Chapter 4

Automatic Grid Adaption

At first sight the problem of designing an algorithm to automatically adapt a computational grid to an evolving flow solution appears to be a Herculean task. Indeed, any adaptive scheme must combine many elements, which must also be sequenced correctly, if it is to be successful. However, the hierarchical grid structure employed by the AMR algorithm enables the adaption process to be straightforwardly sub-divided into a sequence of well defined problems. It is then a relatively simple matter to design a separate solution procedure for each of these elements in turn. Moreover, by judicious choice of properties for these individual procedures we can guarantee certain properties for the adaption process as a whole. The robustness of the AMR algorithm owes much to this *divide and conquer* strategy.

We start this chapter with an example of an adaptive calculation; a plane shock propagating down a duct. Although this is a very simple problem it does nevertheless illustrate all the major elements within the adaption process; these elements are then described separately. For more realistic calculations the adaptions of grids at different levels within the grid hierarchy need to be carefully co-ordinated. The correct co-ordination ensures robustness and and is explained at some length. Following this explanation we are able at long last to present the complete AMR algorithm. Finally, we conclude this chapter with a series of validation problems that demonstrate the integrity of our implementation of the AMR algorithm.

4.1 Elements of the Adaption Process

Consider a plane shock travelling from left to right down a duct. Suppose a coarse grid, G_0, is used to discretize the duct and that a fine embedded grid, G_1, covers the vicinity of the shock, see figure 4.1 (a). Now, if the flow solution on this grid structure is integrated forward in time then sooner or later the shock will be within one mesh cell of the right

hand edge of the grid G_1, see figure 4.1 (b). If the grid structure does not adapt then the shock will run off the edge of G_1. At the very least this will cause the shock to smear to its natural width on the coarse grid; this width is a property of the mesh integration scheme. For strong shocks it may result in a non-monotonic shock profile. So, how can we avoid this problem? The adaption process used by the AMR algorithm effectively results in the embedded grid, G_1, gliding along the coarse grid, G_0, so as to keep pace with the moving shock front.

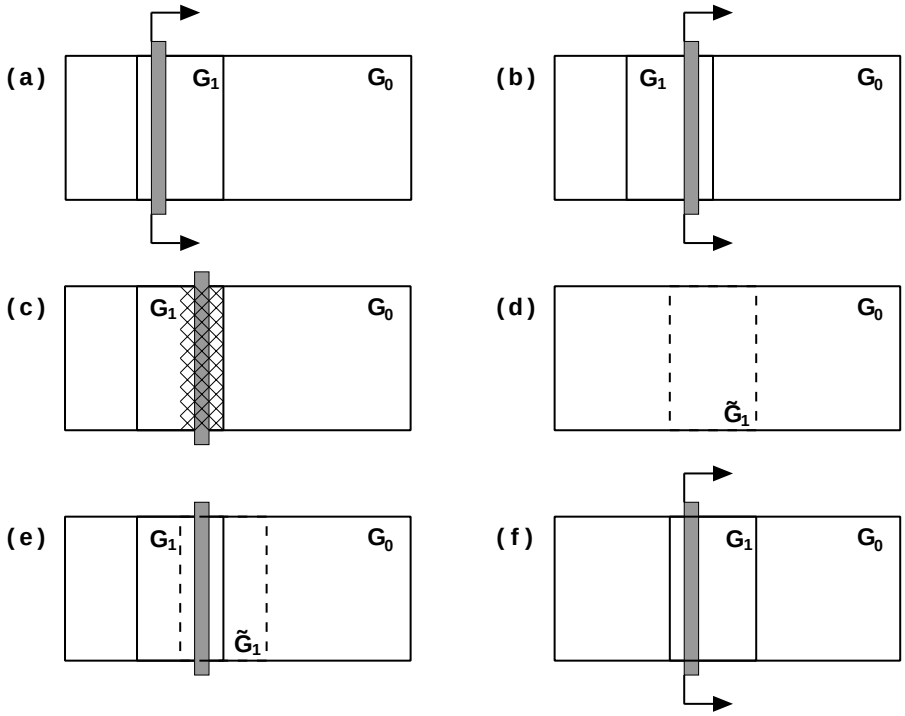


Figure 4.1: Adaption process for a shock propagating down a duct.

The adaption process may be split loosely into three processes. First, given a grid structure and flow solution we identify regions of interest. These regions will be refined, that is they will be covered by an embedded grid. So, for our duct example we need only look at the solution on the grid G_0. Given a suitable refinement criteria then the cells in the vicinity of the shock will be flagged for refinement, see figure 4.1 (c). Second, the flagged cells are grouped into rectangular clusters. Each of these clusters are then covered by a single embedded mesh. This grouping process will result in a new grid structure, $\widetilde{G}$, see figure 4.1 (d). So temporarily we have two grid structures, see figure 4.1 (e), but this new grid structure does not have a flow solution. Finally, we take the solution from the old structure and transfer it to the new one, see figure 4.1 (f). If this sequence of procedures is performed repeatedly then the embedded grid will shadow the moving shock front. Unlike other adaptive schemes, at no stage in the adaption process do we explicitly decide to de-refine part of the grid instead we simply choose not to cover part of it with

an embedded mesh.

The outline of the adaption process given in this section has been rudimentary, but it should help the reader to maintain a sense of direction during the more detailed descriptions of the individual elements that follow below.

4.1.1 Flagging for Refinement

The process of flagging for refinement determines which coarse cells at level l need covering by an embedded grid at level $l+1$. Now, the present work is primarily concerned with computing solutions to unsteady shock hydrodynamic flows; the resolution of numerical solutions to such flows can be improved if the computational grid can be locally refined in the vicinity of shock waves and contact discontinuities. If these basic features can be flagged for refinement, preferably by some economical means, then more complicated flow features such as triple points, roll up vortices and Kelvin-Helmholtz instabilities will also be flagged for refinement. We have adopted the optimistic view that if these types of features are adequately resolved then the rest of the flow field must also be adequately resolved. Results have largely shown that this optimism was not misplaced.

One obvious monitor function with which to flag shock fronts and contact surfaces is the density gradient of the flow solution. Indeed, this method has been used by numerous workers. Here, we simply look at the change in density between two neighbouring mesh cells. If this change is larger than some fraction, F_{tol}, of the maximum density jump between any two neighbouring cells then both cells are flagged as requiring refinement. Typical values for F_{tol} are in the range 0.05–0.15. This somewhat rough and ready approach generally works very well, but it is possible to miss very weak discontinuities. Berger[7] adopted a seemingly more rigorous, and ultimately more expensive, approach. Richardson extrapolation was used to calculate the local truncation error of the solution. The grid was then refined in regions where this error was high. However, near flow discontinuities, the very types of flow feature we are most interested in, Richardson extrapolation will be invalid. Although to be fair, the truncation error calculated by this method will be large and so discontinuities will still be refined. But the approach is somewhat at odds with Occam's razor[1], a useful dictum for any exponent of CFD.

Exactly how this flagging process works in practice may be seen by studying figures 4.2–4.7. Figure 4.2 is a snapshot of the density contours taken during the computation of the reflection of a plane shock by a ramp. The corresponding computational grid is shown in figure 4.3. Underlying the fine grid there is a continuous field solution for the medium grid. Density contours for this medium grid are shown in figure 4.4. We use this solution to flag those cells that require covering by a fine grid. Figures 4.5–4.7 show the three sets of cells that would be flagged for refinement given different values for F_{tol}.

[1] Entities are not to be multiplied beyond necessity.

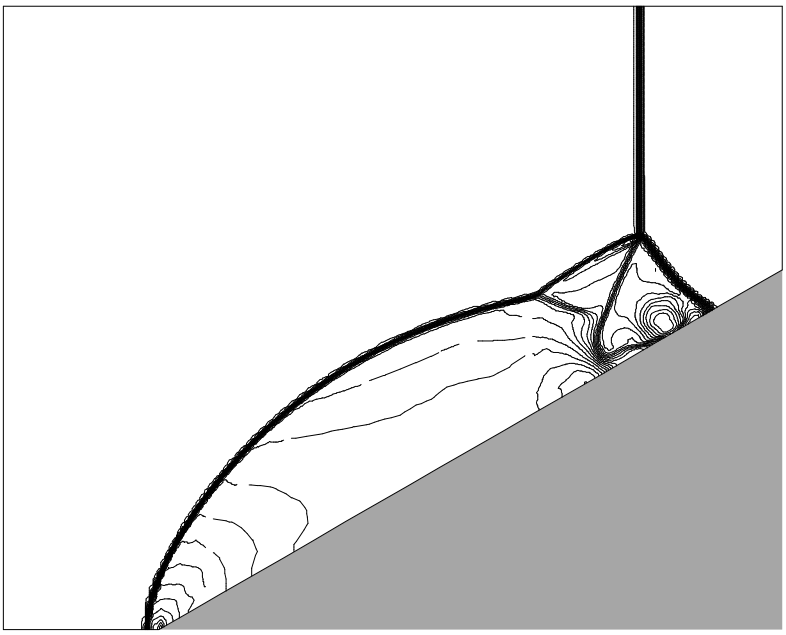


Figure 4.2: Density contours for the reflection of a plane shock by a ramp.

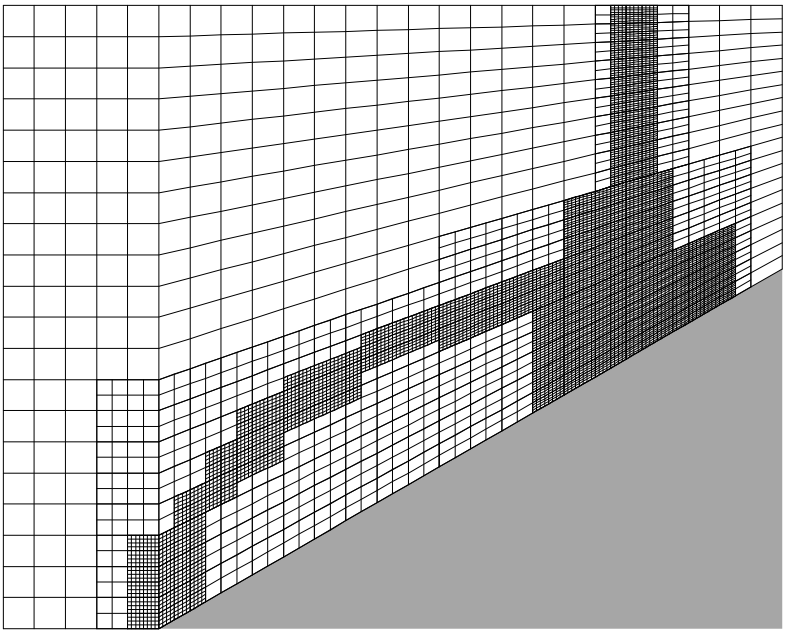


Figure 4.3: Computational grid for the density contours shown above.

