Numerical Analysis Project
Manuscript NA-83-02

March 1983

Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations

by

Marsha J. Berger

Joseph Oliger

Numerical Analysis Project
Computer Science Department
Stanford University
Stanford, California 94305

a

Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations

Marsha J. Berger*
Cou rant Institute
New York University
New York, N.Y. 10012

Joseph Oliger*
Computer Science Department
Stanford University
Stanford, CA 94305

Abstract

We present an adaptive method based on the idea of multiple, component grids for the solution of hyperbolic partial differential equations using finite difference techniques. Based upon Richardson-type estimates of the truncation error, refined grids are created or existing ones removed to attain a given accuracy for a minimum amount of work. Our approach is recursive in that fine grids can themselves contain even finer grids. The grids with finer mesh width in space also have a smaller mesh width in time, making this a mesh refinement algorithm in time and space. We present the algorithm, data structures and grid generation procedure, and conclude with numerical examples in one and two space dimensions.

*This research was supported in part by Office of Naval Research Contract NOOOi4.75C-1132 and in part by the National Science Foundation under Grant No. MCS77-02082.

1. INTRODUCTION

In this paper we describe an adaptive finite difference method for systems of hyperbolic partial differential equations. The solutions of these equations are often smooth and easily approximated over large portions of their domains but contain boundary layers or locally isolated internal regions with steep gradients, shocks, or discontinuities where the solution is difficult to approximate. .We adaptively place finer grids in these regions over a coarse grid covering the domain. The solution on each fine subgrid can then be approximated by standard finite difference techniques, as done on the coarse grid. If we are solving a time dependent problem, the difficult regions will change in time, and thus our grids must adapt in time in response to the solution. Our algorithm is very general, and makes no assumptions about the number or type of these irregular regions, nor about their direction of motion.

The grid refinements we introduce in two space dimensions are rectangles of arbitrary orientation. Our algorithm is recursive, in that subgrids with finer and finer mesh width can themselves be nested in other subgrids. We use oriented rectangles for two reasons: (1) it allows us to approximately align our coordinates with singular surfaces such as shocks, and (2) it allows us to reduce the size of the refined region and the number of mesh points introduced. Furthermore, this allows us a very simple user interface, and requires very little overhead to maintain. Our strategy for grid refinement is to maintain a constant mesh ratio of time step to space step on all grids. We refine in time as well as space, so large time steps are taken on coarse grids and small time steps on fine grids. Values on the

boundaries of the finer grids are defined using interpolation procedures applied to the coarser grid in which the refinement is embedded. We believe that our adaptive methods are the first to use such a space-time grid structure and find that this is a major factor in the efficiency of the method.

We have only implemented these methods for problems in one and two space dimensions over rectangular domains. However, they can Se extended to general regions using techniques like those used by Starius [1980] and B. Kreiss [1982] in a straightforward way. These approaches break up general domains into subdomains which are mapped onto rectangles. Our mesh refinement approach allows us to easily use different methods in different regions -- e.g., higher order approximations where the solution is smooth, and special lower order methods specifically designed for shocks were they occur. our algorithm, which decomposes the computation into regular blocks, can also be easily used for parallel or concurrent processing.

Adaptive methods have long been used as standard practice in many of the classical problems of numerical analysis such as quadrature and ordinary differential equations . Our methods are close in spirit to those discussed by Pereyra and Sewell [1975] for boundary value problems. The original motivation-for our adaptive method can be found in Oliger [1979].

Other adaptive methods have now been proposed for both time dependent and boundary value problems for partial differential equations. Adaptive multigrid methods have been proposed by Brandt [1977] for elliptic problems. Adaptive finite element methods have been developed by Babuska and Rheinboldt [1978] and Rank [1981] for

these same problems. Extensive theory has been developed for these adaptive methods. Recently adaptive finite element methods have started appearing for parabolic equations as well (Davis and Flaherty, [1982]), Gannon [1980], Sherman and Seager [1981]). Miller et al. [1981] have derived moving finite element methods which they have applied to both hyperbolic and parabolic equations. Adaptive grid methods for hyperbolic equations in one space dimension have also been cosidered by Hyman [1981] and Harten and Hyman [1982]. Bolstad [1982] has developed a one dimensional adaptive mesh refinement algorithm using methods very similar to ours. Dwyer et al. [1980] and Winkler [1976] have also done adaptive finite difference calculations, but their grid refinement is done in only one dimension. Our algorithms and data structures are for problems in two space dimensions, and are readily extended to three dimensions since there are no new topological difficulties.

In section 2 of the paper we describe our grid structure. Section 3 describes the integration algorithm for this grid structure. This includes the interactions between the grids as well as the technique for estimating the local truncation error, upon which our adaptive strategy rests. In section 4 we describe our method of subgrid generation, and in section, 5 the data structures used in our method. Numerical results obtained using our programs in one and two space dimensions are presented in section 6. Our computational results obtained with the adaptive programs are compared with computations on uniform grids with mesh intervals which are the same as the finest used in the adaptive computation. Ye have been able to achieve comparable accuracy with considerably less cost using the adaptive method.

2. GRID DESCRIPTION

In this section we describe the type of grids we use in solving a problem with our adaptive mesh refinement strategy. We also develop the notation and terminology needed to discuss these grids.

At the start of a computation only the coarsest, or base grid is specified by the user, This base grid, denoted G_0 , will remain fixed for the duration of the computation. We use the term grid to refer to the convex hull of the point set of the grid, rather than the point set itself. G_0 itself may be composed of several possibly overlapping component grids. Thus, we say that grids overlap if their convex hulls have a nonempty intersection. We call each component grid $G_{0,j}$, and loosely say that G_0 is the union of its components $G_{0,j}$. Each component grid is required to be locally uniform in some coordinate system. They need not form a simply connected domain. In addition, these grid components do not necessarily have the same mesh width. For example, we might use a grid over a region with a boundary layer that has a much finer discretization than that of the grid covering the interior of the domain. Furthermore, within each grid the mesh spacing in the coordinate directions need not be equal. For simplicity of exposition, however, we ignore these points and assume G_0 has mesh spacing $h_x = h_y = h_0$ on all components $G_{0,j}$.

During a computation, refined subgrids will be created adaptively in response to some feature in the transient solution, such as the estimated error in the solution or the appearance of shock fronts. Our goal is to generate the subgrids to best fit the area of the domain where they are needed. The subgrids we create are rectangles of arbitrary orientation. By keeping the subgrids locally uniform,

integration on these subgrids can be very efficient. By allowing the arbitrary orientation, it is possible to have a coordinate system which is locally approximately normal and tangent to some feature in the solution, for example a shock front. For some numerical methods, for example in fluid dynamics problems, it is important to have the flow basically along a coordinate direction. Our rotated grids easily allow for this. In addition, storage requirements can be significantly smaller by allowing rotated rectangles.

It is important to realize that these subgrids are not patched into the coarse grid. Rather, a subgrid should be thought of as overlaying a coarser grid. Each grid is defined independently of the other grids, with its own solution vector, storage, etc. In this way, each subgbrid can be integrated (almost) independently of the other grids. It also easily allows for the possibility cf using moving subgrids, even if the coarsest grid is stationary. By keeping the grids independent, the a lgori thm can be viewed as a method of domain decomposition, and is well suited for multiprocessor architectures currently under development. Other authors (Simpson, [1978]) have created refined meshes which are connected into the underlying coarse grid to make one global grid. In their approach, fine grid points are merged into the set of coarser grid points. In several dimensions this is difficult to do. It destroys-the local uniformity of each grid, substantially slowing down and complicating the integrator, as well as preventing its vectorization. In addition, it requires storage overhead and processing which is typically proportional to the number of refined grid points, instead of the number of refined grids.

It is possible that the fine subgrids will themselves contain even

finer subgrids within their boundaries, i.e., subgrids can be recursively generated. We say that a point in one grid is contained in another grid if it lies in the convex hull of that grid. A grid is contained in another grid if all points in that grid are contained in the other. Similarly, a point of one grid is an interior (boundary) point of another grid if it lies in the interior (on the boundary) of the convex hull of the other grid. We define the level of a grid to be the number of coarser grids the fine grid is contained in. The coarse grid G_0 is at level 0 in the grid hierarchy. The subgrids of G_0 are part of G_1 and they are said to be level 1 refinements. Refined grids within the G_1 grids are at level 2, denoted G_2, and so on. In this way, a nested sequence of grids with finer and finer discretizations may be created over some portion of the spatial domain. Each such grid is one grid component, denoted $G_{\ell j}$, of G_ℓ, where G_ℓ consists of those grids at level ℓ in the hierarchy having mesh width h_ℓ. A point in the problem domain may therefore be interior to several grids, but the approximate solution at that point is defined by interpolation from the finest grid to which that point is interior.

In practice, we assume a set of possible mesh discretizations $\{h_0, h_1, h_2, \ldots, h_{max}\}$ has been specified in advance where each h_ℓ is an integral multiple of $h_{\ell+1}$. A good choice for this refinement ratio will depend on how much of the domain needs what amount of refinement. If only one part of the domain needs to be in a fine grid with mesh width $h_j = h_0/r$, it is more efficient to create one level of refinement with $h_1 = h_0/r$ than two levels each with a ratio of $\sqrt{r}$. In general, however, not all areas needing refinement will need the same amount of refinement, arguing for smaller values of the refinement

ratio r. Since there is some overhead associated with refined grids, we prefer a refinement ratio of 4 over the ratio of 2 typically used with multigrid methods. In special cases where it is expected that all areas needing refinement will need a lot of it, higher values of r can be used efficiently.

We emphasize two points about the grid hierarchy. If at some time during the computation there is a grid with mesh width h_ℓ, we require that it be contained in a grid at level R-l, Which in turn is required to be contained in a level L-2 grid, and so on to the coarsest grid. All intermediate grids between the finest and coarsest are maintained. Furthermore, each point in a fine grid at level ℓ must be in the interior, not just on the boundary, of a grid at the next coarser level, unless it is on the physical boundary of the domain.

Second, not all. points in a fine grid are interior to the same coarse grid. We call this type of nesting level nesting. Figure 2.1

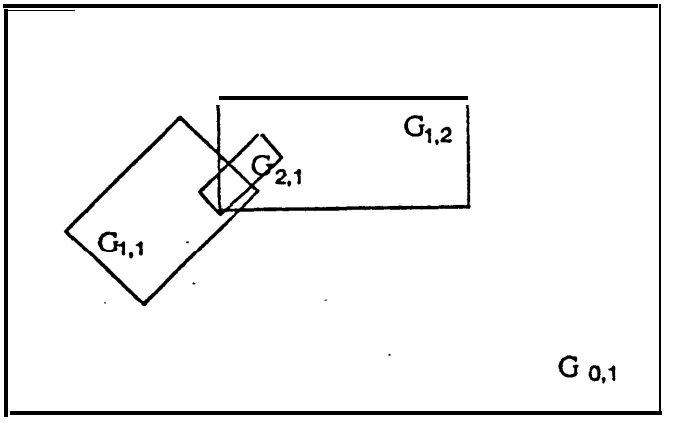


Figure 2.1 Sample Grid Structure

illustrates how the grid at level 2 is nested in the union of grids $G_{1,1}$ and $G_{1,2}$ at level 1. Figure 2.1 also illustrates the complication in two or more dimensions of overlapping refined grids, which we discuss later. In one dimension, where a grid is just an interval, level nesting is identical to straightforward grid nesting, and overlapping of fine grids does not occur. Grids at the same level of refinement with the same mesh width are either disjoint or else they are merged into one grid spanning a larger interval.

