Solution Adapted Nested Grid Refinement for 2-D PDEs

James. M. Hyman[1] and Shengtai Li[1,2]

[1]Theoretical Division, Mail Stop, B284
Los Alamos National Laboratory
Los Alamos, NM 87545

[2] Department of Computer Science
University of California
Santa Barbara, CA 93106

September 14, 2004

Abstract

We have developed a robust, versatile adaptive mesh refinement (AMR) method in 2-D for problems where the fine-scale profile of sharp fronts should be resolved.

After comparing the effectiveness of different data structures and clustering algorithms, we propose an easyly-implemented hierarchical data structure, which can be done by any high level language, and an efficient clustering algorithm. We also investigate and implement the adaptation based on the geometry, the use of staggered grids and the choice of the refinement ratios. We have designed the software to minimize the changes needed in existing codes to make full use of the adaptive algorithms and provide full control by the user. Applications of the 2-D code to several nontrivial problems (including both hyperbolic and parabolic problems) are presented.

1 Introduction

In this paper, we extend the 1-D AMR data structure and algorithms [8] for the 2-D case. An overview of the AMR method and a detail description of the data structure and algorithms can be seen in [8]. To be concise, this paper concentrates on issues unique to the 2-D case. Because of the difference of geometry, a patch (sub-grid) in 2-D AMR hierarchical data structure can have multiple parents and siblings, which will affect the design of the data structure and algorithms.

After briefly describing several data structures for hierarchical data in two (or more) dimensions, we propose a a slight variation in our 1-D hierarchical data structure [8] and Berger's AMR data structure [4].

The clustering algorithm is sensitive, complex and largely determines the efficiency of the 2-D AMR method. Therefore, we compare several clustering algorithms in Section 3.1 and propose a more robust and efficient one by combining different strategies together and using an adaptive variable threshold. We modify the integration algorithm of our 1-D AMR in Section 3.2 so that higher order time integration or implicit temporal integration in method of lines (MOL) approach can be used. We study the impact of the boundary condition on integration and "plug-and-play" approach in Section 3.3. We also investigate other adaptation issues related to complex geometries, the choice of the refinement ratios and the use of staggered grids in Section 3. We intend to designed a software to minimize the changes needed in existing codes to make full use of the adaptive algorithms and provide full control by the user. Therefore, we We describe the differences between the 1-D and 2-D algorithms in the mechanism for user control of the refinement and integration. Finally, we provide numerical examples to demonstrate the effectiveness of our approach.

2 Hierarchical Grid Structure

There are several ways to store the logical and physical positions of these points on each of the AMR levels. The simple approach of storing the grid points on each level as a multidimensional array $p(ix, iy, ilevel)$ is wasteful of memory, because at higher refined levels, only a small part of the grid is used.

The refinement areas in a hierarchical data structure are scattered on the coarse grids, much like the nonzero elements in a sparse matrix. The local uniform grid refinement (LUGR) by Verwer et al. [21] uses storage methods for sparse matrices based on a modified condensed sparse row (CSR) format to store the grid information. This storage scheme does not need clustering, which is the most sensitive and complicated part of AMR [4]. However, data management for CSR is more difficult and is incompatible with most existing PDE software.

Berger's hierarchical data structure clusters the refinement areas into logical rectangles, called "patches", and treats the patch as the basic data unit [5, 8]. The scalar attributes of all the patches on the same level include the *level* number, integration *time*, time step-size, number of ghost boundaries, number of buffer zones, refinement ratios to coarser and finer grids, etc. These shared attributes form a LEVEL class at the top of the data structure for a grid level. The separate attributes for each patch include the pointers to the parents and children, number of grid points, logical grid index for each point, etc. Although a *doubly linked list* is more convenient for operations between different levels, it needs more storage and updating during refinement. Because the AMR process works up from the fine grids to the coarser ones, the pointers to children are not often used and can be eliminated without loss of efficiency.

Our construction of the data structure for 2-D AMR is almost the same as we have done for our 1-D AMR method in Part I [8]. However, because of the difference of geometry and the existence of multiple parents for one patch, pointers p_j in each grid (patch) $G_i = |m_i|p_1|G_{i,1}|p_2|G_{i,2}|...|p_{m_i}|G_{i,m_i}|$ have different meaning from the 1-D case. The variable p_j contains the number of the parent coarse grids for the patch $G_{i,j}$ and we use an auxiliary array to store the indices of the parent grids to access the parent grids.

Another change from the 1-D AMR data structure is that each patch may have one or more sibling patches, which share a common internal boundary. The position and solution on the common internal boundaries and other overlapped ghost boundaries must be consistent during the integration and before the refinement. The sibling pointers are not used by the refinement algorithm. So they can be isolated from the data structure and computed in the integration module after the refinement is completed. Because the siblings are used only to update the boundary values for any two adjacent patches, we need not store the indices of the siblings for each patch. Instead, we store the information to indicate which two patches are siblings. (In practice, we use the index $p_i + N * p_j$, where N is some number larger than the maximum number of patches, to indicate that the p_ith patch and p_jth patch are siblings in the current level).

To make it easier to use the AMR software in existing codes and facilitate reusing the integrator of a single grid, the patch structure contains all the characteristics of a single grid, such as the ghost boundary information, starting logical coordinates and ending logical coordinates. To allow a flexible number of ghost boundary points we use a tensor-product of two 1-D intervals (see [8]), $G_{i,j} = [I_{bx}, I_{ex}] \times [J_{bx}, J_{ex}]$, illustrated in Fig. (2.1). The boundary conditions are used to extend the solution to ghost points outside the patch. This approach can accommodate most other single grid data structures.

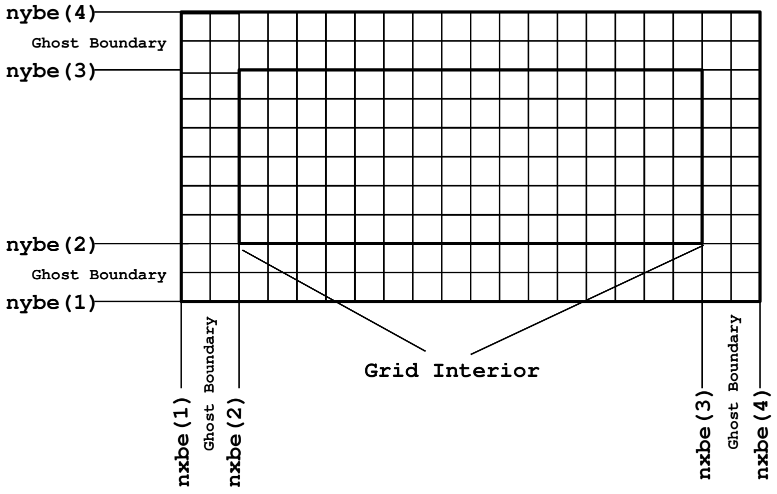

Figure 2.1: Data structure for a single grid. The data stored for a single patch includes the grid, solution, problem parameters and AMR pointers. This information is used by the the dynamic memory allocation algorithm to minimize the workspace and memory (data transfer) operations.

3 The AMR Integration Algorithm for 2-D

The integration algorithm for 2-D AMR differs slightly from the 1-D case because of the differences in data structure. The multiple parents for each patch in 2-D complicated the communications between different refinement levels. **Regrid**(*level*), **Boundary Collection**(*level*) and **Project***(level, level+1)* (see [8]) algorithms should be done carefully. The sibling information must be used to maintain the consistency of any two siblings during **Advance**(*level*). The **Cluster**(*level*) algorithm is the most difficult to extend to higher dimensions and requires special treatment.

For hyperbolic conservation laws the solution data must be updated just before the projection from the fine grid to the coarse grid, based on the flux conservation law at the fine-coarse grid interface. We adopt Berger's flux-correction method [3] at the coarse-fine grid interfaces.

3.1 Clustering

The clustering algorithm is sensitive, complex and largely determines the efficiency of the 2-D AMR method. We consider four similar approaches based on Bergers original algorithm [4].