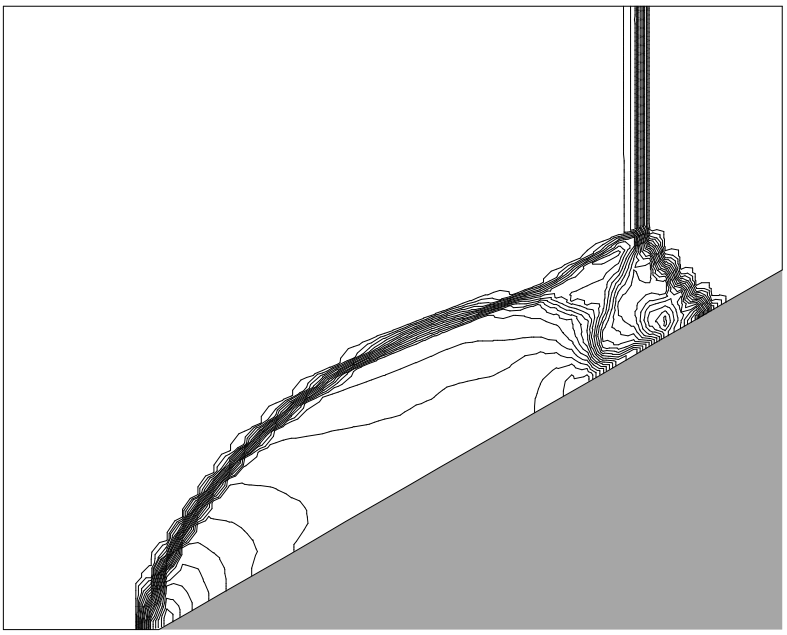


Figure 4.4: Density contours for the medium grid.

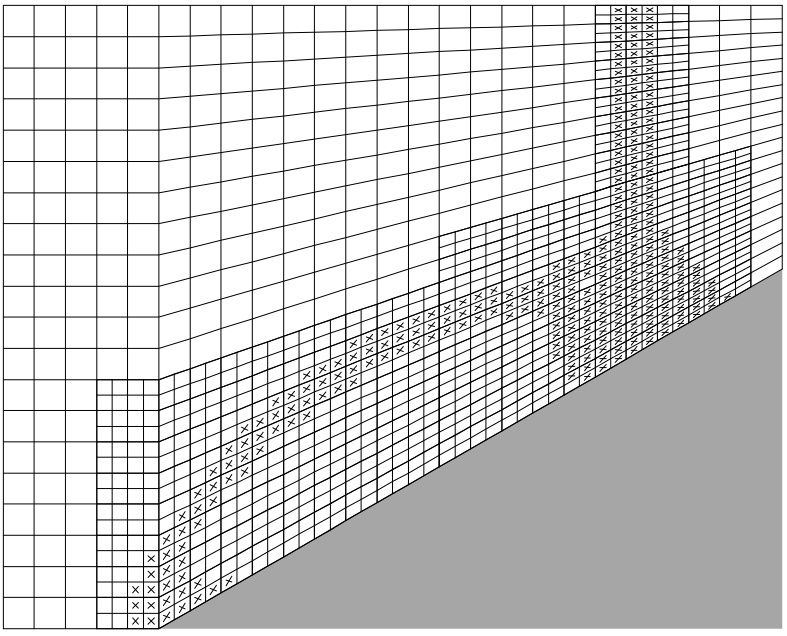


Figure 4.5: $F_{tol} = 0.05$: Cells flagged for refinement.

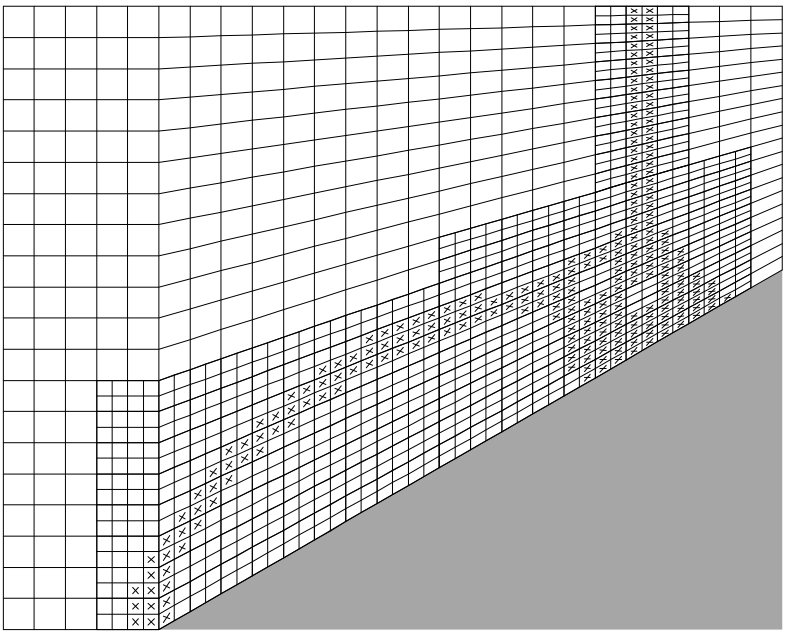


Figure 4.6: $F_{tol} = 0.15$: Cells flagged for refinement.

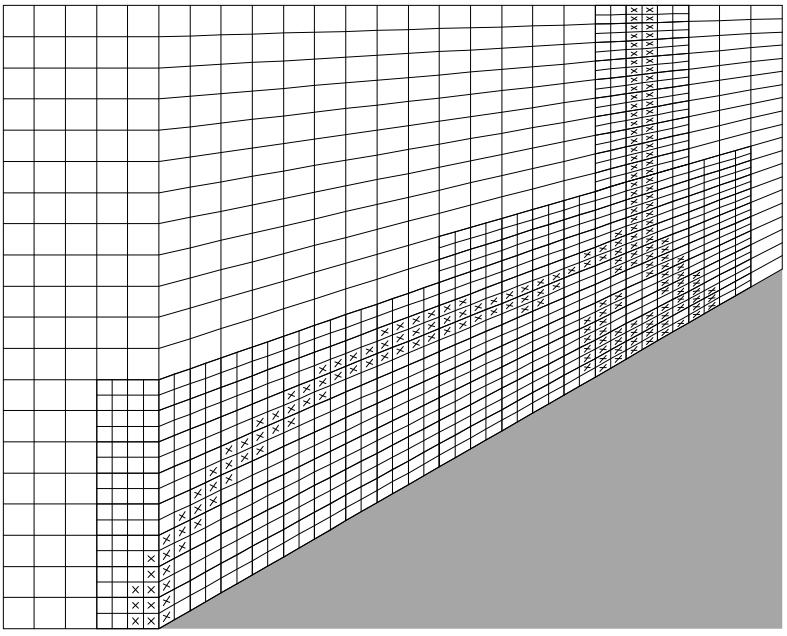


Figure 4.7: $F_{tol} = 0.25$: Cells flagged for refinement.

The flagging for refinement process is the only part of the AMR algorithm that has to be tuned to a particular application. For example, we have used density gradients as a monitor function for inviscid shock hydrodynamic flows and might well use shear stresses for viscous flows. We do not view this need to tune the flagging process as a weakness of the scheme. On the contrary, a single monitor function for all problems would probably be cumbersome and computationally expensive. By keeping the flagging process separate from the rest of the algorithm it is then relatively simple to incorporate the relevant physics for a particular problem into the adaption process. This should allow an expedient monitor function to be found whatever the problem to be solved.

In §3.6 it was shown that spurious oscillations may arise whenever a discontinuity crosses a *fine-coarse* mesh boundary. It would therefore be useful to know how many integrations may be performed on a newly adapted grid before an isolated discontinuity could conceivably cross such a boundary. For simplicity we will consider the linear advection equation in one space dimension. Suppose a perfect discontinuity, travelling from left to right, is contained by an embedded fine mesh. The discrete solution on this fine mesh may be written in normalized form,

$$\{\ldots, 1, 1, 1, 0, 0, 0, \ldots\}.$$

The projection of this solution on the underlying coarse mesh will be,

$$\{\ldots, 1, 1, k/r, 0, 0, 0, \ldots\},$$

where r is the spatial refinement factor of the fine mesh relative to the coarse mesh, and k marks the discrete position of the discontinuity internal to a coarse cell, thus $1 \leq k \leq r$.

Now suppose that the fine grid is adapted such that at least part of the discontinuity on the coarse mesh remains covered by a fine mesh. The gradients between neighbouring coarse cells will determine which cells remain covered by a fine mesh after the adaption process has finished. The gradient at the head of the discontinuity is $-k/r$ and that at the tail is $k/r - 1$. So if $k < r/2$ the coarse cell ahead of the one containing the discontinuity will not necessarily have been flagged for refinement, and so it might not be covered by a fine mesh. Therefore, using a Courant number of ν it could take just $(r-k)/\nu$ integrations of the fine mesh before the discontinuity escapes onto the coarse mesh. The worst case occurs when k is equal to $(r+1)/2 - 1$ then the number of integrations is $(r+1)/2\nu$. Note, if ν is taken as one-half then it will take $r+1$ integrations before the discontinuity can possibly cross the *fine-coarse* mesh boundary.

Now, after the flagging for refinement process we could add a series of safety flags. These flags would be applied to cells adjoining cells that have been flagged for refinement. If the clustering process worked with the combination of safety and refinement flags then this would guarantee that the coarse cell ahead of the one containing the discontinuity would always remain covered by a fine mesh. This in turn would guarantee that for $\nu \leq 1$ it would always take more than $r+1$ integrations of the fine mesh for the discontinuity to escape onto the coarse grid.

4.1.2 Grouping/Clustering

The *grouping/clustering* process produces an embedded grid $\widetilde{G}_l$ when given a set of flagged cells contained by the grid G_{l-1}. This process amounts to finding a set of logically rectangular patches that will completely cover the flagged cells. A single patch will then form a single mesh within $\widetilde{G}_l$. Underpinning this process is a simple area sub-divide algorithm that produces a set of patches for each mesh of G_{l-1} in turn. We present an example of how this algorithm works for a given set of flagged cells. It is then shown how this algorithm may be conveniently coded so as to work for an arbitrary set of flagged cells.

