von Karman Institute for Fluid Dynamics

Lecture Series 1990–03

COMPUTATIONAL FLUID DYNAMICS

March 5–9, 1990

ON UNSTRUCTURED GRIDS AND SOLVERS

T.J. Barth

NASA Ames Research Center, USA

On Unstructured Grids and Solvers

Timothy J. Barth
CFD Branch
NASA Ames Research Center
Moffett Field, Ca

Introduction

As technology in computational fluid dynamics (CFD) matures, the complexity of problems being solved by this technology has soared. To become a useful tool for modern day engineers, numerical algorithms capable of handling complex flows about complicated geometries are essential. It is the lack of generality in treating complex geometries which has prevented the routine use of CFD in everyday engineering. In fact, a NASA sponsored workshop ("Future Directions in Surface Modeling and Grid Generation", Dec. 5-7,1989) was held at NASA Ames to address this problem. When a panel of grid generation experts from universities and research laboratories around the U.S. were asked to evaluate which grid technology or technologies (overset grids, patched grids, unstructured grids) would survive several years into the future, the unanimous consensus was that unstructured meshes would be the only surviving technology. Whether or not this is true, today many aspects of unstructured grids and flow solvers are still relatively unexplored and much work needs to be done.

The intent of these notes is to highlight fundamental and state-of-the-art technology in unstructured grids and flow solvers. Roughly half of the notes (sections 1 and 2) discuss algorithms and techniques pertinent to mesh generation. The first section (pages 1-11) on multi-dimensional searching was provided by Marshal Merriam (NASA Ames). This was included because of the fundamental importance of good searching algorithms; especially in mesh generation. As we will see in section 2, many grid generation and grid manipulation schemes rely on fast multi-dimensional searching. In fact, work estimates for many grid generation algorithms casually assume optimal multi-dimensional searching algorithms. In two dimensions, this subject is rather well developed and a few fundamental algorithms are discussed. The other half of the notes (section 3) discuss flow solver technology. In this case, we specialize the discussion to flow solution techniques for the Euler equations which can be derived from the integral form of the equations. Several different algorithms will be discussed ranging from the finite volume equivalent of the Galerkin finite element method to the more sophisticated upwind algorithms. Using some very simple analysis techniques, we can investigate basic properties of these schemes. We will find that this analysis provides strong motivation for use of Delaunay triangulation. A few sample calculations are shown along the way to demonstrate the methods. Hopefully, these notes will convince readers that the world of unstructured grids contains a great deal of structure.

1.0 Multi-dimensional Searching Algorithms

There is a searching problem that comes up quite often in unstructured grid work. Given a large number of nodes whose locations are known, and an arbitrary point P, find the node which is closest to P. In real life, this problem comes up quite often. For example, when we ask for the location of the nearest public phone (gas station, bathroom, etc.) we are asking for a solution to this problem. The solution to the computational problem draws heavily on insights drawn from the real life problem.

This section reviews a number of the standard techniques for this problem. All of them are viewed as special cases of a fairly general technique. A few theoretical results will also be shown to illustrate which of these techniques may be preferred and why.

1.1 Exhaustive search

By far the simplest technique, in every sense, is exhaustive search. It consists of finding the distance (or the distance squared) between the point P and each node Q_i and then finding the smallest of these distances. So, how much does this cost? The computation of distance between P and Q_i is not expensive. Usually the following formula is used.

$$D(P,Q_i) = (x_P - x_i)^2 + (y_P - y_i)^2$$

If there are N nodes, the cost of finding all the distances is roughly $5N$ operations. For Delaunay triangulation this closest node algorithm needs to be exercised about three times for each node. The total cost of triangulation is proportional to the square of the number of nodes. Generally this $O(N^2)$ cost is prohibitive.

Not surprisingly, this isn't the algorithm we use when deciding where the nearest phone is. Just to list the locations of every telephone in the country would be very difficult and somewhat unnecessary. Usually it is more than enough to know where all the phones in the city are. This approach, sorting the telephones geographically, is the basis for all tree searches.

1.2 Bucket search

Since Delaunay triangulation requires many invocations of the closest node algorithm from widely scattered points, it is usually worthwhile to presort the nodes in some way. Perhaps the simplest arrangement starts by partitioning the entire domain into a two dimensional array of rectangular regions called buckets. These are all the same size. The list of nodes can be sorted in such a way that all the nodes in a particular bucket are contiguous. The index of the first and last node of each bucket can be kept in a two dimensional array for later reference.

The original problem was to find out which node is closest to P. It is easy to find out the indices of the bucket in which P appears. For example, if there are M buckets in each direction, point P appears in bucket (j,k) where

$$j = \frac{M(x_p - x_{min})}{(x_{max} - x_{min})} \quad k = \frac{M(y_p - y_{min})}{(y_{max} - y_{min})}$$

Searching this bucket exhaustively is easy, the nodes contained in it are stored contiguously between known indexes. The closest node to P may, or may not, be contained in

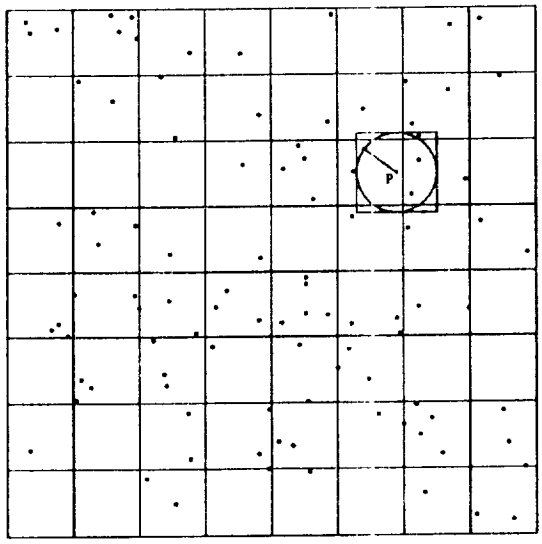

Figure 1.0. Bucket search. First look for the closest node in the same bucket as point P. Then search the buckets touched by the square.

the same bucket as P. In fact, there may be no nodes in the same bucket as P. Therefore other buckets may have to be searched. Which ones?

Computation of the distance from P to the bucket B_{jk} costs about the same as computing the distance to a node. If there are nodes known to be closer than bucket B_{jk}, there is no need to search that bucket. It certainly does not contain any closer nodes. In fact, only a few buckets are even candidates. In the example shown in Fig. 1.0, the bucket containing P has been searched exhaustively. A circle centered at P is drawn through the closest known node. This circle typically covers only a few buckets (six, in this example), sometimes only the bucket in which P appears. In practice, a bounding square of the same radius is easier to use, since truncation can be used to decide the limits for j and k.

So how much does this cost. The original setup is accomplished in two passes, one to count how many nodes are in each bucket, and one to copy the nodes into the appropriate bucket in a duplicate array. The cost is proportional to $O(M^2) + O(N)$ and needs to be done only once for a given set of nodes. The cost of searching the bucket in which P appears is proportional to the number of nodes in that bucket. This is highly dependent on the distribution of nodes and the number of buckets. In the best case, the nodes are evenly distributed, there are N buckets and each bucket contains one node. At most 12 buckets need to be searched in this case, so the cost of each invocation of the closest node algorithm is independent of the number of nodes $O(1)$. A worst case occurs when almost all of the points appear in a single bucket. This generally leads to a lot of looking through empty buckets and a very expensive exhaustive search if the node ultimately lies in the loaded bucket. In this case the bucket search can cost $O(N)$ on each invocation. In practice the situation is usually closer to the worst case. In one example, an airfoil had about 6000 nodes split into 900 buckets. It turned out that 2500 nodes fell in a single bucket (located at the trailing edge). Decreasing the number of buckets made it more expensive to search the loaded bucket. Increasing the number of buckets also slowed things down, due to the problem of searching empty buckets. Nevertheless, a bucket search is a dramatic

improvement over an exhaustive search, both in theory and in practice.

This algorithm is somewhat akin to sorting telephones according to their grid coordinates on a city map. The problem is the constant size of the coordinate squares. A size that is appropriate in an urban area (100 meters say) may be totally inappropriate in a wilderness area. What is needed is a way to size the buckets according to the local node (telephone) density. The quadtree approach is commonly used for this purpose.

1.3 Quadtree search

The quadtree search [1] can be considered as a sort of adaptive bucket search. Many of the features found in the bucket search can be found in the quadtree search also. These include the partitioning of the search space into buckets (called quadrants here), sorting the nodes by quadrant, computation of distance from P to each quadrant to avoid searching them all, and the exhaustive search of certain buckets.

In terms of the telephone analogy, one would look at a list of telephones in the local town. Then, if the town line was close by, one might search nearby towns in the same county. If the county line was nearby, one might search nearby towns in other nearby counties in the same state and so forth to any desired depth. Usually, though, the town line is farther away than the nearest phone in town, so no further looking is necessary.

The setup phase of a quadtree begins exactly like a two by two bucket search. The nodes are sorted in such a way that the nodes in each quadrant are contiguous and the starting and ending indices are stored away for future reference. If the number of nodes in each quadrant is small enough (usually something like 32 nodes), then exhaustive search is practical and quadtree and bucket searches are identical. In most cases there are a lot more nodes than this, so exhaustive search is impractical. A two by two bucket search of each quadrant is then offered as an alternative. Loaded quadrants are divided into sub-quadrants, with a further sort to ensure that nodes within each sub-quadrant are contiguous (as well as nodes within each quadrant).

A sufficient list of things for each quadrant to know is

i. where it is (x_{min}, y_{min})
ii. how big it is (x_{max}, y_{max})
iii. starting and ending indexes for nodes it contains
iv. where all this information is for any subquadrants it contains

This information, and a reordered list of nodes, is the output of the setup phase.