The *nearest neighbor clustering* algorithm originally used by Berger [4, 3] is highly heuristic and time-consuming. First a cluster is defined consisting of a single (arbitrary) tagged node. Other tagged nodes are added to the cluster if they are within a specified minimum inter-cluster distance from the nearest node in the cluster. Finally, clusters are merged when a node is determined to belong to more than one of them. It costs about $O(n^2) + O(p^2)$ on n nodes and p clusters requiring refinement.

The *mean-cut clustering* used by Quirk [17] starts by placing a minimal bounding box around the cells that have been flagged to be refined. Then a ratio of the number of flagged points to the total number of points in the box is computed. If the ratio is greater than a threshold (0.5–0.75), the long edge of the box is bisected. The process is repeated until the ratio is below the cutoff. This algorithm requires $O(nlog(n))$ operations. It operates in computational rather than physical space and it produces no overlaps. Unfortunately, there are some situations where this simple approach fails to produce a good clustering. For example, for the flagged region in Fig.3.1, the mean-cut clustering produces 10 clusters although 4 clusters are sufficient.

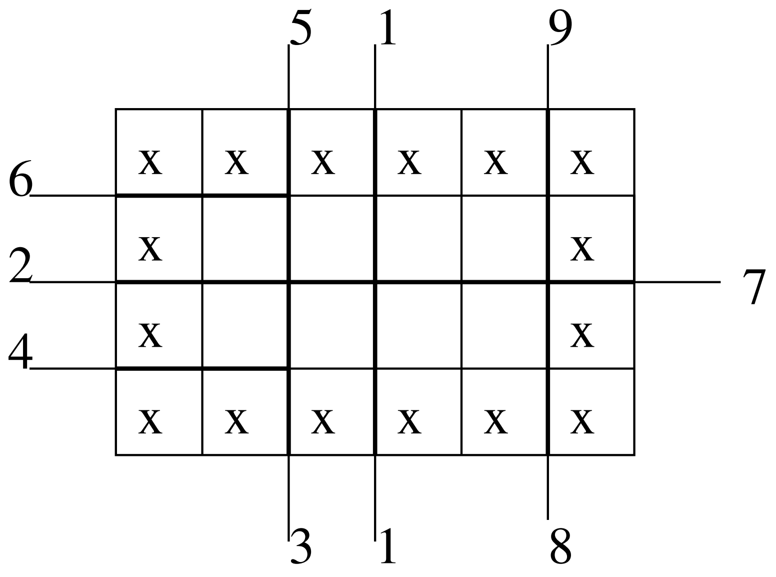

Figure 3.1: Example of mean-cut clustering. The "x" represents the flagged cell. The number beside each line (around the boundary) denotes the order number of the cutting. It is done recursively, and requires a total of 9 cuttings and produces 10 clusters (patches)

The *signature and cut clustering* algorithm proposed by Berger and Rigoutsos [6] modifies the definition of the cutting index in the mean-cut algorithm by computing a signature list along each grid axis. The signature s_i is the number of flagged entries along the i^{th} grid surface. The zero entry in the signature list is used to indicate the best index to subdivide the domain. If there is no zero entry, the second difference of the signatures $\Delta_i = s_{i+1} - 2s_i + s_{i-1}$ is used to define $z_{i+\frac{1}{2}} = |\Delta_{i+1} - \Delta_i|$. The index with the largest $z_{i+\frac{1}{2}}$ is chosen as the cutting index. The recursion stops when either the ratio for the cluster is less than some threshold or no cutting index is found (index is equal to 0).

The signature and cut algorithm produces excellent clusters in most cases and shares the high efficiency of mean-cut clustering. For the same example (above), this algorithm requires four cuttings and produces four clusters. We found that there are some special situations where *signature and cut* clustering algorithm could be slightly improved. For example, suppose the refinement area is along the diagonal and the flagged cells are displayed as in Fig.3.2. The *mean-cut* clustering is necessary for this case.

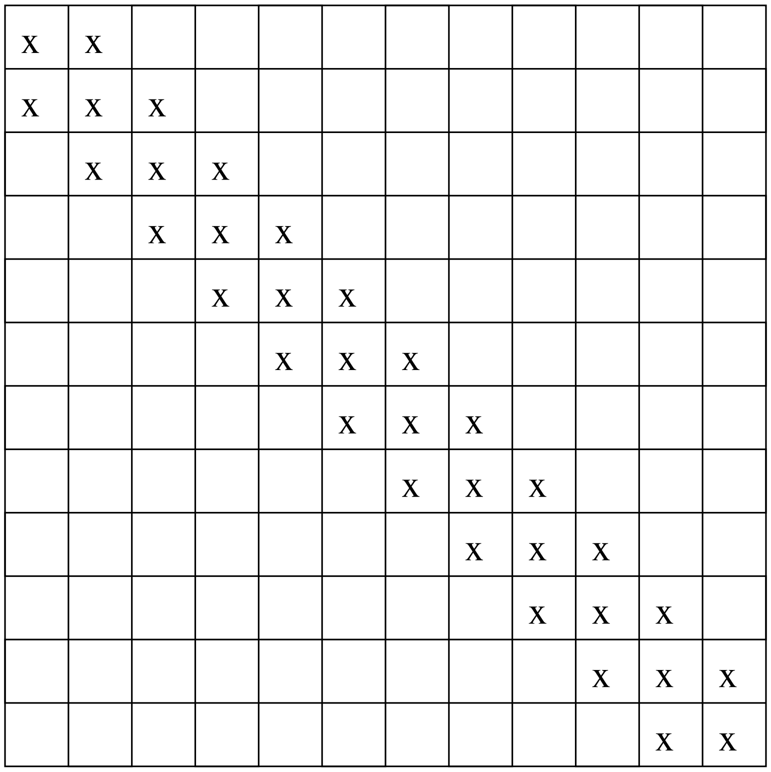

Figure 3.2: Example of *signature and cut* clustering There is no zero entry in the signature list. All the $z_{i+\frac{1}{2}}$ are equal to zero. So there will be no cutting and the whole region will be refined and only one patch is produced.

To ensure the flux correction algorithm [3], which is performed at the fine-coarse interface, works correctly and easily, we require in our clustering that the grids be *properly nested* in the strict sense that, except at physical boundaries, the level l grids are large enough to guarantee that there is a border at least one level l cell wide surrounding each level $l+1$ grid. Although all the flagged points are inside the coarse grids, the new finer grids after clustering may not be properly nested in the coarse grids. This is because some unflagged points are added to the patches during clustering, and those points may not be inside the coarse grids. To ensure the proper nesting, the mean-cut may be used to cut the clusters into two. However, just as we have pointed out, the *signature and cut* algorithm can produce a better clustering in this case.

As we have noted in [8], the clustering procedure can start from any level that is less than maximum level. If the clustering starts from level $l > 1$, the new grids generated at level $l+1$ may not be properly nested in level l in a strict sense, because the flagged cells may reach the boundaries of the level l after a few time steps when level l grids remain the same during that period. To avoid this problem, the coarse grids must have enough buffer zones. In our refinement, we add different number of buffer zones for different levels: the coarser levels have more buffer zones than the finer levels.

For memory allocation and parallel problems, we may wish to produce clusters of no greater than a specified size. Therefore any cluster larger than that is bisected along the longest axis (the mean-cut is performed here). Therefore, if a patch already produced satisfies the threshold condition, it may need further cutting to ensure the proper nesting and/or proper size.

The threshold value does affect the efficiency of the algorithm. The bigger the threshold, the fewer unwanted points are involved, but usually more clusters result. More clusters introduce more artificial internal boundaries which may increase the storage requirement and computation time.

During our implementation, we found that a fixed threshold(> 0.5), which is constant during the clustering, usually results in small patches when the depth of the recursion is large. These small patches dramatically reduce the computational efficiency, especially on computers with vector processing units. Every patch formed during the clustering requires both internal and ghost boundary cells. In very small patches, the number of cells can even be smaller than the number of ghost boundary cells.