Consider the flagged cells shown in figure 4.8 (a), the area sub-divide algorithm proceeds as follows. First, find the smallest rectangular patch that covers all the cells, see (b), we call this patch the extent of the flagged cells. Now, this patch is a candidate for forming one of the meshes of the adapted grid $\widetilde{G}_l$. A simple packing criteria is used to see if the mesh is acceptable. If the number of flagged cells divided by the number of cells contained by the prospective mesh is greater than some value, P_{tol}, then the mesh is deemed acceptable. Typical values for P_{tol} are in the range 0.5–0.75, here we assume P_{tol} is 0.6. The extent shown in (b) is not acceptable, when judged by our packing criteria, so we sub-divide it into two regions. This sub-division is made about the longest side. Every time the mesh is sub-divided the right-hand/top sub-division is stored on a LIFO[2] stack and we concentrate on the left-hand/bottom sub-division, see (c). It takes two further sub-divisions, see (e) and (g), before an extent of flagged cells meets our packing criteria, see (h). This extent is then saved as a mesh patch and we remove an area from the stack for further consideration, see (i). Again this area meets our packing criteria so it is saved as a mesh patch and a further area is taken from the stack, see (j). This general procedure continues until we no longer have to sub-divide an area and there are no more areas left on the stack, see (k).

The procedure **Group**, see figure 4.9, shows how the area sub-divide algorithm can be conveniently coded using recursion. The procedure works by first finding the number, N, and the extent, $< I'_1, J'_1, I'_2, J'_2 >$, of any flagged cells contained within the specified area $< I_1, J_1, I_2, J_2 >$. If there are no flagged cells then the procedure can return directly, otherwise the extent of these flagged cells is checked to see if it would make an acceptable mesh. If the mesh is acceptable then the extent and number of flagged cells contained by the extent are saved and the procedure returns to the caller. However, if the prospective mesh was not acceptable then the extent is sub-divided into two. This sub-division is made along the longest side. So, if the width of the extent is greater than its height then the extent is sub-divided into two rectangles left and right, with extents $< I'_1, J'_1, \frac{1}{2}(I'_1 + I'_2), J'_2 >$ and $< \frac{1}{2}(I'_1 + I'_2) + 1, J'_1, I'_2, J'_2 >$ respectively. Finally, the procedure is called recursively for both these sub-divided areas before returning to the caller. The reader should satisfy himself/herself that this procedure will indeed produce the sequence of

[2]Last In First Out.

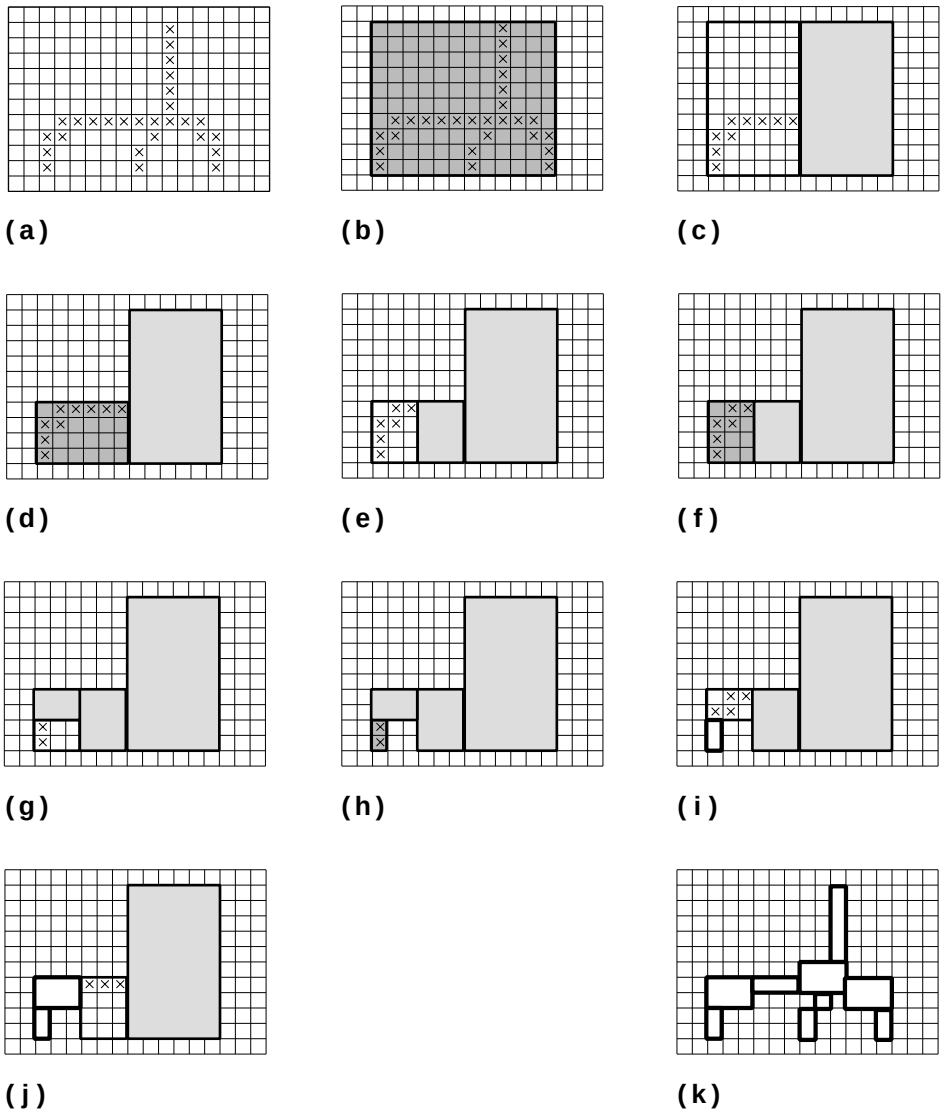


Figure 4.8: Sequence of events during the grouping process.

events shown in figure 4.8.

It should be noted that in order to minimize the amount of computer storage required by the AMR algorithm it is worth placing restrictions on the size of an embedded mesh. Namely, the width or height of a mesh should not exceed NIJ cells, and the number of cells contained by a mesh should not exceed $NIxJ$ cells. Currently NIJ and $NIxJ$ are set to 210 and 5000 cells respectively. Some further storage can be saved by optimizing the patches produced by the area sub-divide algorithm. The optimization process tests to see if certain patches can be combined to form a single alternative patch. This process will decrease the ratio of the grid perimeter length to grid area. However, it is not an essential part of the scheme and details will not be given here, other than to note the fact that after optimization some mesh patches may overlap one another.

It was stated above that typical values for P_{tol} lie in the range 0.5–0.75, however, the exact value is not critical to the success of the AMR algorithm. If the value is too high then there will simply be a large number of embedded meshes, so the grid will have a large mesh perimeter to mesh area ratio. Therefore, the reduced time spent by the flow solver will be offset by the increased time spent looking after mesh boundaries. Similarly, if the value is too low then there will be a few large meshes. So, more time will be spent by the flow solver and less time spent at mesh boundaries. Figures 4.10–4.12 show the sensitivity of the *grouping/clustering* process to different values of P_{tol}. These figures show the outlines of the embedded meshes produced at grid level 2 to cover the flagged cells at grid level 1 shown in figure 4.6.

We end this section by noting a few properties of the *grouping/clustering* process. The area sub-divide algorithm produces non-overlapping meshes, but the optimizing process may produce overlapping ones. Every mesh $\widetilde{G}_{l,k}$ produced by the process will be wholly contained by at least one mesh of G_{l-1}, that is $\widetilde{G}_l \subseteq G_{l-1}$. Finally, one slightly surprising property is that the process will not necessarily produce a symmetric set of meshes given a symmetric set of flagged cells. This will occur if it is necessary to sub-divide a side whose length is not even. In addition, the sub-division process is biased towards splitting square regions into halves left & right rather than top & bottom.

```
Procedure Group(< I1, J1, I2, J2 >)
  Extent(N, < I1, J1, I2, J2 >, < I1', J1', I2', J2' >)
  if N > 0 {
    if Mesh_Acceptable(N, < I1', J1', I2', J2' >) {
      Save_Mesh(N, < I1', J1', I2', J2' >)
    }
    else {
      if width >= height {
        I'mid = 1/2(I1' + I2')
        Group(< I1', J1', I'mid, J2' >)
        Group(< I'mid + 1, J1', I2', J2' >)
      }
      else {
        J'mid = 1/2(J1' + J2')
        Group(< I1', J1', I2', J'mid >)
        Group(< I1', J'mid + 1, I2', J2' >)
      }
    }
  }
End Procedure
```

Figure 4.9: Recursive procedure to perform the area sub-divide algorithm.

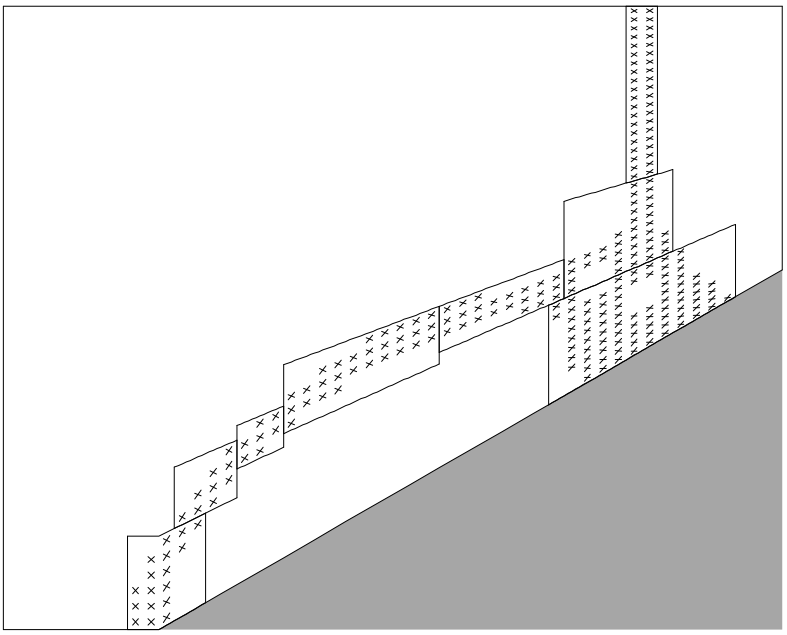


Figure 4.10: Embedded meshes produced for $P_{tol} = 0.5$.

