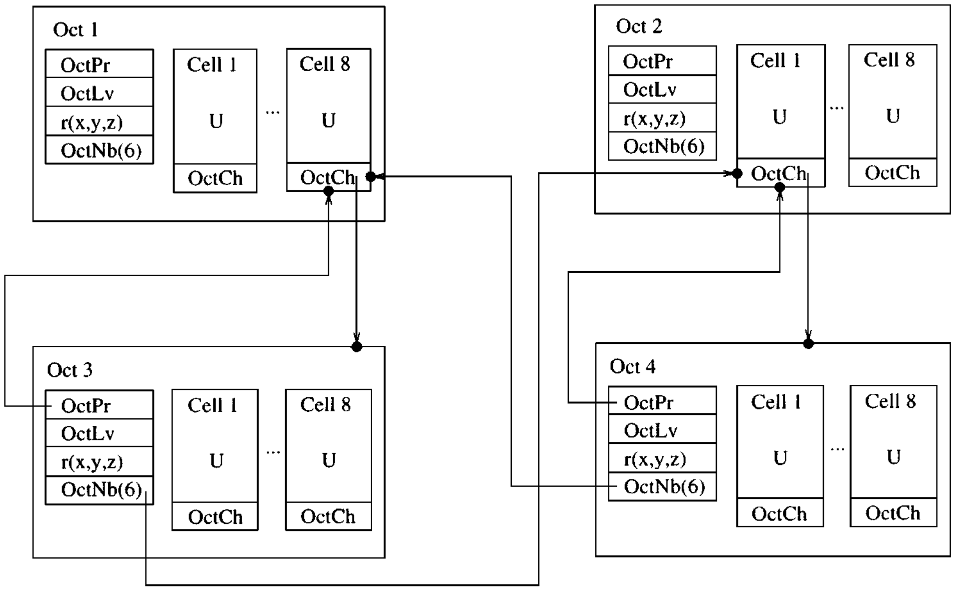


FIG. 2. Relationship between the cells and octs in a fully threaded tree. Pointers from octs to cells and from cells to octs are indicated by arrows.

with computing cell pointers from oct pointers and oct pointers from cell pointers, but this overhead is relatively small (Section 8).

The FTT time and memory overheads may be compared with the corresponding overheads for ordinary quad- and oct-trees that do not store any connectivity information. The memory overhead for a quad- and an oct-tree is four and eight words per cell, respectively. This is two to four times more than that required for FTT. The average number of levels of a quad- or an oct-tree that must be traversed to determine a neighbor of a cell is $\simeq 2$ [14]. In FTT, half of the neighbors belong to the same oct. These neighbors can be found without traversals using an add/subtract operation. Finding others requires accessing a neighboring parent cell to determine a neighboring oct. Finding a neighbor then again requires an add/subtract operation. Thus, the average number of levels that must be traversed in order to find a neighbor in FTT is $\simeq 1/2$, or four times less than for a quad- or an oct-tree. For a quad- or an oct-tree, the actual number differs from cell to cell, and, in the worst case, the tree must be traversed up to its root. On the contrary, the actual number of traversed levels in FTT never exceeds one. As a result, there is a small fixed number of paths of accessing neighbors of a cell, depending on its position inside the oct and on the level of a neighbor. These paths can be stored in a look-up table and selected quickly during the run time.

If needed, octs may also contain coordinates *OctPos* of their centers. These coordinates are equal to coordinates of the corresponding parent cells. Coordinates of cells that belong to an oct can be found by adding or subtracting $\Delta_i/2$ from the corresponding oct's coordinates. The memory requirement for coordinates is thus three words per oct or $\frac{3}{8}$ words per cell, which increases the total memory requirement to $2\frac{3}{8}$ words per cell.

Figure 2 shows that FTT allows parallel modifications at any given tree level. To unrefine a cell, the associated oct pointed to by *OctCh* must be destroyed, and *OctCh* must be set to *nil*. No other changes are required in neighboring octs or cells located at the same tree level. This is because neighboring octs are accessed through parent cells located at a different level of the tree. To refine a cell, a new oct must be created with all necessary pointers, and the corresponding *OctCh* pointer of the refined cell must be set to a new oct. Again, no other changes in any other octs or cells located at the same tree level are required.

All operations on the tree can thus be expressed as a sequence of parallel operations on large groups of cells that belong to the same tree level. Therefore, it is possible to introduce a general iterator for performing operations on these cells and to hide the details of parallelization inside it. This greatly reduces the amount of code and simplifies coding. To perform a parallel operation, each processor first gathers all required information in work arrays using indirect addressing. Then, the CPU-intensive work is done by looping over contiguous blocks of memory. After that, the result is scattered back. In this way, the code performance is virtually unaffected by the distribution of cells in a physical space.

5. INTEGRATION PROCEDURE

As mentioned in the Introduction, integration in time on the tree requires a time-stepping strategy which is different from that typically used on regular grids. In the integration procedure described below, computations are organized not on cell-by-cell, but on interface-by-interface basis. At every interface, fluxes are evaluated, changes to cell values on the left and on the right of the interface are applied, and then fluxes are discarded. Left and right fluxes for any cell can be evaluated in an arbitrary order and even on different processors. Cell-by-cell computations would have doubled the work spent on flux evaluations.

An additional advantage of the interface-by-interface approach is that interfaces between cells of different sizes can be treated in the same way as interfaces between cells of the same sizes.

Equations (1) are integrated in time with different time steps $\Delta t(l)$ at different levels of the tree l; $l_{\min} \leq l \leq l_{\max}$, where $l_{\min}$ and $l_{\max}$ is the minimum and maximum levels of leaves. A global time step $\Delta t \equiv \Delta t(l_{\min})$ is determined from the CFL condition

$$\Delta t = cfl \frac{2^{-l_{\min}} L}{\max_i(\max_j(a_i + |U_{i,j}|))}, \tag{2}$$

where a is the sound speed, $cfl < 1$ is a constant, and the maximum in (2) is taken over all three directions $j = x, y, z$ and over all leaves i. Time steps at various levels are

$$\Delta t(l) = 2^{l_{\min}-l} \Delta t. \tag{3}$$

Integration at different levels of the tree is interleaved with tree refinement. Let us designate the procedure of advancing level l one step $\Delta t(l)$ in time as $\mathcal{A}(l)$, and the procedure of tree refinement at level l as $\mathcal{R}(l)$. The $\mathcal{R}$ procedure consists of refining leaves of level l and unrefining split cells of level l according to certain refinement criteria. This procedure is described in Section 6. The advancement procedure $\mathcal{A}(l)$ is described below.

The solution at every level l is advanced in time using direction splitting. Equations for the X-sweep can be written as

$$\frac{\partial \mathcal{U}}{\partial t} = \frac{\partial \mathcal{F}(\mathcal{U})}{\partial x} + \mathcal{S}, \tag{4}$$

where the flux vector is $\mathcal{F} = (\rho U_x, (E + P)U_x, \rho U_x^2, \rho U_x U_y, \rho U_x U_z)$ and the source term is $\mathcal{S} = (0, \rho U_x g_x, \rho g_x, 0, 0)$. The equations for the Y- and Z-sweeps are similar.

Let us designate right and left neighbors of cell i in the direction of a sweep as $i+$ and $i-$, respectively. Let $\mathcal{U}^o$ and $\mathcal{U}^n$ be a state vector at the beginning and at the end of a global time step, respectively. The $\mathcal{A}(l)$ procedure is described in a form of the following pseudocode:

```
for ( leaves i of level $l-1$ ) { if ( i has split neighbors ) $\mathcal{U}_i^n := \mathcal{U}_i^o$; }
for ( X, Y, Z directions ) {
    for ( leaves i of level l ) {
        if ( $i+$ is a leaf or boundary ) {
            Compute fluxes $\mathcal{F}_{i,i+}$ at the $(i, i+)$ interface;
            $\mathcal{U}_i^n := \mathcal{U}_i^n - \alpha\mathcal{F}_{i,i+} + \Delta t(l)\mathcal{S}_i$;
            $\mathcal{U}_{i+}^n := \mathcal{U}_{i+}^n + \gamma\mathcal{F}_{i,i+}$;
        }
        if ( $i-$ is a leaf of level $l-1$ or boundary ) {
            Compute fluxes $\mathcal{F}_{i-,i}$ at the $(i-, i)$ interface;          (5)
            $\mathcal{U}_i^n := \mathcal{U}_i^n + \alpha\mathcal{F}_{i-,i}$;
            $\mathcal{U}_{i-}^n := \mathcal{U}_{i-}^n - \beta\mathcal{F}_{i-,i}$;
        }
    }
    for ( leaves i of level l ) { if ( i has no neighbors at level $l-1$ ) $\mathcal{U}_i^o := \mathcal{U}_i^n$; }
}
for ( split cells i of level l ) { $\mathcal{U}_i^o := 2^{-\dim} \sum_{j=1}^{2^{\dim}} \mathcal{U}_{iCh(i,j)}^o$; }
for ( leaves i of level l ) { if ( i has split neighbors ) $\mathcal{U}_i^o := \mathcal{U}_i^n$; }.
```

Factors

$$\alpha = 2^{l_{\min}} \frac{\Delta t}{L}, \quad \beta = 2^{-\dim}\alpha, \quad \gamma(i,l) = \begin{cases} \alpha, & \text{if } iLv(i+) = l; \\ \beta, & \text{if } iLv(i+) = l-1, \end{cases}$$

in (5) take into account the difference in volumes of neighboring cells of different sizes around interfaces; $\dim = 1, 2, 3$ is the number of spatial dimensions.