These small patches can be eliminated by post processing the selection decisions after the clustering. However, we have been unable to devise a robust post processing optimization that is not problem-specific. We did observe that dynamically reducing the threshold on the recursion could improve performance. Since a small patch is produced only when the depth of the recursion is large, if we decrease the threshold when the depth of the recursion increases, the recursion will stop and a relatively big patch will be produced. In our implementation, we decrease the threshold by 10% for each depth of the recursion. The threshold at the start of the recursion can be large, such as 0.9 or 0.8. This approach does not add many unflagged points to the flagged regions, because the patch becomes smaller when the recursive depth increases.

Our overall algorithm combines *signature and cut* clustering with *mean-cut* clustering. Given an initial logically rectangular domain and the scattered flagged cells on it, a cluster of small patches that cover the flagged cells are output by the following algorithm. For convenience, we will use the *stack* data structure, which has a property of *last in first out* (LIFO). First push the current domain into stack.

1. If the stack is empty, **STOP**;
2. Pop out one domain from stack and set it as the current domain;
3. Shrink the domain boundaries so that it becomes a minimal bounding box for the flagged cells in the current domain. If there is no flagged cell, go to 1.
4. Compute signature list s_i along each grid line.
5. Computed ratio r of flagged cells to the total number of cells.
6. Compute the threshold for the current recursive level by $\theta_j = \theta_0 \cdot 0.9^j$, where θ_0 is the initial threshold and j is the number of elements in the stack.
7. If $r \geq \theta_j$, then
   (a) if the number of nodes in the current domain exceeds the restricted maximum number, setting the cutting index as the middle of the longest axis (mean-cut) and go to 10;
   (b) if the resulting patch is not properly nested in the coarse grids, go to 8 (using the signature to find the cutting index);
   (c) otherwise, output the current domain as a new patch, and go to 1.
8. Find the cutting index by the following approaches:
   (a) A zero entry in the signature list is used as the cutting index;

(b) If there is no zero entry, the Laplacian second derivative of the signatures is computed,

$$\Delta_i = s_{i+1} - 2s_i + s_{i-1}.$$

If $z_{i+\frac{1}{2}} = |\Delta_{i+1} - \Delta_i|$ is largest, this index is chosen as the cutting index;

9. If the cutting index is 0, then setting it as the middle of the longest axis.

10. Divide the domain into two along the cutting index, push them into stack. Go to 1.

To further reduce the number of the patches in a level, we also merge any two patches that have a common edge. The new patches must have proper size that is no great than the maximum size in any direction.

Our clustering algorithm is optimized for serial computation or parallel computation with distributed memory and does not take into consideration the advantages of long vector operations when there are vector processors. On vector machines there may be significant advantages to modifying the shape of the patches to maximize the vector length.

3.2 Advancing the solution one time step

When there are multiple siblings for a patch, these siblings must be kept consistent during the integration. During the integration, the internal cells are advanced with the PDE, while the ghost boundary cells are interpolated in time and space from internal values on the parent coarser grids. We can integrate each patch separately when the boundary routine is called only once during one time step. The ghost cells are defined from the sibling patches, or collected according to the external boundary conditions in the boundary routine at each evaluation of the time derivative. For these values to be accurate, the patch must be solved and advanced simultaneously with its sibling patches.

To illustrate this process, suppose the solutions at the boundary cells have been updated before integration for a refinement level. To advance the solution with an explicit single-step time integration method:

1. Compute the time derivatives of the boundary cells for each patch on the level using the boundary values at the forward time (collected before integration of the current level) and backward time, and keep them as constant during a one time-step integration.

2. Collect the external boundary values using the external boundary conditions if any patch reaches the external boundaries.

3. Evaluate the time derivative of the internal cells for each patch on the level.
4. Calculate the intermediate solution for each patch with the time derivatives from 1 and 2.
5. Update the solution at the boundary cells for each patch with more accurate values from its sibling patches.
6. Goto 2 until the integration completes one time step.

When the integration method is implicit, then the connectivity between different patches must be accounted for when using an iterative method to be sure the solution on each patch has the appropriate updated value when the implicit method is updating the solution. In particular, the boundary conditions in an overlapping region must be carefully applied be sure the approximation is accurate.

An existing code for a single grid can usually be quickly adapted to this approach by either directing using the time derivative by mapping the solution and data structures back and forth between and using the original solver routine as a "black box" to advance the patch data a single time step. We call this second approach "plug-and-play."

3.3 Boundary conditions

Each of the four sides of a patch can have an external or internal boundary. The boundary conditions are treated the same as in 1-D [8] by defining the solution in a cocoon of ghost points surrounding each patch. The boundary values are assigned during the refinement, while the values of the internal boundary cells at the forward time are collected from the parent coarse grid before the integration.

Four kinds of external boundary condition routines are supplied in our AMR system: inflow (Dirichlet), outflow (extrapolated), reflecting (symmetric and antisymmetric), and periodic. The inflow boundary conditions often have analytic values, for example, the amount of material being ejected from a source. The outflow boundaries typically have extrapolated values. A second order extrapolation is used when needed. The reflecting boundaries are like mirror images. The scalar variables, such as density, may have symmetric values to their internal images, while the vector variables, such as velocities, may have antisymmetric values to their internal images specifies which kind of conditions is used on an external boundary or supply his own external boundary routines. Our AMR system has a switch for the user to input special boundary conditions for some specific problems. Because a patch may not have all of the 4 external boundaries, the user must treat each boundary separately.

There are three sources for the boundary collections: sibling patches, external boundaries, and parent coarse patches. Unlike most AMR implementations, which collect the boundary values from all of the three source at once, we do the boundary collections at different times for different sources. The ghost boundary values are also stored during our integration and refinement. This enables us to use our monitor function to flag the refined cells and allows a high order time integration for an MOL approach.

Using patch translation (see also [14]), we treat the periodic boundary as an internal boundary [8]. The translation can be done easily by adding a constant to the patch structure and does not need additional instrumentation for the boundary treatment. After translation, it is possible that a sibling of one patch at one side may be on the opposite side. This is identified and tracked and monitored by the consistency maintenance algorithm.

If the boundary routine is called only once during one time step integration, we can do the "plug-and-play" directly. The *claw2* solver in CLAWPACK belong to this category can be plugged it into our AMR solver without any recoding. If the boundary routine is called more than once during one time step integration, as in a high-order Runge-Kutta method, we must recode the time integration routines with the algorithm discussed in section 3.2. The recoding is much easier when the method is implemented with time derivative being explicitly defined. Thus, we update all the patches as a single level for each stage in a multigrid method before proceeding to the next stage. We have supplied an interface for second and fourth order Runge-Kutta methods.

3.4 Geometric complexity adaptation

Problems with complex geometry caused by irregular boundary curves and embedded physical objects (see [13] and its references) can be solved by a Cartesian grid method, which covers the irregular regions with uniform grid, and uses the standard Cartesian grid methods for regular cells and special treatment for the boundary cells. This approach has proven effective using rotated boxes [9], flux-redistribution procedures [15], and merging procedures [16]. The Cartesian grid method requires storing the intersection between the boundary curve of the physical domain and the uniform mesh for the boundary conditions and the approximation of spatial derivatives. A major advantage of this approach is the simplicity of discretizing the PDEs on a uniform grid.

An irregular region can also be descritized with a body-fitted structured grids [20]). It is more difficult to derive high order methods for the irregular boundary fitted grids. Because the AMR method is based on the local data structure not the physical location, it can still be used on these grids. If a body-fitted structured grid is used, it affects the storage and memory requirements, and dynamic memory allocation for the AMR system. We know that for a uniform mesh, we do not need to store the physical positions of each point. However, for curvilinear grids, the physical positions of each point must be stored for later use. Many AMR systems consider only one kind of grid or store the physical positions for all kinds of grids. Our AMR system has a switch, which is input by the user, between curvilinear grids and other kinds of grids. If a curvilinear grid is used, we store the physical position for each point. Otherwise, we assume that the grid is tensor-product or uniform, and only physical positions for each direction are stored.