In sum, using the ideas of independent component grids and recursive refinement, a hierarchy of nested grids is formed. The complete grid structure is denoted

$$G = \bigcup_{\ell} G_{\ell} ,$$

where the grid structure at level ℓ is the union of rectangular components

$$G_{\ell} = \bigcup_{j} G_{\ell,j} .$$

3. INTEGRATION ALGORITHM

In this section we describe the integration algorithm that we use to solve a hyperbolic pde using mesh refinement. There are three main components to this algorithm. They are (i) the actual time integration using finite differences that is done on each grid, (ii) the error estimation and subsequent grid generation, and (iii) the special grid-to-grid operations that must be done every time step during the integration that arise because of mesh refinement itself. We describe these three components in turn.

Since each grid is defined as an independent computational entity, with its own solution vector, each grid can be integrated in time independently of the other grids, except for the determination of its boundary values. We must then consider the question of which grids to integrate when, and determine the order of their integration. This is made easy however by t'he following requirement.

Recall from section 2 that in our grid formulation, the mesh widths h_ℓ of grids at level ℓ are an integral factor r of the mesh width $h_{\ell-1}$ of the next coarser level. We use the same factor to set the time step on the level ℓ grids, $k_\ell = k_{\ell-1}/r$. In this way we keep the mesh ratio λ of time step to space step constant on all grids. This makes our algorithm one of mesh refinement in time and space, One of the main reasons our method is efficient is because the overly restrictive small time step of the finest grid is not applied over the entire domain.

This constant mesh ratio λ makes it easy to determine the order of grid integration. The steps are interleaved so that before advancing a grid, all its subgrids are integrated co the same time. At every

coarse grid step, all grids should be at the same time. One coarse grid cycle is then the basic unit of the algorithm. Fig. 3.1 illustrates this in one space dimension and time.

Since our refined grids are rotated rectangles, the difference equations must be transformed into the rotated coordinate system. This can be done in an automatic way, so that the integrator, which is supplied by the user, can be separated from the adaptive mesh routines. For standard finite difference equations, this can even be done in a conservative way (Viviand, [1978]), which is important for problems with discontinuous solutions. To solve

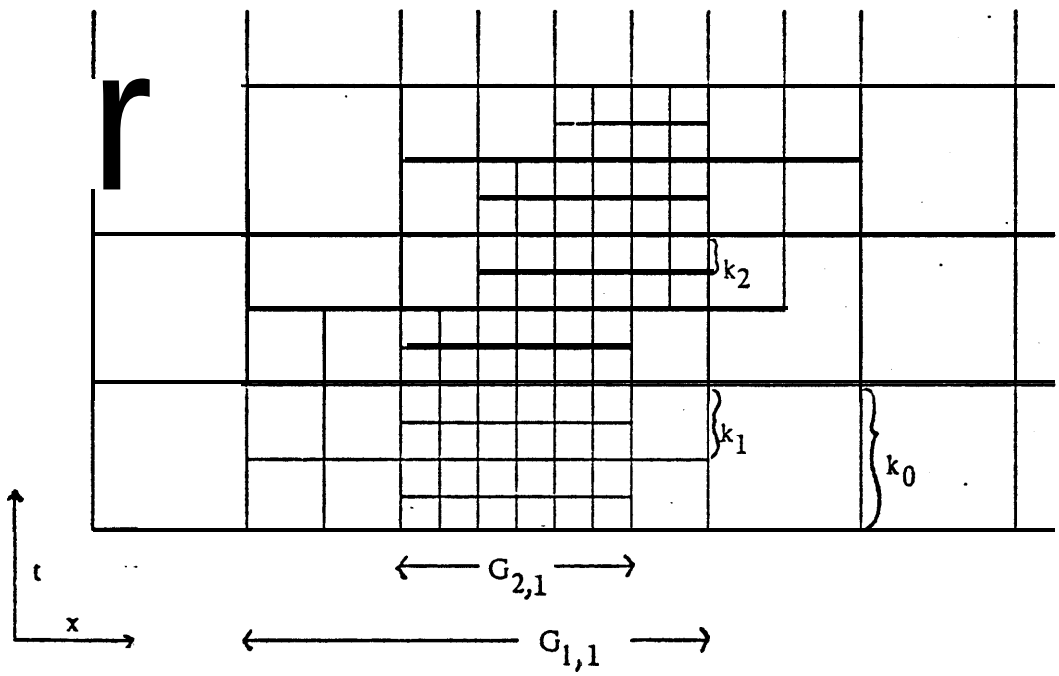


Figure 3.1 1D SPACE TIME MESH

$$u_t = f(u)_x + g(u)_y$$

under the more general coordinate transformation

$$r = r(x,y)$$

$$s = s(x,y)$$

we solve

$$u_t = \hat{f}(u)_r + \hat{g}(u)_s$$

where

$$\hat{f} = \frac{(r_x f + r_y g)}{J}.$$

$$\hat{g} = \frac{(s_x f + s_y g)}{J}$$

and J is the determinant

$$J = \det \begin{bmatrix} r_x & r_y \\ s_x & s_y \end{bmatrix}.$$

We point out that it is sometimes not necessary to transform the difference equations for each grid. If the physical problem is invariant to translation and rotation, we can use the identical difference equations on each grid. It can also happen that the difference scheme itself is invariant under rotation. Jameson [1974]

has proposed a rotated difference scheme for transonic flow calculations. The rotation is built into the scheme to be able to properly difference the potential equations in the streamline and normal directions.

Finally, we note that it is sometimes beneficial to use different integrators in different regions of the solution. For example, in a shock problem one might use a first order integrator on a refined grid around a shock, and a higher order method elsewhere. Another approach could be to use a more expensive method such as the Glimm scheme only on refined grids, and a less expensive integrator on the coarse grid. One can also solve different equations on different grids. For example, it is possible to only add artificial viscosity on a fine grid (around a shock zone), or solve boundary layer equations only on a separate grid in the boundary layer. Several integration nodules can be easily supplied by the user without any further changes to the mesh refinement program.

Error estimation and the subsequent regridding operation is the second major task of the mesh refinement algorithm. This is where most of the canputational overhead of the method lies. Every several time steps, we estimate the error at all grid points and possibly create new fine grids or remove those no longer needed. If a new fine grid is created, its initial values are interpolated using the finest grids from the already existing grid structure. No thing need be done to remove a grid that is no longer needed except reclaim its storage space. .

We first discuss how of ten the error estimation should be done, and then the procedure we use to do it. In hyperbolic problems, one

can estimate the speed of propagation and calculate how fast some phenomenon needing to be in a fine grid will move. If we add a buffer zone around the fine grid we can lengthen the time interval over which grids are appropriate, and thus lengthen the interval between the regridding operations. The larger this buffer zone the less often we have to regrid, but the more work it is to integrate the extra points in the buffer zone.

Typical calculations give an optimal regridding frequency of approximately every 3-4 steps. If there are many levels of refinement, we apply this result at each level. The finest 'grids must be moved more often than the coarsest grids. We call the same regridding procedure with a base level which is finer than the coarsest level. The base grids stay fixed, and finer subgrids will be created within the boundaries of the grids at the base level according to the proper nesting restrictions of section 2.1. The potential problem here is that a refined grid might move off its base or not stay sufficiently far away from the base grid boundaries. If this happens, it is probably time to move the base grids as well.

Finally, we use estimates of the local truncation error to decide where to refine the grid. There are two reasons for this. First, we were motivated by the convergence results of (Gustafsson, [1975]) for initial boundary value problems for hyperbolic systems. Under some assumptions that are mostly about the Cauchy stability of the difference approximations and stability in the sense of Kreiss for the initial boundary value problem, Gustafsson shows that if

(i) the local truncation error $\simeq kh^m d_v(t)$

(ii) the initial error of the approximation $\simeq h^{\beta} e_{\nu}(\sigma \Delta t)$, $\sigma = 0,1,\ldots,s$

(iii) the error for the boundary approximation $\simeq h^{\beta} f_{\nu}(t)$

where d, e, f are bounded functions, and if $\beta \geq m$, then the convergence rate is order m. Mesh refinement can control these errors by decreasing k and h in those regions where $|h^{m} d_{\nu}(t)|$ or $|h^{\beta} f_{\nu}(t)|$ are too large.

Gustafsson's results are for nonadaptive, uniform grid calculations. More recently, Oliger [1982] has a convergence result for adaptive mesh refinement under some stronger hypotheses than those of Gustafsson. Experimentally, the expected rate of convergence is observed along with the expected decrease in constants when the local truncation error tolerance (that is, the refinement criteria) is reduced.

To estimate the error, we use a method based on Richardson extrapolation. For simplicity, let Q be a two-level explicit difference operator. If the solution is smooth enough, the local truncation error is

$$u(x,t+k) - Qu(x,t) = k\left(k^{q_1} a(x,t) + h^{q_2} b(x,t)\right) + k\, O\left(k^{q_1+1} + h^{q_2+1}\right) \tag{3.1}$$

$$= \tau + k\, O\left(k^{q_1+1} + h^{q_2+1}\right),$$

where we denote the leading term by τ. If u is smooth enough, then if we take time two steps with the method Q, to leading order the error is 2τ,

$$u(x,t+2k) - Q^2 u(x,t) = 2\tau + k\, O\left(k^{q_1+1} + k^{q_2+1}\right).$$

Let Q_{2h} be the same difference method as Q but based on mesh widths of 2h and 2k. Also, assume the order of accuracy in time and space are equal. $q_1 = q_2$. Then

$$u(x,t+2k) - Q_{2h}u(x,t) = (2k)\left((2k)q\, a(x,t) + (2h)^q\, b(x,t)\right) + O(h^{q+2})$$

$$= 2^{q+1}\,\tau + O(h^{q+2}) .$$

Since $u(x+2k) - Q^2u(x,t) \simeq 2\tau$, by forming the difference

$$\frac{Q^2u(x,t)-Q_{2h}u(x,t)}{2^{q+1}-2} = \tau + O(h^{q+2}) , \qquad (3.2)$$

we get an estimate of the local truncation error at time t. In words, we take one giant step based on mesh widths of 2h and 2k using the solution at time t, and compare it with the solution obtained by taking two regular integration steps to obtain the pointwise error estimate. This is illustrated schematically in Figure 3.2.

This procedure has several points to recommend it. First, it is not necessary to know the exact form of the truncation error to apply it. The functions a(x,t) and b(x,t) in (3.1), which involve higher derivatives of the solution, need not be known explicitly. Especially for systems of equations in several variables, it can be quite difficult to compute the exact form of the error. Not only is our estimator independent of the pde, the error estimation procedure is independent of the difference method. This is important if mesh refinement is going to be generally applicable to a wide variety of problems without an inordinate amount of programming. The exact same

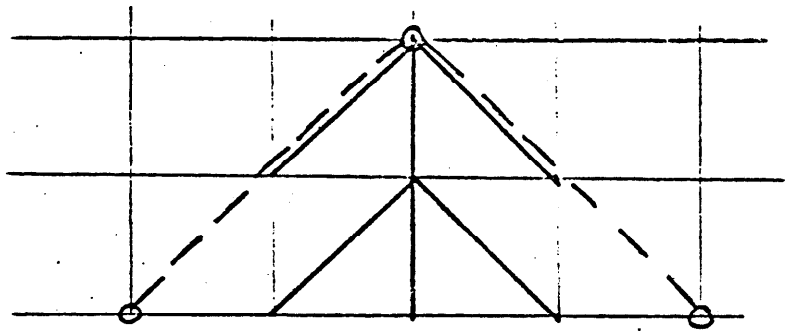


Figure 3.2. Richardson Error Estimation Procedure

user-supplied method of integration can be used for the error estimation. The restriction of this procedure, that the accuracy in time and space be the same, is not a severe one. Many popular finite difference methods fall into this category, for example, second order methods like Lax-Wendroff or MacCormack's method, and Leap Frog, and first order methods such as upstream differencing. For methods where the accuracy in space and time is not the same, a more expensive variant of this procedure is possible. For example, one could estimate the spatial and temporal error separately, by first keeping k constant and taking a step based on 2h differences, then keep h constant and take a step with time step 2k. Other variations are possible too.

