Sixth Mississippi State Conference on Differential Equations and Computational Simulations, *Electronic Journal of Differential Equations*, Conference 15 (2007), pp. 141–151.
ISSN: 1072-6691. URL: http://ejde.math.txstate.edu or http://ejde.math.unt.edu
ftp ejde.math.txstate.edu (login: ftp)

ISSUES IN ADAPTIVE MESH REFINEMENT IMPLEMENTATION

NOUREDDINE HANNOUN, VASILIOS ALEXIADES

Abstract. Physical phenomena often involve discontinuities and/or localized high-gradient areas. The numerical simulation of these problems and conventional techniques (Finite Elements, Finite Volumes, Finite Differences, and Spectral Methods) with a uniform grid is inefficient when high accuracy is required. Adaptive Mesh Refinement (AMR) is a technique that allows local refinement of the grid. In this presentation, we describe a typical AMR technique and address implementation and algorithmic issues. Triangular unstructured grids and a regular 1 to 4 refinement are considered.

1. Introduction

The numerical solution of problems in Science and Engineering usually requires a grid that covers the computational domain. Discretization of the model equations is performed at the grid element level.

Solution accuracy may be increased through the use of higher order discretization, or uniform refinement. Both methods have drawbacks: higher order schemes are prone to producing oscillations (nonmonotone), while uniform refinement is too expensive.

Adaptivity of the mesh is necessary to cluster grid points in the regions where it is most needed, while keeping the grid coarse elsewhere. There are two main adaptive strategies [3]: adaptive mesh redistribution, also called p-refinement, and adaptive mesh refinement, also called h-refinement. The first type of refinement, p-refinement, continuously repositions a fixed number of cells to improve resolution in selected areas. Although easier to implement, it has serious drawbacks: it does not allow for a change of topology in the solution discontinuity, and the grid may get severely distorted. The second type of mesh adaption, h-refinement, or more commonly AMR [2], adds new cells and deletes other cells that are no longer required. The AMR technique is extremely useful for applications involving discontinuities, shock waves [7], phase change [6] or/and combustion [5] interfaces, or any localized large-gradient regions. However, it is more complex to implement and program.

2000 *Mathematics Subject Classification.* 68U99, 65M50, 65Y20.
Key words and phrases. Adaptive mesh refinement; data structure; computational method; object oriented programming; conservation laws.

Published February 28, 2007.

The present work aims at presenting the steps involved in a typical AMR technique for unstructured triangular grids. Emphasis will be on some of the problems that may arise during the implementation.

2. Choosing the data structure

2.1. **Types of triangular mesh refinement.** For grids with triangular elements, there are three common refinement types, which are illustrated in Figure 1. The regular 1:4 refinement, (c), has the advantage of maximizing children angles, thus keeping the element aspect ratio close to one. This refinement type is adopted in the present work.

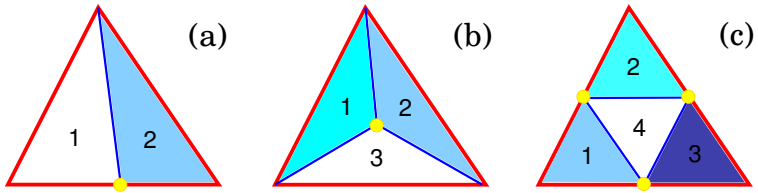


Figure 1. Triangle refinement: (a) longest-edge bisection, (b) Interesection of bisectors, (c) Edge midpoints or regular 1:4 refinement.

2.2. **Elements.** An element is a basic grid entity. Each element is a triangle. Elements are generated upon refinement or deactivated upon coarsening. Therefore, it is necessary to keep in memory both active and inactive elements. A suitable data structure for this type of problems is the so-called k-way tree [4]. Figure 2 displays the tree associated with a given AMR grid. The number of elements may increase or decrease with time (calculations). This change requires making use of dynamic memory allocation and of pointers, two features which are readily available in Fortran 95 [1] and C++. Since an element is one node of the k-way tree, it should have two pointers, one pointer to its parent and an array of four pointers to its four children. Additional data may include the refinement level, the type of element if multiple geometries are allowed, whether the element is a boundary element or not, and all the data related to the type of numerical method being used (such as the coefficients of the expansion of the solution w.r.t. the basis when using a finite element method).