The lookup phase of a quadtree search is similar to that for a bucket search, but with one further refinement. The distance from point P to each quadrant $D(P, q)$ is computed, and these distances are sorted, shortest to longest. The quadrant in which P resides is the closest and the the one diagonally opposite is the farthest away. The four quadrants are searched in this order, and the search is terminated when the next quadrant is farther away than the closest known node. The search of each quadrant is accomplished through exhaustive search, or if the quadrant has too many nodes, through a bucket search.

The example in Fig. 1.1 shows this process graphically. The point P is found in the lower left subquadrant of the upper right quadrant. A search of the subquadrant reveals a node close enough to limit the search to the upper right quadrant. The lower right subquadrant is searched next because it is closer than either of the other two subquadrants. A search of this subquadrant reveals a closer node. The reduced distance further limits

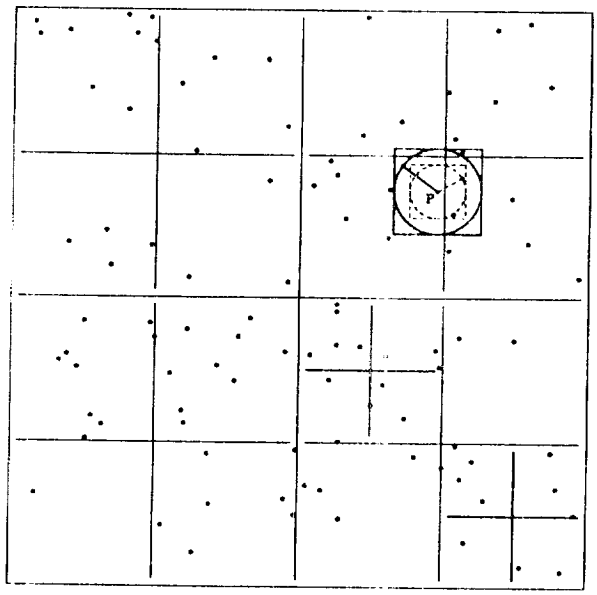

Figure 1.1. Quadtree search. First look for the closest node in the same quadrant as point P. Then search the quadrants touched by the square in order of distance from point P. In this case, the second quadrant searched contains a closer node that shrinks the square and terminates the search.

the search area to that shown by the dotted line. Since all the subquadrants contained in the dotted region have been searched, there is no closer node and the search terminates.

How much does all this cost? In rough terms, probably a lot less than the bucket search. A precise answer depends a lot on the node distribution. In the best case, each quadrant at the bottom level is the same size and exactly full (unlike Fig. 1.1). Thus there are $N/nmin$ quadrants on the bottom level, where $nmin$ is the threshold value. The next level up has $\frac{1}{4}$ as many, leading to an estimate of $\frac{4}{3}N/nmin$ buckets. The number of levels will be $\log_4 N$ or $\frac{1}{2}\log_2 N$. Sorting is required on each level. The total preprocessing cost is proportional to $N \log_2 N$.

The search in this rosy world consists of searching through $\frac{1}{2}\log_2 N$ subdivisions, searching one quadrant at the bottom, and pruning back through $\frac{1}{2}\log_2 N$ levels. The best case has a cost proportional to the number of levels, or $O(\log_2 N)$. This is worse than the best case for the bucket sort.

The worst case is quite similar to the worst case for the bucket sort. It occurs when one point appears in (say) the upper left corner and all the others are tightly clustered in the lower right corner of the domain. This results in lots of empty quadrants, but fewer than for the bucket search. Instead of the costly exhaustive search needed by the bucket search, the quadtree search needs many levels to get down to where the action is. Although the cost of traversing a level is small, there is no limit to the number of levels that might be required. The worst case cost for a pure quadtree search is unbounded! With both the best and worst cases looking inferior to the bucket search, one might expect this algorithm to be in disfavor. There are two reasons why this isn't so. First, the number of levels is usually limited arbitrarily. This keeps the worst case cost from being unbounded. Second, the worst case hardly ever happens. For example, the 6128 node case listed above required

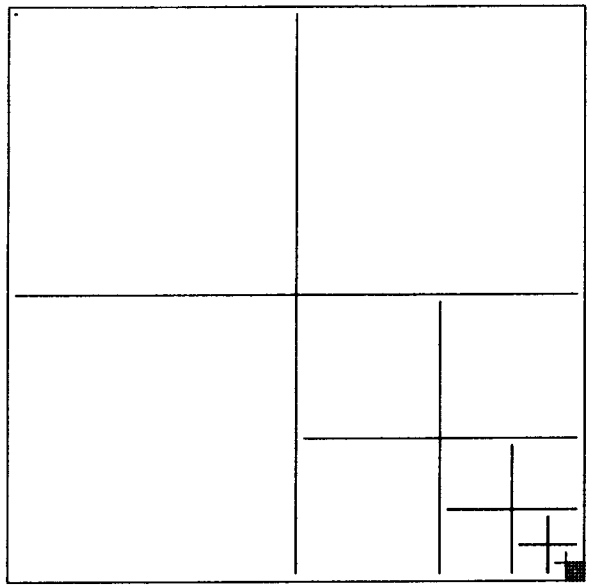

Figure 1.2. Worst case for quadtree search. In principle the number of levels for the tree is unbounded.

653 quadrants on 12 levels. A best case would indicate 256 quadrants on 5 levels. While far from a best case, the quadtree search in this example was substantially less expensive that a bucket search. On an IRIS 4D workstation the running time dropped from 2 minutes to 22 seconds. It is often conjectured that the "average" cost for triangulation is $O(N \log N)$ or even $O(N)$. The later claim is experimental and suggests that the tree traversal cost is negligible (but unbounded!).

1.4 Split Trees

This is a modification to the quadtree approach (quadtree seems to have come first [2]). It contains a few significant improvements. The most striking difference is that the domain is divided into two parts instead of four at each step. The division occurs alternately along lines of constant x and constant y. For this reason the algorithm can be described as a binary, alternating direction tree (no, not a BAD tree). Each half can be searched exhaustively or divided.

Another important difference is that the divisions do not bisect the regions geographically. Ideally, the division should be such that exactly half the points fall in each of the two regions. This could be done by sorting the nodes according to x and choosing the divisor to be the median value of x. Such a choice would always guarantee a best case. A cheaper and simpler alternative is to find the average value of x. The median and the mean are related by the variance, but in general the approximation is acceptable.

A sufficient set of things for each half to know is

i. which nodes it contains
ii. which direction to split it in
iii. where it gets split
iv. where all this information is for the subpartitions.

These things are generated in much the same way as in a quadtree and the resulting data

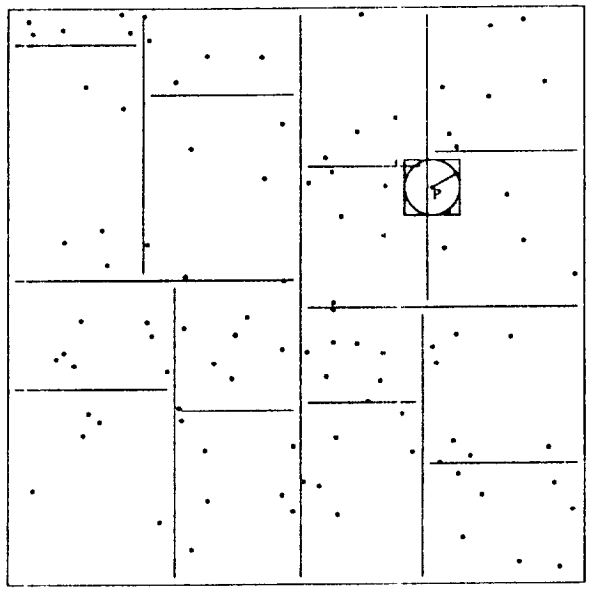

Figure 1.3. Partitions for a splittree search.

structure has many similarities.

In the lookup phase, the advantages of the split tree are significant. Finding the distance to the divisor involves only a single subtraction, which simplifies the computation of distance to each half. Secondly, sorting the two halves by distance involves only a single comparison, which cleans up the code quite a bit.

In the setup phase, it is easier to sort into two piles than four, in fact the node reordering time for each level is cut by a factor of three. On the other hand there may be up to twice as many levels as a quadtree approach, which cuts into the savings. The node reordering time drops by $\frac{1}{3}$, significantly more in three dimensions. This saving is illusory however. A quadtree search could incorporate the same type of setup, sorting first in x and then in y. In any event, this cost is only a few percent of the total.

The cost of finding the mean is $O(N)$ on each of $\log N$ levels for a total cost $O(N \log N)$. If computation of the median is selected, the cost is $O(N \log^2 N)$ which is slightly higher, but promises a best case. In either case, the extra cost reduces the number of levels significantly. This helps a lot in the lookup phase.

In the lookup phase, the time for traversing each level is reduced significantly through simpler distance calculations and cheaper sorting, offset by having slightly more levels. In one case, with 7240 nodes, the quadtree used 8 levels and 777 quadrants. By contrast the split tree (using the mean) generated 705 halves on 11 levels. A best case would be 340 halfs on 9 levels. In this example, and in general, the split tree will get closer to a best case than a quadtree search. On average, the split tree search is slightly faster and more robust than a quadtree search, other things being equal. The advantage lies in slightly simpler coding and in getting closer to a best case. Using a split tree search, triangulation of the 6128 node, three element airfoil case required 11 seconds of CPU time on an IRIS 4D work station. This is less than the time required to read in the nodes and write out the edges.