Another type of geometry complexity is related to the topology of the domain. For instance, the domain may be of L or U shape, or has some holes internally. We treat this case by splitting the domain into several logical rectangular domains, which imply that the base grid may contain several subgrids instead of one. The external boundaries of the base grid will be given by the external boundaries for each hole and the whole logical rectangular domain.

To reduce the effects of the physical coordinates during the integration, we design as many of our algorithms as possible in the computational domain. However, there are some algorithms which must use the physical positions for the grids. The physical positions for the grids are needed to evaluate the spatial derivatives, for the **Select**(*level*) (see Fig. 4.1 of [8]) algorithm (which flags the bad points for the next refinement level) and (when using the CFL condition) to control the time step.

3.5 Choosing the time refinement ratio adaptively

In most AMR systems, the refinement ratios between different levels are given by the user and are not allowed to change during the integration. We have investigated choosing the efficient refinement ratios adaptively during the refinement.

To optimize the refinement ratios, we normalize the problem and define

- r is the refinement ratio for two adjacent levels,
- a is the ratio of the areas between the fine grid regions and coarse grid regions,
- 1 is the integration cost of the coarse grid,

- 1 time step-size for the coarse grid is equal to r time step-sizes on the fine grid (*i.e. for hyperbolic problems*).
- h is the overhead cost introduced by the AMR.

Then the total integration cost for two adjacent levels consists of the cost of one integration on the coarse grid, r times the integration cost on the fine grid, and the overhead cost due to the AMR procedures (including the refinement, injection, projection, boundary-collection, etc.) between them.

The total cost is given by

$$1 + r * ra * ra + h = 1 + r^3a^2 + h.$$

If we do not refine between them and replace the coarse grid with the fine grid completely, the cost will be r^3.

To find the optimal r, let

$$(1 + r^3a^2) + h \leq r^3.$$

Most AMR systems [17], have an overhead cost h below 15% of the integration cost. Therefore, an optimal refinement ratio r satisfies

$$(1 + r^3a^2)(1 + 0.15) < r^3.$$

If $1 - 1.15a^2 > 0$, i.e., $a < 0.9325$, then

$$r > (\frac{1.15}{1 - 1.15a^2})^{1/3}.$$

The refinement ratio should be kept as small as possible to reduce the total integration cost. If $a \geq 0.9325$, the whole coarse grid should be replaced by the fine grid to reduce the overhead due to AMR. For $a < 0.9325$, the following table shows the relationship between r and a.

r	2	3	4	8
$a <$	0.863	0.912	0.924	0.931

If we choose $r = 2$, then $a < 0.863$. That is, the refinement ratio $r = 2$ is optimal, when the ratio between the number of bad cells and the total number of cells is less than 0.863. In practice, the ratio a is usually much less than 0.863. That is why $r = 2$ is an appropriate default refinement ratio in most AMR systems.

We adaptively choose the refinement ratios by starting with $r = 2$ and unpacking and storing all the grids separately. For example, if the current refinement ratio is $r = 4$ between two adjacent levels, then these two levels are separated into three levels, with $r = 2$ between the two adjacent levels and the middle level is just twice as coarse as the finer level. Next, we refine the temporal grid appropriately (e.g., for hyperbolic PDEs the time refinement is the same as in space.) Finally, we merge any two adjacent levels for which $a < 0.8$. This process is done recursively until no levels can be merged. The refinement ratios are reset during the merging process.

This optional adaptive procedure involves a little overhead in storage and computation but is worthwhile if any two levels are merged.

3.6 Finite difference and finite volume method

We have incorporated both the finite difference, which is node-centered, and finite volume method, which is cell-centered, in our AMR system. There are some differences between the two methods. First the communication between the coarse and fine grid is totally different because the solutions locate in different place and have different meaning. The cell-centered solution for finite volume method means an average value in a cell whereas the node-centered solution for finite difference method means the solution value on that nodes. Second, for an n_x by n_y grid, only $n_x - 1$ by $n_y - 1$ positions have solution values for a finite volume method , which will complicate the management of data structure.

We have designed our AMR system in such a way that both the finite difference and finite volume method will work well.

4 User Control over the Adaptively

The user control and format of the grid file for 2-D is just the same as our 1-D AMR software [8] except that each patch consists of two corners (each corner has two coordinates) instead of two ends (each end has one coordinate).

The logical coordinates and any point in a uniform mesh are easily computed from the base grid information and the physical positions of two corners of a patch. However, for a curvilinear grid, we must locate the logical cell in the base grid which contains the corner. To avoid the $O(n^2)$ cost of a line-sweep to check if the cell contains the corner. We use the mean-cut algorithm. The mean-cut algorithm for clustering is done in the computational domain but here it must be done in the physical domain. We recursively check which half of the logically rectangular domain contains the corner, and then eliminate the half that does not contain the corner, divide the remaining half in half and continue until only one cell that contains the corner is left.

To locate the half of the logical rectangular domain which contains the corner, we compare the corner point with the middle grid line of the domain. Because the middle grid line may not be a straight line, we should find a straight line which has the same relationship to the corner point as the middle line. First we locate the point on the middle line closest to the corner. Then the line segment connected to the nearest point on the middle line can be used to determine in which side the corner point is located.

Because overlaps are not allowed in our AMR system when the refinements are input by the user, we check first to see if there is overlap among the inputs and if there is, we re-cluster the input patches to eliminate the overlap.

Instead of inputting patch by patch in the grid file, the user can input a polygonal region which specifies the forced refinement area. The vertices of the polygon are used to define the region. Our program will read the data, find the logical coordinates for the refinement regions, and then cluster them into the patch structure automatically.

5 Numerical Experiments

In the test examples, we refine during the integration of the coarse grid and fix the refinement ratio as $r = 2$. We choose the number of buffer zones to be 1 and the number of ghost boundaries to be 2.

5.1 Cluster Examples

Because the clustering algorithm is crucial to the 2-D AMR system, we first present two examples of clustering.

5.1.1 Burgers' Equation

The first example is Burgers' equation

$$u_t + uu_x + uu_y = 0.005(u_{xx} + u_{yy}), \tag{1}$$

with an analytic solution

$$u(x, y, t) = \frac{1}{1 + \exp(x + y - t)/(2r)},$$

The solution is a steep wave front propagating to the right top corner. We start the integration at $t = 0.75$ and define the analytic solution in our **advance(***level)* routine. The flagged points are distributed along the diagonal, and all of the z_i for the signature list are zero. Therefore, there is no cutting return for the signature and cut algorithm, and only one patch is produced for any level if the signature and cut clustering is used. After combining the mean-cut clustering with the signature and cut algorithm, we get a better result (see Figure 5.1.1). We also compare the results for a variable threshold ($theta_0$=0.8) and a fixed threshold (0.7) in our clustering algorithm. We use 4 refinement level. The total number of points for the finest level with the variable threshold increases about 10% compared with the fixed threshold (0.7) method. However, the total number of patches for the level decreases from 78 to 18.

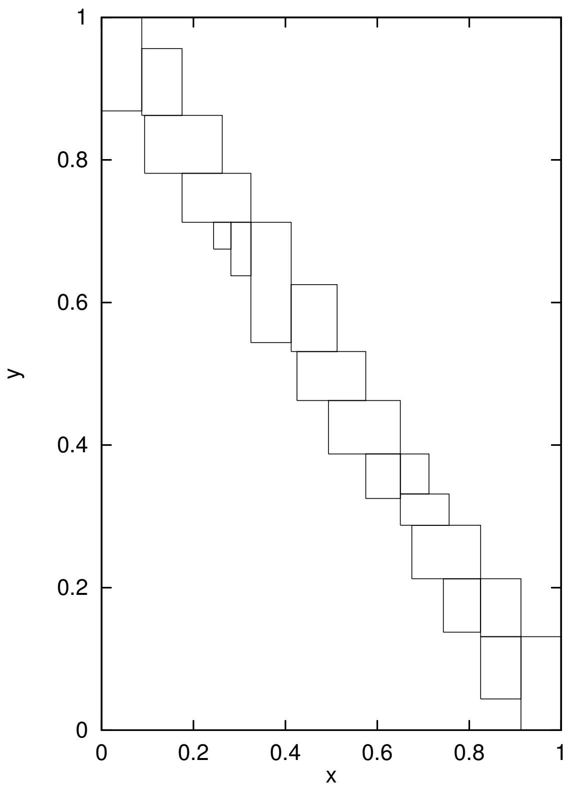

Figure 5.1-a: Clusters of the finest level with a variable threshold (start with 0.8) for Eq. (1) at t=1.0. A total of 18 clusters are generated.