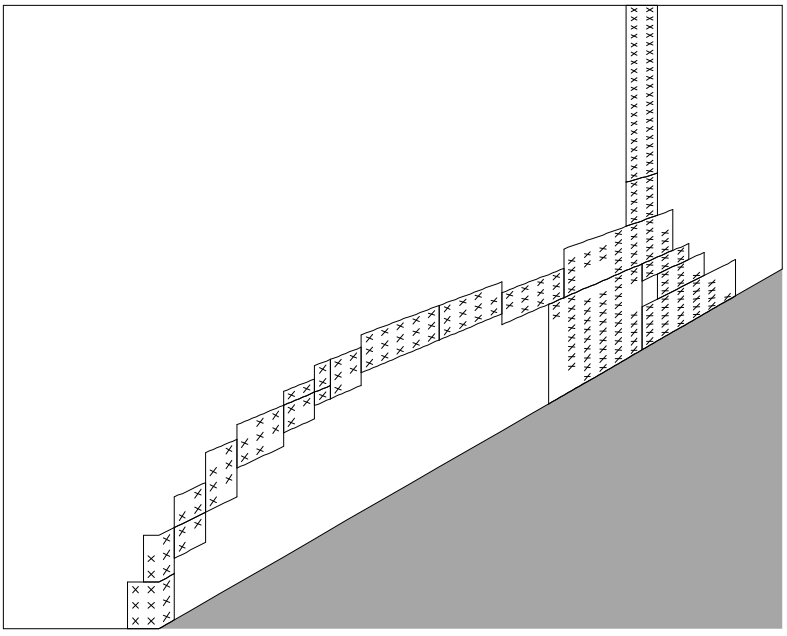


Figure 4.11: Embedded meshes produced for $P_{tol} = 0.7$.

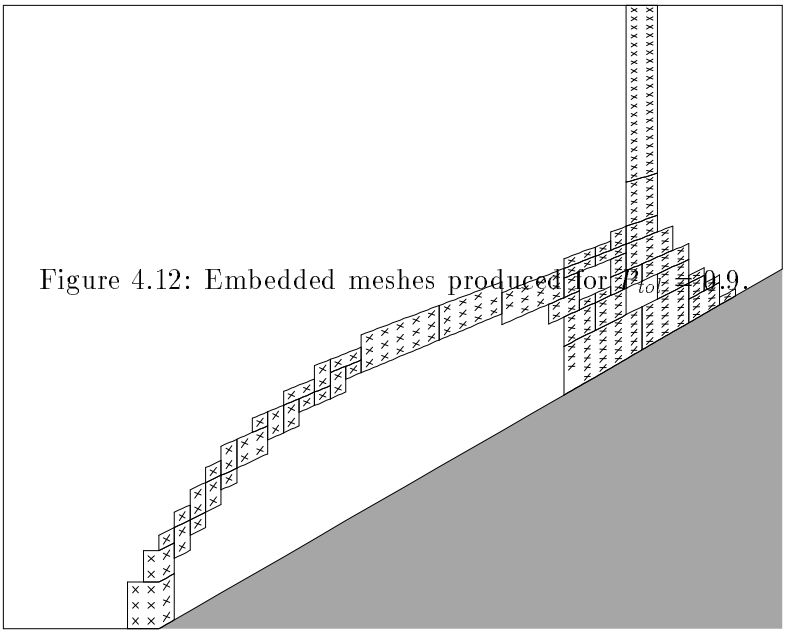


Figure 4.12: Embedded meshes produced for $\alpha_{cut} = 0.9$.

4.1.3 Transfer of Solution

The newly adapted grid, $\widetilde{G}_l$, produced by the *grouping/clustering* process does not have a field solution; the *transfer of solution* process sets up a field solution, $\widetilde{\mathbf{W}}_l$, for this grid from the known field solution, $\mathbf{W}$, on the grid structure G. This transfer process should have the following properties: it must work for any grid produced by the *grouping/clustering* process; it should not compromise the accuracy of the mesh integration process; it should be monotonicity preserving; it must be conservative.

Any newly adapted grid, $\widetilde{G}_l$, will always be wholly contained by the grid G_{l-1}. Therefore, it is possible to provide a complete field solution for the fine grid $\widetilde{G}_l$ by interpolation from the known field solution on the coarse grid G_{l-1}. However, the new grid $\widetilde{G}_l$ will partially overlap the old grid G_l. For the regions of overlap, it is possible to replace the interpolated solution with a more accurate solution taken directly from $\mathbf{W}_l$. Note, that this procedure does not involve interpolation. The regions of overlap are geometrically identical, therefore, the solution is simply shovelled from a cell in G_l to the corresponding cell in $\widetilde{G}_l$. One of the secrets to the success of the AMR algorithm is that flow discontinuities always fall within these regions of overlap between G_l and $\widetilde{G}_l$. So the adaption process cannot introduce further errors in these problem regions. Consequently, near flow discontinuities the numerical solution largely exhibits the properties of the numerical scheme used for the single mesh integration process.

The method used to interpolate the field solution on the coarse grid, G_{l-1}, needs to be chosen with care. For example, bi-linear interpolation will not lead to a conservative transfer of solution. This could present serious problems for evolutionary flow calculations, but would be immaterial for steady-state calculations. Moreover, using bi-linear interpolation does not guarantee that the transfer process is monotonicity preserving. A simple one-dimensional example demonstrates this fact. Consider the grid structure, G, and corresponding field solution, $\mathbf{W}$, shown in figures 4.13 (a) and (b). Now suppose that the grid structure has adapted to that shown in figure 4.13 (d). If we take the coarse solution $\mathbf{W}_0$, see figure 4.13 (c), and linearly interpolate between cell centres, we get the monotonic profile shown in figure 4.13 (e). However, after we have partially overwritten this interpolated solution with the more accurate solution from $\mathbf{W}_1$, we get the non-monotonic profile shown in figure 4.13 (f).

We have chosen to use the MUSCL procedure described in chapter 3 to perform the interpolation of the coarse grid field solution. This will guarantee that the transfer process is conservative. Moreover, in non-smooth regions of the flow the interpolation will reduce to zeroth-order and will thus be monotonicity preserving. Finally, it should be noted that the MUSCL procedure is another method of achieving a second-order extension to a first-order Riemann based solver. Therefore it is unlikely that this type of interpolation could compromise the accuracy of our flux-limited Riemann based solver.

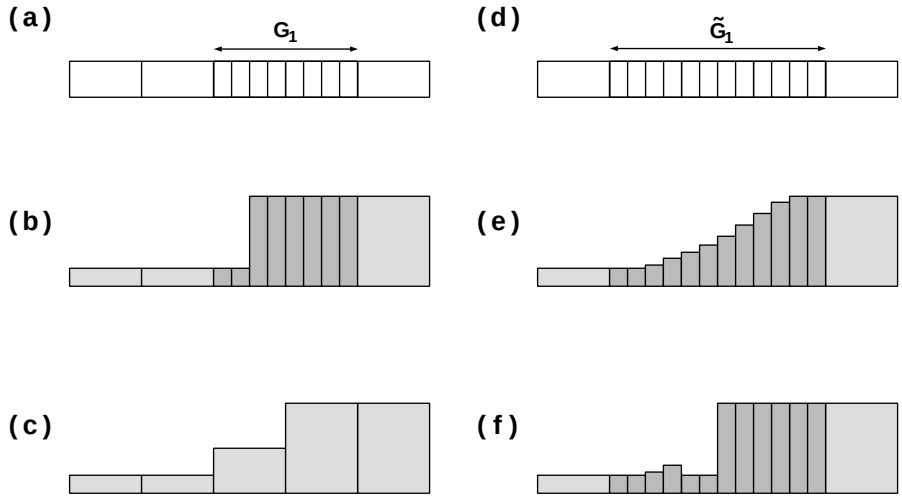


Figure 4.13: Bi-linear interpolation is not monotonicity preserving.

4.2 Co-ordinating the Adaption Process

So far in this chapter we have described how to adapt a single fine grid, G_l, relative to its underlying coarse grid, G_{l-1}. This forms the basis of the automatic adaption procedure used by the AMR algorithm. We now show how to co-ordinate the adaptions at different grid levels so as to provide a robust means of dynamically adapting an arbitrary grid structure to an evolving flow solution. This robustness is achieved by ensuring that discontinuities cannot escape their respective embedded meshes between grid adaptions, and that the adaption process will never fail to produce a *properly nested* grid structure.

In §4.1.1 we noted two conditions under which a discontinuity would not escape a newly adapted grid, $\widetilde{G}_l$, in fewer than $r_l + 1$ integrations of the solution vector $\mathbf{W}_l$. Therefore assuming one of these conditions is met, we can prevent a discontinuity from ever escaping the grid at level l by adapting G_l after every r_l integrations of the solution vector, $\mathbf{W}_l$. This suggests that the order of grid adaption should follow the order of grid integration. Consider a grid structure that has four levels G_0, G_1, G_2 and G_3, with refinement factors of $1, 2, 4$ and 2 respectively. The order in which the grid levels are integrated may be found by applying the *recursive interleaving* procedure described in §??. The required order of grid adaption then follows straightforwardly, see figure 4.14.

An examination of this figure shows that the grid structure undergoes eight separate adaptions during a single coarse time step. During most of these adaptions only the finest grid level is moved. However, there are times when two or more grid levels must be

adapted simultaneously. The simultaneous adaption of several grid levels introduces one further element to the adaption process. Namely, how do we ensure that a *properly nested* grid structure will always adapt to another *properly nested* grid structure. This property is vital because without it the grid integration process will fail.

A property of the *grouping/clustering* process is that a newly adapted grid, $\widetilde{G}_l$, will be wholly contained by the grid G_{l-1}, that is $\widetilde{G}_l \subseteq G_{l-1}$. But on its own, this property is not sufficient to ensure that a grid structure remains *properly nested* after two or more grid levels have been adapted. For, $\widetilde{G}_l \subseteq G_{l-1}$ and $\widetilde{G}_{l-1} \subseteq G_{l-2}$ do not guarantee that $\widetilde{G}_l \subseteq \widetilde{G}_{l-1}$, this problem is clearly shown in figure 4.15. Fortunately, we can overcome this problem if we are careful to adapt G_l before adapting G_{l-1}. By performing the adaption in this order we will already know $\widetilde{G}_l$ when we come to flag those cells that will eventually be clustered to form $\widetilde{G}_{l-1}$. Now, these cells are contained by the grid G_{l-2}. But, $\widetilde{G}_l \subseteq G_{l-1}$ and $G_{l-1} \subseteq G_{l-2}$, therefore, $\widetilde{G}_l \subseteq G_{l-2}$. This enables us to enforce *proper nesting*, $\widetilde{G}_l \subseteq \widetilde{G}_{l-1}$, by simply flagging all those cells contained by G_{l-2} that are covered by $\widetilde{G}_l$.