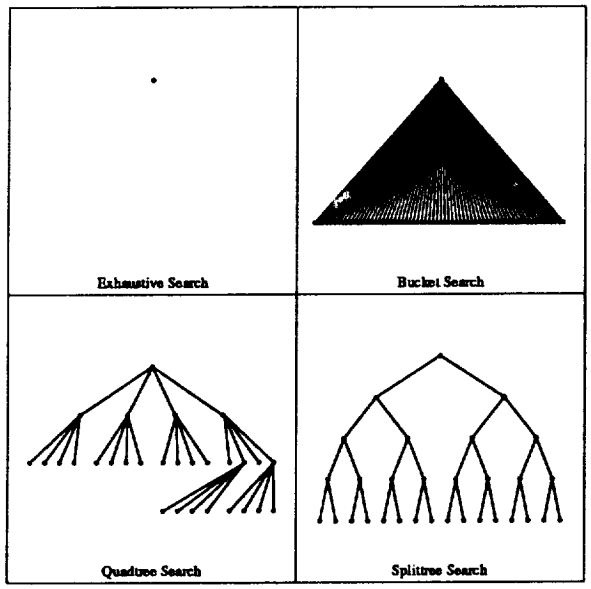

Figure 1.4. All four algorithms are tree searches. They have different branching factors.

1.5 General Trees

All four of the algorithms described above, and many others, can be viewed as tree searches, as shown in Fig. 1.4 The exhaustive search is a degenerate tree with only one level. The bucket search is more of a bush, with two levels and a branching factor of M^2. The quadtree has a branching factor of 4, the binary tree has a branching factor of 2.

A general tree searching algorithm is defined (recursively) as follows:

```
procedure treesearch(tree,N,P,D)
if (N ≤ nmin) then
  ExhaustiveSearch(tree,N,P,D)
else
  NextClosestBranch(P,branch,n,d)
  do while (d < D)
    treesearch(branch,n,P,D)
    NextClosestBranch(P,branch,n,d)
  enddo
endif
return
end
```

The procedure "treesearch" will search through the N nodes in tree and return the distance D to the node nearest to point P. If there are a small number of nodes, an exhaustive search is performed. The procedure "NextClosestBranch" returns the unsearched branch closest to P, along with the number of nodes it contains (n) and its distance from P, (d). If there are no unsearched branches, a large value of d is returned, terminating the search.

Essentially all the work in traversing the tree goes into finding the next closest branch. The relative cost of NextClosestBranch and ExhaustiveSearch determines the optimum

value of *nmin*. This value occurs where the probable work avoided by searching fewer nodes equals the probable work of finding the next closest branch a number of times.

The simplest implementation of NextClosestBranch finds the distance to all the branches at once and sorts these distances. This may be inefficient if the number of branches is large (*e.g.*, bucket search), since the search will probably terminate long before all the branches are searched. A lot of unnecessary distances are computed. On the other hand, this implementation is typical of a quadtree or splittree search.

What is an optimum branching factor? Suppose that the nodes are equally divided among b branches at each of L levels. This means that there will be $\frac{N}{nmin}$ "leaves" which might have to be exhaustively searched. The operative relation is

$$b^L = \frac{N}{nmin}$$

The simplest search consists of L calls to NextClosestBranch and an exhaustive search of *nmin* nodes. Calls to NextClosestBranch cost $O(b)$ and exhaustive search costs $O(nmin)$ so the total cost can be expressed as

$$\text{Cost} = \alpha L b + \beta nmin$$

Where α and β are implementation dependent constants. Combining this with the previous expression gives

$$\text{Cost} = \alpha b \frac{\log(N/nmin)}{\log b} + \beta nmin$$

Differentiating with respect to b gives a root at $\log b = 1$ (but what is the base of the log?). At this value the cost expression is

$$\text{Cost} = \alpha b \log(N/nmin) + \beta nmin$$

Here it is obvious that b should be as small as possible. Since the log of 1 vanishes in any base, $b = 2$ is the best integer value. At this value, the best choice for *nmin* is

$$nmin = \frac{2\alpha}{\beta}$$

Which leads to a total cost of

$$\text{Cost} = 2\alpha(log_2 N - log_2(\frac{2\alpha}{\beta}) + 1)$$

the leading term of which is independent of β. Notice that the number of dimensions does not explicitly appear, though in some cases it might hide in the values of α and β. In a split tree search, for which this analysis is valid, the value of α is independent of the number of dimensions.

The assumptions used in this optimization analysis do not always apply and it may be possible to do better. For example the bucket search, as described above, did not require computing the distance from point P to each of M^2 regions. In the best case, one might expect the cost of finding the next branch to be independent of the number of branches $O(1)$. In that case, the cost uniformly decreases as the number of branches increases, provided that the nodes are evenly divided among the branches.

A general algorithm for setting up a tree search (sorting the nodes) is defined recursively as

```
procedure treesort(startnode,endnode,tree)
n = startnode-endnode + 1
if (n ≥ nmin) then
  partition(nodes,startpartition,endpartition)
  do for each branch
    treesort(startpartition(b),endpartition(b),subtree)
    concatenate(subtree)
  enddo
endif
return
end
```

If the number of nodes is small enough, no further partitioning is required The procedure "partition" divides the search domain into b separate partitions and reorders the nodes so that nodes within a partition are contiguous. Each of these are then treesorted. The procedure "concatenate" accumulates pointers to these subtrees. When all partitions have been treesorted the procedure terminates, returning pointers to its subtrees or ,if $N \leq nmin$, null pointers.

The main cost is in the routine "partition" and has two parts. Dividing the search domain can be almost free if done geographically as in the quadtree, or it can be costly, $O(N)$ or $O(N \log N)$, if an attempt is made to put equal numbers of nodes into each partition. Reordering the nodes usually costs $O(N)$. Since there are $O(\log N)$ levels, the overall cost is $O(N \log N)$ or $O(N \log^2 N)$.

1.6 Balanced Trees

In the quadtree search, a worst case is shown in Fig. 1.2. This occurs because most of the divisions put all the nodes in one quadrant. This leads to a cost that (in principle) is unbounded. How can such a disaster be avoided? In a related question, how can the best case be (perhaps approximately) realized?

The answer in both cases is tree balancing. The trick is to put roughly the same number of nodes in each partition. In the case of the binary trees, two balancing tricks have already been shown. These are the use of the median (exact balance) and the use of the mean (approximate balance). There are several other techniques available.

In the simplest balancing technique, an arbitrary node is chosen as the divisor. This technique (used in QUICKSORT for example) at least limits the worst case to slightly more work than exhaustive search. In the binary tree, only one of its two coordinates would be used. In the quadtree, it would form the intersection of the four quadrants. The node itself would (arbitrarily) belong to the first quadrant. It must be noted that the quadtree is inherently more difficult to balance than a binary tree. For example, it may not be possible to exactly balance an arbitrary set of nodes among the four branches.

A simple variant chooses a small number of nodes (typically three) and sorts them, choosing the median as the divisor. This improves the worst case somewhat and makes

it much less likely to occur. For the quadtree, one might take the x coordinate from the median x value and the y coordinate from the median y value.

One final technique is the histogram (also called radix) sort. This is basically a bucket sort in one dimension. The range of values is divided into a moderate number (perhaps 100) of subranges. With one pass through the nodes, each subrange can know how many nodes fall in that range and what the extreme values are. This can be used to get an excellent approximation for the median at a cost proportional to the number of nodes plus the number of subranges. To realize significant savings over a true sort, the number of subranges must be smaller than $N \log N$.

One drawback to balancing is an increase in complexity and storage: the divisors must be saved. Another can be an increase in the time required to compute distance from P to each partition: truncation can not be used to locate P. Finally, balance is difficult to maintain if nodes are being added or removed as sometimes required. These disadvantages must be weighed against the improved robustness and the reduced number of levels that come with a balanced tree. Experience in two dimensions has shown that the tree setup takes only about 5% of the total triangulation time, even though (in the case of the splittree) its complexity is formally equivalent to the lookup time. This suggests that extra time spent balancing the tree may be less than the time saved by a balanced tree.

1.7 Conclusions

Significant improvements in operation count are possible if tree searches are used instead of exhaustive search. A summary of the methods discussed here is given in table 1. This list is by no means complete, but gives a flavor for the options available. Each of the entries has its place. Exhaustive search is best if the number of nodes is fairly small. Bucket searching is best if the number of nodes is large, but fairly evenly distributed. Quadtree and splittree searches are best if the nodes are highly clustered. Splittree is easier to balance than quadtree, and therefore more robust.

Name	Best Case	Worst Case	Typical Case
Exhaustive	$O(N^2)$	$O(N^2)$	$O(N^2)$
Bucket	$O(N)$	$O(N^2)$	$O(N^2)$
Quadtree	$O(N \log_2 N)$	$O(\infty)$	$O(N \log_2 N)$
Splittree	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$

Table 1 - Complexity of various tree searches

A variety of other choices spring immediately to mind, but the search algorithm no longer dominates the running time anyway. It is difficult to make general statements about the triangulation times since they depend so heavily on the data. In Figure 1.5, though, some experience with various search methods is summarized. Notice that the bucket search in particular varied quite a bit depending on the node distribution. At one extreme it was nearly as fast as the quadtree search, at the other it was only twice as fast as exhaustive search.