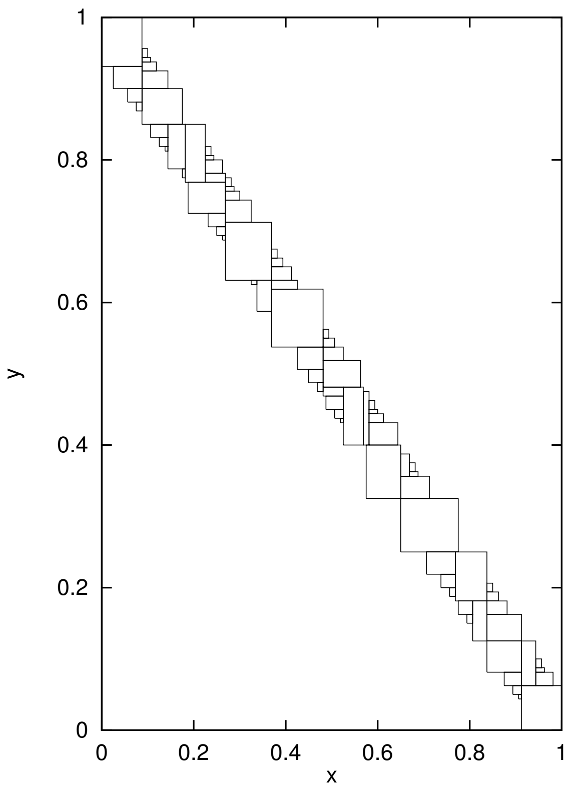

Figure 5.1-b: Clusters of the finest level with a fixed threshold (0.7) for Eq. (1) at t=1.0. A total of 78 clusters are generated.

5.1.2 Rotating Cone Problem

The next example is a hyperbolic problem used in [4].

$$u_t - yu_x + xu_y = 0, \tag{2}$$

with initial condition

$$u(x,y,0) = \begin{cases} 0, & \text{if } (x-0.5)^2 + 1.5y^2 \geq \epsilon \\ 1 - \epsilon((x-0.5)^2 + 1.5y^2), & \text{if } (x-0.5)^2 + 1.5y^2 < \epsilon \end{cases}$$

on a rectangular domain $-1.1 \leq x \leq 1.1$, $-1.1 \leq y \leq 1.1$.

The parameter ϵ in [4] is $\frac{1}{2}$. Here we set $\epsilon = \frac{1}{6}$ (also see [1]). The solution

$$u(x,y,t) = \max(0, 1 - 16[(x\cos(t) + y\sin(t) - 0.5)^2 + 1.5(y\cos(t) - x\sin(t))^2])$$

is a cone rotating counterclockwise about the origin. We use four refinement levels and a base grid, $\Delta x = \Delta y = 0.05$. The clustering and refinement algorithms were first tested using the exact solution in the integrator. The variable threshold (starting from 0.8) method results in 13 patches in the finest level (Fig. 5.3-a) while fixed threshold (0.7) method results in 30 patches in the finest level (Fig. 5.3-b). The figures illustrate how the smaller threshold results in many small patches at the edge of the cone. The smallest patch contains only 4 cells while the number of ghost boundary cells for this patch is 36.

5.2 Shock wave problem

5.2.1 Inviscid Burgers' Equation

As pointed out in Part I [8], our AMR system can easily incorporate legacy codes for a single mesh. The inviscid Burgers' equation,

$$u_t + uu_x + uu_y = 0, \tag{3}$$

with discontinuous initial condition

$$u(x,y,0) = \begin{cases} 1.0 & \text{if } 0.1 \leq x \leq 0.6 \text{ and } 0.1 \leq y \leq 0.6, \\ 0.1 & \text{otherwise,} \end{cases}$$

and periodic boundary conditions, is chosen from the test cases of CLAWPACK [12] for 2-D,

The solution is a shock wave propagating to the top right corner of the domain. When the shock arrives at the boundary, it emerges from the opposite side and corner because of the periodic boundary condition. We use CLAWPACK routine *claw*2 as a solver for the single grid and provide a dummy (stub) boundary routine but do not modify any internal code. We provided an interface that mapped between our parameters and those in CLAWPACK. We use three refinement levels and a 50×50 base level. The clusters and the contour plots are shown in Fig. 5.4-a. To investigate the effectiveness of the algorithm of handling periodic boundary in our AMR system, we compare the solution with one solved on an extended domain to $[0, 1.2] \times [0, 1.2]$, where the shock does not reach the boundaries yet. The contour plots are shown in Fig. 5.4-b.

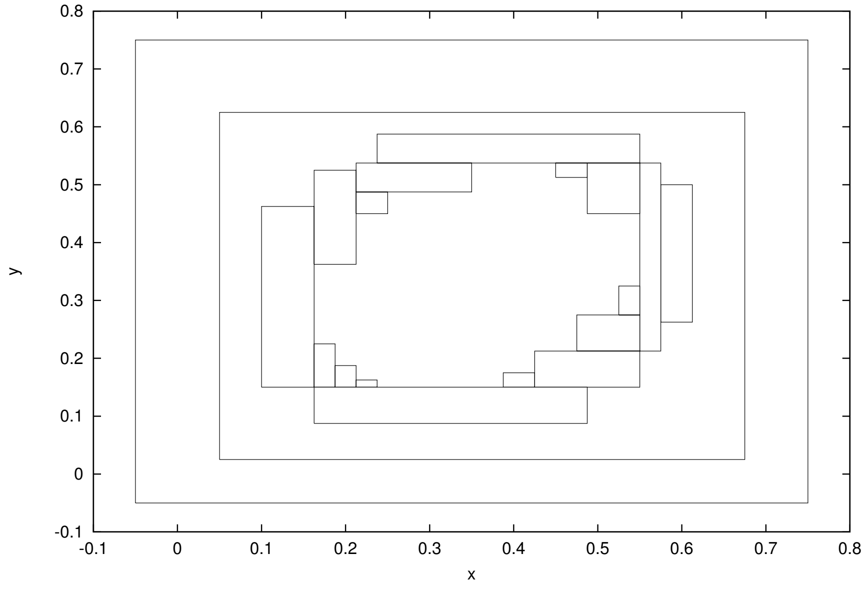

Figure 5.3-a: Result of clustering with variable threshold for Eq. (2) at time $t = 0.78$

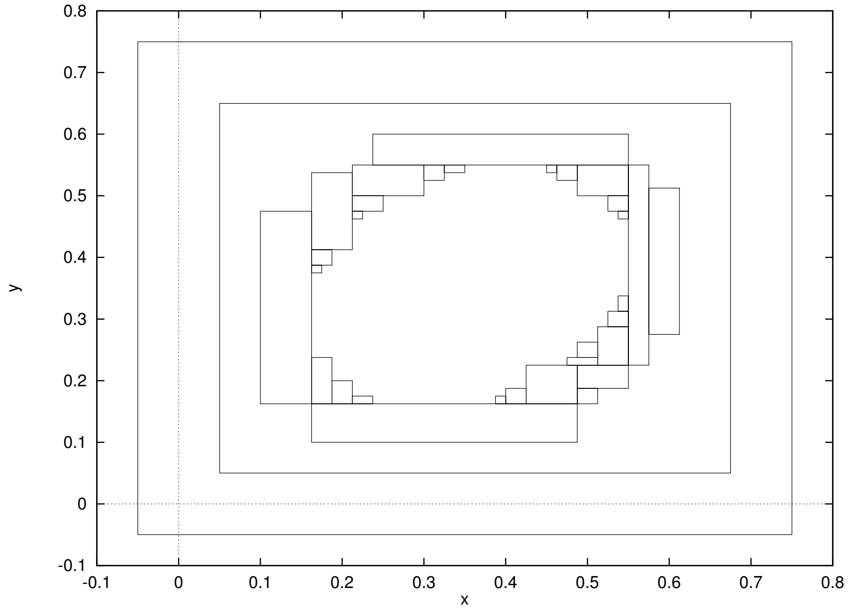

Figure 5.3-b: Result of clustering with fixed threshold (0.7) for Eq. (2) at time $t = 0.78$