The global time step of integration consists of going through all levels of the tree, starting with l_{min}, and performing a sequence of refinements and advances at every tree level. This can be expressed as a recursion,

$$\begin{gathered} \textit{for (all cells i) } \{ \mathcal{U}_i^n := \mathcal{U}_i^o; \}, \\ \textit{Global Time Step } = \mathcal{S}(l_{\min}), \end{gathered} \tag{6}$$

where the procedure $\mathcal{S}(l)$ is a combination of advancing and refinement procedures,

$$\mathcal{S}(l) = \mathcal{A}^\dagger(l)\mathcal{S}(l+1)\mathcal{A}\,\mathcal{S}(l+1)\mathcal{R}(l), \tag{7}$$

and $\mathcal{S}(l)$ does nothing if $l > l_{\max}$. The procedures $\mathcal{S}(l+1)$ and $\mathcal{A}(l)$ are repeated in (7) twice when directional time-step splitting is used in $\mathcal{A}$. After every $\mathcal{A}$ procedure, the sequence of XYZ one-dimensional sweeps is recursively reversed. This is indicated by the $\dagger$ superscript. All procedures in (7) are performed from right to left, i.e., $\mathcal{R}(l)$ first, and $\mathcal{A}^\dagger(l)$ last. For example, for $l_{\min} = 6$ and $l_{\max} = 8$, the sequence generated by (6) and (7) would be $[\mathcal{R}(6)\ [\mathcal{R}(7)\ [\mathcal{R}(8)\ \mathcal{A}(8)\ \mathcal{A}^\dagger(8)]\ \mathcal{A}(7)\ [\mathcal{R}(8)\ \mathcal{A}(8)\ \mathcal{A}^\dagger(8)]\ \mathcal{A}^\dagger(7)]\ \mathcal{A}(6)\ [\mathcal{R}(7)\ [\mathcal{R}(8)\ \mathcal{A}(8)\ \mathcal{A}^\dagger(8)]\ \mathcal{A}(7)\ [\mathcal{R}(8)\ \mathcal{A}(8)\ \mathcal{A}^\dagger(8)]\ \mathcal{A}^\dagger(7)]\ \mathcal{A}^\dagger(6)]$. Square brackets separate different levels of recursion.

The algorithm works in the following way: In the beginning of every global time step, a new state vector $\mathcal{U}^n$ is set equal to an old state vector $\mathcal{U}^o$ for all leaves and split cells in the tree. During a global time step, every invocation of $A(l)$ changes $\mathcal{U}^n$ at levels l and $l-1$, and $\mathcal{U}^o$ at level l. Each $\mathcal{A}(l)$ visits all leaves i at level l, evaluates numerical fluxes at right interfaces of i (interfaces between cells i and $i+$), and changes $\mathcal{U}_i^n$ and $\mathcal{U}_{i+}^n$, accordingly. This is done only if level $i+$ is less than or equal to level i. Left fluxes are evaluated, and $\mathcal{U}_i^n$ and $\mathcal{U}_{i-}^n$ are changed only if level $i-$ is less than level i. Fluxes are not evaluated if level i is less than the level $i+$, or if it is less than or equal to level $i-$. These fluxes will be accounted for during the $\mathcal{A}(l+1)$ advancement procedure. A total change of $\mathcal{U}^n$ comes from both right and left fluxes. Each flux is evaluated, applied, and then discarded immediately. Fluxes associated with one cell may be evaluated in any order and even on different processors (if more than one are available).

Euler fluxes in (5) are evaluated by solving a Riemann problem at every interface according to [26]. Left and right states for the Riemann problems are obtained by a piecewise linear reconstruction [27] from $\mathcal{U}^o$ state vectors. A small amount of dissipation is introduced into the code as an artificial diffusion added to the numerical fluxes [28]. For cells that have neighbors at the same tree level, the entire integration procedure is that of a usual conservative finite differencing

$$\mathcal{U}_i^n = \mathcal{U}_i^o + \frac{\Delta t(l)}{\Delta x_i}(\mathcal{F}_{i,i-} - \mathcal{F}_{i,i+}) \tag{8}$$

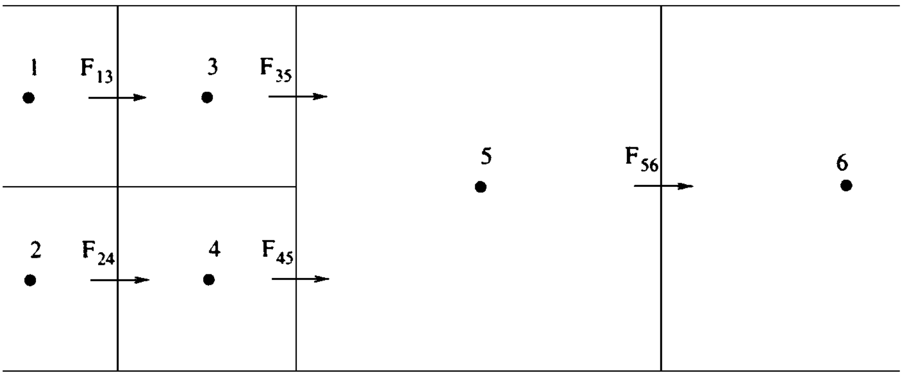


FIG. 3. An illustration of flux evaluation at two different levels of the tree. Fluxes at interfaces between fine cells (1–3 and 2–4), and fluxes between fine and coarse cells (3–5 and 4–5) are evaluated twice as often as fluxes between coarse cells (5–6). Four fluxes from the fine side (3–5 and 4–5 computed at two different times), and one flux (5–6) from the coarse side contribute to changes of the state vector of cell 5 during a coarse time step.

with left and right fluxes centered at the same time $t = t^o + \Delta t(l)/2$. It is formally second-order accurate both in space and in time. For cells that have neighboring leaves at different levels, l and $l-1$, the flux contributions from more than two neighbors and at two different moments of time, $t = t^o + \Delta t(l)/2$ and $t = t^o + 3\Delta t(l)/2$, are taken into account, again in a conservative fashion. For these cells, the accuracy reduces to first order. This is because physical states $\mathcal{U}^o$ of leaves at level $l-1$ are used twice at $t = t^o$ and at $t = t^o + \Delta t(l)$ for the left/right state reconstruction at level l. However, $\mathcal{U}^o$ at level $l-1$ are updated only once at time $t = t^o + \Delta t(l-1) = t^o + 2\Delta t(l)$.

The algorithm is schematically illustrated in Fig. 3 for the two-dimensional case. In this particular example, the tree has only two levels occupied by leaves, $l_{\max} = l_{\min} + 1$. Cells 1, 2, 3, and 4 belong to level $l_{\max}$, and cells 5 and 6 belong to level $l_{\min}$. During a time step at level $l = l_{\max}$, fluxes accross interfaces (1, 3), (2, 4) (3, 5), and (4, 5) are evaluated, new state vectors $\mathcal{U}^n$ for cells 1, 2, 3, and 4 are computed, and old state vectors $\mathcal{U}^o$ in these cells are updated. For cell 5, $\mathcal{U}_5^n$ is changed due to fluxes $\mathcal{F}_{3,5}$ and $\mathcal{F}_{4,5}$. However, state vector $\mathcal{U}_5^o$ is not updated yet. During the second time step $\Delta t(l)$, this procedure is repeated: States $\mathcal{U}_1^0, \mathcal{U}_2^0, \mathcal{U}_3^0, \mathcal{U}_4^0$ are updated, $\mathcal{U}_5^n$ is changed again due to fluxes $\mathcal{F}_{3,5}$ and $\mathcal{F}_{4,5}$, but state vector $\mathcal{U}_5^o$ is still not updated. After two time steps at level l, one time step at level $l-1$ is performed. Fluxes at interface (5, 6) are computed, $\mathcal{U}_5^n$ is changed due to these fluxes, and the state vector $\mathcal{U}_5^o$ is finally updated.

If necessary, accelerations $\mathbf{g}$ are computed in the beginning of every advancement procedure. To maintain the second-order accuracy in time for the source term $\mathcal{S}$, the state $\mathcal{U}^o$ is corrected after every advancement procedure as

$$\tilde{\mathbf{U}} := \mathbf{U} + \frac{\mathbf{g}^{t+\Delta t} - \mathbf{g}^t}{2}\Delta t, \quad E := E + \rho\frac{\tilde{U}^2 - U^2}{4}, \quad \mathbf{U} := \tilde{\mathbf{U}}. \tag{9}$$

6. MESH REFINEMENT

The most difficult part of an AMR simulation is to decide where and when to refine or unrefine a mesh. It is also the most problem-dependent part. Different problems may require different criteria for refinement and unrefinement. The refinement procedure currently implemented consists of four steps:

1. For every cell, a refinement indicator, $0 \le \xi \le 1$, is computed. Large $\xi > \xi_{\text{split}}$ indicates that a leaf must be refined, and small $\xi < \xi_{\text{join}}$ indicates that a split cell can be unrefined; ξ_{split} and ξ_{join} are some predefined constant values.
2. ξ is smoothed in order to prevent cells from being falsely refined (mesh trashing) in places where ξ fluctuates around critical values ξ_{split} and ξ_{join}.
3. Leaves are refined if $\xi > \xi_{\text{split}}$.
4. Split cells are unrefined if $\xi < \xi_{\text{join}}$ and if they are not just split.

There are two approaches to computing ξ. First, it may be computed to measure the convergence of a solution [2]. This allows us to control the accuracy of the solution "on flight," but requires that the solutions on the fine and coarse meshes be generated simultaneously. Another approach is to use an indicator proportional to gradients in the solution. Such an indicator shows where to expect a large error in the solution [16, 19, 20]. The convergence of the solution must be checked afterwards with a different resolution. We adopt the second approach, but note that the tree structure allows both approaches to be implemented.