In three dimensions it is a different story. Preliminary estimates place the running time for 3-D triangulation in the neighborhood of 1 to 4 Cray CPU hours. For problems

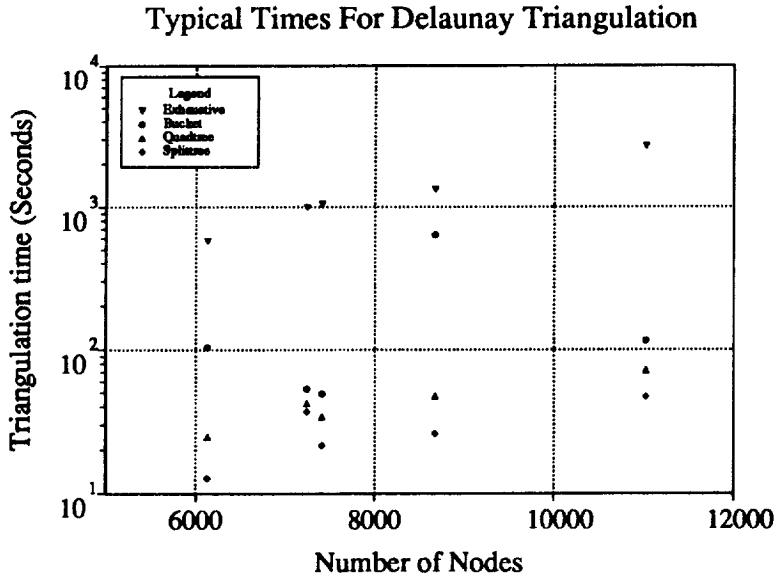

Figure 1.5 Performance of a triangulation algorithm using various search methods. The splittree search was the clear winner. Timings are for an IRIS-4D/70G

of this scale, even small percentage improvements are significant. Furthermore, I/O is less likely to dominate, since it is essentially of $O(N)$ complexity.

2.0 Mesh Generation Methods

In this section several two-dimensional mesh generation techniques are discussed. Particular emphasis is placed on the method of Delaunay triangulation which has proven to be a very powerful technique with rich mathematical theory. Other methods which have become popular in recent years are also discussed. Finally, data dependent meshes and adaptive meshes are reviewed.

In order to concisely describe various mesh generation and eventually flow algorithms, we adopt some elementary graph notation. Specifically, we consider a two-dimensional mesh M of vertices, edges, and cells (faces) denoted by v,e, and, c respectively. For a given mesh vertex, define the degree of a vertex, $d(v)$, as the number of edges incident to v. Similarly, define the degree of a cell $d(c)$ as the number of edges bounding the cell. Figure 2.1 shows a simple mesh generated from random vertices.

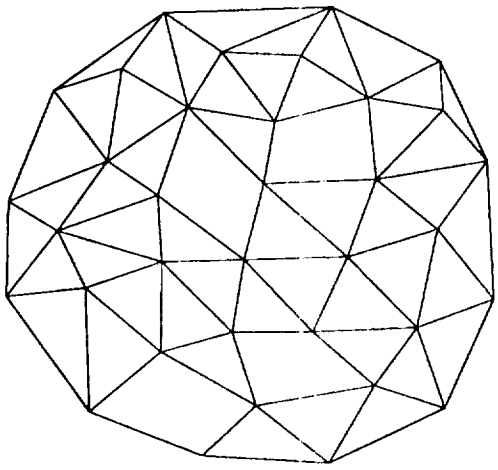

Figure 2.1. Simple Planar Mesh.

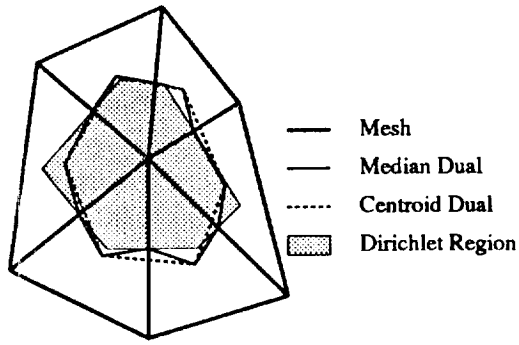

Figure 2.2. Triangulation Duals.

One of the most often used formulas in graph theory precisely relates the number of cells, edges, and vertices $n(c)$, $n(e)$ and $n(v)$ of a planar simple graph:

$$n(c) = n(e) - n(v) + 2 \qquad \text{Euler's formula} \tag{2.0}$$

The usual convention in graph theory is to number all cells including the "infinite" face which extends from the outer boundary to infinity in all directions. We can eliminate the need for an infinite face by describing the outer boundary in terms of boundary edges which share exactly one cell (interior edges share two). We also consider boundary edges which form simple closed curves in the interior of the mesh. These curves serve to describe possible objects embedded in the mesh (in this case, the polygon which they form is not counted as a cell of the mesh). We denote the number of these embedded "holes" by $n(h)$. The modified Euler's formula now reads

$$n(c) + n(v) = n(e) + 1 - n(h) \tag{2.1}$$

Since interior edges share two cells and boundary edges share one cell, the number of interior and boundary edges can be simply related to the number of cells by the following formula:

$$2n(e)_{interior} + n(e)_{boundary} = \sum_{i=3}^{max\ d(c)} i\ n(c)_i \tag{2.2}$$

where $n(c)_i$ denotes the number of cells of a particular edge degree, $d(c) = i$. Note that for pure triangulations T, these formulas can be used to determine, *independent of the triangulation method*, the number of triangles or edges given the number of vertices $n(v)$, boundary edges $n(e)_{boundary}$, and holes $n(h)$:

$$n(c)_3 = 2n(v) - n(e)_{boundary} - 2 + 2n(h) \tag{2.3}$$

or

$$n(e) = 3n(v) - n(e)_{boundary} - 3 + 3n(h) \tag{2.4}$$

This is a well known result for planar triangulations. (For brevity, we will sometimes use N to denote $n(v)$ in the remainder of these notes.) In many cases boundary edges are not explicitly given and the boundary is taken to be the convex hull of the vertices. (To obtain the convex hull in two dimensions, envision placing an elastic rubber band around the cloud of points. The final shape of this rubber band will be the convex hull.) When boundary edges are given, the triangulation is termed a constrained triangulation. Of great importance is the observation that the number of cells and edges is linearly proportional to the number of vertices. This gives a great deal of flexibility in choosing a particular data structure while still maintaining linear storage requirements. Unfortunately, these linear relationships do not hold in three dimensions. As we will note in the next section, even when we consider "good" triangulation methods in 3-D, the number of cells and faces can be $O(N^2)$ in the number of vertices.

Given the mesh M, we informally define a dual mesh M_{Dual} to be any mesh with the following three properties: each vertex of M_{Dual} is associated with a cell (face) of M; each edge of M is associated with an edge of M_{Dual}; if an edge separates two cells, c_i and c_j of M then the associated dual edge connects two vertices of M_{Dual} associated with c_i and c_j. In Fig. 2.2, edges and faces about the central vertex are shown for duals formed from median segments, centroid segments, and by Dirichlet tessellation. The Dirichlet tessellation of a set of points is a pattern of convex regions in the plane, each region being the portion of the plane closer to some given point P of the set of points than to any other point (the shaded region in Fig. 2.2). These Dirichlet regions are sometimes also called Voronoi regions or Thiessen polygons. The Voronoi regions partition the plane into a net which is referred to as the Voronoi diagram. Voronoi diagrams play a useful role in the method of Delaunay triangulation which we now discuss in the next section.

2.1 Delaunay Triangulation

The Delaunay triangulation of a set of points can be defined as the dual of the Dirichlet tessellation of the set. The Delaunay triangulation is formed by connecting two points if and only if their Voronoi regions have a common border segment. This also implies that vertices of the Voronoi polygons are the centers of circumcircles of the triangles. The

Delaunay triangulation possesses several alternate characterizations and many properties of importance. Several of these properties are listed below (see [3,4,5,6,7] for complete details).

(1)*Uniqueness.* The Delaunay triangulation is unique. This assumes that no four vertices are cocircular. The uniqueness follows from the uniqueness of the Dirichlet tessellation.

(2)*The circumcircle criteria.* A triangulation of $N \geq 2$ vertices is Delaunay if and only if the circumcircle of every interior triangle is point-free. Related to the circumcircle criteria is the *incircle* test for four points as shown in Fig. 2.3a-b.

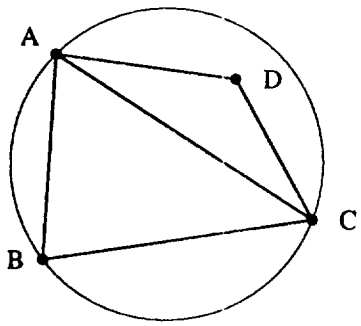

Figure 2.3a. Incircle test for $\triangle ABC$ and point D (true).

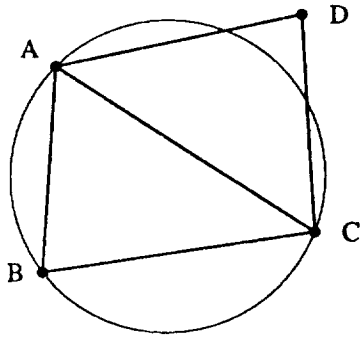

Figure 2.3b. Incircle test for $\triangle ABC$ and point D (false).

This test is true if point D lies interior to the circumcircle of $\triangle ABC$ and is equivalent to asking whether $\angle ABC + \angle CDA$ is less than or greater than $\angle BCD + \angle BAD$. More precisely we have that

$$\angle ABC + \angle CDA = \begin{cases} < 180^\circ & \text{incircle false} \\ 180^\circ & \text{A,B,C,D cocircular} \\ > 180^\circ & \text{incircle true} \end{cases} \tag{2.5}$$

Since interior angles of the quadrilateral sum to 360°, if the circumcircle of $\triangle ABC$ contains D then swapping the diagonal edge from position $A - C$ into $B - D$ guarantees that the new triangle pair satisfy the circumcircle criteria. Furthermore, this process of diagonal swapping is local, i.e. it does not disrupt the Delaunayhood of any triangles adjacent to the quadrilateral.

(2)*Equiangular property.* Delaunay triangulation maximizes the minimum angle of the triangulation. For this reason Delaunay triangulation often called the MaxMin triangulation. This property is also locally true for all adjacent triangle pairs which form a convex quadrilateral. This is the basis for the local edge swapping algorithm of Lawson [8] described below.