Finally, we mention that for nonsmooth solutions we no longer have an accurate error estimate. However, the Richardson estimates still provide a good criterion for refinement since near a singularity the estimate will probably be large. For piecewise constant initial

conditions, the Richardson algorithm gives an estimate proportional to the jump in the solution.

The last major component of the mesh refinement algorithm concerns the interaction between the component grids. There are three tasks which fit under this heading. The first of these deals with boundaries. Refined grids have boundaries which are in the interior of the problem domain, and so they will need boundary values not supplied with the pde. These values can be calculated in three ways. First, if a boundary point is in the interior of a different fine grid, we can get the boundary value at the next time step from the interior integration of the intersecting grid. If there are no intersecting fine grids, the boundary values are calculated using values from underlying *coarse* grids. We use either the Coarse Mesh Approximation Method (Ciment, [1971]) or interpolation from a coarser grid to get the boundary values. In Berger [1982] we prove that if we use Lax Wendroff as the interior difference scheme with either of these interface equations, mesh refinement in time and space by any integer is stable in the sense of Kreiss. In addition, stable boundary conditions have been derived which maintain the conservation form of the difference scheme at the interface between the fine and coarse grid. These will also be reported elsewhere.

The second item of intergrid communication is updating. If a fine grid is nested in a coarse grid, then when they are integrated to the same point in time the coarse grid values are updated by injecting the fine grid solution values onto the coarse grid points. If grid points do not match up at regular intervals, interpolation is used to find the value from the fine grid which replaces the coarse grid value. For

explicit methods, it is only necessary to update as many coarse grid points at the perimeter of the fine grid as the number of points in the stencil for the coarse grid integrator. For implicit methods, it is necessary to update the entire coarse grid, and in fact, some sort of averaging might be desirable before injection onto the coarse grid. This is similar to the residual weighting used in multigrid methods (Brandt, [1977]).

Updating is necessary for two reasons. If the coarse grid is not updated, the solution can become so dispersed and dissipated that after a while it will look as if refinement is no longer necessary. Second, this prevents a train of bad values on the coarse grid from spreading into the buffer zone and contaminating the values that will be used for the boundary approvima tion for the fine grid.

The last grid communication task is that of averaging. This only arises when two subgrids at the same level of refinement overlap. In general, the region of overlap is at most a few coarse mesh widths wide. Still, the question arises when updating the underlying coarse grid, which fine grid should inject the solution values. The solution on either fine grid is as accurate as the other. Since they are only overlapping by O(h) width, and they are coupled through the boundary values, the solutions do not diverge from each other. However, if one is not careful about injecting onto the coarse grid, thee can be a jump in the solution representation on the coarse grid. So far, this has only been important for graphical output.

The overall mesh refinement algorithm is presented in Figure 3.3 in outline form. It can be written quite simply as a recursive

procedure. Of course, in writing the mesh refinement program in Fortran, we have converted it to an iterative procedure.

```
Recursive Procedure Integrate (level)
    Repeat (h_f/h_c)^level times
        Regridding time? -- error estimation for grids at level level and finer
        Step Δt_level on all grids at level (level)
        If (level + 1 exists)
        Then Begin
            Integrate (level + 1)
            Update (level, level + 1)
            End
    end
level = 0 (*coarsest grid level*)
Integrate (level).
```

Figure 3.3. Coarse Grid Integration Cycle

4. CLUSTERING AND GRID GENERATION

Much of the success of this adaptive mesh refinement algorithm lies in the generation of efficient subgrids. The idea is to estimate the error at all grid points in level ℓ grids, and flag those points where the error (or some other measure for determining the need for refinement) exceeds a tolerance ε. The grid generation procedure creates a new level of grids with mesh width $h_{\ell+1}$ so that every fiagged point is in the interior of a fine grid. The cost of deciding where fine grids are needed, and generating them, is small since it is proportional to the number of coarse grid points. The most expensive cost is the cost per step of integrating the fine grids, which is proportional to their area. Thus we seek to minimize the total area of these refined grids. In addition, we want to create grids whose coordinate lines are approximately aligned with the solution.

More precisely, when it is time to regrid, a new grid level may be created, an existing level recreated, or no longer necessary existing levels removed. Even if a fine grid should simply be translated in some direction we use the more general approach of creating a new grid, and initializing it with solution values taken from the old refinement before it is deleted.

The outline of the regridding algorithm is as follows. Suppose the current grid structure G has ℓ levels. Based on the error estimates of level ℓ, we might create new grids at level $\ell+1$. Next, based on estimates from the (larger) level R-l grids, we recreate a new level ℓ, making sure it includes the new level $\ell+1$. Continuing, the error estimates on level $\ell-2$ are used to generate level $\ell-1$, making sure level& is properly nested, and so on to the coarsest level. It

is important to work from the finest to the coarsest levels, even though this entails the extra work of ensuring proper level nesting. This way, grids are generated using the most accurate error estimates taken from the finest grid at any given point.

Thus, for each existing level of grids, we apply the same procedure to generate the next finer level. This regridding procedure consists of four steps:

(1) flag points needing refinement

(2) cluster the flagged points

(3) generate a grid for each cluster

(4) evaluate, possibly repeat.

Steps (2) and (3) are the difficult steps.

The first step in the algorithm is to identify those grid points at level ℓ which need to be in a finer grid at level $\ell+1$. In section 3

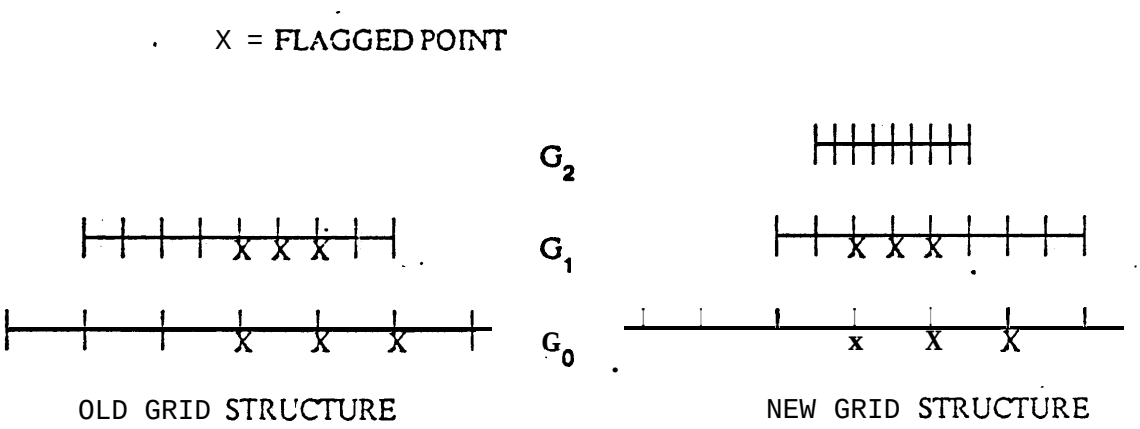


Figure 4.1. 1D Regridding Algorithm

we discussed the use of local truncation error estimates in deciding where these refined meshes are needed. Using the procedure described there, we estimate the error at all grid points at level ℓ, flagging those points $\underset{\sim}{x}$ where $e(\underset{\sim}{x}) > \varepsilon$. At this step we also flag grid points in level ℓ grids which are interior to grids at level $\ell+2$, even if $e(\underset{\sim}{x}) < \varepsilon$ based on the level ℓ grid. Since each flagged point at level ℓ will be in a level $\ell+1$ grid, proper level-nesting is assured.

The second step of the algorithm is the separation of flagged points into distinct clusters. In step three, each cluster will be fit with a fine grid containing all the flagged points of the cluster. We describe the procedure for the one dimensional case here separately. Since in one dimension a grid is just an interval, clustering is trivial and can be done concurrently with grid generation. The leftmost and rightmost flagged points of the coarser grid form the left and right boundary of the new subgrid. The cluster in this case would consist of all flagged points between and including the leftmost and rightmost flagged points. Possibly, if a long enough gap of unflagged points is found, two or more separate subgrids may be formed instead. The exact definition of long enough depends on the size of the buffer zone. After a grid is created, it will be enlarged to include a safety zone around all its flagged points. Recall from section 3 that this buffer zone determines how often grids must be examined versus how large they are. The size of this buffer zone is what determines how large the intergrid spacing should be. Flagged points which are closer together than twice the size of the buffer zone should be in the same grid refinement.

Figure 4.1 illustrates the regridding procedure in one dimension.

On the left is the grid structure before regridding, and on the right is the new grid structure. The x's are the grid points which have been flagged with high error estimates. The grid structure is illustrated schematically by drawing each grid separately, instead of superimposing them. We have used a buffer width of one coarse grid point in this illustration.

2D Grid Generation

The clustering algorithm serves two purposes: one is to separate spatially distinct phenomena so that different features of the solution will be in separate grids. The second purpose is to subdivide points when one rather large region should be fit with several grids. This situation is illustrated in Figure 4.2. If the entire front (represented by the darkened line) were fit with one large grid, it

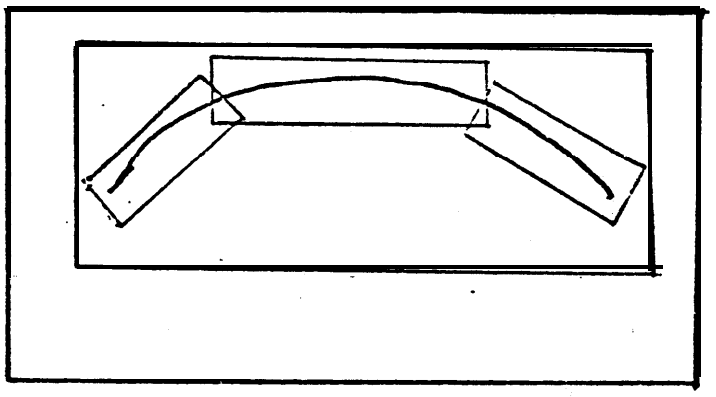


Figure 4.2. Multiple Grids - One Front

wouldhave an unacceptably large area of refinement. If we had some information about the directional layout of the points, a smart subdivision of the points could be made.

It is very tricky to find a general clustering algorithm that is almost foolproof and is not very expensive. Our approach is to start with a simple algorithm which works in the easy cases, and try to detect when it does a bad job of clustering. In these cases we use a more expensive algorithm, and try to tailor it to the special cases when the first approach fails. We are lucky to be able to draw on the large literature in pattern recognition and Artificial Intelligence (see, for example, Duda and Hart, [1973]; Hartigan, [1973]). There are algorithms for feature extraction or edge detection as well as more general clustering algorithms With goals similar to ours. We report here on the simplest clustering algorithm in two dimensions, and refer the reader to Berger [1982] for a detailed discussion of alternatives. However, this is still an open problem where more research is needed.

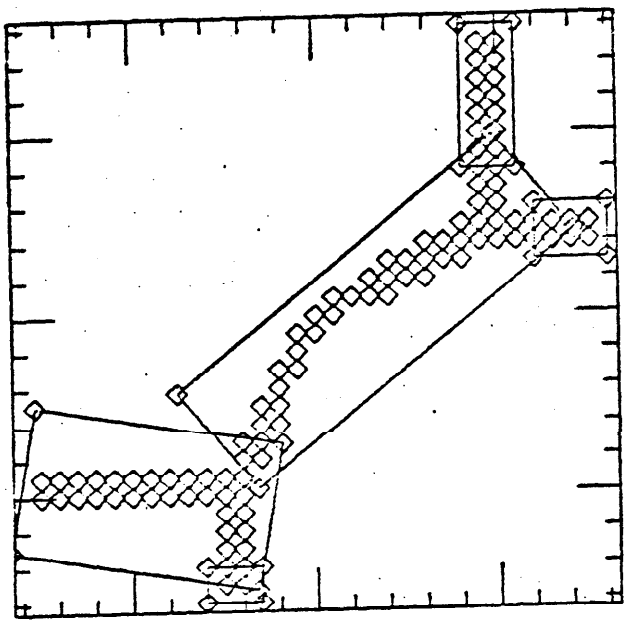
The first approach we use to cluster points is the nearest neighbor algorithm. The nearest neighbor algorithm forms clusters which are distinguished by having interpoint distances for points in the same cluster smaller than the intercluster distances. We start with one point forming a new cluster. Successive points are included in this cluster if the distance from the point to the cluster is less than some specified tolerance, whch we usually take to be two mesh widths.