The *pseudo-code* shown in figure 4.16 will co-ordinate the adaption process for an arbitrary grid structure. Called with a parameter l_p this procedure will adapt grid levels $l_p + 1$ and above. Following our descriptions for the different elements which form the adaption process this code is largely self explanatory. However, it is worth pointing out one feature of the adaption process that allows a simplification of the procedures that are required to manage the data storage heaps which were described in §2.4.2. Levels 0–l_p of the newly adapted grid structure, $\widetilde{G}$, will always be the same as that for the old grid structure, G. So given G, $\widetilde{G}$, and $\mathbf{W}$ the procedure **Transfer_Solution** need only calculate the field solutions $\widetilde{\mathbf{W}}_{l_p+1}$ to $\widetilde{\mathbf{W}}_{l_{max}}$. As for the old grid structure, these field solutions are stored on a heap as a series of data blocks, one block for each mesh. But we choose to store them on a series of temporary heaps. The procedure **Transfer_Data_Structure** takes the data describing the new grid structure and overwrites the data that described the old structure. It then calculates a set of heap pointers such that the new mesh data blocks may be stored contiguously on the working storage heaps. Finally, it transfers the new field solution from the temporary heaps to the working storage heaps. Thus the old grid structure and field solution have been replaced by the newly adapted ones ready for another series of mesh integrations followed by yet another grid adaption.

Because the heap data blocks are always stored contiguously there is no need to incorporate *garbage collection*[34] routines into the heap management system. Admittedly this simplification has been bought at the expense of having to transfer data from the temporary heaps to the working heaps. But an analysis of our program using the UNIX utility *gprof* suggests that the work required to shuffle the data in this manner typically accounts for less than 0.1% of the total run time. We therefore conclude that the benefits to be gained from incorporating a more sophisticated heap management system into our scheme would be inconsequential.

Grid Integration	Grid Adaption
G_0	
G_1	
G_2	
G_3	
G_3	
.....	G_3
G_2	
G_3	
G_3	
.....	G_3
G_2	
G_3	
G_3	
.....	G_3
G_2	
G_3	
G_3	
.....	G_3, G_2
G_1	
G_2	
G_3	
G_3	
.....	G_3
G_2	
G_3	
G_3	
.....	G_3
G_2	
G_3	
G_3	
.....	G_3
G_2	
G_3	
G_3	
.....	G_3, G_2, G_1

Figure 4.14: The order of grid adaption follows the order of grid integration.

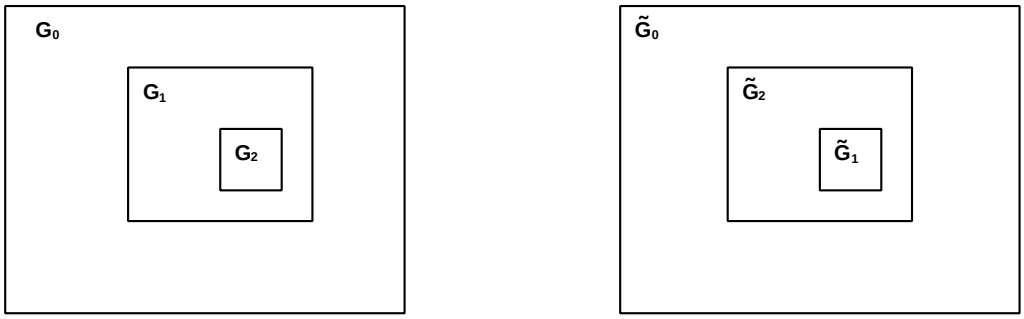


Figure 4.15: $\widetilde{G}_l \subseteq G_{l-1}$ does not guarantee *proper nesting*.

```
Procedure Adapt(l_p)
  Initialise_G̃
  for l = l_max − 1 down to l_p {
    Set_Refine_Flags(l)
    Ensure_Proper_Nesting(l)
    Cluster(l)
  }
  Transfer_Solution
  Transfer_Data_Structure
  Set_BdyTypes(l_p + 1)
  Calculate_Geometry(l_p + 1)
End Procedure
```

Figure 4.16: *Pseudo-code* to co-ordinate the adaption process.

4.3 The Complete AMR Algorithm

We have now presented procedures for all the major elements of the AMR algorithm. All that remains is to show how these procedures are co-ordinated in order to form the complete algorithm. The material required to understand this co-ordination has already been given. So here we shall simply present the algorithm together with a cursory explanation that makes reference to this earlier material.

The *pseudo-code* shown in figure 4.17 will co-ordinate the complete AMR algorithm. The basic structure for this procedure follows that used for the integration of a fixed grid structure which was given in §3.10. In fact there are only two additional statements both of which are easily explained. We have noted that in order to prevent a discontinuity from escaping an embedded mesh then the order of grid adaption should follow the order of grid integration, see §4.2. Therefore the grid G_{l+1} is adapted, by calling **Adapt**(l), after its sequence of rt_{l+1} integrations is complete and conservation with the grid G_l has been enforced. Note that the call **Adapt**(l) will in fact adapt all grid levels from $l+1$ to l_{max}. Consequently there is no need to explicitly adapt the grid G_{l+1} if this is to be closely followed by an adaption of the grid G_l, hence the logical test to see if it is strictly necessary to call the adaption procedure. Now, our process for flagging for refinement uses density gradients of the flow solution. So, in order to calculate the correct gradients along mesh boundaries the dummy cells bordering these boundaries should be primed with the most up to date values possible. Therefore, before adapting the grid at level $l+1$ we prime the dummy cells contained by the grid G_l by calling **Set_bc**(l,N_t).

The call **AMR**(0,0) will perform one complete sequence of grid integrations and adaptions such as that shown in figure 4.14. So, only a very simple harness is required to form a procedure for marching a flow solution over some specified period of time, see figure 4.18. The procedure **Find_Stable_Time_Step** should be able to calculate the set of time steps $\{\Delta t_l : l \in \mathcal{N}, 0 \leq l \leq l_{max}\}$ such that the calculation remains stable. While the function **The_Calculation_Is_Finished** simply checks to see if we need to stop or not.

We have chosen to present fragments of *pseudo-code* because they are a succinct means of describing accurately the intricacies of the AMR algorithm. It is not our intention that these fragments be cribbed verbatim in order to build a working code. Indeed, this would not be possible because for reasons of clarity the various procedures simply do not contain the mundane details such as variable typing and scoping declarations that are required by real programming languages. Nevertheless the structure of our implementation follows directly from these *pseudo-code* fragments. Therefore they should prove to be invaluable to anyone wishing to implement the ideas described in this thesis.

```
Procedure AMR(l,Nt)
  Integrate_Grid(l,Nt)
  if l < l_max {
    Initialise_Conservative_Fixup(l + 1)
    for Nft = 1 to rt_{l+1} {
      AMR(l + 1,Nft)
    }
    Apply_Conservative_Fixup(l + 1)
    Project_Solution(l + 1)
    Set_bc(l,Nt)
    if Nt < rt_l Adapt(l)
  }
End Procedure
```

Figure 4.17: *Pseudo-code* to orchestrate the AMR algorithm.

```
...     ...     ...
  repeat {
    Find_Stable_Time_Step
    AMR(0,0)
  } until The_Calculation_Is_Finished
...     ...     ...
```

Figure 4.18: A harness for the AMR algorithm.

4.4 Validation

We are not able to prove that all the properties which we have been careful to design into our automatic grid adaption process do in fact hold true when the scheme is applied to the Euler equations. However, we now present solutions to three test problems which give ground for the belief that our implementation of the AMR algorithm is sufficiently accurate and robust to be of practical use. The three problems were chosen so as to check specific features of the scheme under adverse conditions, and the resultant computational solutions are not specifically intended to be scrutinized as accurate flow predictions. Indeed, one of the test problems highlighted a hitherto unpublicised deficiency of the standard numerical scheme which we had adopted as the basis for our mesh integrator; a full discussion of this deficiency is given in chapter 5.

4.4.1 Validation Test #1

It was shown in §3.6 how spurious oscillations may arise whenever a strong shock propagates across a *fine-coarse* mesh boundary. Our adaption process has been designed to avoid these oscillations by simply preventing a shock from crossing such a boundary. In §4.1.1 we gave two conditions under which a one-dimensional discontinuity could be guaranteed not to escape an embedded fine mesh. These conditions were derived for the linear advection equation, however, in practice they are also applicable to the Euler equations. But, we shall only demonstrate this fact for the stronger of the two conditions, namely no safety flags.

Figure 4.19 shows a sequence of density profiles taken from the computation of a strong shock, $M_s = 10$, propagating from left to right with a Courant number of 0.5. Three grid levels were used for the calculation. The base level grid contained 100 equally spaced cells and the two finer grids each had a spatial refinement factor of 10. Thus the finest mesh spacing would be equivalent to that of a single uniform mesh containing $10,000$ cells. Density gradients were used to flag for refinement and F_{tol} was set to 0.99 so as to ensure that an embedded mesh would at most cover just two cells of the underlying coarse mesh. Now, this is quite a severe test for the pressure ratio across the shock is fairly large at 116.5, and the finest grid has been adapted some 1720 times. But we can see from this figure that the shock profile remains reasonably well-behaved. Note because the embedded meshes are so narrow the shock sometimes reaches the point of just being about to cross the *fine-coarse* boundary. Under such circumstances it would be better if the *fine-coarse* boundary procedure could refrain from interpolating in time. Figure 4.20 shows results for this test problem where we have simply switched off the temporal refinement at *fine-coarse* boundaries. Pleasingly, the shock profile now remains perfectly monotone[3].

[3]Ignoring the start up error.

We have repeated this test problem using a Courant number of 0.8, see figure 4.21. Under these conditions it is inevitable that the shock front will sometimes just escape an embedded fine mesh between grid adaptions. Consequently, the numerical shock profile is forced to alternate between being too wide and too narrow compared with its natural profile on the coarse and fine meshes. As discussed in §3.6, this gives rise to a continuous series of start up errors, here such errors produce severe oscillations. But note, these test conditions are somewhat contrived and in practice it is extremely unlikely that a shock will escape an embedded mesh. Firstly, the recommended values of F_{tol} ensure that the strongest discontinuities are contained by embedded meshes whose widths fluctuate between two and three coarse mesh cells. The extra cell is sufficient to prevent a discontinuity running off the edge of an embedded mesh. Secondly, the majority of discontinuities in a complicated flowfield will be moving much slower than the fastest characteristic upon which the choice of stable time step is based. Thirdly, the speed of a shock wave must be less than the speed of sound in the post-shock state. So all in all, it is extremely unlikely for a discontinuity to cause problems by escaping across a *fine-coarse* mesh boundary. Indeed, we have repeated this test using $F_{tol} = 0.15$ and $\nu = 0.8$, and to plotting accuracy these results are indistinguishable from those shown in figure 4.20. Moreover, none of the calculations that are presented in chapter 6 employed safety flags, and yet the time steps were chosen such that the fastest characteristic had a notional Courant number of 0.8.