(4)*Nearest neighbor property.* An edge formed by joining a vertex to its nearest neighbor is an edge of the Delaunay triangulation. Note that this does not describe all edges of the Delaunay triangulation. This property makes Delaunay triangulation a powerful tool in solving the closest proximity problem.

(5)*Minimal roughness.* The Delaunay triangulation is a minimal roughness triangulation for arbitrary sets of scattered data, Rippa [9]. Given arbitrary data f_i at all vertices of the mesh and a triangulation of these points, a unique piecewise linear interpolating surface can be constructed. The Delaunay triangulation has the property that of all triangulations it minimizes the roughness of this surface as measured by the following Sobolev semi-norm:

$$\int_T \left[\left(\frac{\partial f}{\partial x} \right)^2 + \left(\frac{\partial f}{\partial y} \right)^2 \right] dx\, dy \tag{2.6}$$

This is a interesting result as it does not depend on the actual form of the data. This also indicates that Delaunay triangulation approximates well those functions which minimize this Sobolev norm. One example would be the harmonic functions satisfying Laplaces equation with suitable boundary conditions which minimize precisely this norm. In section 3, we will also prove that a Delaunay triangulation guarantees a maximum principle for the discrete Laplacian approximation (with linear elements). As we will see, this is a quite different result from that obtained by Ciarlet and Riviart which requires that all angles of the mesh by less than $(\pi/2) - \epsilon$ for fixed ϵ.

We now consider several techniques for Delaunay triangulation in two dimensions. These methods were chosen because they perform optimally in rather different situations. For instance, the incremental algorithm works very well for mesh enrichment while Lawson's algorithm is ideal when a complete mesh (not Delaunay) already exists.

(a) Incremental Delaunay Triangulation

For applications such as mesh adaptation and mesh refinement, we desire an algorithm which allows modification of existing (Delaunay) meshes by adding additional vertices. We would also like to use the same algorithm to generate an initial mesh given some arbitrary cloud of points. Incremental algorithms satisfy these requirements and can be made very efficient owing to the nature of the Delaunay triangulation. Incremental algorithms have been proposed by Green and Sibson [4], Bowyer[3], and others. In the following discussion we give a brief description of an incremental algorithm. For simplicity, we begin by assuming that the vertices to be added lie within a bounding polygon of the existing triangulation. If we desire a triangulation from a new set of points, three initial phantom points can always be added which define a triangle large enough to enclose all points to be added. In addition, interior boundaries are usually temporarily ignored for purposes of the Delaunay triangulation. After completing the triangulation, spurious edges are then deleted as a postprocessing step. In any case, the algorithm begins by considering the insertion of a new point P to the existing triangulation T, (see Fig 2.4a).

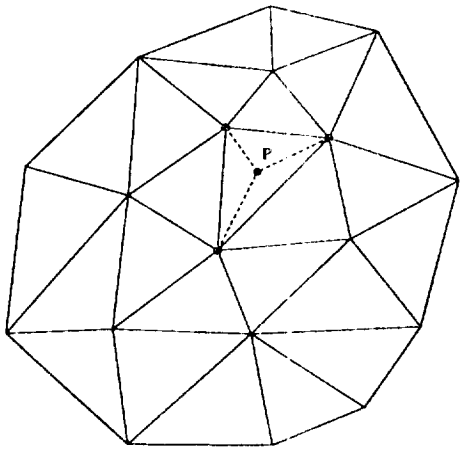

Figure 2.4a. Insertion of new vertex.

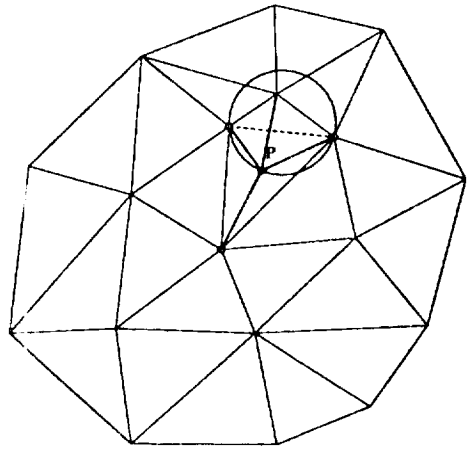

Figure 2.4b. Swapping of suspect edge.

The first step is location, i.e. find the triangle containing point P. Once this is done, three edges are then created connecting P to the vertices of this triangle. If the point falls on an edge, then the edge is deleted and four edges are created connecting to vertices of the newly created quadrilateral. Using the circumcircle criteria it can be shown that the newly created edges (3 or 4) are automatically Delaunay. Unfortunately, some of the original edges are now incorrect. We need to somehow find all "suspect" edges which could possibly fail the circle test. Given that this can be done (described below), each suspect edge is viewed as a diagonal of the quadrilateral formed from the two adjacent triangles. The circumcircle test is applied to either one of the two adjacent triangles of the quadrilateral. If the fourth point of the quadrilateral is interior to this circumcircle, the suspect edge is then swapped as shown in Fig. 2.4b, two more edges then become suspect. At any given time we can immediately identify all suspect edges. To do this, first consider the subset of all triangles which share P as a vertex. One can guarantee at all times that all initial edges incident to P are Delaunay and any edge made incident to P by swapping must be Delaunay. Therefore, we need only consider the remaining edges of this subset which form a polygon about P as suspect and subject to the incircle test. The process terminates when all suspect edges pass the circumcircle test.

Algorithm: Incremental Delaunay Triangulation
Step 1. Locate existing cell enclosing point P.
Step 2. Identify suspect edges.
Step 3. Perform edge swapping of all suspect edges failing the incircle test.
Step 4. Identify new suspect edges.
Step 5. If new suspect edges have been created, go to step 3.

Note that aside from point-cell location, the algorithm is asymptotically optimal: only edges that need to be inserted are created and only edges that need be deleted are removed. The only remaining questions concern point-cell location. Two extreme situations arise in this regard. Typical mesh adaptation and refinement algorithms determine the particular cell for point insertion as part of the mesh adaptation algorithm, thereby avoiding

point location altogether. In the other extreme, initial triangulations of randomly distributed points usually require advanced searching techniques for point location to achieve asymptotically optimal complexity $O(N \log N)$. At first glance this may seem simple given the searching algorithms of section 1. In actuality, it is not quite so simple since the tree structure must be modified each time a point is included. Alternatively, search techniques based on "walking" algorithms are frequently used because of their simplicity. These methods work extremely well when successively added points are close together. The basic idea is start at the location in the mesh of the previously inserted point and move one edge (or cell) at a time in the general direction of the newly added point. In the worst case, each point insertion requires $O(N)$ walks. This would result in a worst case overall complexity $O(N^2)$. For randomly distributed points, the average point insertion requires $O(N^{\frac{1}{2}})$ walks which gives an overall complexity $O(N^{\frac{3}{2}})$. For many applications where successive points tend to be close together, the number of walks is roughly constant and these simple algorithms can be very competitive.

(b) **Divide-and-Conquer Algorithm**

The idea is to partition the cloud of points T (sorted along a convenient axis) into left (L) and right (R) half planes. Each half plane is then recursively Delaunay triangulated. The two halves must then be merged together to form a single Delaunay triangulation. Note that we assume that the points have been sorted along the x-axis for purposes of the following discussion (this can be done with $O(N \log N)$ complexity).

Algorithm: Delaunay Triangulation via Divide-and-Conquer
Step 1. Partition T into two subsets T_L and T_R of nearly equal size.
Step 2. Delaunay triangulate T_L and T_R recursively.
Step 3. Merge T_L and T_R into a single Delaunay triangulation.

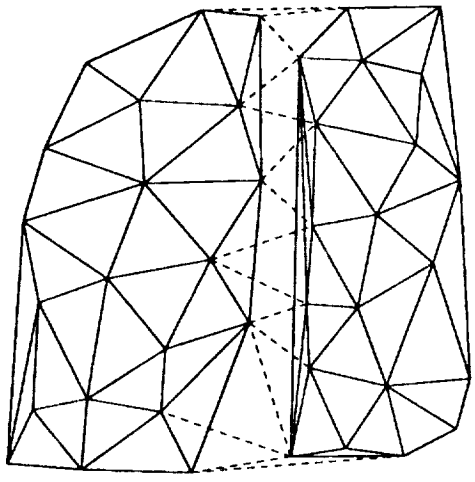

Figure 2.5. Triangulated subdivisions.

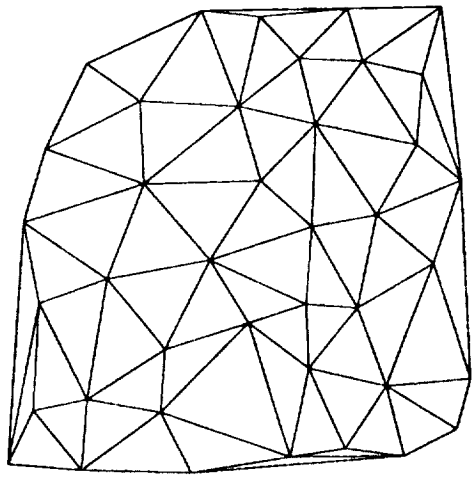

Figure 2.6. Triangulation after merge.

The only difficult step in the divide-and-conquer algorithm is the merging of the left and right triangulations. The process is simplified by noting two properties of the merge:

(1) *Only cross edges (L-R or R-L) are created in the merging process.* Since vertices are neither added or deleted in the merge process, the need for a new R-R or L-L edge indicates that the original right or left triangulation was defective. (Note that the merging process will require the deletion of edges L-L and/or R-R.)

(2) *Vertices with minimum (maximum) y value in the left and right triangulations always connect as cross edges.* This is obvious given that the Delaunay triangulation produces the convex hull of the point cloud.