The nearest neighbor algorithm is very successful in accomplishing the first goal of clusteing, but fails in the second. In these cases we use special data structures, such as minimal spanning trees and

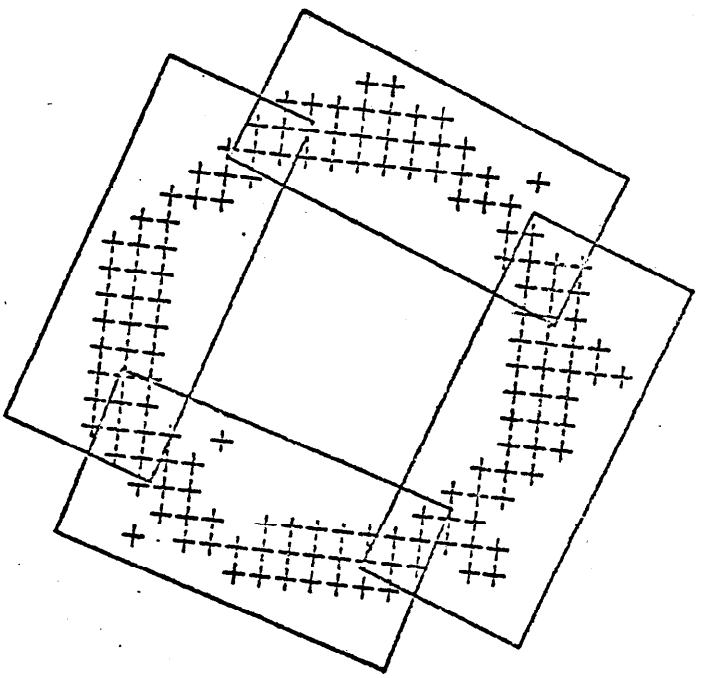
relative neighbor graphs, to organize and process the points into separate clusters. These data structures possess certain properties that should also hold for points in the same cluster. For example, two points in the relative neighbor graph are connected if no other point is closer to both of them. Once the points are connected to each other in an organized way, we use an iterative method of grid generation. Starting with a core group of points, the algorithm proceeds by merging the points connected to the core cluster through the data structure and immediately generating the fine grid to the new cluster, until no more merges can be successfully done. A merge is considered successful if it has an acceptable efficiency evaluation. This is step 4 in the grid generation algorithm.

Our practical criterion for measuring the efficiency of a new grid uses the fraction of the area of the rectangle which is unnecessarily refined. This can be estimated quickly by taking the ratio of flagged points to the total number of coarse grid points in the new fine grid. If this ratio falls below a cutoff tolerance, typically between 1/2 and 3/4, the merge is rejected, and the previous cluster remains. The pictures in Figure 4.3 illustrate the different subgrids that are formed using the efficiency parameter indicated.

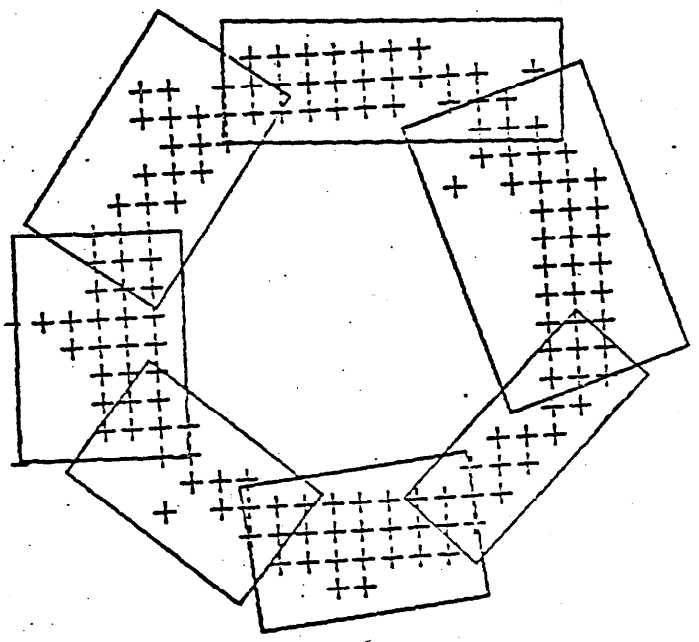
A last important observation is that once we have good clusters they do not change very fast. If at some initial time an expensive clustering algorithm is used, the same clusters can continue to be used for many time steps. Flagged pints can be grouped in the same clusters they were grouped in at the previous step, and only the orientation of a new refined grid need be calculated. If grid points are flagged on a



(a) eff. = .45



(b) eff. = .50



(c) eff. = .75

Figure 4.3. Efficiency of Subgrids

section of a coarse grid not previously refined, the flagged points should be added to the nearest cluster.

Given a cluster of flagged points, the next step is to generate a rectangular grid so that grid lines are in some sense aligned with the points. The rectangle should have as small area as possible to minimize the work of integrating that grid in time. The algorithm we use for this first determines the orientation of the rectangle, and then the length of the sides needed to enclose the points. For simplicity we present it in two space dimensions, but it generalizes immediately to higher dimensions.

Let A be the n-by-2 matrix of the coordinates of the n flagged points, and A_m the matrix of the same dimension with the x and y coordinates of the mean of the points, (x_m, y_m). The matrix $M^tM = (A-A_m)^t(A-A_m)$, where

$$A = \begin{pmatrix} x_1 & y_1 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ x_n & yn, \end{pmatrix} \qquad A_m = \begin{pmatrix} x_m & y_m \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ x_m & ym \end{pmatrix}$$

and

$$M^tM = \begin{pmatrix} \Sigma_i\, x_i^2 - x_m^2 & \Sigma_i\, x_i y_i - x_m y_m \\ \Sigma_i\, x_i y_i - x_m y_m & \Sigma_i\, y_i^2 - y_m^2 \end{pmatrix}$$

contains the second rime nt s of the points about their mean. This matrix M^tM determines an ellipse with the same first and second moments as the original set of points (Cramer, [1951]), and so provides a good

approximation of the layout of the points. Since M^tM is real and symmetric, it has two real eigenvectors. These eigenvectors are the major and minor axes of the ellipse. We use these eigenvectors to determine the orientation of the rectangular subgrid. This algorithm is invariant under translations and rotations of the points, and is extremely simple. It is easy to find the eigenvectors since M^tM is a 2-by-2 matrix. Furthermore, if clustering is done concurrently with grid generation, the first and second moments can be updated instead of recalculating them for every additional point. A similar technique of clustering and fitting ellipsoids has been used by (Gennery, [1979]) for stereo vision processing. The goal of his work is obstacle avoidance for exploring vehicles, for example, the Viking Lander's exploration of Mars.

Once the grid orientation has been determined, the dimensions of the rectangle are calculated to include all points in the cluster. This is the expensive part of the algorithm. We take the dot product of a point with the orientation vectors for every point in the cluster. (Some of this work can be avoided if the convex hull of the set of flagged point is kept. For iterative algorithms, there are also ways to update the convex hull of a set of points for the addition of new points.) Once the dimensions of-the rectangle are calculated, a buffer zone of predetermined size is added around the rectangle perimeter to complete the new subgrid.

Two additional points will complete the discussion of the regridding algorithm. As outlined above, this algorithm creates one new level of refinement for each invocation. For time dependent boundary conditions, if no assumptions are made on the smoothness of

the boundary data, an incoming wave might need several new levels of refinement for it to be well resolved. To handle this case, the error estimation and grid generation procedure can be re-applied to a newly created fine grid at the boundary to see if even finer new grids are needed.

Finally, we mention that the initial grid creation at time $t = 0$ employs a slightly different strategy than the regridding procedure used at later times. Only at this time can we take advantage of the initial conditions specified with the problem. For example, when a level 1 refinement is created, it is initialized using the initial conditions rather than interpolation. The error estimation and regridding procedure can then be applied again on the level 1 grids to see if even finer subgrids are needed, and so on, until a pointwise error estimate $e(\underset{\sim}{x}) < \varepsilon$ holds at all points.

5. DATA STRUCTURES

The data structures in our adaptive mesh refinement strategy turn out to be surprisingly simple, although crucial to the feasibility of the entire approach. A structure is needed which keeps track of the relationships between the grids, as well as the solution storage for each grid.

We will first describe the data structure we use in one dimension. A generalization of this data structure is what we use in two or more dimensions.

Recall fran section 2 the nesting requirement for one dimensional mesh refinement: each fine grid must be entirely contained in a coarser grid at the next level. We use this to define a tree data structure, *where each node represents a grid, and make a correspondence between a parent (of a) node, and a coarser, parent grid. Subgrids are considered the offspring of their parent grid. Siblings are subgrids within the same coarser grid. If fine grids are at the same level of refinement with different parents, we call them neighbors. Technically, we use an ordered tree data structure where each node can have multiple descendents. In this representation, we see that a node will have multiple descendents if the coarse grid corresponding to that node has several fine grids contained in it. In this one dimensional case, it is also possible to order the nodes using the coordinate value of the left-most grid point in the associated grid. Figure 5.1 shows a grid structure and its related tree structure.

All the grid-to-grid operations, such as fine grids updating coarse grids and setting internal boundary values for fine grids, have an information flow which follows path links in the tree. The only

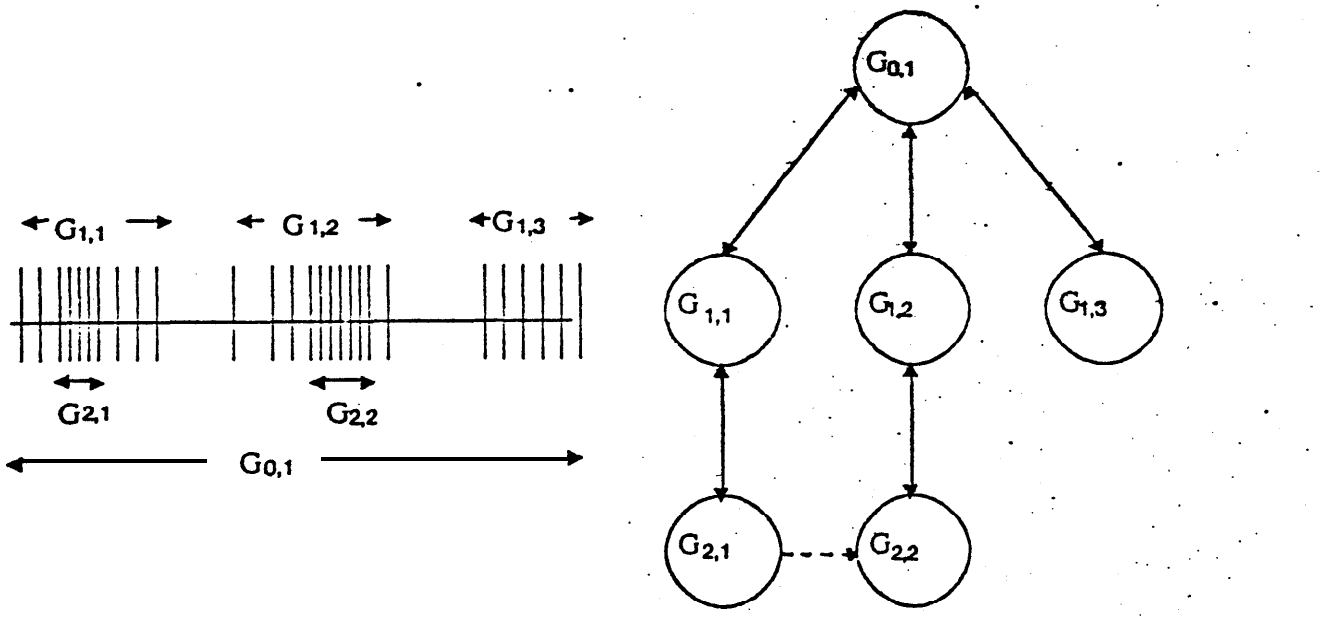


Figure 5.1. 1D Data Structure

nonstandard link in the tree is for the neighbor pointer we described above. This thread is indicated by the dashed line in Figure 5.1. This additional link makes it easy to implement the operation of taking one integration step on all grids which are at the same level of refinement.