2.3. **Nodes/Vertices.** There are two approaches to the storage of vertices (the three vertices of a triangular element). In the first approach (Figure 3), each vertex is a unique entity and the vertices are stored in a queue. In the second approach, a vertex is part of the data structure for the element to which it belongs. Advantages of the second approach include faster access to vertices and simpler algorithms, while there are two drawbacks. First, if a vertex belongs to several elements, it will be stored in each element and this results in waste of memory. Second, it is necessary to identify which vertex of one element corresponds to the same physical vertex in an adjacent element. This may be accomplished by comparing coordinates of the two vertices although comparison between real numbers may be tricky. When using the second approach, it seems difficult to assign a global number to a vertex.

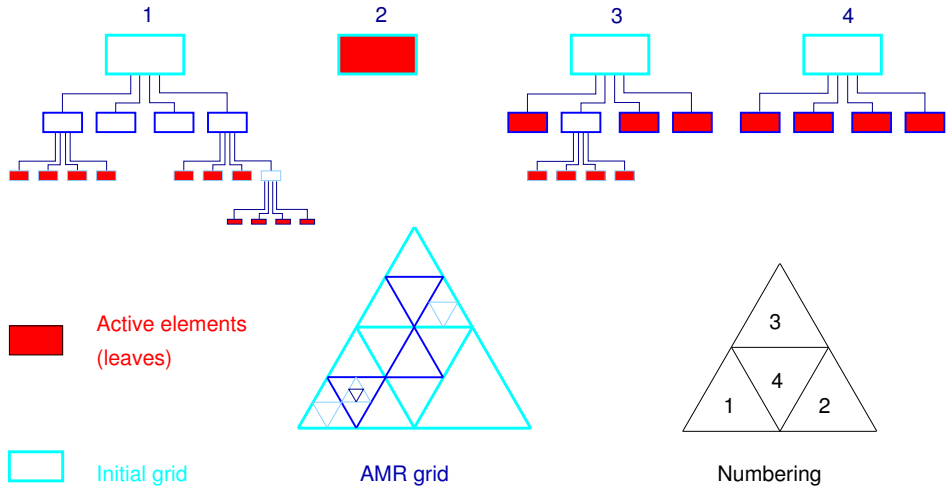


FIGURE 2. k-way tree for the grid elements

The second approach is adopted in this work and the corresponding data structure is described next.

One very important issue about the data structure of an element is the storage of neighbors and edges, which are needed for both refinement/coarsening and for flux evaluation for the numerical simulation. The problem is complicated by the occurrence of dangling nodes (nodes in the middle of an edge). Figure 4 shows a numbering that results in a one-to-one neighbor-edge-vertex mapping. This is achieved by adding dangling vertices to the list of actual vertices of the element while splitting edges into smaller ones as appropriate. A maximum of six vertices (edges or neighbors) is thus allowed. If the actual vertices (the original three of the triangle) are needed for refinement or solver algorithms, these can be stored in an array of three integers "ind" corresponding to the indices of the actual vertices in the list of all vertices of the element. For example, if there are 5 vertices and the actual vertices are 1, 3, and 4, then $\mathsf{ind}(1) = 1$, $\mathsf{ind}(2) = 3$, and $\mathsf{ind}(3) = 4$.

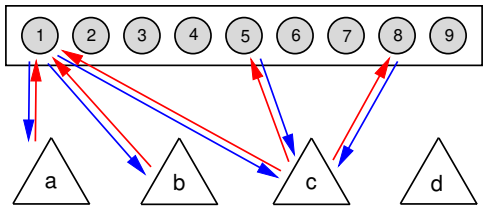


FIGURE 3. Storage of vertices in queue

Neighbors of an element may be accessed through a pointer to an array of pointers (Figure 5-a). This allows for easy accounting of the change of number of neighbors upon grid refinement/coarsening. Null pointers are used for boundary edges with no neighbors, thus preserving the one-to-one edge-neighbor-vertex mapping (Figure 5-b). Vertices may be stored either in an allocatable array or an array of pointers. When dealing with loops over indices of vertices, it is advantageous to avoid modulo