The indicator ξ is constructed as a maximum of several indicators,

$$\xi = \max(\xi^1, \xi^2, \ldots), \tag{10}$$

each of which is either a shock indicator, a contact discontinuity indicator, or a gradient indicator, all normalized to unity, $0 \le \xi^k \le 1$.

As a shock indicator, the quantity used is [28]

$$\begin{aligned} \xi_i^s &= \max_{j=1,\ldots,6} \xi_{i,j}^s, \\ \xi_{i,j}^s &= \begin{cases} 1, & \text{if } \dfrac{\left|P_{iNb(i,j)} - P_i\right|}{\min\left(P_{iNb(i,j)}, P_i\right)} > \epsilon_s;\ \delta \cdot \left(U_{iNb(i,j),k} - U_{i,k}\right) > 0, \\ 0, & \text{otherwise,} \end{cases} \end{aligned} \tag{11}$$

where $\delta = 1 - 2\cdot\text{mod}(j, 2)$, $k = \text{int}((j+1)/2)$, and $\epsilon_s = 0.2$ determines the minimum shock strength to be detected.

A discontinuity indicator is defined as

$$\begin{aligned} \xi_i^c &= \max_{j=1,\ldots,6} \xi_{i,j}^c, \\ \xi_{i,j}^c &= \begin{cases} 1, & \text{if } \dfrac{\left|P_{iNb(i,j)} - P_i\right|}{\min\left(P_{iNb(i,j)}, P_i\right)} < \epsilon_s;\ \dfrac{\left|\rho_{iNb(i,j)} - \rho_i\right|}{\min\left(\rho_{iNb(i,j)}, \rho_i\right)} > \epsilon_c, \\ 0, & \text{otherwise,} \end{cases} \end{aligned} \tag{12}$$

where $\epsilon_c = 0.2$.

A gradient indicator for a variable a is constructed as

$$\xi_i^a = \max_{j=1,\ldots,6} \left(\frac{\left\|a_{iNb(i,j)}\right| - \left|a_i\right\|}{\max\left(\left|a_{iNb(i,j)}\right|, \left|a_i\right|\right)} \right), \tag{13}$$

where a may be a mass density, energy density, pressure, velocity, vorticity, etc. According to (10)–(12), shocks and discontinuities are marked with $\xi = 1$ and refined to the maximum refinement level allowed. Thus, the gradient indicators (13) are applied only to a smooth flow.

As explained in the beginning of this section, the indicator ξ computed according to (10)–(13) must be smoothed before it is used for refinement. Smoothing is based on the analogy with the propagation of a reaction-diffusion front. Let us consider ξ as a concentration of a reactant obeying a reaction-diffusion equation,

$$\frac{\partial \xi}{\partial \tilde{t}} = K \nabla^2 \xi + Q, \tag{14}$$

where $\tilde{t}$ is a fiducial time, $K = 2^{-2l} L^2$ is a constant diffusion coefficient, and

$$Q = \begin{cases} 1, & \text{if } 1 > \xi > \xi_{\text{split}}, \\ 0, & \text{otherwise}, \end{cases} \tag{15}$$

is a reaction rate. A steady-state form of (14),

$$-S_\xi \frac{\partial \xi}{\partial x} = \frac{\partial^2 \xi}{\partial x^2} + Q, \tag{16}$$

describes a reaction front which moves with a constant speed,

$$S_\xi = 2^{-l} L \sqrt{\xi_{\text{split}}}, \tag{17}$$

and has a thickness

$$\delta_\xi \simeq 2^{-l} L / \sqrt{\xi_{\text{split}}}\,. \tag{18}$$

As the front moves, it leaves behind cells marked with $\xi = 1$.

The values of ξ from (10) are used as the initial condition for (14). Equation (14) then describes a reaction front which starts at places where $\xi > \xi_{\text{split}}$ and propagates outwards, normal to itself with the speed S_ξ. By integrating (14), this front is advanced approximately 2–3 computational cells. According to the Huygens principle, the front curvature tends to decrease with time, so that boundaries of regions marked for refinement become smoother. Diffusion prevails over reaction in isolated areas with $\xi > \xi_{\text{split}}$ if these areas have a size less than $\simeq \delta_\xi$. These small regions do not trigger a reaction front and disappear under the refinement threshold ("extinguish"). This reduces the numerical noise in selecting cells for refinement and unrefinement. To further reduce mesh trashing, split cells are not joined if this produces an isolated leaf.

7. NUMERICAL EXAMPLES

Many numerical tests including advection, shock propagation, and interaction in one, two, and three dimensions were used to validate the algorithms. Below several test examples are presented which illustrate the performance of the FTT-based adaptive mesh refinment algorithm described in this paper. These examples are then used in Section 8 to evaluate the FTT efficiency.

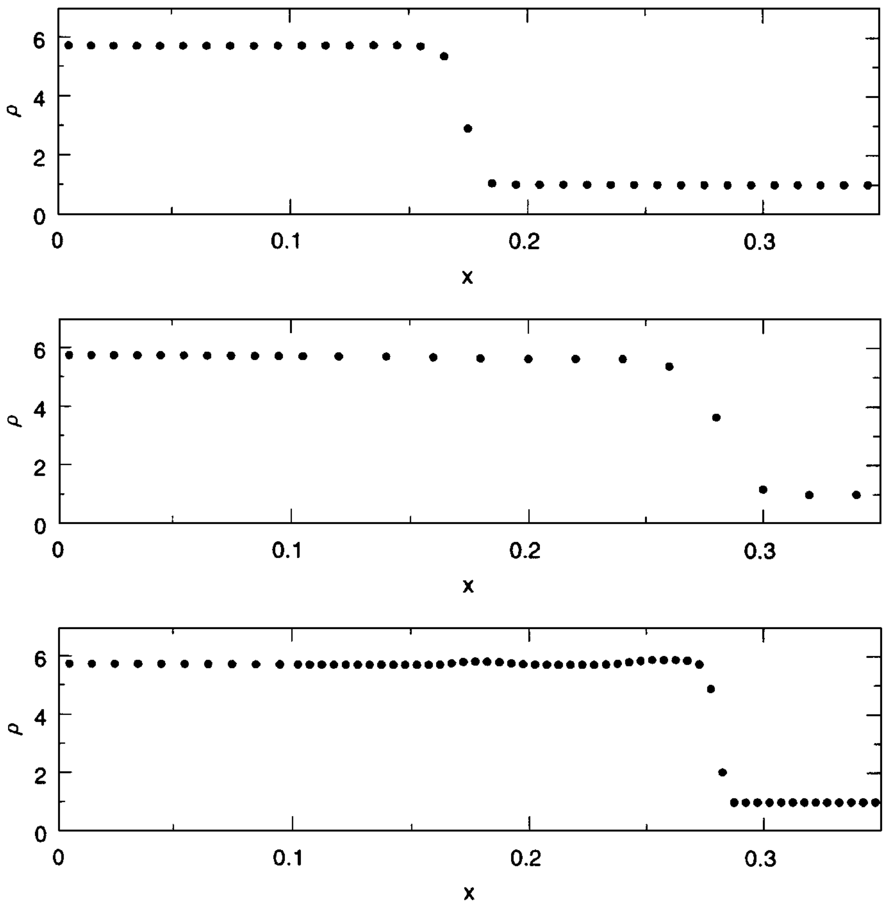


FIG. 4. An isolated shock wave with a Mach number $M = 10$ in an ideal gas with $\gamma = 1.4$; computed with $cfl = 0.7$. Density is shown by dots whose position corresponds to centers of computation cells. In all panels, the shock is moving from left to right. Top panel—shock on a uniform mesh. Middle panel—shock after passing from fine to coarse mesh. Bottom panel—shock after passing from coarse to fine mesh. Coarse-to-fine interface is located near $x \simeq 0.11$. Two disturbances generated during the passage of the coarse-to-fine interface are seen on the bottom panel.

7.1. *Isolated Shock Wave Passing through a Coarse-to-Fine Interface*

Figure 4 shows a computation of an isolated $M = 10$ strong shock in an ideal gas with $\gamma = 1.4$. In all three panels, the shock is moving from left to right. The upper panel shows the shock profile at one time during the shock propagation through a uniform mesh. The shock profile is two cells wide, and oscillation-free. When the shock passes from a fine to a coarse mesh (middle panel), the amplitude of disturbances generated is less than 1%. When passing from a coarse to a fine mesh (low panel), two types of disturbances, acoustic and entropy waves, are generated at the $\simeq$5% amplitude.