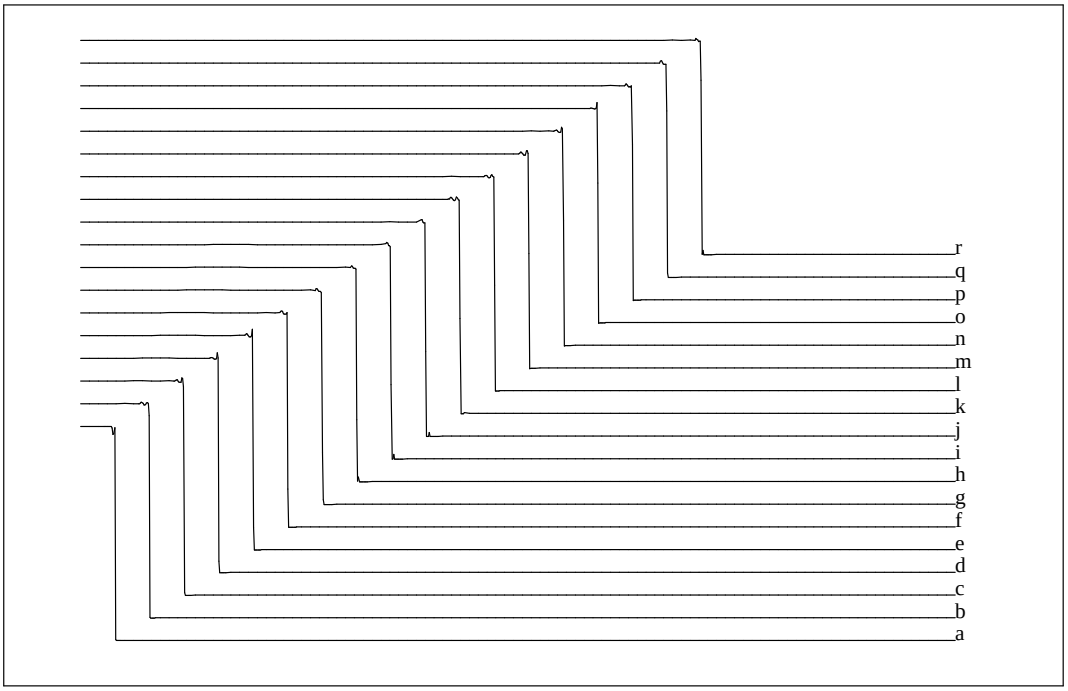


Figure 4.19: A sequence of Density profiles during the propagation of a shock.

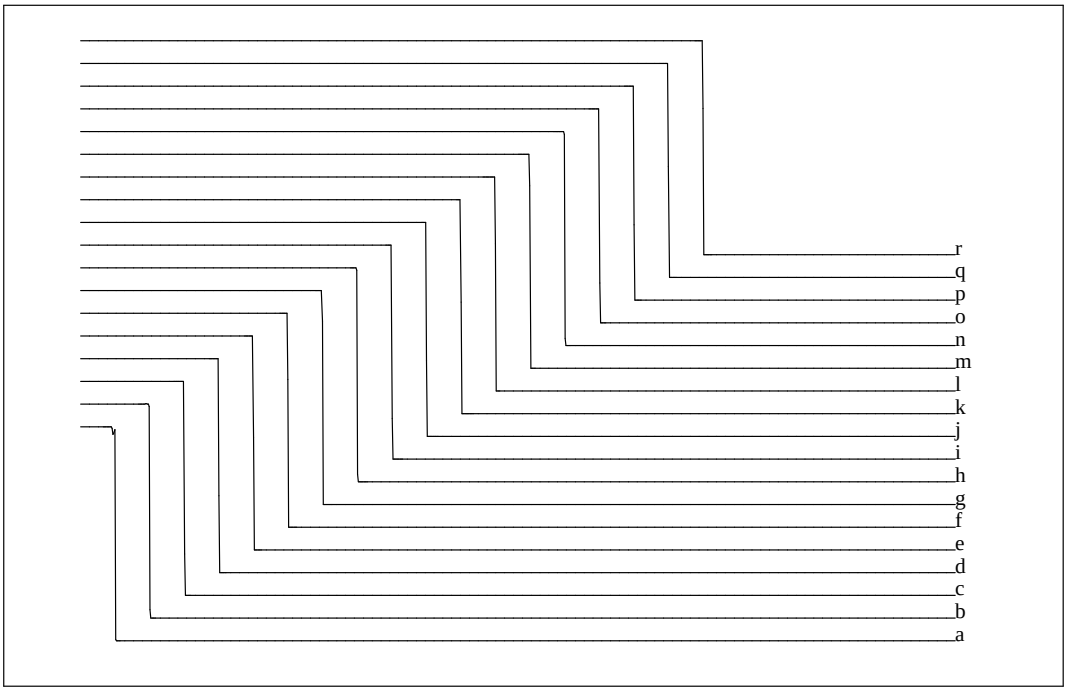


Figure 4.20: The *fine-coarse* boundary procedure is prevented from interpolating in time.

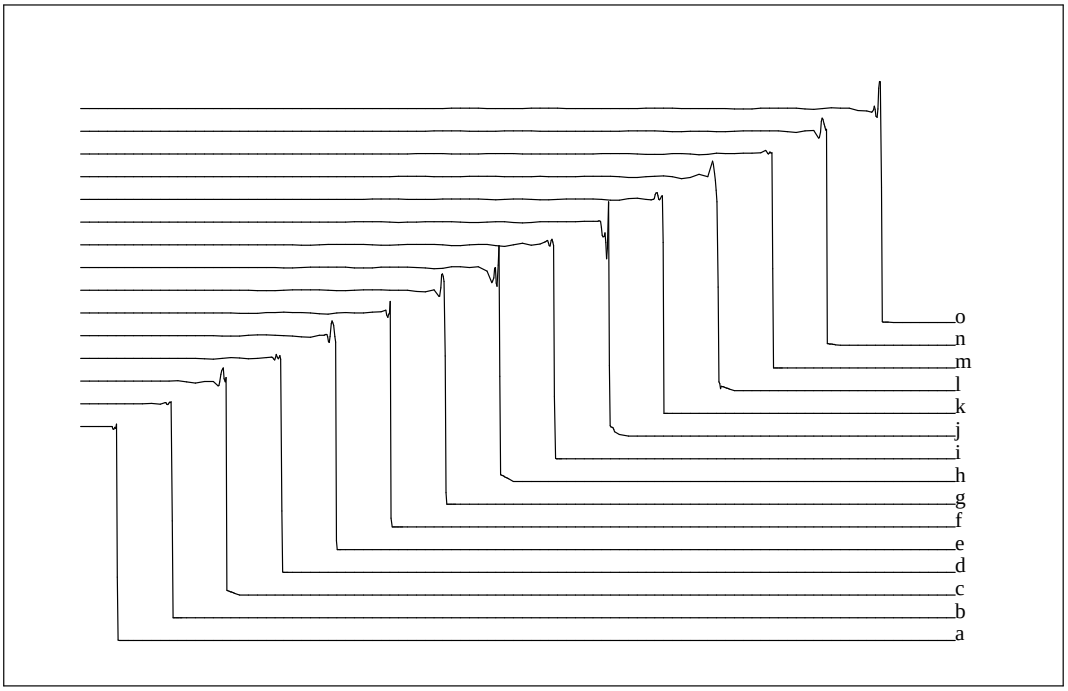


Figure 4.21: Oscillations due to Courant numbers larger than 0.5.

4.4.2 Validation Test #2

The robustness of the AMR algorithm is due to the integrity of its parts. For example, starting from one *properly nested* grid structure, no matter how complex, the automatic grid adaption process is guaranteed to produce another *properly nested* grid structure. Similarly, the grid integration process is guaranteed to work[4] for an arbitrary *properly nested* grid. But we can never prove that our implementation is bug-free and therefore we cannot guarantee that it will always work in the manner in which we intended. Consequently, in order to have confidence in our implementation we must demonstrate that it is robust under extreme conditions. To this end we have computed the reflection of a plane shock, $M_s = 2.12$ and $\gamma = 1.4$, by a 30^o wedge using five grid levels. The solution to this type of problem is well documented[28] and for our specific problem one would expect a self-similar solution known as single Mach reflection.

Figure 4.22 shows the grid structure used to start the calculation. A single coarse mesh, 25 by 20 cells, covers the computational domain. A further four grid levels are used to cover the initial position of the plane shock. From the insert it can be seen that for each of these four grid levels both rI and rJ were set to 2. The values used to control the adaption process, F_{tol} and P_{tol}, were the same for each grid level and were set to 0.05 and 0.6 respectively. Figure 4.23 shows the grid structure after 40 integrations of the coarse grid. Therefore at this point in the calculation the grids G_1, G_2, G_3 and G_4 have been adapted 40, 80, 160 and 320 times respectively. We would like to stress that the process of adapting the grid from that shown in figure 4.22 to that shown in figure 4.23 was completely automatic. In view of the many grid levels and grid adaptions involved in this calculation, coupled with the fact that the grid has remained *properly nested* we believe our implementation is sufficiently free of bugs to be of practical use.

Now, in this section we are primarily concerned with validating the robustness of our implementation of the AMR algorithm. But, in passing we note that the finest grid shown in figure 4.23 immediately suggests that the correct type of reflection has been computed, namely single Mach reflection. The corresponding density contours confirm this, see figure 4.24. The incident shock, reflected shock and Mach stem are all well resolved, as is the slip line emanating from the triple point. Furthermore, as expected, the Mach stem is at right angles to the surface of the wedge. Finally as shown in figure 4.25, we note that the salient features of our solution to this test problem agree well with experiment[24].

[4] But this is not to suggest that it will always give desirable results!