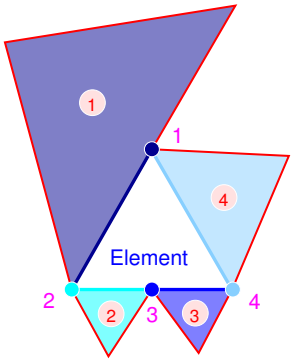
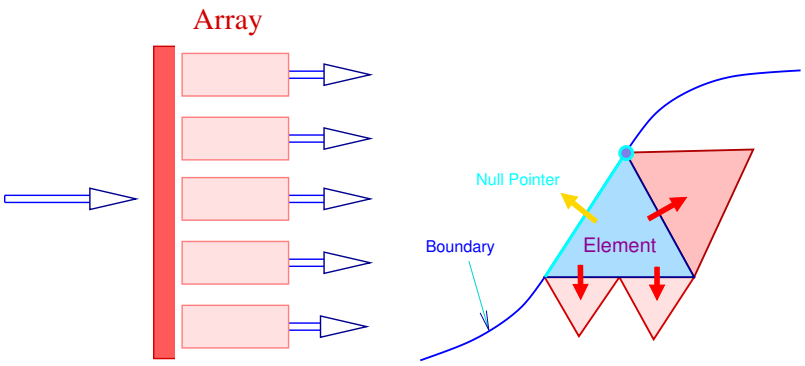


FIGURE 4. Numbering of neighbors, edges, and vertices

arithmetic by the use of an integer array having twice as many elements as the number of vertices (edges, neighbors); e.g. if there are four vertices, define an array perm = [1 2 3 4 1 2 3 4] and, for example, use neighbor perm$(i+2)$ to refer to neighbor $(i+2)$



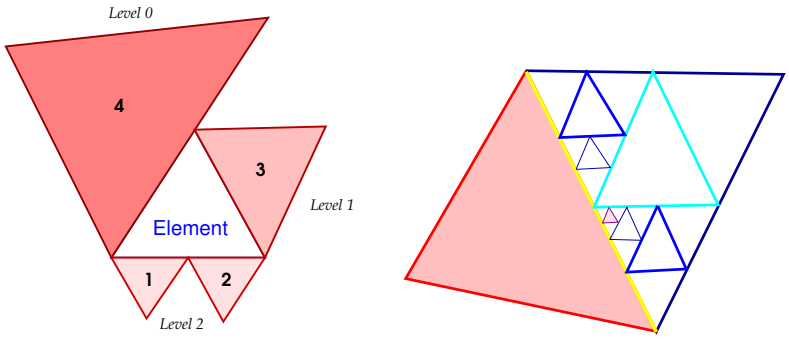
(a) Neighbor data structure (b) Boundary neighbor pointer

FIGURE 5. An element's neighbors

3. Before the refinement

3.1. **When should the grid be refined?** Refinement of the grid is performed locally according to some criterion which depends on the type of problem being solved. When the solution is known in advance, a suitable criterion is $\|\nabla f\|h > \varepsilon$, where f is the solution, h the size of the mesh, and ε the threshold value for refinement. If the solution has a discontinuity, an element would be refined if it is intersected by the discontinuity. If the solution is not known in advance, the refinement criterion is often based on the residual of the model equation. There is extensive research on this type of criterion, usually referred to as Error Estimates for AMR [8, 9].

3.2. **Refinement constraints.** Two fundamental issues arise when dealing with the refinement process. First, it is necessary to choose a maximum level of refinement, otherwise the algorithm may result in an infinite loop or simply exceed available computer memory. A second constraint, that two neighboring elements should not differ my more than one refinement level [7], may be imposed to simplify both book-keeping and algorithm, as well as ensure accurate inter-element flux calculations. Figure 6 shows the three possible neighbor configurations (a), and an example of a not-allowed configuration (b).



(a) Possible neighbor configurations

(b) Impossible configuration

FIGURE 6. Refinement/coarsening constraints

3.3. **Refinement strategy (order).** The process of refinement of the grid is complex and involves recursion. It is not enough for an element to satisfy the refinement criterion in order to be refined. Indeed, the constraints mentioned in the previous section may result in the following scenario: if any of the element's neighbors are larger than the element itself, they must be refined before the element is refined. Therefore, recursive programming features need be used.