A generation of spurious disturbances by a shock passing from a coarse to a fine mesh is a known problem [2, 4, 16]. An illustrative example for an $M = 10$ shock in an ideal gas with $\gamma = 1.4$ is presented by Berger and Colella [2]. Berger and Colella argue that the disturbances are caused by the $O(1)$ errors always present in the numerical fluxes in the vicinity of a strong shock. The errors do not cancel each other if mesh spacing is changing. Formally,

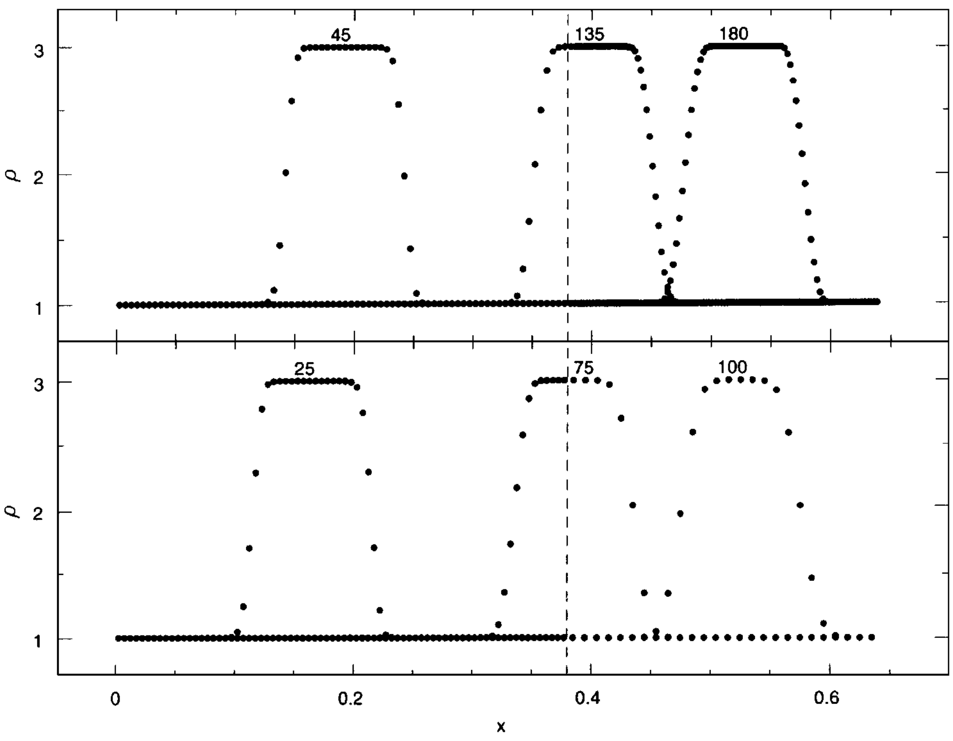


FIG. 5. Advection of a high density, cold slab of gas with $\rho_{\text{high}} = 3$ surrounded by a low density hot gas with $\rho_{\text{low}} = 1$. Pressure is $P = 0.01$, and velocity is $U = 2$ everywhere. An ideal gas EOS with $\gamma = 1.4$; computed with $cfl = 0.7$. Advection is taking place from left to right. A coarse-to-fine interface is located at $x \simeq 0.38$ (dashed line). Density is shown by dots whose position corresponds to centers of computation cells. Time-step number is indicated at the top of each density profile.

Berger and Colella's integration was second-order accurate at interfaces. In our case, the formal order of integration at interfaces is first order. However, our example shows smaller disturbances, which could be attributed to the smaller refinement ratio (2:1 instead of 4:1).

7.2. *Density Discontinuity Passing through a Coarse-to-Fine Interface*

Figure 5 shows the advection of a slab of high density, cold gas surrounded by a hot gas of lower density. Initial conditions for this problem are $P = 0.01$ and $U = 2$ everywhere, $\rho_{\text{cold}} = 3$, $\rho_{\text{hot}} = 1$. Initially, the slab is 20 cells wide. The equation of state is that of an ideal gas with $\gamma = 1.4$ constant. Integration is done with $cfl = 0.7$. The only result of passing a slab through coarse-to-fine interfaces is changes in a number of cells representing contact discontinuities. The pressure and velocity field (not shown) both remain constant to machine accuracy. The test shows that passing a discontinuity through coarse-to-fine interfaces does not generate any disturbances. However, if a discontinuity exits a refined area, the resolution is lost. If it enters a refined area, its width does not decrease, and no gain in resolution is obtained.

The main conclusions from the tests above is that both shocks and contact discontinuities should not be allowed to enter a more refined mesh. They must be refined all the time with the maximum resolution allowed. This substantiates our approach to the evaluation of a mesh refinement indicator ξ, where shocks and discontinuities are always marked with the maximum $\xi = 1$ (Section 6).

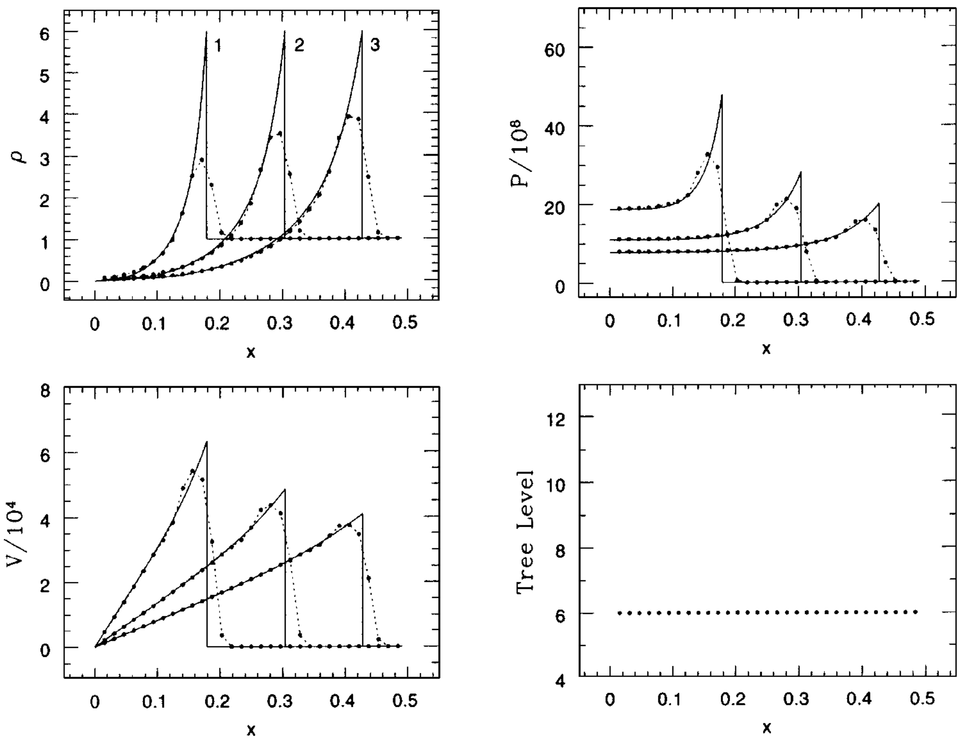


FIG. 6. Strong point explosion in an ideal gas $\gamma = 1.4$ in planar geometry. The initial pressure is $P_0 = 10^{-3}$, initial density is $\rho_0 = 1$, and the explosion energy is $E = 2.5 \times 10^9$. Computed on a uniform mesh using $cfl = 0.7$. Plots of density, pressure, velocity, and cell level in the tree are shown for three different moments of time (1) 1.57×10^{-6}, (2) 3.49×10^{-6}, and (3) 5.82×10^{-6}. The variables are shown by dashed lines and by dots whose position corresponds to centers of computation cells. Solid lines—the exact solution.

7.3. *Planar Strong Point Explosion*

Figures 6 and 7 shows a one-dimensional (planar) strong point explosion in an ideal gas with $\gamma = 1.4$. Initial conditions are $\rho_0 = 1$, $P_0 = 10^{-3}$ everywhere. The explosion energy $E = 2.5 \times 10^9$ is put in a single cell by increasing the pressure in that cell to $P = P_0 + (\gamma - 1)E/\Delta$, where Δ is the cell size. Figure 6 shows the explosion computed on a uniform mesh with $\Delta = \frac{1}{64}$. The exact solution to the problem is shown for comparison [29]. The solutions agree near the center. The difficulty is to reproduce the solution near the shock.

Figure 7 shows the same explosion computed with eight levels of refinement, $l = 5, \ldots, 12$. Two refinement indicators were used, one for shocks (11) and another one for the pressure gradient (13). The refinement criteria were $\xi_{\text{split}} = 0.5$ and $\xi_{\text{join}} = 0.01$. In this case, the flow behind the shock has been resolved. The number of cells used in the simulation was $\simeq$250, and only a few ($<$20) located at the highest level $l = 12$. A uniform 4096 mesh would be required to get the same resolution. Small disturbances at fine–coarse interfaces are seen at the pressure and velocity plots. Since the gradients in the solution are large everywhere, these disturbances may be caused by the same inexact cancelling of numerical errors in fluxes at the fine–coarse interfaces discussed in [2] and in Section 7.1 above, or they may be the result of decreasing the order of the accuracy of integration at interfaces, or both. Which effect is important and whether it is worthwhile to maintain formal second-order accuracy at interfaces requires further study.