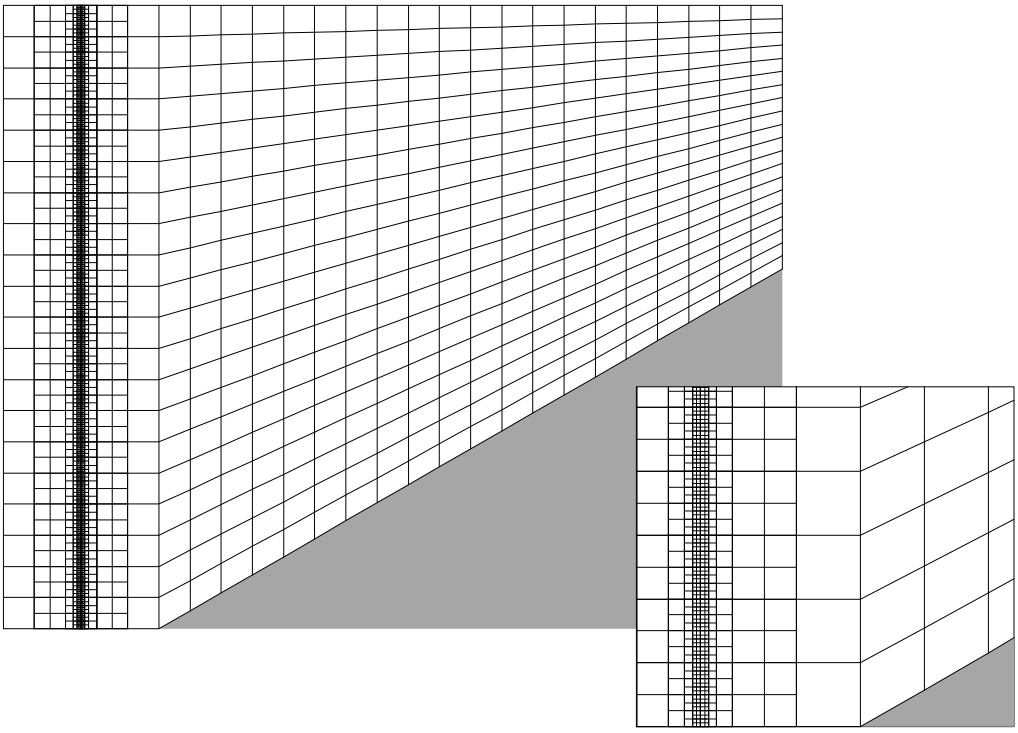


Figure 4.22: Initial grid structure for test problem #2.

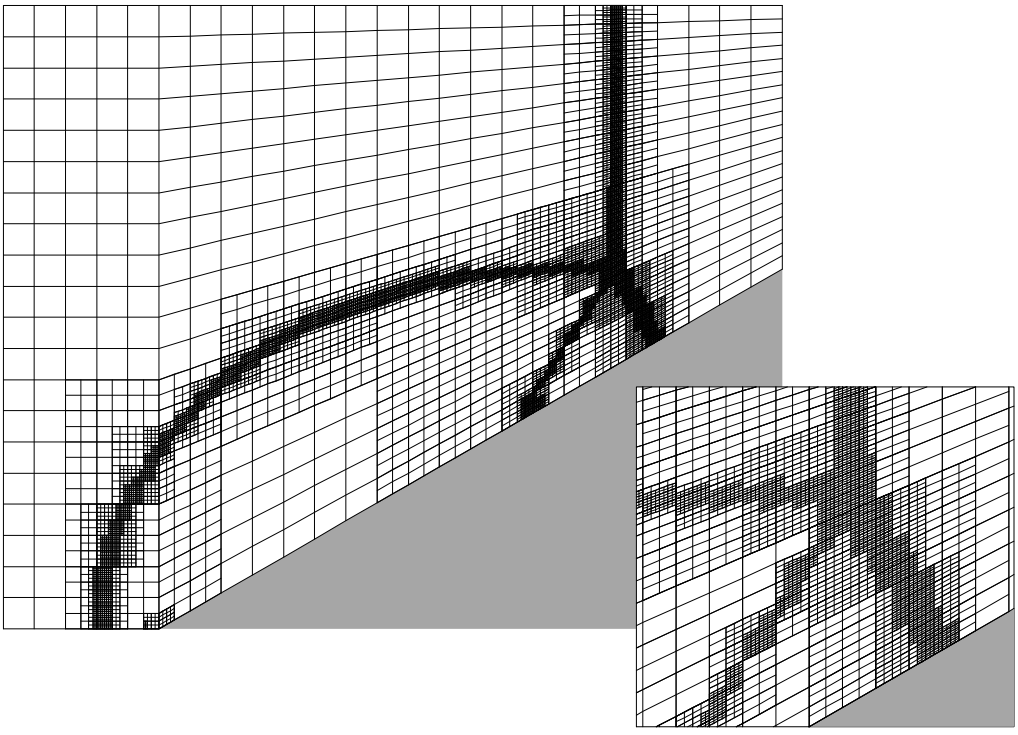


Figure 4.23: Final grid structure for test problem #2.

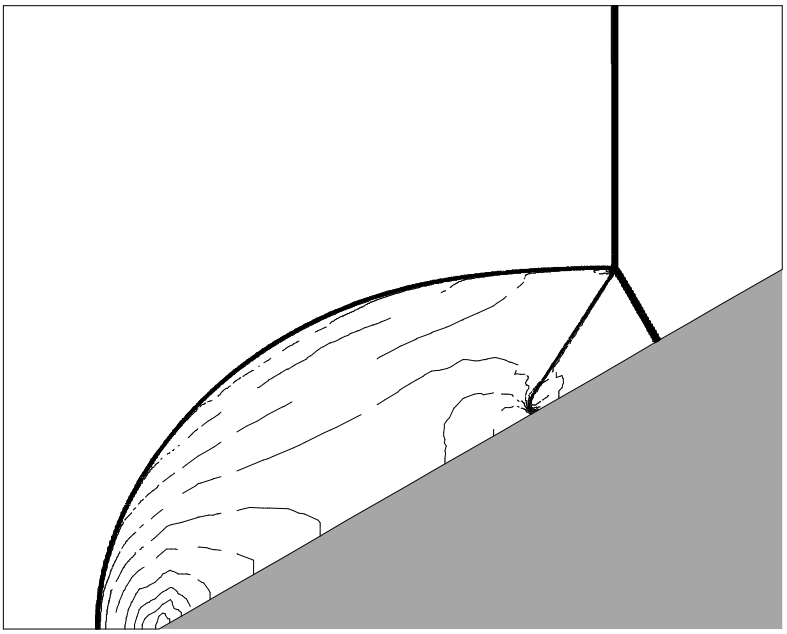
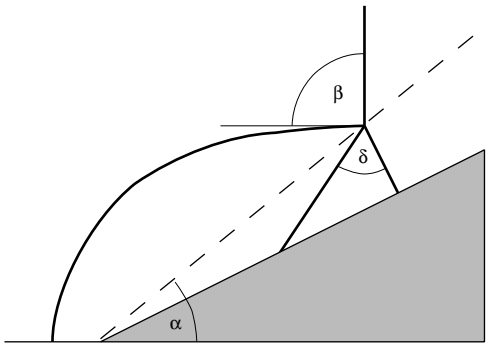


Figure 4.24: Single Mach reflection: $M_s = 2.12$, $\theta = 30^o$, $\gamma = 1.4$.



Angle	Experiment	Computation
α	38.5	38.4
β	92.0	91.6
δ	63.5	63.7

Figure 4.25: Comparison with experimental results.

4.4.3 Validation Test #3

It takes considerably more effort to implement the AMR algorithm than it does to implement a scheme that operates on just a single quasi-rectangular mesh. Consequently, if our algorithm is to be viable then we must be able to demonstrate that it greatly reduces the computational effort required to perform realistic shock-hydrodynamic calculations, and that in so doing, it produces results that are comparable in quality to those that could be obtained using a single fine mesh. To this end we again compute the reflection of a plane shock from a 30^o wedge. But we change the shock speed, $M_s = 5.5$ and $\gamma = 1.4$, such that we now expect a type of solution known as double Mach reflection.

We have used our implementation of the AMR algorithm to compute two sets of results for this problem, one with grid adaption and one without. Figure 4.30 shows a sequence of density contours[5] taken from the calculation performed without grid adaption; the computational grid consisted of a single fine mesh of 200 by 160 cells. At time $t = 1.1$ the solution clearly exhibits the salients features of double Mach reflection, except that the Mach stem is kinked. It transpires that this kinking is attributable to the numerical scheme which formed the basis of the mesh integrator used for these calculations, and so we feel justified in ignoring this anomaly here. However, an examination of the cause for this spurious solution will be given in chapter 5.

The adaptive calculation, see figure 4.31, used three grid levels. The base level grid, G_0, consisted of 25 by 20 cells, while the various factors used to generate the grids G_1 and G_2 are shown tabulated in figure 4.26. Note, at all times during the calculation, any one mesh cell from the grid G_2 will exactly overlay a mesh cell from the grid used for the non-adaptive calculation. Thus, the two grids used for these calculations effectively have the same resolution.

Pleasingly, these two sets of results, figures 4.30 and 4.31, are remarkably similar. At all three times the position of the incident shock, reflected shock and the Mach stem are identical. Furthermore, the two sets of contours for the rectangular region bounded by the Mach stem, surface of the wedge and the two triple-points are virtually indistinguishable from one another. The only criticism of the adaptive calculation is that it has not resolved the very weak slip line emanating from triple-point at the end of the reflected shock. But anyhow this is not particularly well resolved by the non-adaptive calculation. Note, the striations that appear in figure 4.30, for time $t = 0.8$, mark the positions of the two perturbations generated by the start up error, see §3.6. Because these perturbations are so small whether they appear in a contour plot or not is a matter of chance.

Both sets of calculations were performed on a Sun SPARCstation 1 which has a nominal rating of 1.4 Mflops. Timings for these calculations were obtained using the UNIX command *time* and are given in figure 4.27. Apart from the job which was being timed,

[5] 40 contours drawn at 2.5% intervals.

the machine was idle. Consequently, *User time* roughly equates to *Real time*, that is the elapsed time for the job to run, less any time spent by the virtual memory system on disk caching. From these timings it can be seen that the non-adaptive calculation takes almost 20 times longer to run than the adaptive calculation. Even if we exclude the virtual memory access time required by the non-adaptive calculation due to its larger storage requirements then there is still a factor of 8 between the run times for these two calculations.

At best these factors can only be indicative of the computational savings to be gained by using the AMR algorithm. For in general, any savings that are realised will vary not only from problem to problem but also from computer architecture to computer architecture. Consequently, we do not present these results in order to try and substantiate some outrageous claim as to the general computational efficiency of our scheme. Nevertheless the savings for this test problem are sufficiently large to leave little doubt that for a large class of problems the complexity of the AMR algorithm is vindicated. If further evidence is required then the results presented in chapter 6 speak for themselves. Given our computing resources, it simply would not have been possible to produce these very high resolution results without resorting to some form of grid adaption.

l	rI	rJ	F_{tol}	P_{tol}
0	1	1	0.05	0.60
1	2	2	0.10	0.60
2	4	4	N/A	N/A

Figure 4.26: Grid adaption parameters for validation test #3.

DMR. Execution Times hrs:mins:secs				
Period of calculation	Non-Adaptive		Adaptive	
	Real	*User*	*Real*	*User*
$0.0 \rightarrow 0.4$	2:34:57	0:59:27	0:05:02	0:04:57
$0.4 \rightarrow 0.8$	3:26:31	1:15:25	0:08:48	0:08:44
$0.8 \rightarrow 1.1$	2:20:58	1:03:33	0:11:44	0:11:41

Figure 4.27: Run times for validation test #3.