Given these properties we now outline the "rising bubble" [5] merge algorithm. This algorithm produces cross edges in ascending y-order. The algorithm begins by forming a cross edge by connecting vertices of the left and right triangulations with minimum y value (property 2). This forms the initial cross edge for the rising bubble algorithm. More generally consider the situation in which we have a cross edge between A and B and all edges incident to the points A and B with endpoints above the half plane formed by a line passing through $A - B$, see Fig 2.7 .

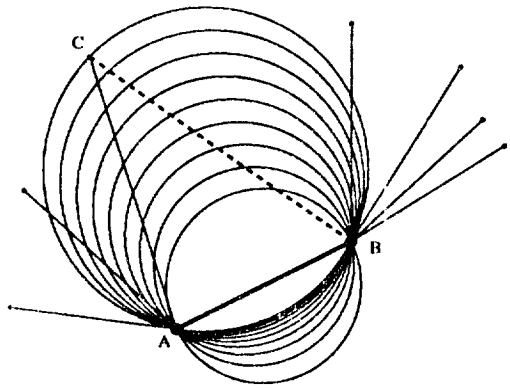

Figure 2.7. Circle of increasing radius in rising bubble algorithm.

This figure depicts a continuously transformed circle of increasing radius passing through the points A and B. Eventually the circle increases in size and encounters a point C from the left or right triangulation (in this case, point C is in the left triangulation). A new cross edge (dashed line in Fig. 2.7) is then formed by connecting this point to a vertex of $A - B$ in the other half triangulation. Given the new cross edge, the process can then be repeated and terminates when the top of the two meshes is reached. The deletion of $L - L$ or $R - R$ edges can take place during or after the addition of the cross edges. Properly implemented, the merge can be carried out in linear time, $O(N)$. Denoting $T(N)$ as the total running time, step 2 is completed in approximately $2T(N/2)$. Thus the total running time is described by the recurrence $T(N) = 2T(N/2) + O(N) = O(N \log N)$.

(c) **Delaunay Triangulation Via Edge Swapping**

This algorithm due to Lawson [8] assumes that a triangulation exists (not Delaunay) then makes it Delaunay through application of edge swapping such that the equiangularity of the triangulation increases. The equiangularity of a triangulation, $A(T)$, is defined as

the ordering of angles $A(T) = [\alpha_1, \alpha_2, \alpha_3, ..., \alpha_{3n(c)_3}]$ such that $\alpha_i \leq \alpha_j$ if $i < j$. We write $A(T^*) < A(T)$ if $\alpha_j^* \leq \alpha_j$ and $\alpha_i^* = \alpha_i$ for $1 \leq i < j$. A triangulation T is globally equiangular if $A(T^*) \leq A(T)$ for all triangulations T^* of the point set. Lawson's algorithm examines all interior edges of the mesh. Each of these edges represents the diagonal of the quadrilateral formed by the union of the two adjacent triangles. One must first check if the quadrilateral is convex so that a potential diagonal swapping can place without edge crossing. If the quadrilateral is convex then the diagonal position is chosen which maximizes the local equiangularity. This is equivalent to maximizing the minimum angle of the two adjacent triangles. Lawson's algorithm continues until the mesh is locally equiangular everywhere. It is easily shown that the condition of local equiangularity is equivalent to satisfaction of the incircle test described earlier. Therefore a mesh which is locally equiangular everywhere is a Delaunay triangulation. Note that each new edge swapping (triangulation T^*) insures that the global equiangularity increases $A(T^*) > A(T)$. Because the triangulation is of finite dimension, this guarantees that the process will terminate in a finite number of steps.

As Babuška and Aziz [10] point out, from the point of view of finite elements the MaxMin (Delaunay) triangulation is not essential. What is essential is that no angle be too close to 180°. In other words, triangulations which minimize the maximum angle are more desirable. These triangulations are referred to as MinMax triangulations. These meshes can also be generated with Lawson's algorithm using similar arguments as given previously. In this case define $\overline{A}(T) = [\alpha_1, \alpha_2, \alpha_3, ..., \alpha_{3n(c)_3}]$ such that $\alpha_i \geq \alpha_j$ if $i < j$. A global optimum is reached when $\overline{A}(T^*) \geq \overline{A}(T)$ for all possible T^* of the point set. Again we use Lawson's algorithm which guarantees that with each new edge swapping we have that $\overline{A}(T^*) < \overline{A}(T)$. Thus the process will terminate in a finite number of steps.

Iterative Algorithm: Triangulation via Lawson's Algorithm

```
swapedge = true
While(swapedge)do
  swapedge = false
  Do (all interior edges)
    If (adjacent triangles form convex quadrilateral)then
      Swap diagonal to form T*.
      If (MaxMin/MinMax criteria satisfied)then
        T = T*
        swapedge = true
      EndIf
    EndIf
  EndDo
EndWhile
```

Figures 2.8 and 2.9 present Delaunay (MaxMin) and MinMax triangulation for 100 random points.

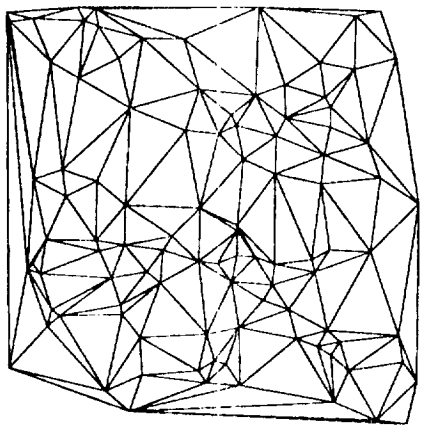

Figure 2.8 Delaunay Triangulation.

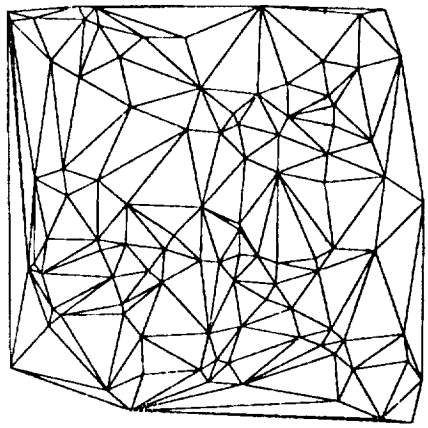

Figure 2.9 MinMax Triangulation.

(d) **Advancing Front Triangulation**

Another algorithm for performing Delaunay triangulation is the advancing front method developed by Merriam [17]. Here the idea is to start with a known boundary edge and form a new triangle by joining both endpoints to one of the interior points. This may generate up to two additional edges, providing they aren't already part of another triangle. After all the boundary edges have been incorporated into triangles, the new edges will appear to be a (somewhat ragged) boundary. This moving boundary is often called an advancing front. The process continues until the front vanishes. The problem here is to make the triangulation Delaunay. This can be done by taking advantage of the *incircle* property; the circumcircle of a Delaunay triangle contains no other points. This allows the appropriate point to be selected iteratively as shown in Fig. 2.10.

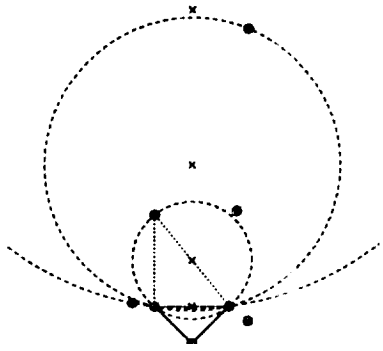

Figure 2.10. A straightforward iteration procedure selects the node which generates the smallest circumcircle for a given edge. The absence of nodes inside the circumcircle establishes convergence.

The iteration begins by selecting any node which is on the desired side of the given edge. If there are no such nodes, the given edge is part of a convex hull. Next, the

circumcircle is constructed which passes through the edge endpoints and the selected node. Finally, check for nodes inside this circle. If there are any, replace the selected node with the node closest to the circumcenter and repeat the process. When the circumcircle is empty of nodes, connect the edge endpoints to the selected node. A prescribed boundary edge may not meet the Delaunay condition for edges as seen in the divide and conquer algorithm. This means that the resulting triangulation may not be Delaunay around the edges. On the other hand, the selection of nodes from only one side of the edge helps prevent edges that cross a boundary (the so called "breakthrough" problem). Generally the resulting triangulation is as close to Delaunay as one can get within the confines of the given nodes and the given boundary edges.

An interesting variant of this approach specifies no interior nodes, only boundary nodes. Proceeding as before, one soon encounters a proposed triangle with an unacceptably large circumcircle. In this case, an additional node is generated inside the circumcircle positioned such that the resulting triangle has a suitably small circumcircle. One degree of freedom remains, and this is usually used to make the resulting triangle equilateral. Another good choice would be to make it a right triangle. The resulting triangulation may not be Delaunay because the nodes are not all present at the outset. A few passes of diagonal swapping takes care of this in two dimensions but there is no good three dimensional analog except to retriangulate. Additionally a number of special cases require care to avoid crossed edges. These also appear in more traditional advancing front algorithms like those implemented by Löhner et. al. [11]. Sometimes it is more convenient to specify interior nodes. For example in certain adaptive refinement algorithms, the positions of the new nodes are given. In other cases it is more convenient not to specify the exact location of interior nodes, perhaps relying on adaptive refinement to correct any deficiencies. The advancing front Delaunay approach even allows these two approaches to be mixed, perhaps to specify a node distribution along a singular line. Thus there is no clear preference for for algorithms which do/do not specify interior nodes in advance.