5.2.2 Double Mach Reflection

The *double Mach reflection of a strong shock* [22] [3] is given as Euler equations for 2-D gas-dynamics:

$$\begin{aligned} \rho_t + (\rho u)_x + (\rho v)_y &= 0, \\ (\rho u)_t + (\rho u^2)_x + (\rho uv)_y &= 0, \end{aligned}$$

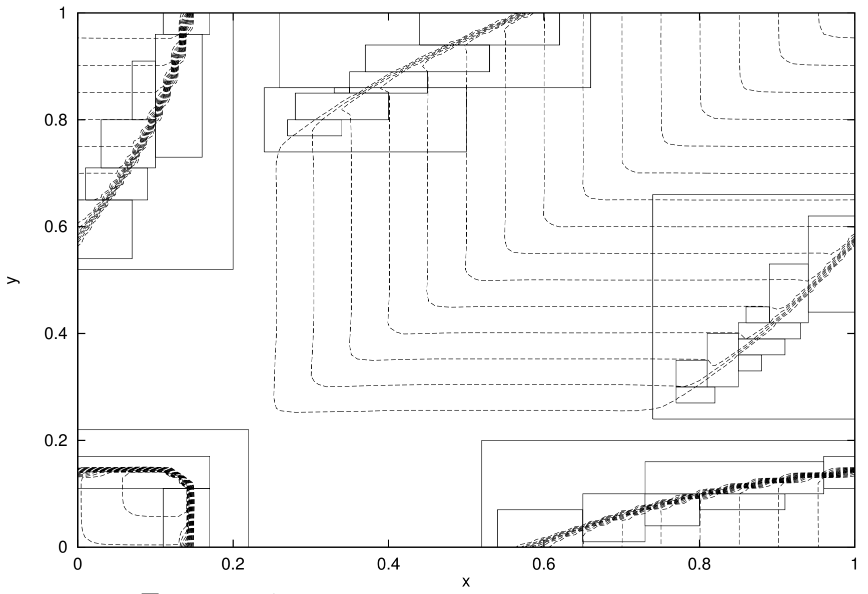

Figure 5.4-a: Clusters for Eq. (1) at time $t = 1.0$.

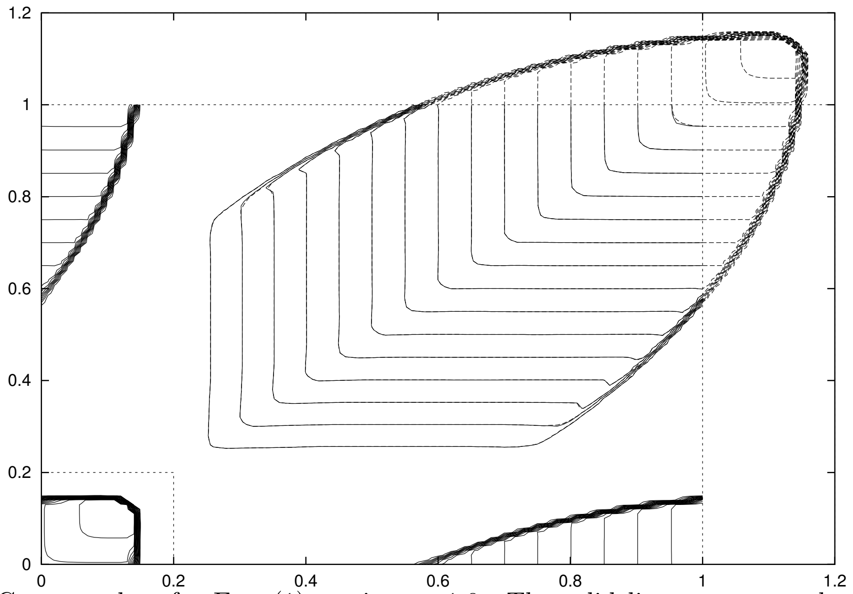

Figure 5.4-b: Contour plots for Eq. (1) at time t=1.0. The solid lines represent the contours for the domain $[0,1] \times [0,1]$. The long dashed lines represent the contours for the extended domain $[0,1.2] \times [0,1.2]$. 17 contours from 0.15 to 1.0 are plotted.

$$\begin{aligned} (\rho v)_t + (\rho uv)_x + (\rho v^2)_y &= 0, \qquad (4) \\ e_t + ((e+p)u)_x + ((e+p)v)_y &= 0, \end{aligned}$$

where $p = (\gamma - 1)(e - \frac{1}{2}\rho(u^2 + v^2))$ is pressure.

The reflecting wall lies at the bottom of the $[0,4] \times [0,1]$ computational domain starting from $x = 0.1$. Initially a right-moving Mach 10 shock is positioned at $x = 0.1, y = 0$, and makes a 60° angle with the x-axis. For the bottom boundary, the exact post shock condition is imposed for the part from $x = 0$ to $x = 0.1$ and a reflective boundary condition is used for the rest. At the top boundary of the computational domain, the flow values are set to describe the exact motion of the Mach 10 shock (see [22] for a detailed description of this problem).

The Mach stem solution obtained using *claw2* on a single grid is severely kinked (see Fig. 5.5-d). We tracked the problem to the CLAWPACK approximate Riemann solver (Roe's method) which can admit spurious solutions that are triggered by an incorrect treatment of shear waves. The kink point on the Mach stem has also been observed by by Quirk [17].

The solution obtained with, second order MUSCL, provided by Colella [7] did not have this problem. We use three refinement levels and a 200×50 base grid. The clustering for the finest level grid at $t = 0.2$ is shown in Fig. 5.5-a. The contour plots on part of the domain: $[0, 3] \times [0, 1]$ are shown in Fig. 5.5-b.

5.2.3 A Mach 3 wind tunnel with a step

This model problem, which has been used in [22] for uniform grid, solves the same equation (5) . The setup of the problem is the following: The wind tunnel is unit length wide and 3 length units long. The step is 0.2 length units high and is located 0.6 length units from the left-hand end of the tunnel. The problem is initialized by a right-going Mach 3 flow. Reflective boundary conditions are applied along the walls of tunnel and in-flow and out-flow boundary conditions are applied at the entrance (left-hand end) and the exit (right-hand end). We did not perform any special techniques (see [22]) on the corner of the step. The second order MUSCL [7] is used as a solver for the single grid. The initial base grid is divided into two patches: [0.0,0.6]×[0.0,1.0] and [0.6,3.0]×[0.0,1.0]. Two refinement levels are used during the integration. The clustering and the contour plot is shown in Fig. 5.6. The CLAWPACK has the same difficulty as in double Mach reflection problem to resolve the Mach stem on the left side the step.

5.3 Nonlinear Diffusion Equation

The nonlinear diffusion equation

$$u_t = (u^3)_{xx} + (u^3)_{yy}, \tag{5}$$

proposed by Aronson [2], originates from a diffusion phenomena of the flow when the internal object is removed from the flow field.