In order to determine the amount of time spent adapting the computational grid as opposed to integrating the numerical flow solution contained by the grid, the adaptive calculation was analysed using the UNIX utility *gprof*. Figure 4.28 gives the profiling information for the procedure **AMR** which co-ordinates the complete AMR algorithm, this procedure was presented in figure 4.17. While figure 4.29 gives the profiling information for the procedure **Adapt** which co-ordinates the adaption process, this procedure was

presented in figure 4.16. Note, a tabulated value of 0.0 implies that the % of the total run time is less than 0.1 and not zero. The most striking feature of these results is that the entire adaption process accounted for less than 2.5% of the total run time. Compared to other adaptive mesh schemes this figure is remarkably low. Furthermore, over half of this time was spent simply calculating the metrics for the newly adapted grids so that the flow integration process would not have to calculate them. So, the actual adaption process accounted for less than 1.2% of the total run time. Again we must emphasize that these timings are only indicative of the behaviour of the AMR algorithm. Nevertheless, these results go some way to explain why the algorithm proves competitive for simulating transient flows.

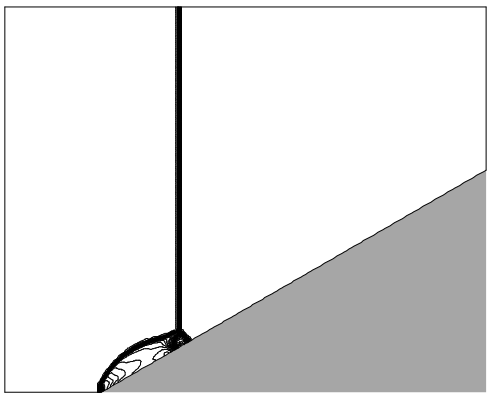
Procedure	% of total run time
Integrate_Grid	93.5
Adapt	2.5
Set_bc	1.6
Project_Solution	0.9
Apply_Fixup	0.1
Initialise_Fixup	0.0

Figure 4.28: Profiling information for the procedure **AMR**.

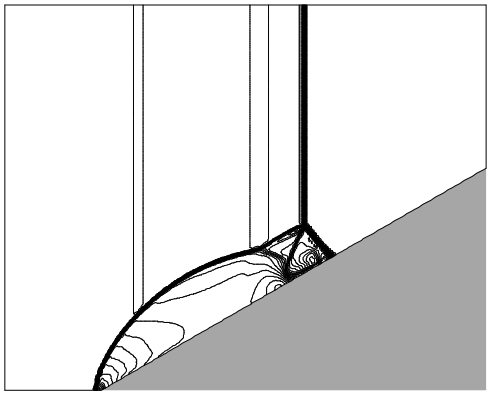
Procedure	% of total run time
Calculate_Geometry	1.3
Transfer_Solution	0.9
Set_BdyTypes	0.1
Transfer_Data_Structure	0.1
Set_Refine_Flags	0.1
Cluster	0.0
Ensure_Proper_Nesting	0.0
Initialise_$\widetilde{\mathrm{G}}$	0.0

Figure 4.29: Profiling information for the procedure **Adapt**.

$t = 0.4$



$t = 0.8$



$t = 1.1$

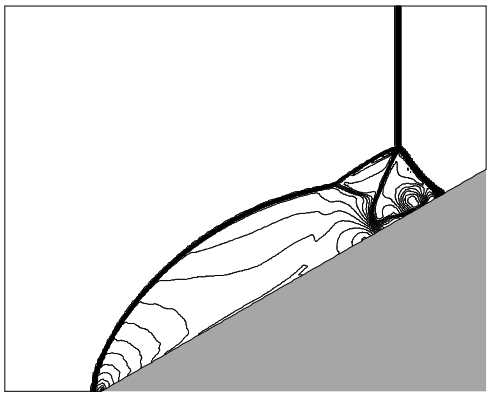
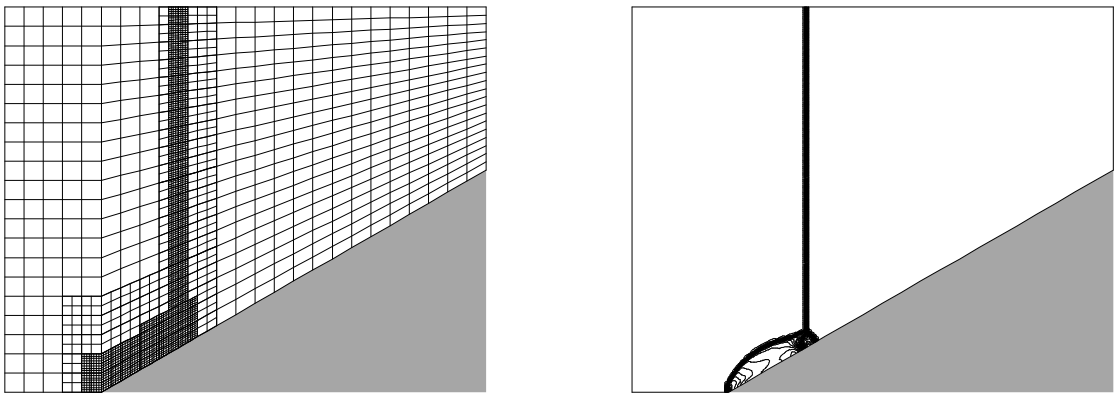
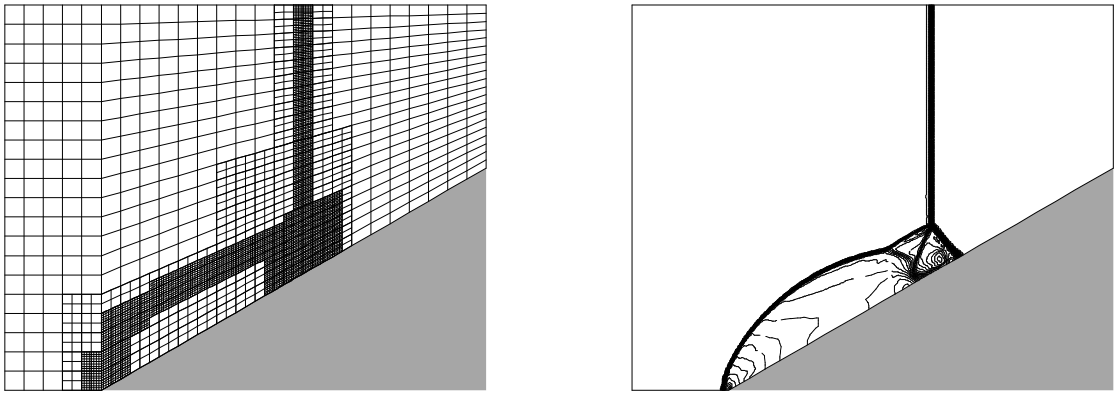


Figure 4.30: Validation test #3, without grid adaption.

$t = 0.4$



$t = 0.8$



$t = 1.1$

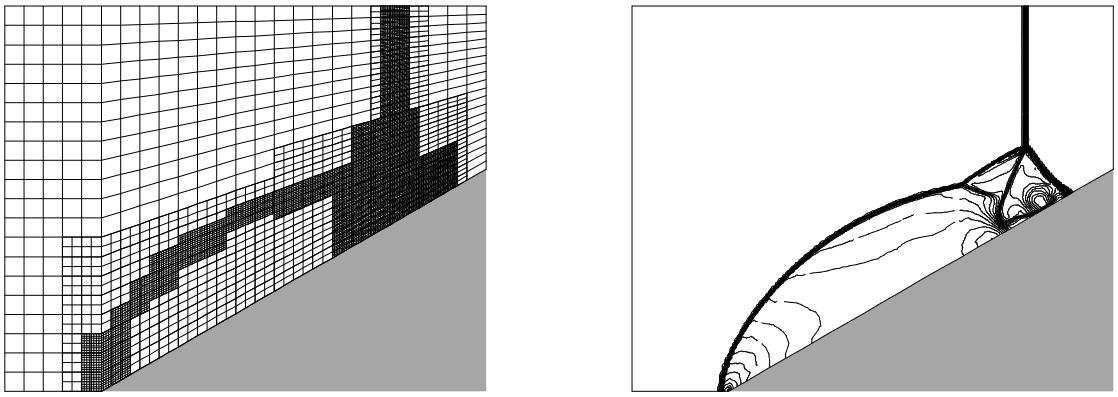


Figure 4.31: Validation test #3, with grid adaption.

4.5 Closing Comment

The validation tests presented in this chapter demonstrate that the AMR algorithm is both computationally efficient and robust. Here we recap the main reasons for this impressive performance.

In contrast to other adaptive schemes, the AMR algorithm has not been designed with the express purpose of minimizing the number of grid cells required to achieve a given numerical resolution. Instead, a balance has been sought between the demands of the spatial and temporal discretizations. This has led to a scheme that refines in time as well as space, and it is this combination of refinement strategies that primarily accounts for the efficiency of the algorithm. The presence of a few extremely fine mesh cells in one part of the flow domain does not have an adverse affect on the rate at which the rest of the flow domain may be integrated. Also in contrast to other schemes, very little effort is required to adapt the computational grid. This economy of effort stems from the algorithm's structured grid system. The majority of the operations for adapting the grid G_l work with the grid G_{l-1}. Note, each cell flagged for refinement at level $l-1$ will typically produce 16 cells at level l. So, the grid adaption operators process far fewer cells than do the grid integration operators. Moreover, these adaption operators are much cheaper than the integration operators. Therefore it is not surprising that the entire adaption process amounts to only 1% or so of the total computational workload.

There are several reasons why the AMR algorithm proves to be robust. One, the algorithm is built from a number of foolproof procedures. For example, given an arbitrarily complex *properly-nested* grid — the adaption process is guaranteed to produce another *properly-nested* grid. Similarly, the process that gathers mesh connectivity information cannot fail. Two, the flow integration process can utilize any *cell-centred* numerical scheme that has been developed for a single quasi-rectangular mesh. Thus the algorithm can assume the inherent robustness of schemes such as Roe's method. Three, many numerical difficulties are circumvented. For example, a strong discontinuity never actually crosses a *fine-coarse* mesh boundary. Also, the process of transferring the numerical solution from the old computational grid to the newly adapted grid does not involve interpolation in the vicinity of discontinuities. Near a discontinuity the numerical solution need only be shovelled from the old grid to the new grid, thus the exact numerical representation of the discontinuity is preserved between grid adaptions.

In conclusion, the impressive performance of the AMR algorithm is the direct result of adopting a balanced strategy which places the emphasis on circumventing numerical difficulties rather than attempting to effect sophisticated remedial procedures.