Because this tree structure can grow or shrink dynamically, some form of dynamic storage allocation is needed, both for the grid information in each node, and the solution storage for each grid. Since Fortran does not provide such a facility, the storage management must be explicitly provided by the mesh refinement program. We keep a linked list of free nodes which are assigned to newly created grids and

reclaimed when a grid is removed. Sometimes when the regridding procedure is called, it is to move grids on only the finest levels. The coarser level grids stay fixed. In this case, the top half of the tree will remain as is. A new bottom half will be generated, and the two halves connected. Although the number of nodes in the tree vary, we would like each node to contain a fixed amount of information to represent each grid. Since the number of offspring of each node is variable, the tree is implemented with each node having an offspring pointer only for the first descendent, with one sibling pointer per node to connect the rest of the subgrid nodes (Knuth, [1968]).

A grid then is characterized by the following pieces of information stored in each node of the tree.

1) Grid location

2) Number of grid points

3) Level in tree

4) Offspring pointer

5) Sibling pointer

6) Parent pointer

7) Pointer to the next grid at the same level

8) Time to which this grid has been integrated

9) Index into main storage array where approximate solution values are stored.

The same information characterizes two dimensional grids as well. However, the two dimensional version of this data structure is more intricate than the one dimensional version since in two dimensions a grid can be partially nested in more than one coarser grid. An additional complication is that grids at the same level of refinement

can intersect. Finally, in two dimensions there is the possibility of having several base grids $G_{0,j}$ in an initial domain specification. We have generalized the ne dimensional data structure to account for these differences in the following way. We start with an n-tuply rooted tree, where each of the n root nodes correspond to a canponent grid in the coarsest mesh. Technically, a tree with more than one root is called a forest, which is simply a collection of trees. Next, since a subgrid can have many possible parent grids, we replace this single slot of information in each node with a pointer to a short linked list of parent grids. Lastly, we add to the information for each grid a pointer to another linked list of intersecting grids at the same level of refinement. Schematically this data structure is illustrated in Figure 5.2.

The final data structure used in our mesh refinement program manages the large array which is the storage area for the solution values on all grids. For problems in p space dimensions, we use a p-dimensional array. For vector rather than scalar problems, we use a p+1 dimensional array where the extra dimension is the number of variables in the problem. This storage area is managed as a linked list of used and available blocks of storage. When a grid is created, contiguous storage space is reserved from the sorted list of free blocks using a first-fit algorithm (Knuth, [1968]). In this algorithm, the list of free blocks of storage is scanned until a large enough block is found. The requested space is allocated, and the unused portion returned to the list. Reclaimed space, which occurs when a grid is no longer necessary, is inserted back into the linked list of free space. For quick memory access, space is never allocated in a

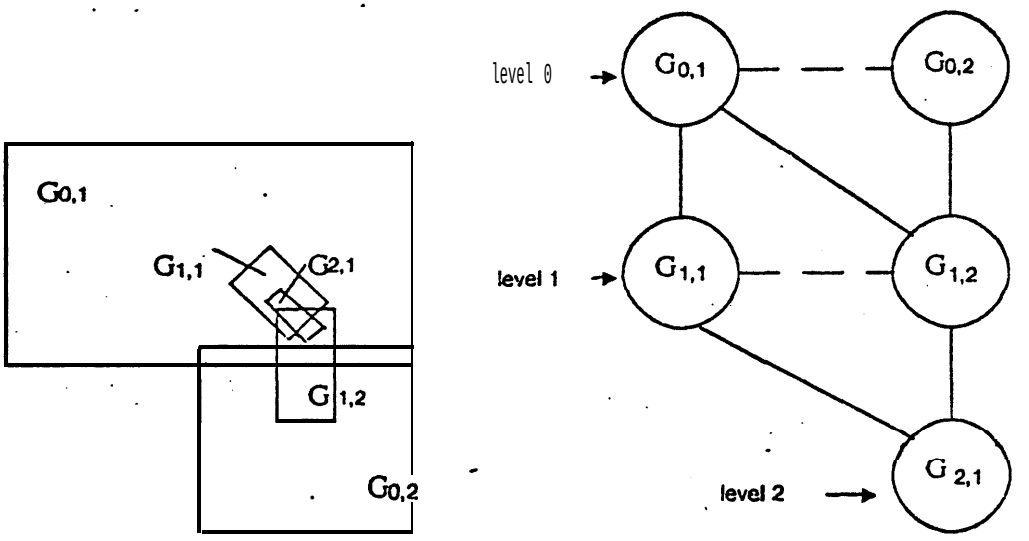


Figure 5.2. 2D Data Structure

circular fashion across the array boundary: all memory allocation must be contiguous. A compaction, or garbage collection, routine could be included to provide for case where there is enough total but noncontiguous space available in this array to satisfy a storage request. However, as Knuth [1968] reports, if storage is too fragmented to service a request, canpaction usually adds only a few more transactions before space is exhausted. A routine which would allow the user to restart with a larger memory area would be more useful.

6. COMPUTATIONAL EXAMPLES

In this section we will present some numerical experiments in one and two dimensions to illustrate how our adaptive mesh refinement algorithm works. Our procedure for comparing results is the following. We solve each problem with mesh refinement with a specified coarse grid, a maximum number of levels of refinement, and an error tolerance which is the criterion we use in deciding to refine a grid. We compare the solution to that obtained on a uniform grid with first the coarsest, then the finest (if possible) mesh spacing used in the mesh refinement calculation. We measure the computer time without I/O costs (when possible) and the error in the solution. In all these cases, we compute the 2 norm of the error at only the coarse grid points. In one dimension, this means we compute

$$\|\text{error}\|_{2_c} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} \text{error}(x = ih_c)^2}\ ,$$

and similarly in two dimensions. For the fine grid this means we compute the error only at every h_c/h_f grid points. The one dimensional examples were run on an IBM 370/168 using the Fortran H Extended compiler, optimization level 3. The two dimensional examples were run using the same compiler on an IBM 370/3081.

Example 1. Shock Tube Problem

In this example, we compute the solution to the shock tube problem in one space dimension. This Riemann problem is taken from Sod [1978] where it was used to compare a number of methods for solving gas dynamics problems. The initial conditions are chosen so that the

solution contains a shock, contact discontinuity, and a rarefaction wave. The problem is:

$$u_t = f(u)_x$$

for $0 \leq x \leq 1$, and $0 \leq t \leq 0.15$, where

$$u = \begin{pmatrix} \rho \\ \rho u \\ e \end{pmatrix}, \quad f = \begin{pmatrix} \rho u \\ \frac{m^2}{\rho} + p \\ \frac{m}{\rho}(e + p) \end{pmatrix}$$

Here, ρ is the density, e is energy per unit volume, u is velocity, and m is the momentum density pu. The equation of state is $p = (\gamma - 1)\rho\varepsilon$, where we take $\gamma = 1.4$ and ε is the internal energy per unit mass $\varepsilon = e - \frac{1}{2}\rho u^2$. The initial conditions we use are

$$u(x,0) = 0$$

$$p(x,0) = \begin{cases} 1.0, & \text{if } x < 0 \\ 0.1, & \text{if } x > 0 \end{cases}$$

$$\rho(x,0) = \begin{cases} 1.0, & \text{if } x < 0 \\ 0.125, & \text{if } x > 0 \end{cases}$$

The integration method we use is the two-step Lax-Wendroff scheme. The coarse grid step size is the same as in Sod [1978]. Reflecting boundary conditions are used. Hopscotch artificial viscosity is used to smooth the solution. The coefficient of the hopscotch artificial viscosity was the same for all but the uniform coarse grid run. For the coarse grid this led to too much smearing and so a smaller coefficient is used.

The parameters for the mesh refinement calculation are shown below.

buffer size	4
grids move every	5 steps
error tolerance	0.0005
refinement ratio	10
λ	0.1

In Figures 6.1 and 6.2 and Table 6.1 we show the results of our tests. Figure 6.1 shws the solution, and Figure 6.2 the plots of the errors. In this experiment, we have done two uniform grid runs with a mesh spacing of 0.01, and 0.001. We compare this to a two level mesh refinement run with a refinement ratio of 10, and a three level refinement run, which means the mesh spacing on the finest grid is 0.0001. During the calculation, the amount of the coarse grid which was refined varied from 20 % to 70 %.

The most interesting results here are that in less than one fourth the time, mesh refinement is able to calculate a solution which is as accurate as the uniform fine grid calculation. As we see in Figure 6.2, most of this error is due to the smearing of the corners of the rarefaction and contact discontinuity. To improve the calculation of the contact discontinuity, a better method than Lax-Wendrof f should be used.