4. Performing the refinement of an element.

The process of refining an element, once it qualifies for refinement, is described next.

4.1. **Creating the children.** First, the four children need to be created and an appropriate numbering scheme should be adopted. Figure 7 displays a possible numbering choice. Edge nodes are numbered according to their location vis-a-vis the element vertices and similarly for the children. Children vertices may be numbered as follows: the first vertex of a child corresponds to the vertex of the parent element where the child is located, while for the fourth child (central one) the first vertex would correspond to the first edge of the parent element.

Data also needs to be transferred from the parent element to the children in a consistent way. Accuracy, conservation, monotonicity, and many other considerations may need to be taken into account during the transfer operation.

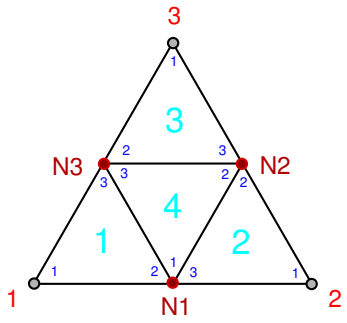


FIGURE 7. Numbering of childrens

4.2. **Finding neighbors of children.** Once the children are created, it is necessary to create the list of vertices as well as find their neighbors. Vertices are easy to determine. For neighbors, this can be accomplished using an edge-based analysis. Figure 8 shows the two possible configurations as well as appropriate nomenclature for child iv corresponding to main vertex iv. Keeping in mind that there is a one-to-one mapping between vertices, edges, and neighbors of an element, it is possible to determine the three new neighbors $N1$, $N2$, and $N3$ of a child iv corresponding to the vertex $i = \mathsf{ind}(iv)$. As Figure 8 shows, the first neighbor $N1$ is always the ith neighbor of the parent element, the second neighbor $N2$ is always the 4th child, while the third neighbor $N3$ is either the $\mathsf{ind}(iv+2)$ neighbor of the parent element or the next one, depending on whether or not neighbor $\mathsf{ind}(iv+2)$ of the parent element has the same refinement level as the parent element.

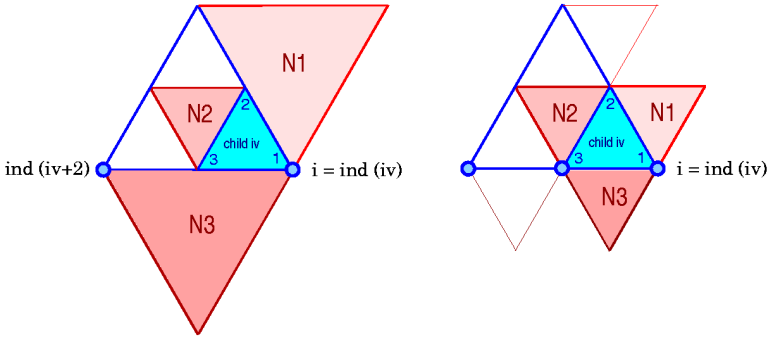


FIGURE 8. Finding children's neighbors after refinement

4.3. **Updating neighbors.** Proceeding edge-wise, the neighbor(s) of the parent element on edge iv need(s) an update of its (their) vertices and list of neighbors after the refinement process is performed. Figure 9 shows the two possible configurations. When the neighbor has the same level of refinement as the element being refined, there is only one neighbor along the edge. The refinement adds a vertex for that neighbor. Morover, the neighbor element i no longer has the element being refined as its neighbor since it is deactivated. The two children iv and $iv+1$ are now the

new neighbors. This can be accomplished by changing the target of one neighbor pointer as well as inserting a new neighbor in the list (Figure 9).