The algorithms described above do not cover the complete repertoire of methods for Delaunay triangulation. Conspicuously missing is the method of Bowyer [3] and algorithms of this type which compute the Voronoi diagram directly. This is the method implemented by Baker [7] in three dimensions. Note that in both two and three dimensions, a good point placement algorithm is also crutial to the success of the Delaunay triangulation method. We also have not discussed methods which obtain the Voronoi diagram in $\mathbf{R}^d$ by calculating the convex hull of the point set lifted into $\mathbf{R}^{d+1}$ by way of the unit paraboloid transformation $x_{d+1} = x_1^2 + x_2^2 + ... + x_d^2$. Figure 2.11 demonstrates this technique for a 1-D Voronoi diagram. In this figure the intersection of the dashed lines with the x-axis indicates the boundary of the Voronoi regions. This technique is further discussed in [12].

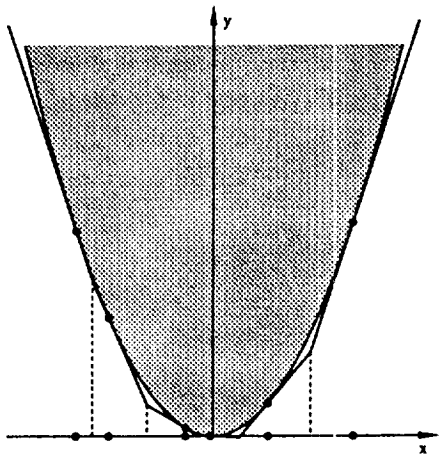

Figure 2.11. Voronoi diagram in $\mathbf{R}^1$ from convex hull in $\mathbf{R}^2$.

Before leaving the topic of Delaunay triangulation, we now return to the question of how many edges and vertices we may expect to generate in the construction of a tetrahedral mesh from N vertices. A rather depressing result of Klee [13] states that in a d-dimensional space where $M(d,N)$ is the maximum number of vertices of the Voronoi diagram on N points we have the following bound on $M(d,N)$

$$\lceil d/2 \rceil ! N^{\lceil d/2 \rceil} \leq M(d,N) \leq 2\lceil d/2 \rceil ! N^{\lceil d/2 \rceil}, \quad for\ d\ even$$

and

$$\frac{(\lceil d/2 \rceil - 1)!}{3} N^{\lceil d/2 \rceil} \leq M(d,N) < \lceil d/2 \rceil ! N^{\lceil d/2 \rceil}, \quad for\ d\ odd$$

so in three dimensions we have

$$\frac{1}{3}N^2 \leq M(3,N) < 2N^2$$

Note that vertices of the 3-D voronoi regions are in one-to-one correspondence with cells of the tetrahedral mesh. Further note that eqn. (2.2) holds in 3-D as well. This establishes that in a worst case scenario, both the number of cells and faces can be $O(N^2)$ for 3-D tetrahedral meshes. Fortunately, vertex configurations which give rise to $O(N^2)$ tetrahedra are extremely rare. In practice the number of tetrahedra exceeds vertices by a factor of 5-6.

2.2 Greedy Triangulation

As explained in Preparata and Shamos [6], a greedy method is one that never undoes what it did earlier. The greedy triangulation continually adds edges compatible with the current triangulation (edge crossing not allowed) until the triangulation is complete, i.e. Euler's formula is satisfied. One objective of a triangulation might be to choose a set of edges with shortest total length. The best that the greedy algorithm can do is adopt a local criterion whereby only the shortest edge available at that moment is considered

for addition to the current triangulation. (This does not lead to a triangulation with shortest total length.) Note that greedy triangulation easily accommodates constrained triangulations containing interior boundaries and a nonconvex outer boundary. In this case the boundary edges are simply listed first in the ordering of candidate edges. The entire algorithm is outlined below.

Algorithm: Greedy Triangulation
Step 1. Initialize triangulation T as empty.
Step 2. Compute $\binom{n}{2}$ candidate edges.
Step 3. Order pool of candidate edges.
Step 4. Remove current edge e_s from ordered pool.
Step 5. If(e_s *does not intersect edges of* T) add e_s to T
Step 6. If(*Euler's formula not satisfied*) go to Step 4.

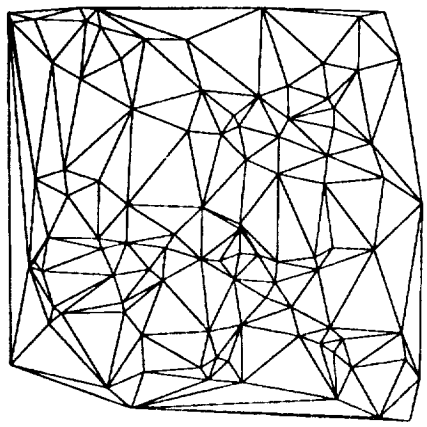

Figure 2.12 Delaunay Triangulation.

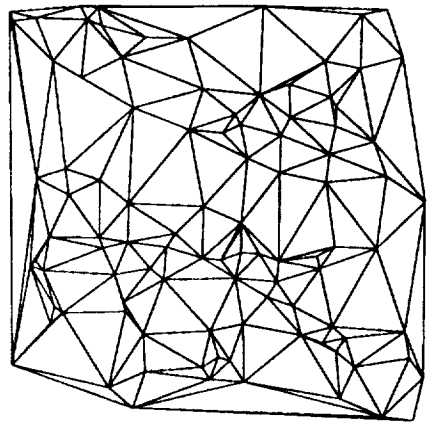

Figure 2.13 Greedy Triangulation.

Figures 2.12 and 2.13 contrast the Delaunay and greedy algorithm. The lack of angle control is easily seen in the greedy triangulation. The greedy algorithm suffers from both high running time as well as storage. In fact a naive implementation of Step 5. leads to an algorithm with $O(N^3)$ complexity. Efficient implementation techniques are given in Gilbert [14] with the result that the complexity can be reduced to $O(N^2 \log N)$ with $O(N^2)$ storage.

2.3 Data Dependent Triangulation and Mesh Adaptation

Without question, one of the most valuable attributes of unstructured meshes is their ability to locally adapt and refine the mesh to better resolve features of the flowfield: shocks, contacts, boundary layers, vortices, etc. Perhaps it is not quite as clear how this should always be done, i.e. by adding/deleting grid points, moving grid points, changing connections. Even less clear is how to decide when adaptation should be done. After the adaptation, we need to transfer a solution computed on a previous mesh to the new mesh in a conservative manner. This is not a trivial operation. In a later section we describe a strategy for doing this.

Data Dependent Triangulation [15,16]

Unlike mesh adaptation, a data dependent triangulation assumes that the number and position of vertices is fixed and unchanging. Of all possible triangulations of these vertices, the goal is to find the best triangulation under data dependent constraints. In Nira, Levin, and Rippa [15], they consider several data dependent constraints together with piecewise linear interpolation. In order to determine if a new mesh is "better" than a previous one, a local cost function is defined for each interior edge. Two choices which prove to be particularly effective are the JND (Jump in Normal Derivatives) and the ABN (Angle Between Normals). Using their notation, consider an interior edge with adjacent triangles T_1 and T_2. Let $P(x,y)_1$ and $P(x,y)_2$ be the linear interpolation polynomials in T_1 and T_2 respectively:

$$P_1(x,y) = a_1x + b_1y + c_1$$

$$P_2(x,y) = a_2x + b_2y + c_2$$

The JND cost function measures the jump in normal derivatives of P_1 and P_2 across a common edge with normal components n_x and n_y.

$$s(f_T,e) = |n_x(a_1 - a_2) + n_y(b_1 - b_2)|, \qquad \text{JND cost function}$$

The ABN measures the acute angle between the two normals formed from the two planes P_1 and P_2. Again using the notation of [15]:

$$s(f_T,e) = \theta = \cos^{-1}(A), \quad A = \frac{a_1a_2 + b_1b_2 + 1}{\sqrt{(a_1^2 + b_1^2 + 1)(a_2^2 + b_2^2 + 1)}}, \qquad \text{ABN cost function}$$

The next step is to construct a global measure of these cost functions. This measure is required to decrease for each legal edge swapping. This insures that the edge swapping process terminates. The simplest measures are the l_1 and l_2 norms:

$$R_1(f_T) = \sum_{edges} |s(f_T,e)|$$

$$R_2(f_T) = \sum_{edges} s(f_T,e)^2$$

Recall that a Delaunay triangulation would result if we were to choose cost functions which maximize the minimum angle between adjacent triangles (Lawson's algorithm). (We also claim that this is equivalent to minimizing the roughness of the triangulation.) Although we would like to obtain a global optimum for all cost functions, this could be very costly in many cases. An alternate strategy is to abandon the pursuit of a globally optimal triangulation in favor of a locally optimal triangulation. Once again Lawson's algorithm is used. Note that in using Lawson's algorithm, we require that the global measure decrease at each edge swap. This is not as simple as before since each edge swap can have an effect on other surrounding edge cost functions. Nevertheless, this domain of influence is very small and easily found.

Iterative Algorithm: Data Dependent Triangulation via Modified Lawson's Algorithm

swapedge = true
While(swapedge)do
  swapedge = false
  Do (*all interior edges*)
    If (*adjacent triangles form convex quadrilateral*)then
      Swap diagonal to form T^*.
      If $(R(f_{T^*}) < R(f_T))$then
        $T = T^*$
        swapedge = true
      EndIf
    EndIf
  EndDo
EndWhile

Because edge swapping only occurs when $R(f_{T^*}) < R(f_T)$ the method terminates in a finite number of steps. Figures 2.14-2.15 plot the Delaunay triangulation of 100 random vertices in a unit square and piecewise linear contours of $(1 + \tanh(9y - 9x))/9$ on this mesh. The exact solution consists of straight line contours with unit slope.