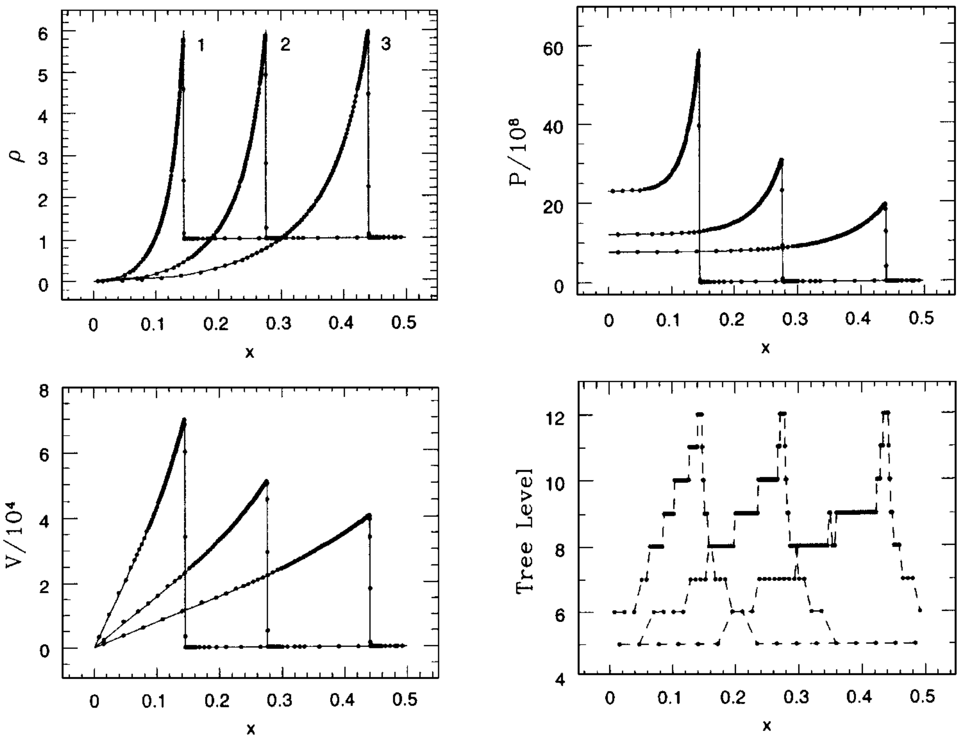
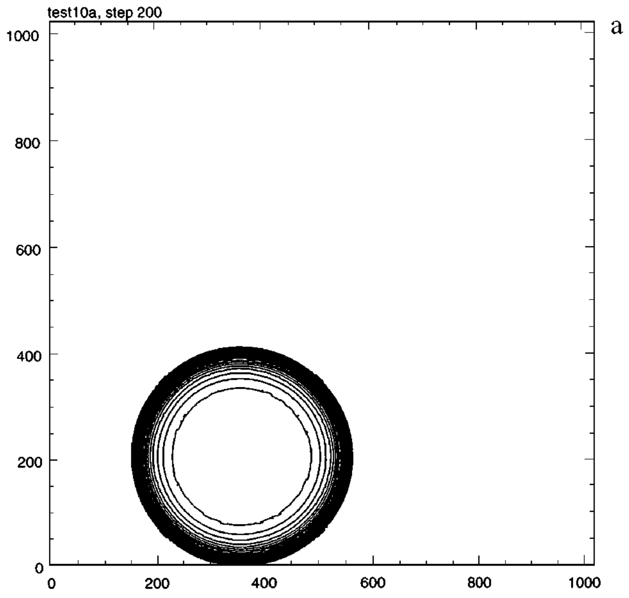


FIG. 7. Strong point explosion in an ideal gas $\gamma = 1.4$ in planar geometry. The initial pressure is $P_0 = 10^{-3}$, initial density is $\rho_0 = 1$, and the explosion energy is $E = 2.5 \times 10^9$. Computed using *cfl* $= 0.7$ and eight levels of mesh refinement. Plots of density, pressure, velocity, and cell level in the tree are shown for three different moments of time (1) 1.15×10^{-6}, (2) 3.02×10^{-6}, and (3) 6.07×10^{-6}. The variables are shown by dots whose position corresponds to centers of computation cells. Solid lines—the exact solution.

7.4. *Cylindrical Strong Point Explosion in a Box*

Figures 8 and 9 show a cylindrical strong point explosion in a square box and illustrate how the adaptive mesh algorithm treats a very complicated, rapidly evolving flow with multiple shocks, contact discontinuities, and vortices. Initial conditions in the box are $\rho_0 = P_0 = 1$. The equation of state is that of an ideal gas with $\gamma = 1.4$. The explosion energy $E = 10^5$ is put in a single cell of size $\frac{1}{1024}$ located at $x = 0.35$, $y = 0.2$. The computation was done with six levels of refinement, $l = 5, \dots, 10$, using shock and pressure gradient refinement criteria with $\xi_{\mathrm{split}} = 0.5$ and $\xi_{\mathrm{join}} = 0.05$. The Courant number, *cfl*, is 0.7. The equivalent uniform resolution would require a 1024×1024 grid or $2^{20} = 1{,}048{,}576$ cells.

A cylindrical blast wave propagates from the explosion point, first hits the lower, and then the left, right, and upper walls of the box. Reflections of the primary shock begin as regular reflections, but later transform into Mach reflections. The sound speed is very high in the material behind the primary shock. As a result, reflected shocks quickly engulf the inside of the hot bubble created by the initial blast, catch up with the primary shock, and interact with walls and with each other. Figures 8a (step 200) and 8b (step 300) show the interaction of the primary shock with the lower wall. In Fig. 8c (step 400) this interaction continues, but the shock is also reflecting off the left wall. Reflections of the primary shock off the



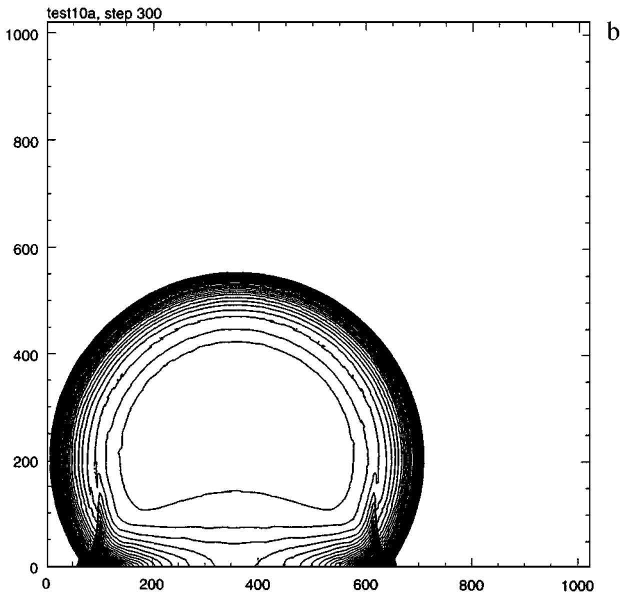
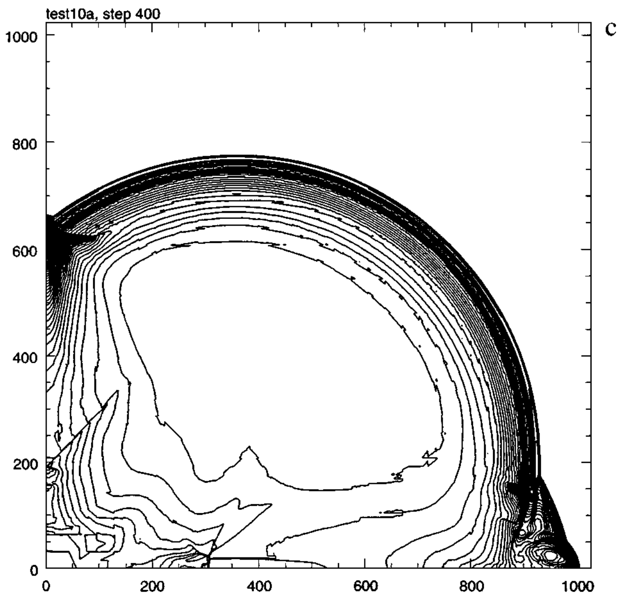


FIG. 8. Cylindrical strong point explosion in a square box with solid walls. Computed with levels of refinement $l = 5, \ldots, 10$. Density is shown for time steps: (a) 200, (b) 300, (c) 400, (d) 500, and (e) 600; (f) pressure for time step 600. Density contours are $\Delta\rho = 0.1$, beginning with $\rho_{\min} = 0.05$. Pressure contours are $\Delta P = 10^3$, beginning with $P_0 = 0$. Coordinates are in units of the finest cell size $\Delta(10) = \frac{1}{1024}$.

walls are irregular. Next, Fig. 8d (step 500), a part of the primary shock is still interacting with the upper and right walls. In Fig. 8e (step 600), the primary shock disappeared, and the collision of the two reflected shocks just occurred near the upper right corner of the box. Figure 8f shows the pressure contours for step 600. The mesh for the two time steps, 400 and 600, is shown in Fig. 9. Figures 8 and 9 show a complicated pattern of secondary shocks resolved by the mesh refinement algorithm.



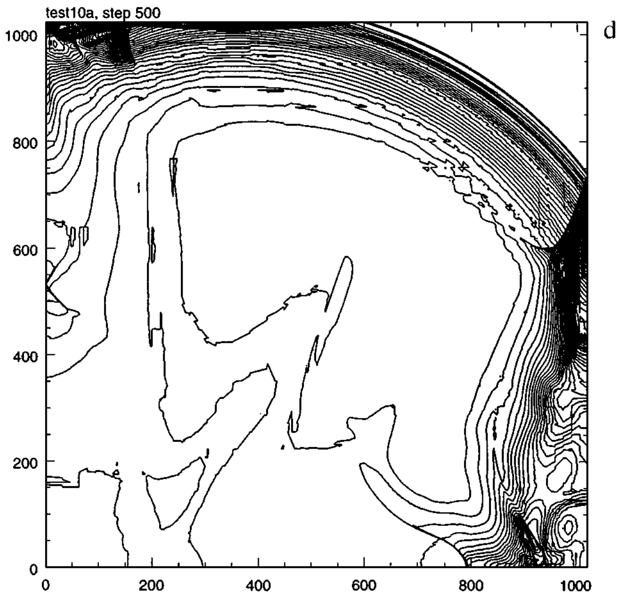
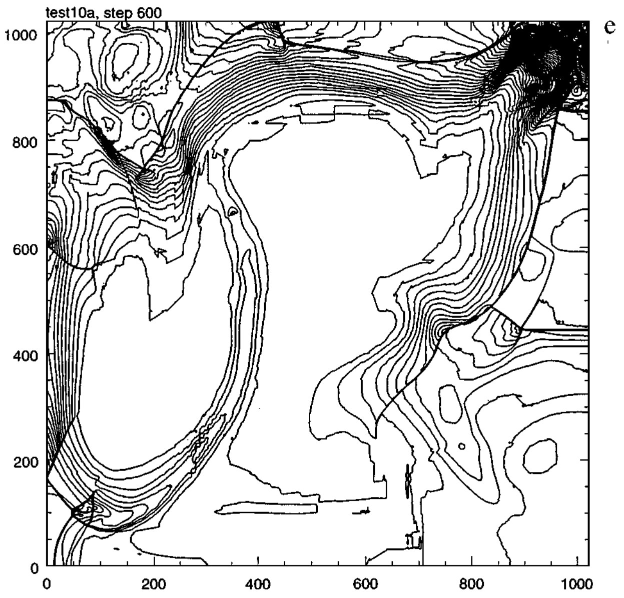


FIG. 8—*Continued*

7.5. *Shock–Bubble Interaction in Three Dimensions*

The interaction of shock waves with fluid inhomogeneities is an important mechanism of vorticity generation [30, 31]. The interaction of a planar shock with a low-density spherical bubble of gas is one of the four representative cases studied theoretically in [31]. It is assumed in [31] that the gas in the bubble is in pressure equilibrium with the ambient gas, but it has a higher temperature. This is different from the conditions of the experiments [30], where the density contrast was created by using gases with different molecular weight. Formulation of the problem in [31] is highly relevant to the shock-generated vorticity in flames.