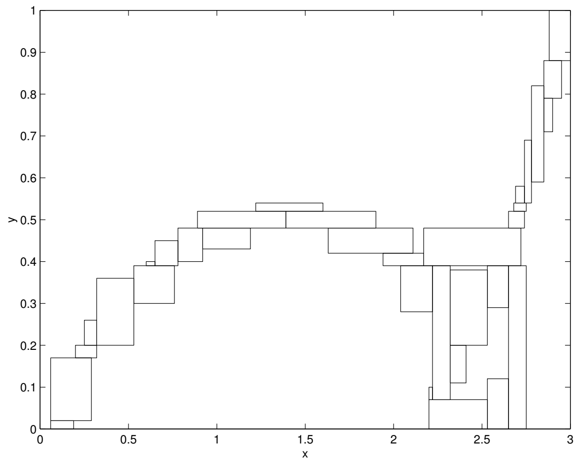

Figure 5.5-a: Clusters of the finest level for double Mach reflection problem at t=0.2 by AMR with plugging in the MUSCL solver. The base grid is of 200×50.

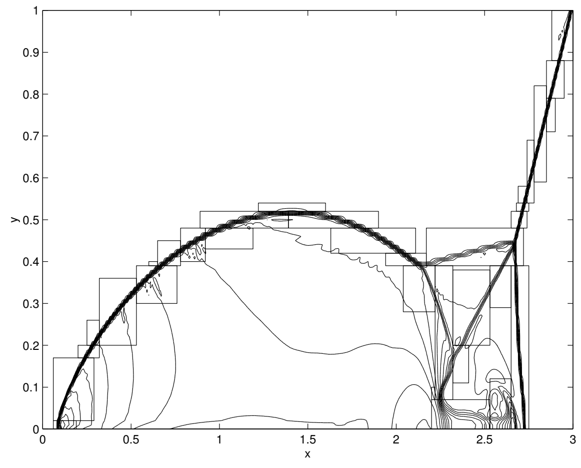

Figure 5.5-b: Contour plot for double Mach reflection problem at t=0.2 by AMR with plugging in the MUSCL solver. Only density is plotted. 30 contours from 1.731 to 20.91 are used.

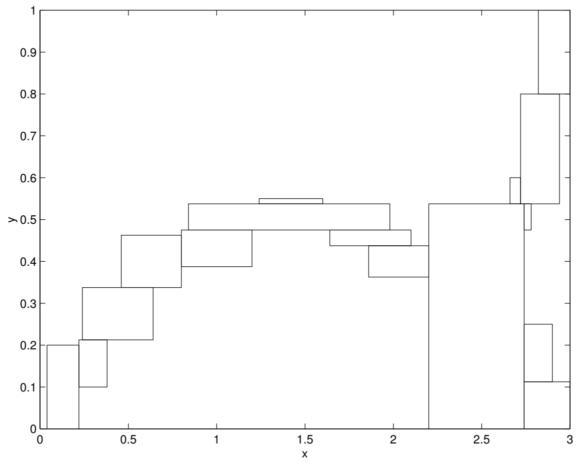

Figure 5.5-c: Clusters of the finest level for double Mach reflection problem at t=0.2 by AMR with plugging in the CLAWPACK solver. The base grid is of 200×50.

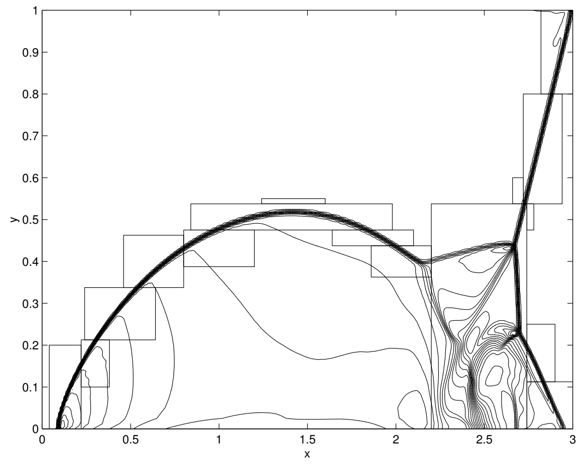

Figure 5.5-d: Contour plot for double Mach reflection problem at t=0.2 by AMR with plugging in the *claw2* solver of the CLAWPACK. Only density is plotted. 30 contours from 1.731 to 20.91 are used.

The initial condition

$$u(x,y,0) = \begin{cases} 0, & \text{if } (x-0.5)^2 + (y-0.5)^2 < 0.04 \\ 0.5, & \text{otherwise in } [0,1] \times [0,1]. \end{cases}$$

collapses in to fill the vacant hole. The wave front is steep and moves fast before it becomes only one point. Because only the region near the wave front requires high accuracy, and that region is becoming smaller and smaller due to the diffusion, AMR can be used to advantage. The position of a circular steep wave front can be derived via theory. We compare our numerical solution for the position of the front versus time in Fig. 5.7-c. In this comparison,

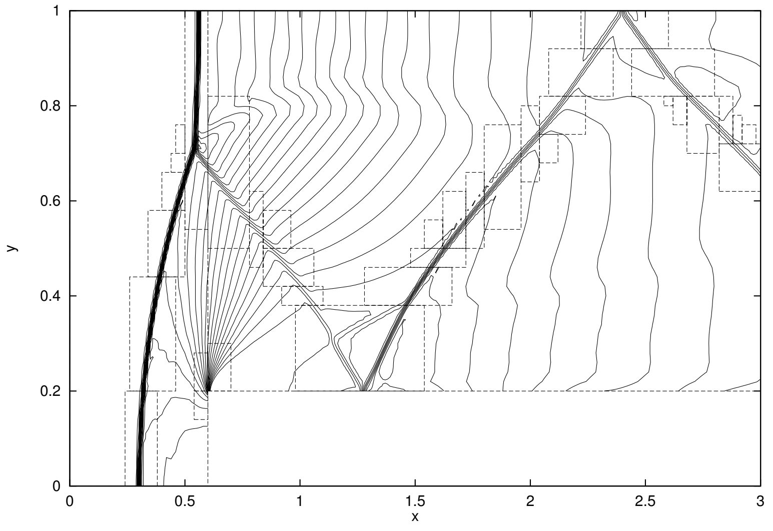

Figure 5.6: Cluster and contour plot for Mach 3 wind tunnel problem at t=4.0 by AMR with plugging in the MUSCL solver. of the CLAWPACK. Only density is plotted. 28 contours from 0.25 to 7.0 are used.

we define position of the wave front to be the average distance to (0.5,0.5) from all the points whose solution values are between 0.002 and 0.02.

We use three-level refinement and a 50×50 base grid. We use centered difference for the spatial discretization and a second order Runge-Kutta method for the time integration. Because it is an explicit method, the time step is chosen based on both stability and accuracy. For each level, we compute the time steps based on the requirement of accuracy and stability, and choose the smaller one. To keep the tolerance for this problem less than 1.0E-5, we choose the initial time step for the base grid to be 0.0001.

The contour plots of the solution and the grid clusters are displayed in Figs 5.7-a and 5.7-b. We also compared these solution with four levels of refinement and observed few differences between the results 5.7-c.

6 Conclusions

We have presented our implementation of AMR in 2-D and the differences between it and our 1-D AMR [8]. We propose a slight change in the existing clustering algorithms, a new algorithm for adaptively choosing the refinement ratios, user control over the adaptation, and high-order or implicit temporal integration, etc. The data structure and algorithms involve relatively little overhead in refinement and data management, and can be implemented effectively with any high level language.