Table 6.1 shows the computation time and the $\|\cdot\|_{2_c}$ errors for the

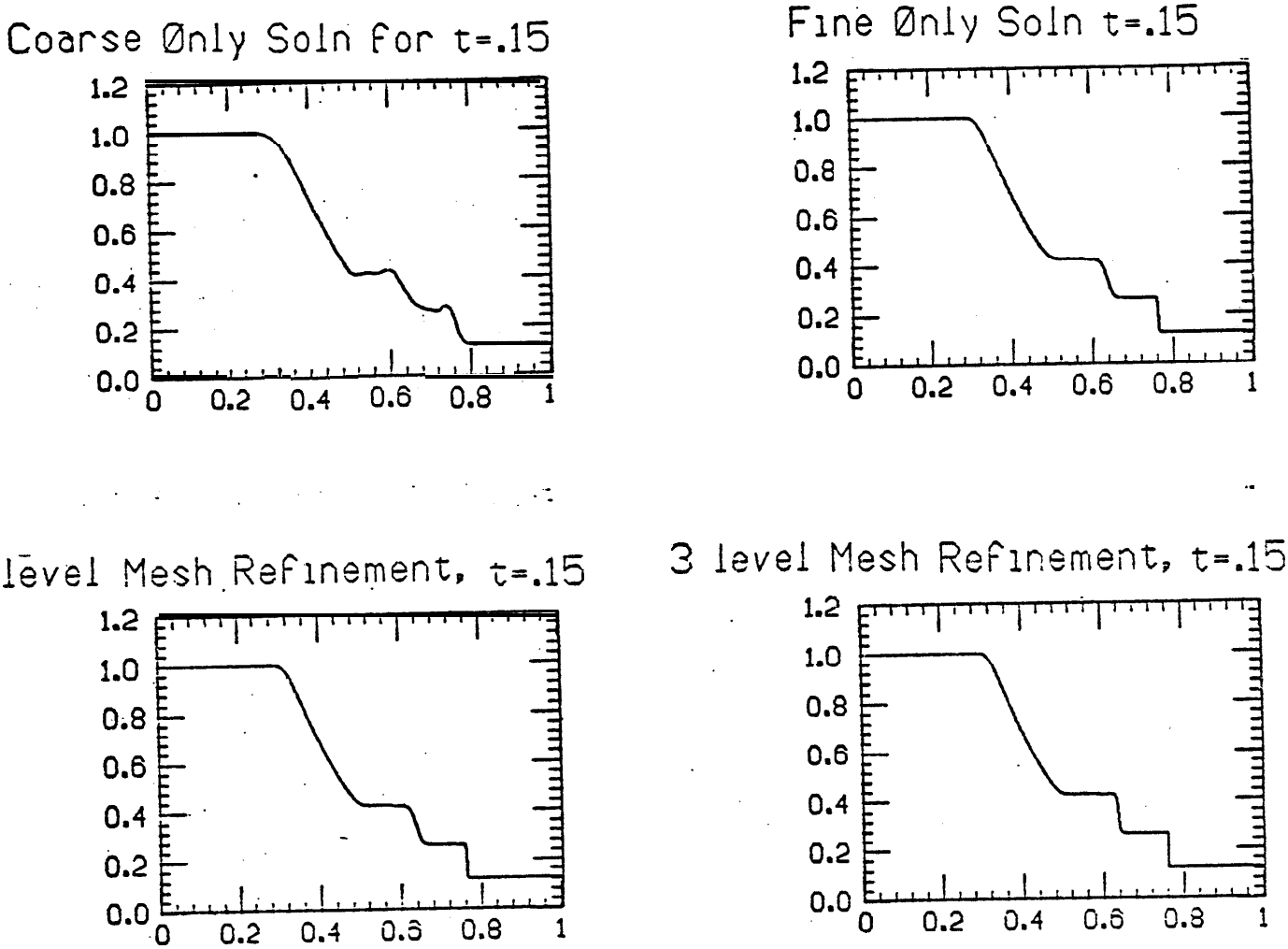


Figure 6.1. Solutions for Riemann Problem

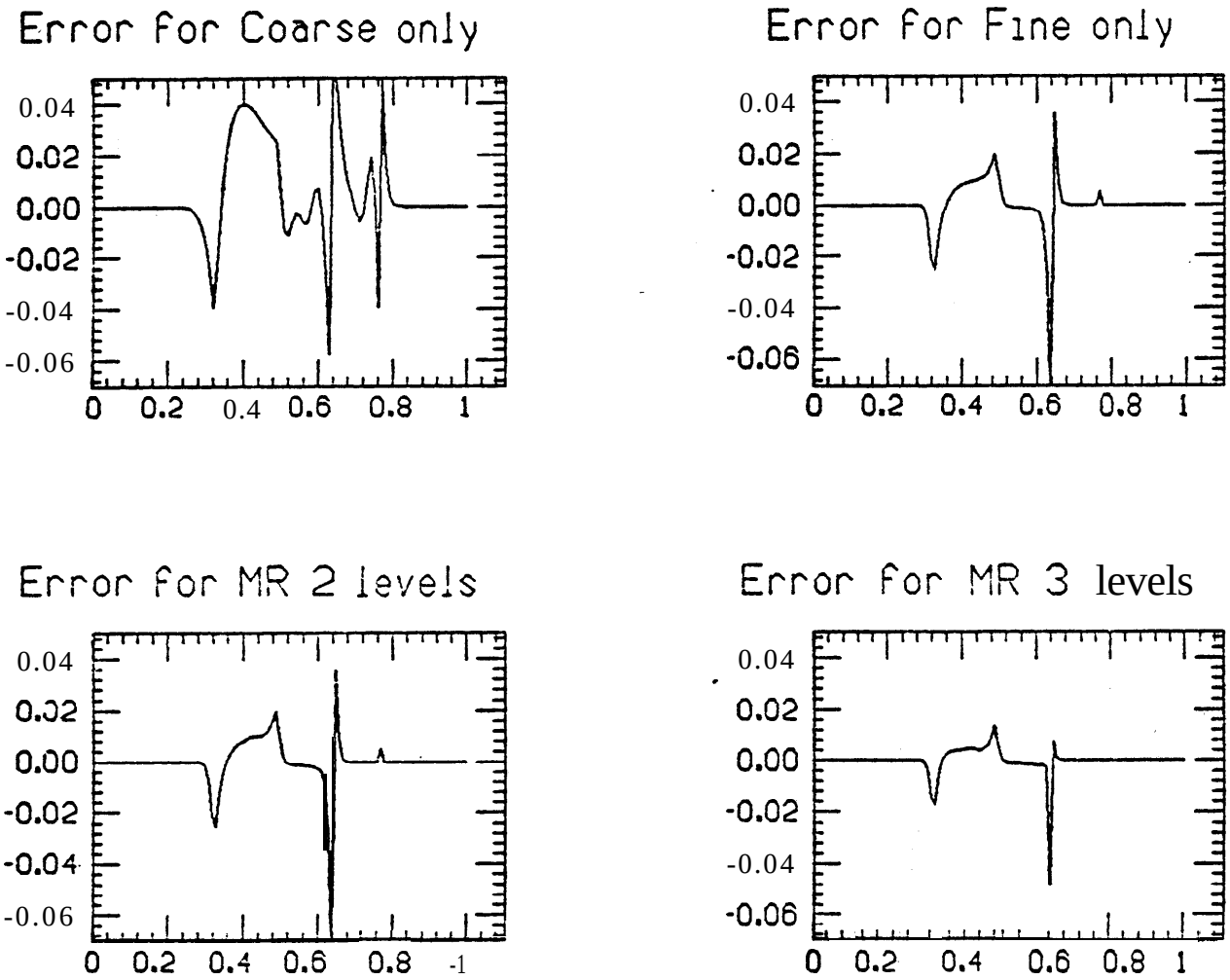


Figure 6.2. Errors for Riemann Problem

four computations. There are a large number of floating point underflows, each of which is handled very inefficiently as an interrupt to the processor. This skews the timings and so the correct asymptotic ratio of fine to coarse timings of 100 is not observed.

An important feature to note here is that while a refinement by h_c^2 (which is a factor of 100) is possible by using mesh refinement (given our computing resources), a computation with a uniform h_c^2 grid would not be possible (it would take about 347×10^2 seconds, or about 9 and a half hours).

method	h	$\|error\|_{2_c}$	time (secs)
coarse	0.01	2.14 E - 2	7.36
fine	0.001	1.15 E - 2	347
MR 2 lev	$h_f = 0.001$	1.15 E - 2	79.9
MR 3 lev	$h_f = 0.0001$	6.53 E - 3	591

Table 6.1. Results of Computations for 1-D Nonlinear Example.

Example 2. 2D Rotating Cone

The rotating cone problem has been used by Gottlieb and Orszag [1977] to compare numerical methods for convection problems. The problem is:

$$u_t = yu_x - xu_y$$

$$u(x,y,0) = \begin{cases} 0, & \text{if } (x - \frac{1}{2})^2 + 1.5\ y2 > \frac{1}{2} \\ 1 - 2\left((x - \frac{1}{2})^2 + 1.5\ y^2\right), & \text{if } (x - \frac{1}{2})^2 + 1.5\ y^2 < \frac{1}{2} \end{cases}$$

on a rectangular domain $-1 \leq x \leq 1$, $-1 \leq y \leq 1$, and $0 \leq t \leq 3.375$. The solution to this problem is a cone with elliptical base, which rotates counterclockwise about the origin. We integrate the solution until the cone is approximately halfway through the first revolution.

We use Lax-Wendroff as the integrator. The boundary conditions are zero inflow and first order extrapolation outflow. The parameters for the mesh refinement calculation are:

buffer size	2
grids move every	8 steps
error tolerance	0.001
refinement ratio	4
λ	0.25

In Figure 6.3 we show snapshot views of the location of the one subgrid in this problem at various intermediate time steps.

The results of this computation are what we would expect. The mesh refinement computation achieves about the same error as the uniform fine grid in about one sixth of the time. In this example, we can also get a rough estimate of the overhead of the method. A uniform fine grid run should asymptotically take 64 times the computer time for the coarse grid run. This is because the grid is refined by 4 in both coordinate directions, and there is a factor of 4 for refinement in time as well. In this rotating cone problem, roughly 12 % of the grid is refined during the computation. Adding 12 % of the cost of the fine grid run to the coarse grid time gives an estimated time of 78 seconds,

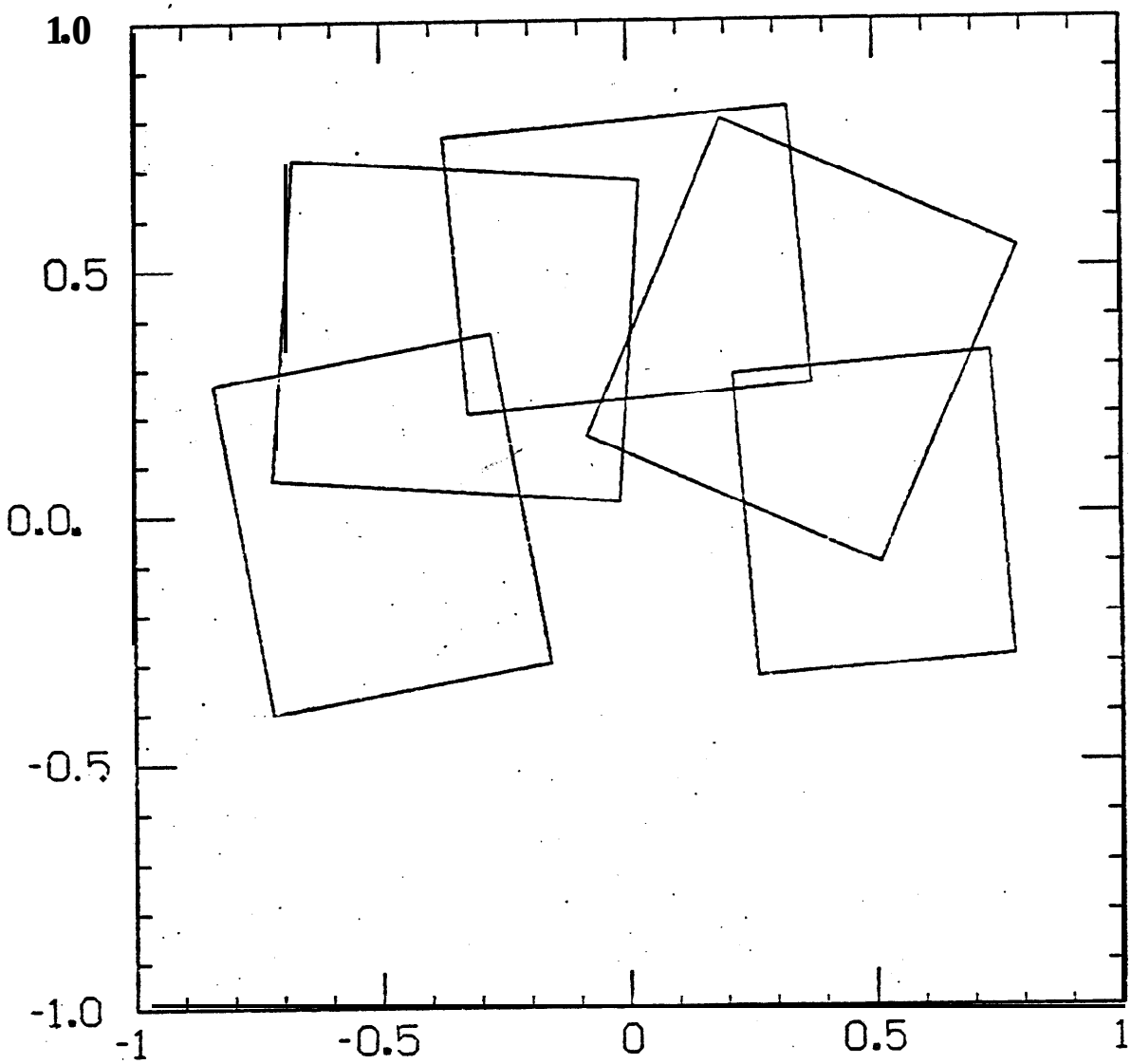


Figure 6.3. Subgrids for Rotating Cone Problem

and so 9 seconds of the total mesh refinement computation time are spent on other things bes ides integrating the grids. This is roughly 12 % of the computing time, most of which is present estimating the error and generating the subgrids. Yowever, the entire run costs only 16 % of the cost of a run on the uniform fine grid with the same accuracy.