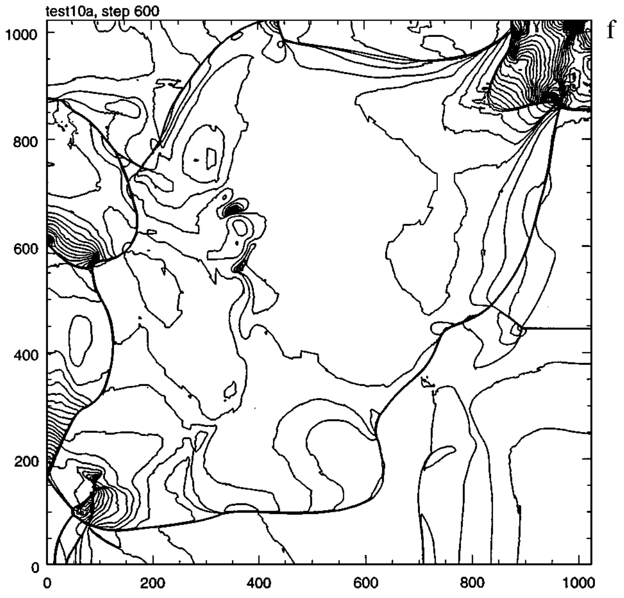
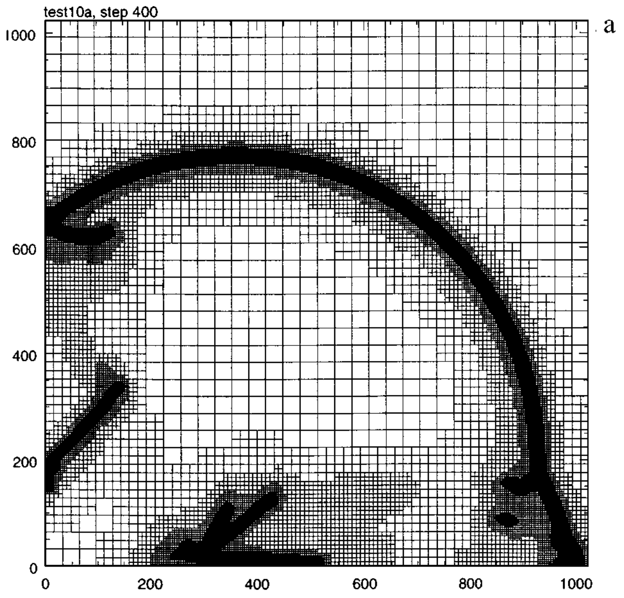


FIG. 8—*Continued*

The computational setting for the shock–bubble problem is shown in Fig. 10. A spherical bubble of hot, low density gas is located inside a rectangular, solid wall tube of length 1 and cross section $\frac{1}{2} \times \frac{1}{2}$. The computational domain of size $L_x \times L_y \times L_z = 1 \times \frac{1}{4} \times \frac{1}{4}$ represents a quarter of the tube cross section, assuming symmetry with respect to the middle planes $y = \frac{1}{4}$ and $z = \frac{1}{4}$. The radius of the bubble is $R_b = \frac{1}{8}$. The bubble is centered at a distance $x_b = \frac{1}{4}$ from the tube entrance. The bubble and the background gas are in pressure equilibrium at a common pressure $P_0 = 1$. The background density is $\rho_0 = 1$ and the bubble density is $\rho_b = 0.166$. The equation of state is that of an ideal gas with constant $\gamma = 1.4$. A shock wave with the Mach number $M = 1.25$ hits the bubble from the left. Boundary conditions are inflow on the tube entrance, and outflow on the tube exit. Simulations are performed



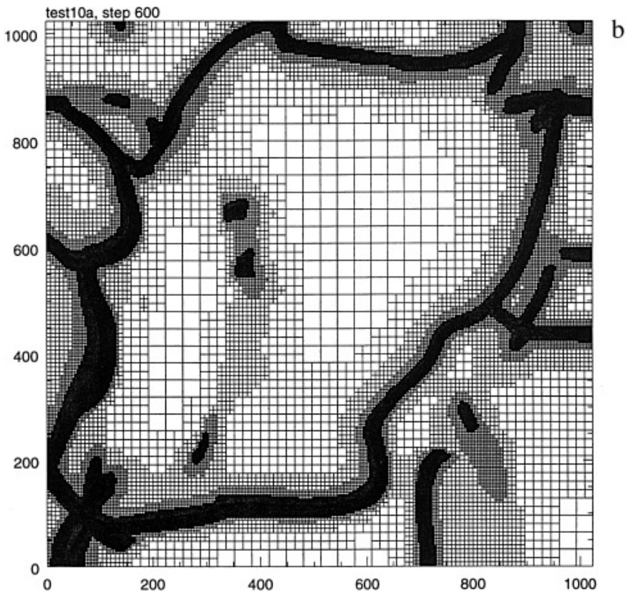


FIG. 9. Same as Fig. 8 but shows the computational mesh: (a) step 400, and (b) step 600.

using five levels of refinement $l = 5, \ldots, 9$. The maximum equivalent uniform resolution would correspond to a $512 \times 128 \times 128$ grid, and would require $2^{23} = 8{,}388{,}608$ cells of size $\frac{1}{512}$. The three-dimensional resolution in this test case is 20% higher than that used in two-dimensional simulations [31]. Four refinement indicators were used: for shocks (11), for contact discontinuities (12), and for density and pressure gradients (17). The refinement criteria are $\xi_{\text{split}} = 0.5$ and $\xi_{\text{join}} = 0.05$. The Courant number $cfl = 0.7$ was used.

Figure 11 shows the density contours and the mesh in the XY plane $z = \frac{1}{4}$ after 20 and 70 global time steps of integration. Figure 12 shows the density contour and the mesh for the same time steps in the perpendicular YZ-planes $x = 0.35$ and $x = 0.6$, respectively. The shock compresses the bubble, and vorticity is generated due to misalignment of the

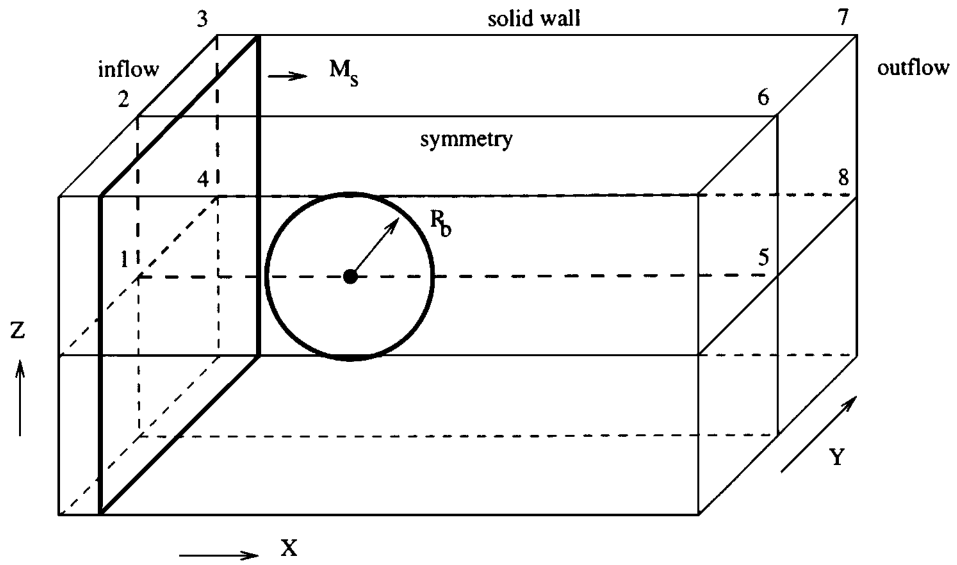


FIG. 10. Computational setting of the shock wave–spherical bubble interaction problem. Depicts a square tube of size $1 \times \frac{1}{2} \times \frac{1}{2}$. The computational domain is a quarter of the tube cross section shown by thick lines and marked by the numbers 1 to 8, assuming symmetry with respect to the XZ plane $y = \frac{1}{4}$ and XY plane $z = \frac{1}{4}$.