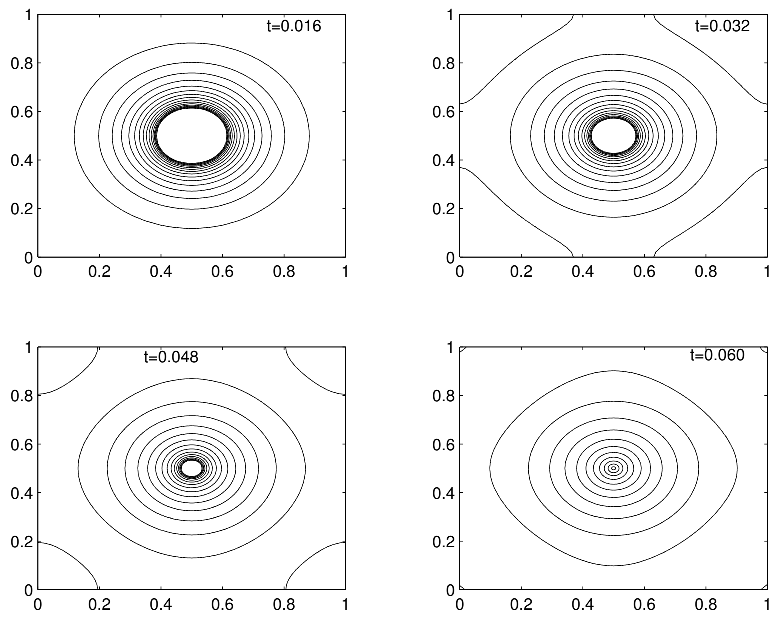

Figure 5.7-a: Contour plot for diffusion problem (5). 15 contours from 0.3 to 0.48 are used.

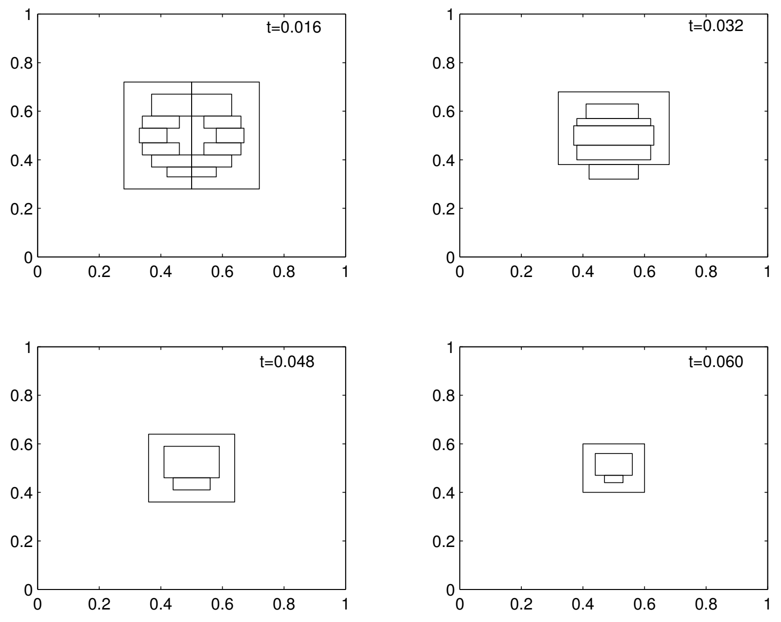

Figure 5.7-b: Clustering for diffusion problem (5)

The adaptive mesh modules are completely separated from the PDE solver for a single grid and can usually be incorporated into existing codes, which work for a single grid, with little effort. We have proposed a new boundary collection and integration algorithm, which allows a semi-discretized PDE solver and method of lines approach to plug into our AMR system easily. Our design is general-purpose and can be used to solve a wide class of time-

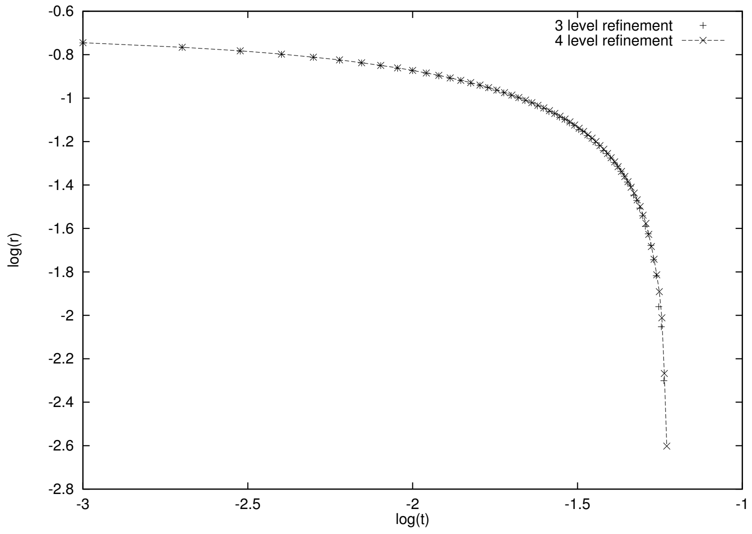

Figure 5.7-c: Log-Log plots for front position versus time

dependent PDEs.

References

[1] D. C. Arney and J. E. Flaherty, A two-dimensional mesh-moving technique for time-dependent partial differential equations, *J. Comput. Phys.*, 67 (1986), 124-144.

[2] D. Aronson, Personal communication, 1997.

[3] M. J. Berger and P. Colella, Local adaptive mesh refinement for shock hydrodynamics, *J. Comput. Phys.* 82 (1989), 64-84.

[4] M. J. Berger and J. Oliger, Adaptive mesh refinement for hyperbolic partial differential equations, *J. Comput. Phys.* 53 (1984), 484-512.

[5] M. J. Berger, Data structures for adaptive grid generation, *SIAM J. Sci. Stat. Comput.* 3 (1986), 904-916.

[6] M. J. Berger and I. Rigoutsos, An algorithm for point clustering and grid generation, *IEEE Trans. on Systems, Man, and Cybernetics*, 21 (1991).

[7] P. Colella, Multidimensional upwinding methods for hyperbolic conservative laws, *J. Comput. Phys.*, 87 (1990), 171-200.

[8] J.M. Hyman, and S. Li, Interactive and dynamic control of adaptive mesh refinement with nested hierarchical grids, Los Alamos National Laboratory Report (1998).

[9] R. LeVeque and M. Berger, A rotated difference scheme for Cartesian grids in complex geometries. AIAA Paper (1991), CP-91-1602.

[10] S. Li and L. Petzold, Moving mesh method with upwinding schemes for time–dependent PDEs, *J. Comput. Phys.* 131 (1997), 368–377.

[11] S. Li, L. Petzold and Y. Ren, Stability of moving mesh systems of partial differential equations, *SIAM J. Sci. Comput.* 20 (1998), 719–739.

[12] R. LeVeque, CLAWPACK codes for hyperbolic conservative laws, in Netlib, 1995.

[13] J.E. Melton, Automated Three-Dimensional Cartesian Grid Generation and Euler Flow Solutions for Arbitrary Geometries, PhD Thesis (1996), UC Davis.

[14] H. Neeman, Autonomous Hierarchical Adaptive Mesh Refinement for Multiscale Simulation, Ph.D thesis, (1996), Department of Computer Science, University of Illinois at Urbana-Champaign.

[15] R. B. Pember, J. B. Bell, P. Colella, W. Y. Crutchfield, and M. L. Welcome, An Adaptive Cartesian Grid Method for Unsteady Compressible Flow in Irregular Regions, *J. of Comput. Phys.* 120 (1995), 278-304.

[16] J. Quirk, An alternative to unstructured grids for computing gas dynamic flows around arbitrary complex two-dimensional bodies, *Computers Fluids* 23(1994), 125-142.

[17] J. Quirk, An Adaptive Grid Algorithm for Computational Shock Hydrodynamics, Ph.D thesis (1991), College of Aeronautics, Cranfield Institute of Tech.

[18] C. W. Shu and S. J. Osher, Efficient implementation of essentially non-oscillatory shock capturing schemes II, *J. Comput. Phys.*, 83(1989), 32-78.

[19] G. A. Sod, A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws, *J. Comput. Phys.* 43 (1978) 1-31.

[20] J. F. Thompson, Z. U. Wasi and C. E. Mastin, Numerical grid generation, (North-Holland, New York, 1985).

[21] J. G. Verwer, J. G. Blom, VLUGR2: A vectorized local uniform grid refinement code for PDEs in 2D, Report NM-R9307, (1993) CWI.

[22] P. R. Woodward and P. Colella, The numerical simulation of two-dimensional fluid with strong shocks, *J. Comput. Phys.*54 (1984), 115-173.