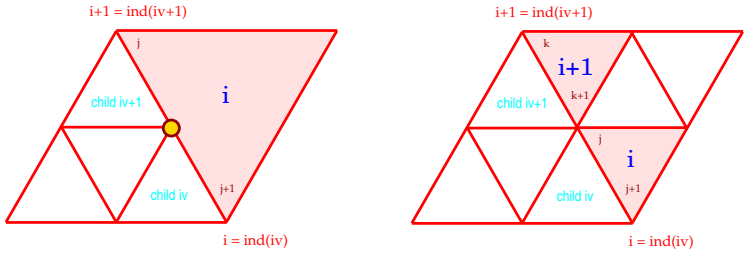


FIGURE 9. Udpating vertices and neighbors of the refined element neighbors.

On the other hand, if the neighbors are smaller than the element being refined, the only change to make is to change the pointer assignment of the two neighbors i and $i+1$ on that edge by making them point to the appropriate children iv and $iv+1$.

4.4. **Deallocating data storage of the parent element.** After refinement, the parent element is no longer in the list of active elements and is not updated during the numerical simulation. It may be useful to free the space allocated to store information about its vertices, neighbors, as well as data values. If this element is reactivated upon coarsening, reallocation of adequate space will be needed.

5. The coarsening process

5.1. **Why coarsening?** For unsteady problems, the regions in need of a finer mesh move with time. In some regions, a refined grid may become useless. To avoid wasting memory and computation, it is necssary to coarsen the grid in those regions.

5.2. **Coarsening strategy (order).** As for refinement, recursive programming is required for coarsening. Coarsening involves an additional difficulty as an element may be coarsened if and only if the neighbors that are finer can be coarsened recursively. However, it may turn out that at the end of a refinement cascade, one neighbor may not be coarsened because of accuracy requirements. Hence, it is first necessary to check that all the elements in the pyramid can be coarsened before actually proceeding to the coarsening of an element. It may be advantageous to proceed with coarsening level-wise from the finest levels to the coarsest levels. This trick avoids the need for a recursive procedure.

5.3. **Coarsening of an element procedure.** Reactivation of a cell is equivalent to the removal of its four children, which must all meet the coarsening criteria. The coarsening process proceeds with the reallocation of the parent vertices and neighbors and their update. Afterwards, data can be transferred from the children to the parents and the children may be deleted.

5.4. **The refinement-coarsening conflict.** Visiting all grid elements once is usually not enough to complete the operations of refinement and coarsening. Visiting the elements several times, on the other hand, raises a new problem. Elements that are refined in a sweep may turn out to qualify for coarsening during the next sweep. Moreover, some elements are refined/coarsened due to constraints (e.g. two neighbors should not differ by more than one refinement level) and not because they meet the refinement/coarsening criterion. This leads to the possiblilty of "chatter" between the refinement and coarsening operations and could result in infinite loops. A way around this is to tag the cells that are refined during the refinement process so as to prevent their coarsening subsequently.

6. Maintaining an active list of elements (and output list)

Rather than having to search the tree each time an operation is to be performed on active elements, it may be useful to create an array of pointers to the active elements in the tree. This array could be used for calculations, output, and refinement/coarsening operations.

7. Pointers and associated problems

Dealing with pointers may be a nightmare. Figure 10 displays a typical setup for a common problem. The allocated memory is created via a pointer `a`. Later, it is accessed via pointer `b` which modifies its content. If pointer `a` is accidentally deallocated (not nullified), then pointer `b` will point to a location that is no longer used for the purpose it was initially set for. The dangerous outcome is this error can go undetected for long if that particular location in memory happens not to be reused right away by the computer's operating system. One day, the memory configuration will be different, the operating system will decide to assign that memory location to another purpose, and the program will abort.

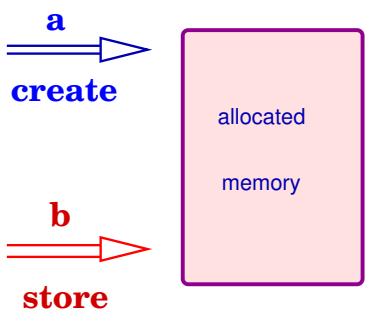
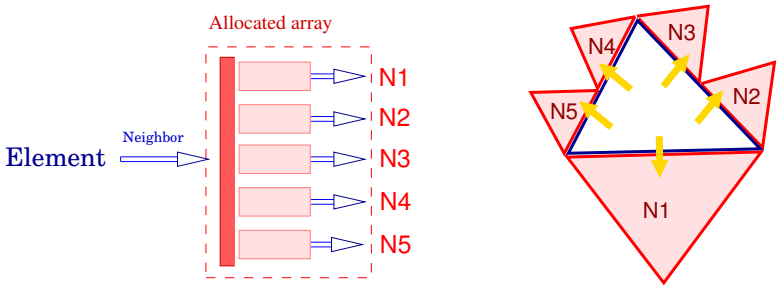


Figure 10. Using two pointers to access allocated memory.