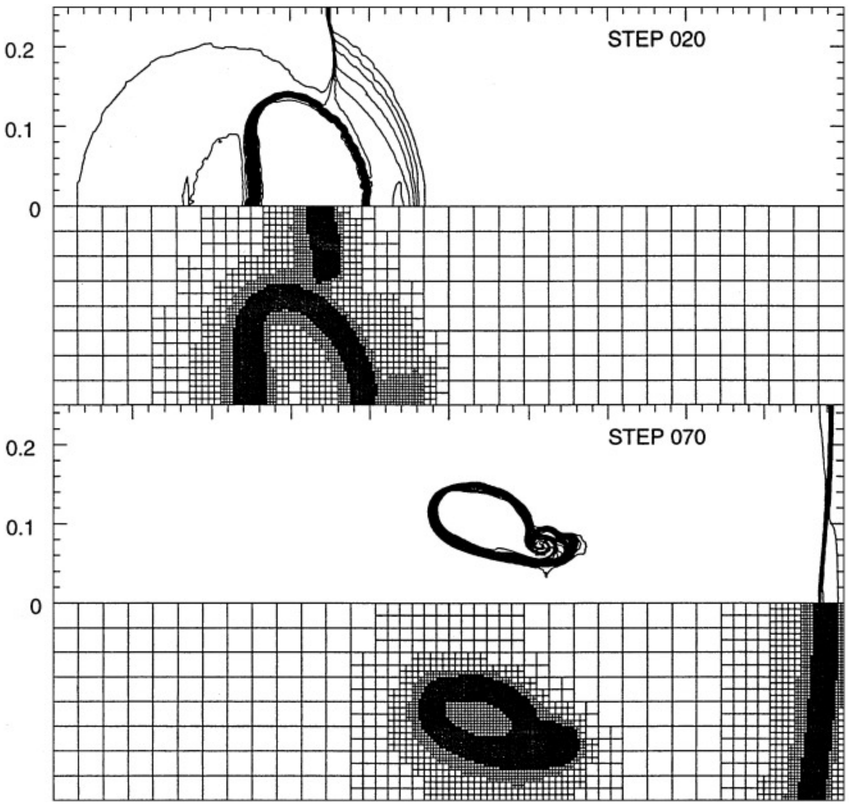


FIG. 11. Shock wave–spherical bubble interaction. Density contours $\Delta\rho = 0.05$ and the mesh in the XY plane 1-2-5-6 (see Fig. 10) for time steps 20 and 70.

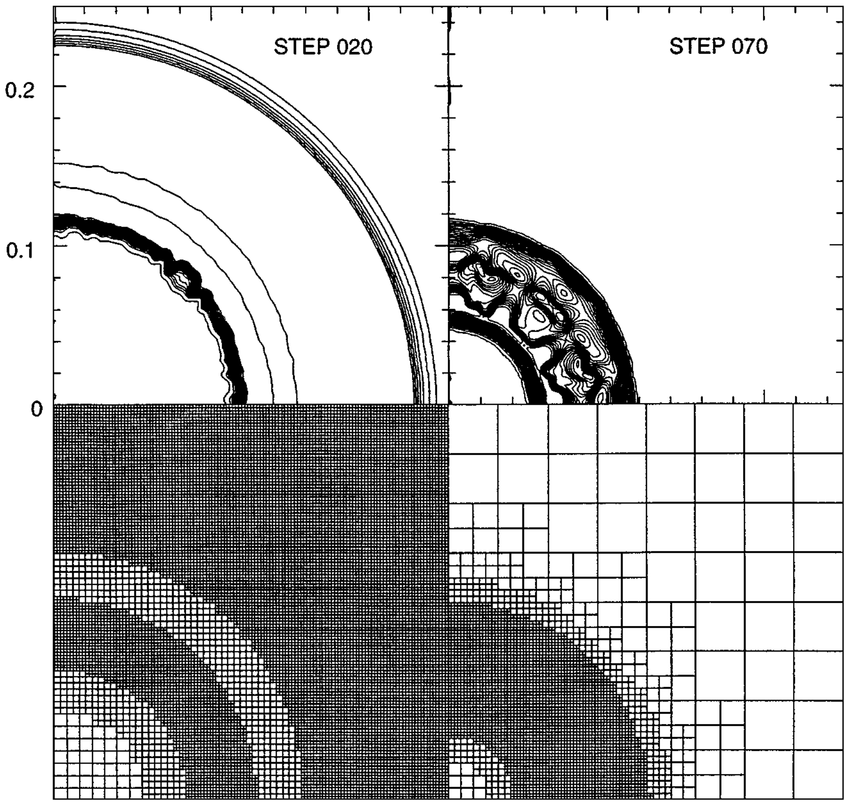


FIG. 12. Shock wave–spherical bubble interaction. Density contours $\Delta\rho = 0.05$ and the mesh in the YZ plane for time steps 20 and 70. The plane coordinates are $x = 0.35$ (step 20) and $x = 0.6$ (step 70).

pressure and density gradients during the interaction. Vorticity is generated mainly at the bubble surface where transmitted shocks are refracted. Rotational motions generated as a result of shock refraction further deform the bubble, and eventually transform it into the rotating ring. The results of the simulation presented here are in a good general agreement with the results of two-dimensional axially symmetric simulations [31].

8. ALGORITHM PERFORMANCE

The algorithm consists of four parts: (1) FTT subroutines for tree modifications, dumping, performing global parallel operations, finding neighbors, parents, and children, etc.; (2) fluid dynamics subroutines; (3) refinement subroutines; (4) service subroutines: driver, print, plot, etc. The code is written entirely in Fortran 77. The line count for each part is given in Table 1 (comments included). The critical FTT part on which the rest of the algorithm is built contains less than 800 lines.

The fourth column of Table 1 shows the breakdown of computer time for different parts of the algorithm compiled using an SGI f77 compiler with -O3 optimization and run on a single R10K processor of Origin 2000. The figures show that the FTT part of the code consumes a relatively small fraction of time, less than 20%. Another 10% of time is spent on gathering information into stride-one working arrays used in floating point computations, and on scattering the results back. The gather/scatter overhead is not specific to FTT. It is

TABLE 1

Algorithm's part	Lines	Time, % (R10000)	Time, % (J90)
FTT	800	17.5	20
Hydro	1485		
Computations		66.0	61
Gather/scatter		10.5	15
Refinement	601	3	2
Services	626	3	2
Total	3512	100	100

inherent to all tree-based AMR algorithms. The reason for noticeable FTT and gather/scatter overheads is an indirect addressing. In the fluid dynamics part of the code, work is done by looping over contiguous blocks of memory, and all loops are effectively optimized. Actual hydro computations take $\simeq$66% of time. The fifth column gives the time breakdown for the same code compiled using Cray cf77 compiler with -Zu parallel option enabled and run on a single Cray J90 processor. The relative numbers for J90 are similar to that for a R10K. The J90 computers lack hardware gather/scatter capabilities, have only two paths between the vector processor unit and the memory, and have a rather long memory latency. The code will run more efficiently on Cray C90 or T90 computers. On a single R10K processor, the computer time to advance one cell one substep is 17 μs per dimension. The corresponding time for a single processor of Cray J90 is 30 μs. In a multiprocessor mode, a speedup of $\simeq$10 was achieved on a 16-processor J90 in a batch environment.

For the cylindrical strong point explosion and for the shock–bubble interaction tests, Figs. 13 and 14 show the ratio of the number of computational cells actually used in the

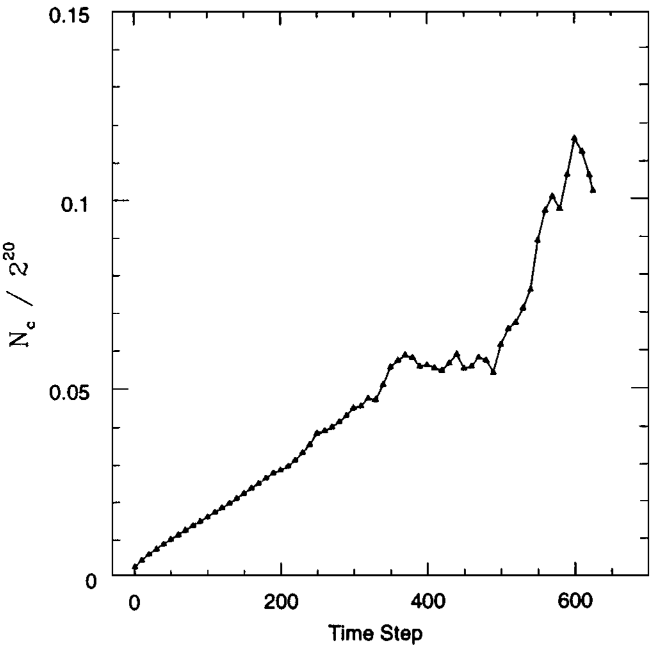


FIG. 13. The ratio of actually used computational cells to the number of uniform grid cells for various time time steps of the cylindrical strong point explosion problem (Section 7.4).

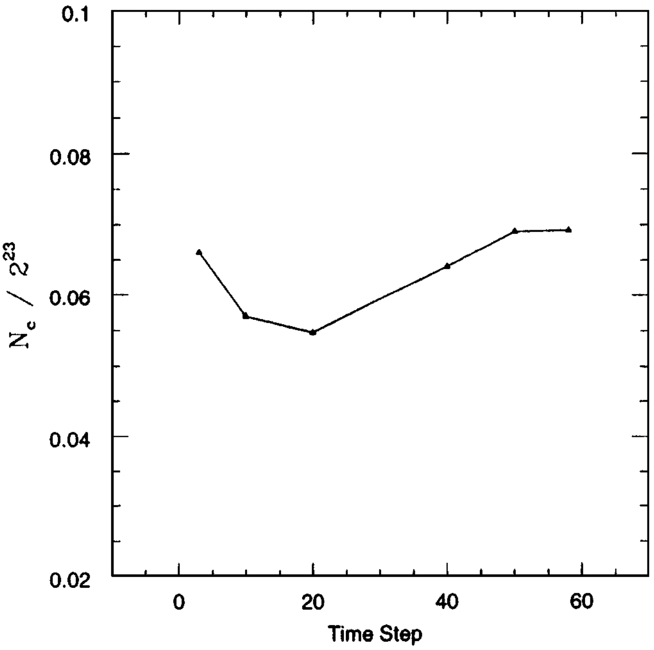


FIG. 14. The ratio of actually used computational cells to the number of uniform grid cells for various time time steps of the shock–spherical bubble interaction problem (Section 7.5).

simulation to the number of cells required to achieve an equivalent uniform resolution. The ratio varies with time. For the cylindrical shock test, for example, it is almost zero in the beginning, when the shocked material occupies only a small fraction of the computational domain. Then the ratio grows with time almost linearly because fine cells tend to concentrate around the surface of the shock only. At the end of the simulation, the ratio reaches its highest value $\simeq$0.1. At this time the entire computational domain is filled with shocks, and the regions of constant flow have disappeared. On average, the ratios for both simulations are $\simeq$0.05–0.07. This represents a factor of $\simeq$15–20 savings in memory compared to a brute force uniform grid computations. Savings in computer time should be $\simeq$30% less if we take into account that the FTT and gather/scatter overheads would be absent in grid-based computations.