Notice that in the mesh refinement computations, the wake behind the cone is greatly reduced over the wake in the coarse grid 'computations. This shows that the moving grid is correctly computing the solution and keeping the coarse grid computation under the cone from contaminating the solution. This also shows why the coarse grid must be updated from the fine grid.

me thod	h	$\|error\|_{2_c}$	time (secs)
coarse	1/20	5.27 E - 2	6.86
fine	1/80	9.34 E - 3	588.
MR	$h_f = 1/80$	9.78 E - 3	86.6

Table 6.2. Results for computation of the solution to 2-D linear example.

Example 3. 2-D Burgers Equation

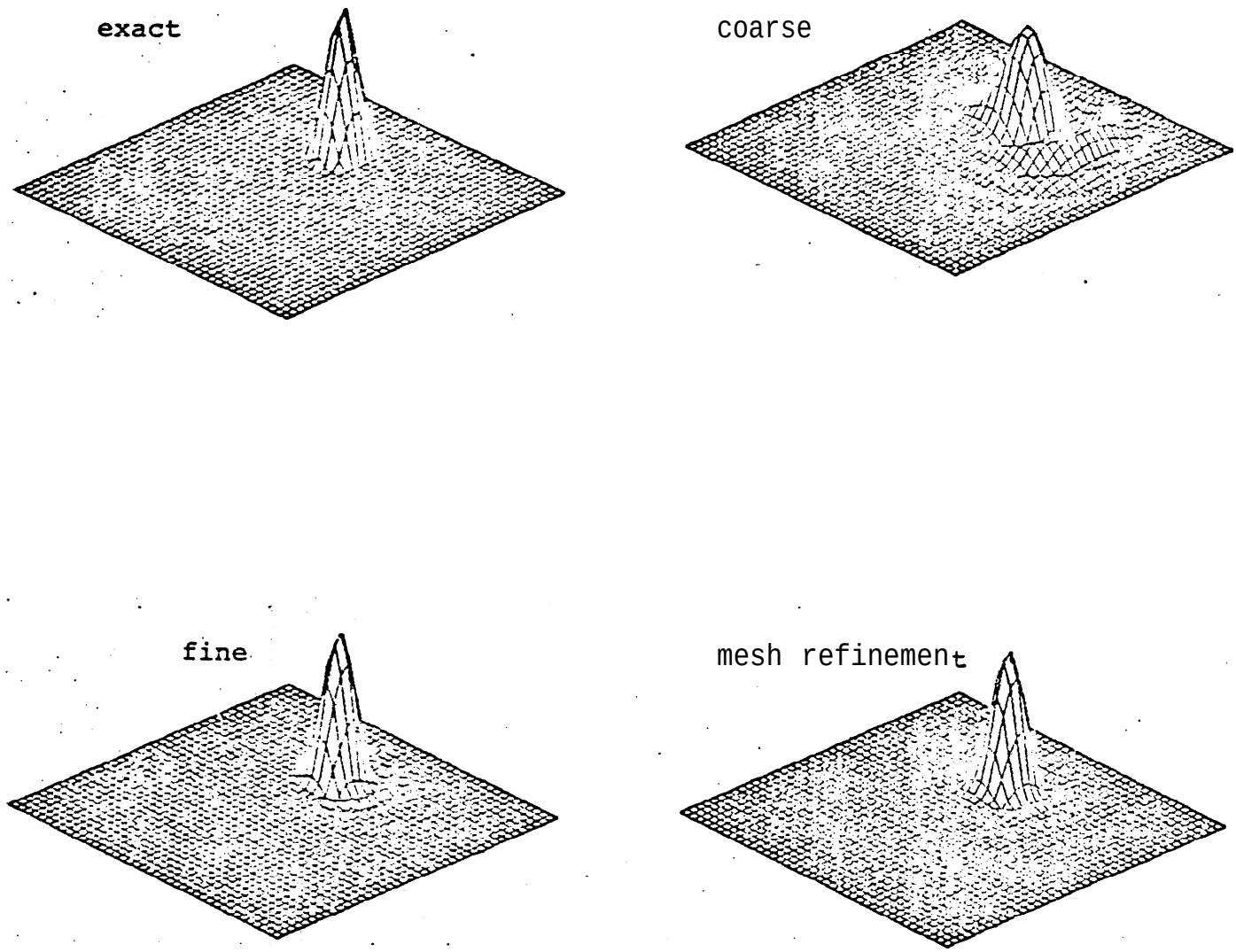


Figure 6.4. Solutions for Rotating Cone Problem

For a nonlinear example we have chosen a problem from Gropp [1980]. This problem contains complicated shock interactions, and is the most difficult example for the clustering and grid generation algorithms. The problem is:

$$u_t + uu_x + uu_y = 0$$

where

$$u(x,y,0) = \begin{cases} \frac{1}{2}, & \text{if } x < \frac{1}{2} \text{ and } y < \frac{1}{2} \\ -\frac{1}{2}, & \text{if } x > \frac{1}{2} \text{ and } y > \frac{1}{2} \\ \frac{1}{4}, & \text{otherwise} \end{cases}$$

on the unit square $0 \leq x \leq 1$, $0 \leq y \leq 1$. At the discontinuities $x = 1/2$ or $y = 1/2$, the initial data is taken to be the average of the values on either side.

We use MacCormack's method to integrate the problem. We specify the inflow boundary conditions to maintain the piecewise constant solution, and use first order extrapolation for the outflow boundaries. The parameters for the mesh refinement calculation are shown below:

buffer size	1
grids move every	4 steps
error tolerance	.001
refinement ratio	10
λ	0.5

In Figure 6.5 we show the solution, and in Figure 6.6 the error. The plots for this problem might be a little misleading. Because of the limitations in graphics packages, both the error and the solution