A typical remedy is the following. If a subroutine is called with a pointer argument pointing to a location to be removed, then it is best to use an additional temporary pointer that points to that location to be removed as a calling argument.

8. DO loops and recursive refinement/coarsening

The recursive nature of the refinement/coarsening processes results in some unusual problems for the programmer. Consider the element shown in Figure 11, along with the data structure used for its neighbors (Fortran 95 is considered here). Before coarsening the element, it is necessary to coarsen any of the neighbors that are already finer than the element. Assuming a DO loop is used for that purpose, the process will go as follows: for $i = 1, 5$, if Neighbor N_i is too small, coarsen it. When the program gets to $i = 2$, N_2 will be coarsened and the number of pointers will decrease to 4 while the allocated pointer array will be deallocated, then reallocated to meet the number of neighbors. This means, the DO loop will not terminate !



(a) Neighbor data structure

(b) Element with neighbor pointers

FIGURE 11. Neighbor data structure

A similar problem arises when considering recursive coarsening of active elements. If an element is to be coarsened, this means its three brothers/sisters are to be coarsened simultaneously. Hence a DO loop over the children will abort after the coarsening of the first child.

9. Output and visualization problems

Using a standard visualization software package, such as Tecplot, to draw two-dimensional contour plots may return surprising results. A nonlinear field will generally display gaps near the dangling nodes (Figure 12). This is due to the fact that the contours are drawn element-wise when the Finite-Element data structure is being used with Tecplot (version 9). Using the FE data structure is necessary with unstructured grids such as AMR grids.

A way around this problem is to use a dual grid obtained by partitioning an element having a dangling node into smaller triangles with no dangling nodes. This is shown in Figure 13, where three possible configurations (1, 2, or 3 dangling nodes) are considered. The dual grid is used only for output purposes. Figure 12 shows the discontinuous contours (top) as well as the corrected continuous contour field along with two grids, the AMR grid and the dual grid.