9. CONCLUSIONS

A fully threaded tree structure (FTT) for adaptive refinement of regular meshes has been described. The FTT allows all operations and modifications of the tree to be performed in parallel, which makes this structure well suited for the use on vector and parallel computers. A filtering algorithm has been described for removing high-frequency noise in mesh refinement.

A FTT-based AMR algorithm for the integration of the Euler equations of fluid dynamics was presented. Time stepping and mesh refinement algorithms specific to the integration of the Euler equations were described. The integration in time is done using different time steps at different tree levels. The integration and mesh refinement are interleaved to avoid the creation of buffer layers of fine mesh ahead of moving shocks and other discontinuities. The time-stepping algorithm described can utilize any method of evaluating numerical fluxes which is at least second-order accurate in space and time on a uniform mesh.

The FTT performance on a single R10K processor of Origin 2000 and on a shared memory Cray J90 computer is reported. The FTT has low memory and computer time overheads, two words and $\simeq$30% computer time per computational cell, respectively. A factor $\geq$10 savings in memory and computer time, compared to equivalent uniform-grid computations have been achieved in two- and three-dimensional test simulations presented in the paper.

There are many uses of FTT other than the integration of the Euler equations, such as the solution of parabolic and elliptical partial differential equations, pattern recognition, computer graphics, particle dynamics. An application of the adaptive refinement tree to dark matter cosmological simulations and to reactive Navier–Stokes equations are described in [25, 36]. In some of these simulations, savings in memory and computer time were more than a factor of 100.

ACKNOWLEDGMENTS

Parts of the work were supported by NSF Grant AST94-17083, NASA Grant NAG52888, by the DARPA Applied and Computational Mathematics Program, and by the Office of Naval Research. The author is grateful to Almadena Chtchelkanova (Berkeley Research Associates, Inc.) for her advice on various data structures and algorithms, to Sandy Landsberg for references, to Elaine Oran and Jay Boris for useful discussions, and to Elaine Oran for reading the manuscript. Discussions with Andrey Kravtsov and Anatoly Klypin (NMSU) during the final stage of code development resulted in several improvements of the FTT structure. The author thanks anonymous referees for useful comments.

REFERENCES

1. M. J. Berger and J. Oliger, Adaptive mesh refinement for hyperbolic partial differential equations, *J. Comput. Phys*. **53**, 484 (1984).
2. M. J. Berger and P. Colella, Local adaptive mesh refinement for shock hydrodynamics, *J. Comput. Phys*. **82**, 64 (1989).
3. M. J. Berger and R. J. LeVeque, *An Adaptive Cartesian Mesh Algorithm for the Euler Equations in Arbitrary Geometries*, AIAA Paper 89-1930-CP (1989).
4. J. J. Quirk, *An Adaptive Grid Algorithm for Computational Shock Hydrodynamics*, Ph.D. thesis, Cranfield Institute of Technology, U.K., 1991.
5. J. J. Quirk, An alternative to unstructured grids for computing gas dynamic flows around arbitrary complex two-dimensional bodies, *Comput. & Fluids* **23**, 125 (1994).
6. J. J. Quirk and S. Karni, On the dynamics of a shock–bubble interaction, *J. Fluid Mech*. **318**, 129 (1996).
7. R. B. Pembert, J. B. Bell, P. Colella, W. Y. Crutchfield, and M. L. Welcome, *An Adaptive Cartesian Grid Method for Unsteady Compressible Flow in Complex Geometries*, AIAA Paper 93-3385-CP (1993).
8. J. Bell, M. J. Berger, J. Saltzman, and M. Welcome, Three-dimensional adaptive mesh refinement for hyperbolic conservation laws, *SIAM J. Sci. Comput*. **15**, 127 (1994).
9. M. Ruffert, Collisions between a white dwarf and a main-sequence star, *Astronom. & Astrophys*. **265**, 82 (1992).
10. M. Ruffert and W. D. Arnett, Three-dimensional hydrodynamic Bondi–Hoyle accretion, *Astrophys. J*. **427**, 351 (1994).
11. R. I. Klein, C. F. McKee, and P. Colella, On the hydrodynamic interaction of shock waves with interstellar clouds, *Astrophys. J*. **420**, 213 (1994).
12. G. G. Duncan and P. A. Hughes, Simulations of relativistic extragalactic jets, *Astrophys. J. Lett*. **436**, L119 (1994).

13. D. P. Young, R. G. Melvin, M. B. Bieterman, F. T. Johnson, S. S. Samant, and J. E. Bussoletti, A locally refined rectangular grid finite element method: Application to computational fluid dynamics and computational physics, *J. Comput. Phys*. **92**, 1 (1991).

14. D. D. Zeeuw and K. G. Powell, An adaptively refined cartesian mesh solver for the Euler equations, *J. Comput. Phys*. **104**, 56 (1993).

15. M. G. Edwards, J. T. Oden, and L. Demkowicz, An $h-r$-adaptive approximate Riemann solver for the Euler equations in two dimensions, *SIAM J. Sci. Comput*. **14**, 185 (1993).

16. W. J. Coirier, *An Adaptively-Refined, Cartesian, Cell-Based Scheme for the Euler and Navier–Stokes Equations*, Ph.D. thesis, University of Michigan, 1994.

17. W. J. Coirier and K. G. Powell, An accuracy assessment of cartesian-mesh approaches for the Euler equations, *J. Comput. Phys*. **117**, 121 (1995).

18. M. J. Berger and J. E. Melton, An accuracy test of a cartesian grid method for steady flow in complex geometries, preprint. [*Proc. 5th Intl. Conf. Hyp. Prob., Stonybrook, NY, 1994*], to appear.

19. M. J. Aftosmis, J. E. Melton, and M. J. Berger, *Adaptation and Surface Modeling for Cartesian Mesh Methods*, AIAA Paper 95-1725-CP (1995).

20. J. E. Melton, M. J. Berger, M. J. Aftosmis, and M. D. Wong, *3D Applications of a Cartesian Grid Euler Method*, AIAA Paper 95-0853 (1995).

21. Yu-L. Chiang, B. van Leer, and K. G. Powell, *Simulation of Unsteady Inviscid Flow on an Adaptively Refined Cartesian Grid*, AIAA Paper 92-0433 (1992).

22. S. A. Bayyuk, K. G. Powell, and B. van Leer, *A Simulation Technique for 2-D Unsteady Inviscid Flows around Arbitrary Moving and Deforming Bodies of Arbitrary Geometry*, AIAA Paper 93-3391-CP (1993).

23. S. A. Bayyuk, K. G. Powell, and B. van Leer, An algorithm for simulation of flows with moving boundaries and fluid–structure interactions, in *Proceedings, The First AFOSR Conference on Dynamic Motion CFD, (New Brunswick, New Jersey, 3–5 June 1996)*.

24. N. Dale and H. M. Walker, *Abstract Data Types: Specifications, Implementations, and Applications* (Jones & Bartlett, Boston, 1996).

25. A. V. Kravtsov, A. A. Klypin, and A. M. Khokhlov, Adaptive refinement tree—A new high-resolution N-Body code for cosmological simulations, *Ap. J. Suppl*. **111**, 73 (1997).

26. P. Colella and H. M. Glaz, Efficient solution algorithms for the Riemann problem for real gases, *J. Comput. Phys*. **59**, 264 (1985).

27. B. van Leer, Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method, *J. Comput. Phys*. **32**, 101 (1979).

28. P. Colella and P. R. Woodward, The piecewise parabolic method (PPM) for gas-dynamical simulations, *J. Comput. Phys*. **54**, 174 (1984).

29. V. P. Korobeinikov, *Zadachi Teorii Tochechnogo Vzryiva* (Nauka, Moskow, 1985), p. 78.

30. J.-F. Haas and B. Sturtevant, Interaction of weak shock waves with cylindrical and spherical gas inhomogeneities, *J. Fluid Mech*. **181**, 41 (1987).

31. J. M. Picone and J. P. Boris, Vorticity generation by shock propagation through bubbles in a gas, *J. Fluid Mech*. **188**, 23 (1988).

32. P. R. Woodward and P. Colella, The numerical simulation of two-dimensional fluid flow with strong shocks, *J. Comput. Phys*. **54**, 115 (1984).

33. D. H. Porter, P. R. Woodward, W. Yang, and Q. Mei, Simulation and visualization of compressible convection in two and three dimensions, in *Nonlinear Astrophysical Fluid Dynamics* edited by R. J. Buchler and S. T. Gottesman (New York Academy of Sciences, New York, 1990), p. 234.

34. E. S. Oran and J. P. Boris, Computing turbulent shear flows—A convenient conspiracy, *Comput. in Phys*. **7**, 523 (1993).

35. E. S. Oran and J. P. Boris, *Numerical Simulations of Reactive Flows* (Elsevier Science, New York, 1987).

36. A. M. Khokhlov and E. S. Oran, Interaction of a shock with sinusoidally perturbed flame, *Combust. & Flame*, 1997, submitted.