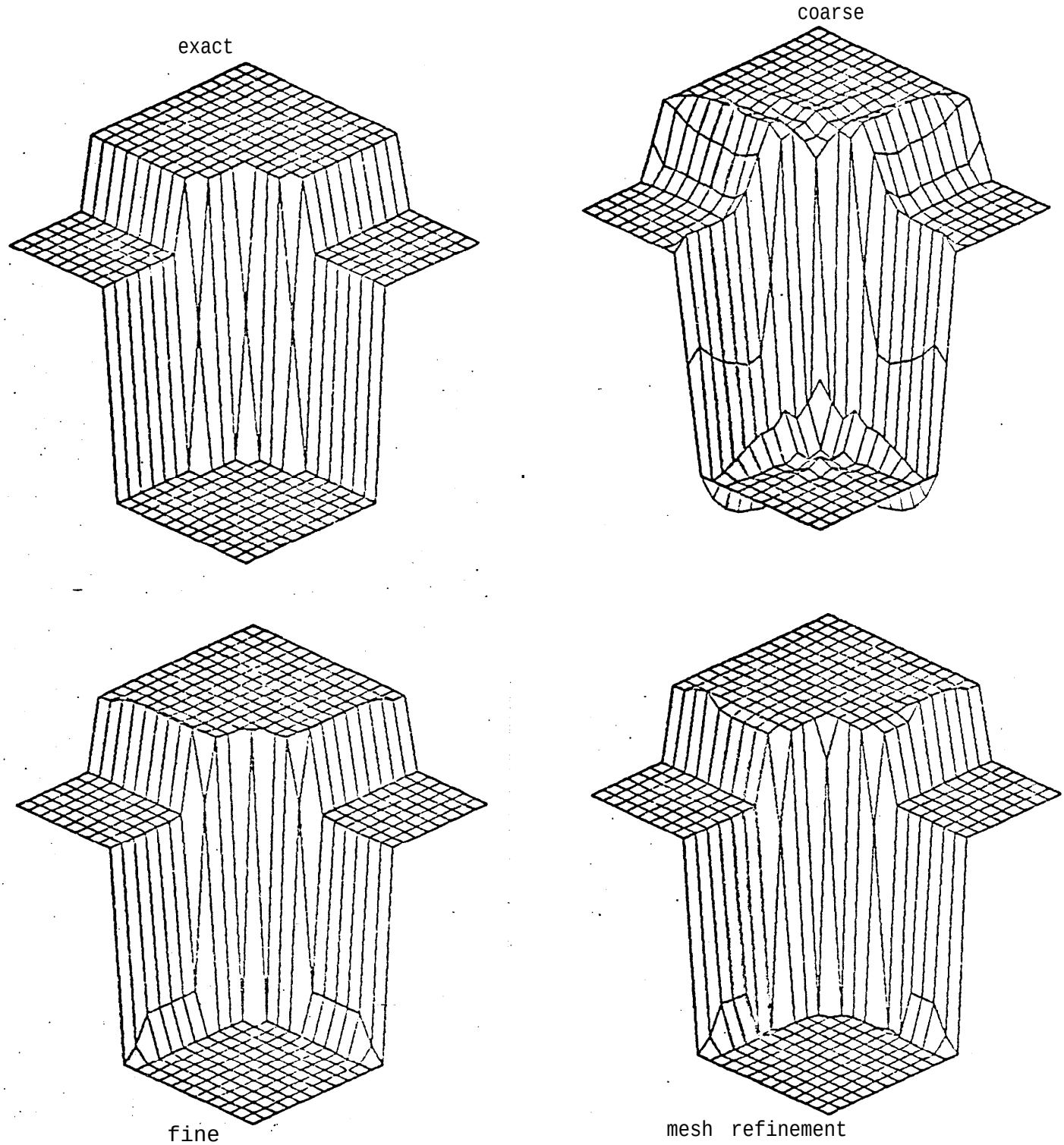


Figure 6.5. Solutions for Burgers Equation

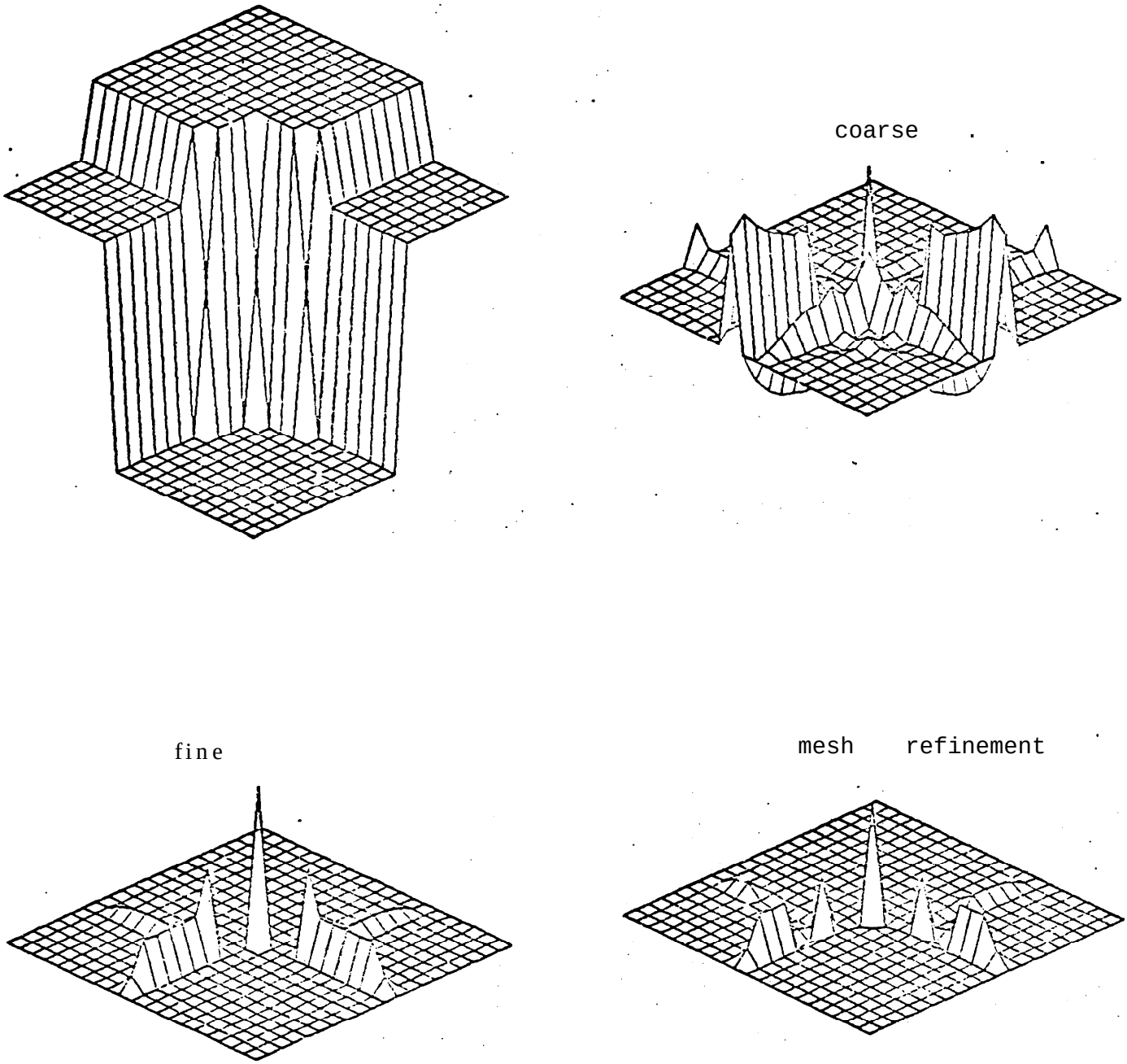
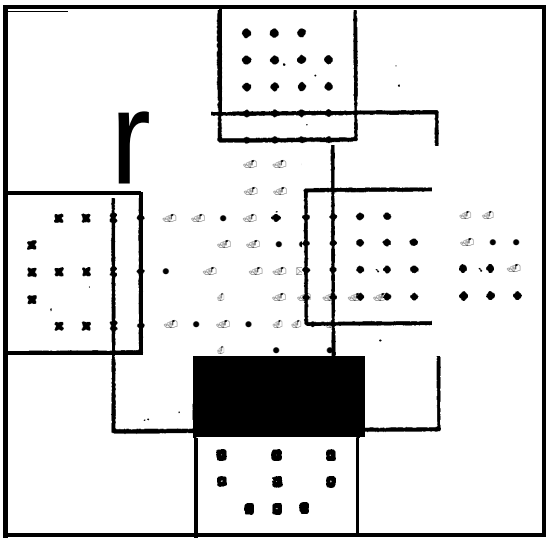


Figure 6.6. Errors for Burgers Equations

plots for all runs are done with the resolution of the coarser grid. For the runs with a finer grid, this means the solution is plotted only every 10 fine grid points. In a problem with discontinuities like this, the size of the overshoot is independent of the mesh width h (Hedstrom, [1976]). However, it appears as if the error on the fine grids (both the uniform and mesh refinement calculations) is smaller. This is because the overshoot in the solution is confined to the region close to the discontinuity, and 10 fine grid points away from the discontinuity the oscillation has decayed.

This problem is a hard test for mesh refinement because such a large fraction of the region is refined. However, even in this example, the mesh refinement calculation is faster than a uniform fine grid. In Figure 6.7, we show the subgrids the algorithm generates at the initial time t = 0, and at the final time when we output the solution. In this case, we have the slightly surprising result of the error for the mesh refinement run being slightly better than the uniform fine grid run. This is due to the grid rotation (Figure 6.7(b)). When the fine grid values are injected onto the coarse grid (both for updating and graphics), we use linear interpolation for the rotated grid, since the grid points do not match up. This has the effect of smoothing the solution in this region, which contains most of the discontinuities. This is why the error is less for the mesh refinement run.

TIME = 0.0



TIME = 0.500

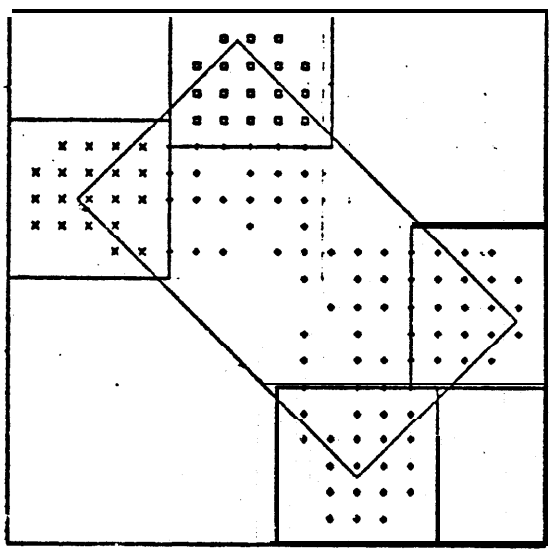


Figure 6.7. Subgrids for Burgers Equations

method	h	$\|error\|_{2_c}$	time secs
coarse	1/20	8.0 E - 2	0.18
fine	1/200	3.86E - 2	155
MR	h_f = 1/200	2.75E - 2	109

Table 6.3. Results of computations for 2-D nonlinear example.

7. CONCLUSIONS

We have presented an algorithm for the numerical solution of pdes using automa tic grid refinements. Several novel features make this algorithm possible. An automatic procedure which estimates the local truncation error determines the points to be included in finer subgrids. Our method of generating the subgrids is a key feature of the algorithm. We cluster the parts of the domain needing refinement, using a nearest neighbor algorithm for simple regions or an all nearest neighbors graph with an iterative procedure for complicated shapes, and to each cluster we fit a rotated rectangular subgrid. Another feature of this algorithm is our use of data structures, which has made such an automatic algorithm posible. We have implemented the mesh refinement algorithm in both one and two space dimensions, and it generalizes immediately to three (or more) dimensions. We have demons tated with several numerical experiments that with our grid structure, we can do calculations with the same accuracy for a fraction of the cost of a calculation on a conventional, uniform grid.

There are several areas in mesh refinement still needing research. We list some of the more important ones. The best solution strategy for steady state computations is still unknown. For example, is It better to iterate to near convergence on the coarse grid before int roduc ing a refinement , or should the solution on two grid levels be mixed. The use of implicit finite difference methods with our grid structure needs to be developed further. The development of data structures for component grids in different coordinate systems is an important project, with applications beyond our adaptive mesh refinement strategy. Finally, adaptive subgrid generation is a

relatively new topic, and the best grid generation procedure is an important open question.

ACKNOWLEDGMENTS

We thank William Gropp for many helpful discussions, and for providing the graphical output for this paper. Computer time for this work was provided by the Stanford Linear Accelerator Center of the U.S. Department of Energy.

1. Babuska and W. Rheinboldt, Error Estimates for Adaptive Finite Element Computations, SIAM J. Numer. Anal. 15 (1978), 736-754.

R. Bank, A Multi-Level Iterative Method for Nonlinear Elliptic Equations, in Elliptic Problem Solvers, M. Schultz (ed.), Academic Press (1981), 1-16.

M. Berger, Ph.D. thesis, Department of Computer Science, Stanford University, 1982.

J. Bolstad, Ph.D. Thesis, Computer Science Department, Stanford University, 1982.

A. Brandt, Multi-Level Adaptive Solutions to Boundary Value Problems, Math. Coup. 31 (1977), 333-390.

M. Ciment, Stable Difference Schemes with Uneven ?Iesh Spacings, Math. Comp. 25 (1971), 219-227.

H. Cram&r, Mathematical Methods of Statistics, Princeton University Press, 1951.

S. Davis and J. Flaherty, An Adaptive Finite Element Method for Initial-Boundary Value Problems for Partial Differential Equations, SIAM J. Sci. and Stat. Comp. 3 (1982), 6-27.

R. I&da and P. Hart, Pattern Classification and Scene Analysis, Wiley, 1974.

H. Dwyer, R. Kee and B. Sanders, -Adaptive Grid Method for Problems in Fluid Mechanics and Heat Transfer, AIAA J. 18 (1980), 1205-1212.

D. Gannon, Self Adaptive Methods for Parabolic Partial Differential Equations, Dept. of Computer Science, Univ. of Iliinois-U. C., UIUCDCS-R-80-1020, 1980.

D. Gennery, Object Detection and Measurement Using Stereo Vision, Proc. 6th IJCAI (1979), 320-327.

D. Gotlieb and S. Orszag, Numerical Analysis of Spectral Methods: Theory and Applications-, SIAM, 1977.

W. D. Gropp, A Test of Moving Mesh Refinement for 2-D Scalar Hyperbolic Problems, SIAM J. Sci. and Stat. Comp. 1 (1980), 191-197.

B. Gustafsson, The Convergence Rate for Difference Approximations to General Mixed Initial Value Problems, Math. Comp. 29 (1975), 396-406.

A. Harten and J. Hyman, Self-Adjusting Grid Methods for One-Dimensional Hyperbolic Conservation Laws, to appear in J. Comp. Physics, 1983.

G. W. Hedstrom, Yodels of Difference Schemes for $u_t + u_x = 0$ by Partial Differential Equations, Math. Comp. 29 (1975), 964-977.

J. Hartigan, Clustering Algorithms, Academic Press, 1973.

A. Jameson, Iterative Solution of Transonic Flows over Airfoils and Wings, Including Flows at Mach 1, Comm. Pure Appl. Math. XXVII (1974), 283-309.

D. Knuth, The Art of Computer Programming, Vol. 1, 2nd ed., Addison-Wesley, 1973.

B. Kreiss, Construction of a Curvilinear Grid, preprint, 1982. Submited to SIAM J. Sci. and Stat. Comp.

K. Miller and R. Miller, Moving Finite Elements. I, SIAM J. Numer. Anal. 18 (1981), 1019-1032.

J. Oliger, to appear.

J. Oliger, Approximate Methods for Atmospheric and Oceanographic Circulation Problems, in Lecture Notes in Phycsics 91, R. Glowinski and J. Lions (eds.), Springer-Verlag (1979), 171-184.

V. Pereyra and E. Sewell, Mesh Selection for Discrete Solution of Boundray Problems in Ordinary Differential Equations, Numer. Math. 23 (1975), 261-268.

A. Sherman and M. Saeger, An Approach to Automatic Software for Parabolic Partial Differential Equations, in Advances in Computer Methods for Partial Differential Equations - IV, Vichnevetsky and Stepleman (eds.), (1981), 88-92.

R. B. Simpson, Automatic Local Refinement for Irregular Rectangular Meshes, Reearch Report CS-78-19, Department of Computer Science, University of Waterloo, 1978.

G. Sod, A Survey of Several Finite Difference Methods for Systems of Nonlinear Hypebolic Conservation Laws, J. Comp. Physics 27 (1978), 1-31.

G. Starius, On Composite Mesh Difference Methods for Hyperbolic differential Equations, Numer. Math. (1980), 241-255.

H. Vivfand, Conservative Forms of Gas Dynamic Equations, La Recherche Aerospatiale, No. 1, Jan./Feb. 1974, 65-68.

K. H. Winkler, A Numerical Procedure for the Calculation of Nonsteady spherical Shock Fronts with Radiation, Ph.D. Thesis, Max Planck Institute for Physics and Astrophysics, 1977.