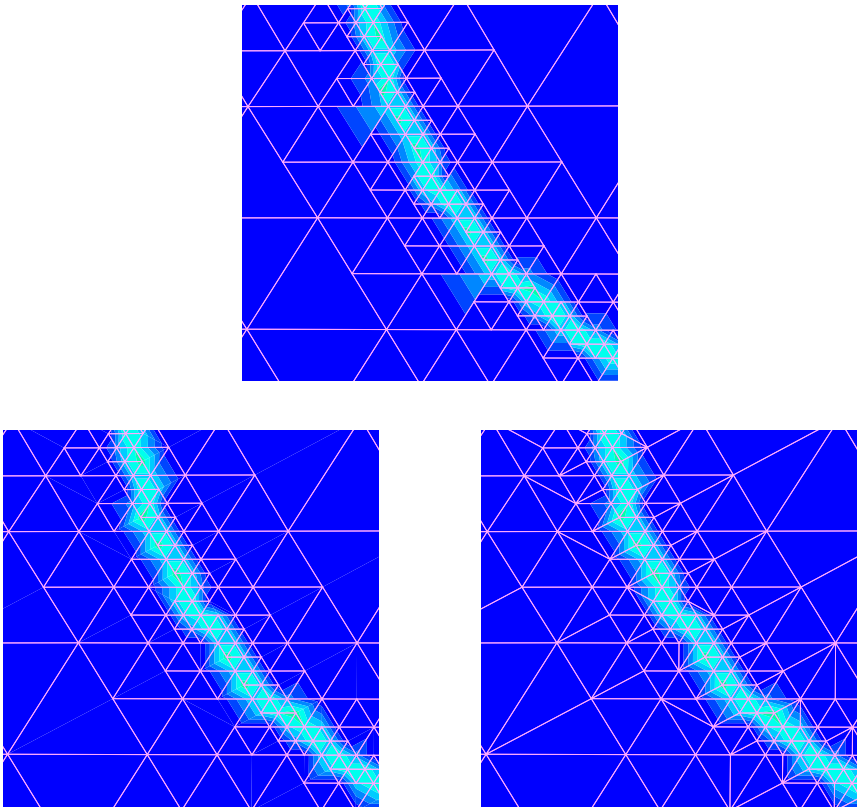


FIGURE 12. Contour plots with AMR and dual grids: top (discontinuous with AMR grid), bottom (continuous with both AMR and dual grids)

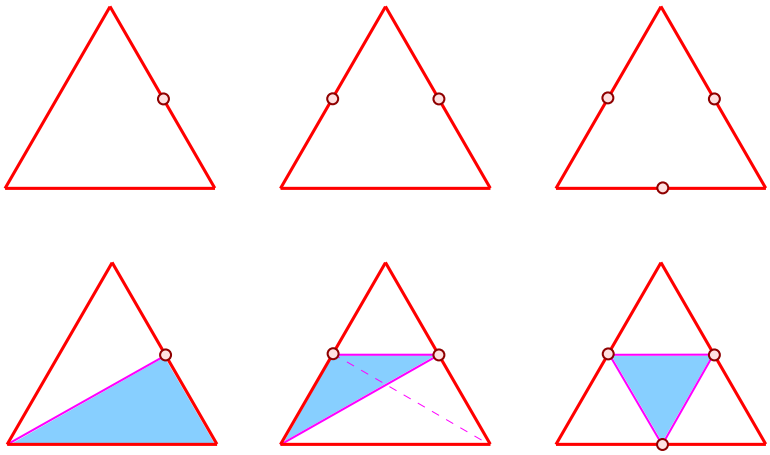


FIGURE 13. Generating the dual grid from the AMR grid.

10. Conclusion

Although there is abundant literature on AMR grids, it seems difficult to find a detailed presentation of the problems related to their implementation. This paper examined the process of implementing a typical AMR procedure and discussed some of the issues that arise during this process. A particular choice of data structure was selected and problems related to that choice were presented and discussed.

References

[1] Akin, E. *Object-Oriented Programming via Fortran 90/95.* Cambridge University Press, 2003.
[2] Berger, M. and Oliger, J. *Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations.* Journal of Computational Physics, Vol 53 (1984) pp. 484–512.
[3] Chung, T. J. *Computational Fluid Dynamics.* Cambridge University Press, 2002.
[4] Mehta, D. P. and Sahni, S. Editors, *Handbook of Data Structures and Applications.* Chapman and Hall/CRC, 2005.
[5] Michaelis, B. and Regg, B. *FEM-Simulation of Laminar Flame Propagation I: Two-Dimensional Flames.* Journal of Computational Physics, Vol 196 (2004) pp. 417–447.
[6] Provatas, N., Goldenfeld, N., and Dantzig, J. *Adaptive Mesh Refinement Computation of Solidification Microstructures Using Dynamic Data Structures.* Journal of Computational Physics, Vol 148 (1998) pp. 265–290.
[7] Scalabrin, L. C. and Azevedo, J. L. F. *Adaptive Mesh Refinement and Coarsening for Aerodynamic Flow Simulation.* International Journal for Numerical Methods in Fluids, Vol 45 (2004) pp. 1107–1122.
[8] Stein, E. (Editor), *Error-Controlled Adaptive Finite Elements in Solid Mechanics.* John Wiley and Sons, 2003.
[9] *Adaptive Mesh Refinement - Theory and Application.* Proceedings of the Chicago Workshop on Adaptive Mesh Refinement methods, Sept. 3–5, 2003. Lecture Notes in Computational Science and Engineering, Springer 2005.

Noureddine Hannoun
Department of Mathematics, University of Tennessee, Knoxville, TN 37996-1300, USA
E-mail address: hannoun@math.utk.edu

Vasilios Alexiades
Department of Mathematics, University of Tennessee, Knoxville, TN 37996-1300 USA, and Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
E-mail address: alexiades@utk.edu