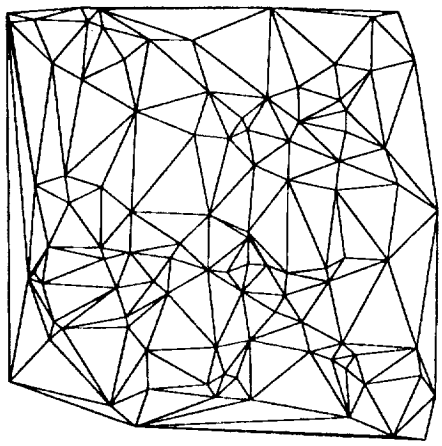

Figure 2.14 Delaunay Triangulation.

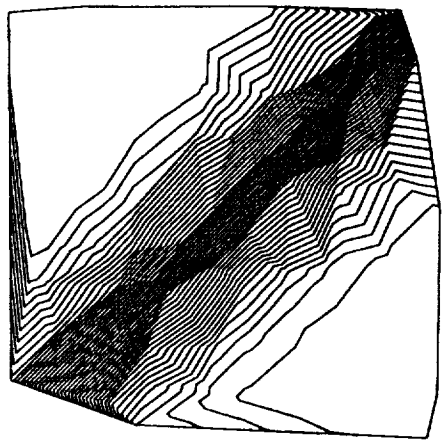

Figure 2.15 Piecewise Linear Interpolation of $(1 + \tanh(9y - 9x))/9$

In Figs. 2.16-2.17 the data dependent triangulation and solution contours using the JND criteria and l_1 measure suggested in [15] are plotted.

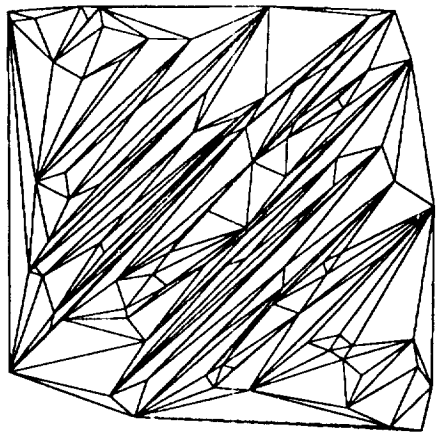

Figure 2.16 Data Dependent Triangulation.

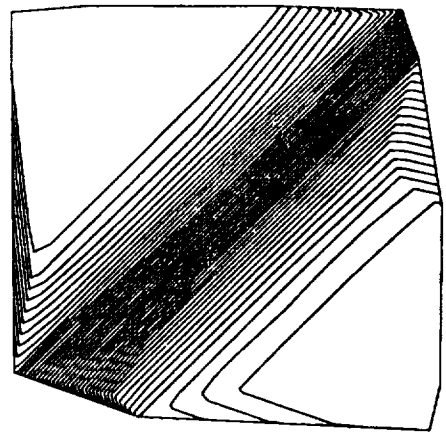

Figure 2.17 Piecewise Linear Interpolation of $(1 - \tanh(9y - 9x))/9$

Note that the triangulations obtained from this method are not globally optimal and highly dependent on the order in which edges are accessed. Several possible ordering strategies are mentioned in [16].

Mesh Refinement Schemes

Conventional mesh adaptation algorithms perform two basic operations. The first operation is to locate all regions requiring mesh refinement or coarsening. The second is to generate a new mesh either by modification of the existing mesh or by regeneration of all or part of the mesh using new mesh generation parameters. The most difficult part of the process is the identification of regions to be refined. Essentially all methods for locating these regions attempt to estimate the magnitude of the solution error. This can be done using one of several classical and nonclassical methods:

(1) Interpolation error estimates. These estimates are discussed in Ciarlet [19], Ciarlet and Raviart [20], Carey and Oden [18]. A typical finite element interpolation estimate relates the Sobolev norm of degree s of the error between the interpolating polynomial $\widetilde{u}^h$ and the exact solution u in terms of the Sobolev norm of the exact solution:

$$\|u - \widetilde{u}^h\|_s \leq c \, \frac{h^{k+1}}{\rho^s} \|u\|_{k+1} \tag{2.6}$$

where k is the degree of the element polynomial, h is maximum element circumcircle in the mesh, and ρ is the minimum element incircle in the mesh. The solution is assumed regular in $k+1$ derivatives. Usually s is chosen equal to zero (which gives an l_2 norm).

$$\|u - \widetilde{u}^h\|_0 \leq c \, h^{k+1} \|u\|_{k+1} \tag{2.7}$$

For piecewise linear elements similar results can be obtained using a semi-norm of the exact solution:

$$\|u - \widetilde{u}^h\|_0 \leq c_1 \, h|u|_1 \tag{2.8}$$

$$\|u - \tilde{u}^h\|_0 \leq c_2 \; h^2 |u|_2 \tag{2.9}$$

In actual implementation, additional assumptions must be made. The first concerns locality. We really desire to use these estimates on a cell by cell basis. This is justified since typical interpolation operators have compact support. In this light, h is usually chosen as the cell diameter. This implies a certain degree of uniformity over the local support. The second assumption concerns the calculation of the $k+1$ Sobolev norm (semi-norm) of the exact solution. This norm is usually not available and must be approximated from the numerical solution. This is not easily done since k degree elements do not generate $k+1$ degree derivatives. To approximate these derivatives requires considering a group of elements and some type of finite difference approximation. This leads to added complexity and lower confidence in the accuracy of the estimate. The other approach is to use lower order estimates such as (2.8) rather than (2.9). Unfortunately, refinements using (2.8) for linear elements may not give the best refinement criteria. The third assumption concerns the overall strategy. We really desire to estimate the error between the exact solution and the numerical solution. The interpolation estimates (2.6) only gives part of the estimate.

$$u - u^h = \underbrace{(u - \tilde{u}^h)}_{\text{interpolation error}} + (\tilde{u}^h - u^h)$$

Ideally, we would like to use error estimates for the entire scheme of the form

$$\|u - u^h\|_0 \leq c \; h^{k+p} |u|_{k+1} \tag{2.10}$$

This would provide use with refinement criteria best suited to the scheme.

(2) Error Extrapolations. One classic technique used in numerical methods to estimate the solution error is to compute solutions on a single mesh using two or more schemes of different order accuracy. A second technique uses a single scheme but computes solutions on multiple meshes of different density. Both of these methods have implementation problems. The first technique requires that both schemes be uniformly of different order accuracy. This can be difficult to guarantee, especially at boundaries and singular points. The second technique requires uniform refinement or coarsening of the mesh. Refinement is easily done, but requires a significant increase in computer memory. Mesh coarsening lessens the burden on memory but the task of finding a "uniformly" coarser unstructured mesh is not trivial (unless the original mesh was constructed by uniform refinement of coarser meshes).

(3) Heuristic Indicators. This is most easily done by interrogating solution quantities (entropy, total enthalpy, vorticity, etc.) or solution gradients (density gradients, pressure gradients, velocity/Mach number gradients, etc.). Figures 2.13 and 2.14 show adaptive meshes generated by Dr. D.J. Mavriplis, ICASE / NASA Langley. These meshes were adapted based on a Mach number gradient function. The incremental Delaunay algorithm was used to retriangulate the added points. Because of the adaptation criteria, a majority of the points added are located in the boundary layer region.

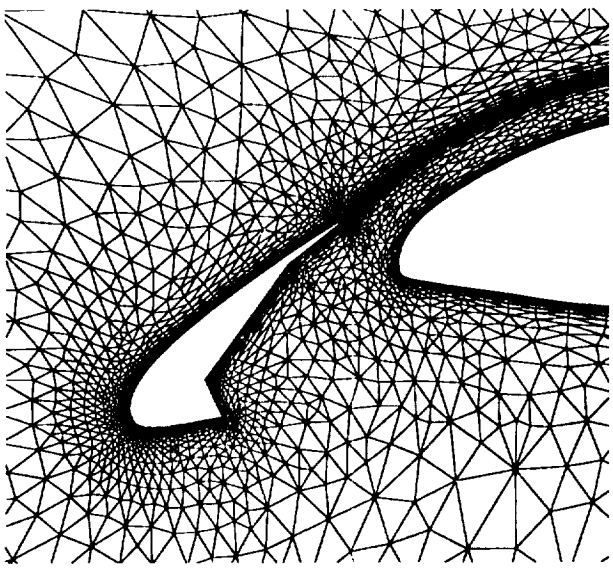

Figure 2.13 Original Mesh.
(Courtesy D.J. Mavriplis)

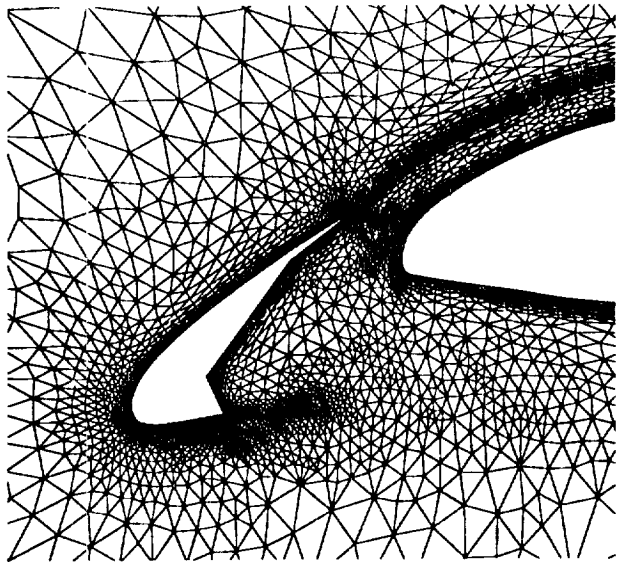

Figure 2.14 Adapted Mesh.
(Courtesy D.J. Mavriplis)

This method works remarkably well when the solution error is dominated by a few flowfield features. This is especially true when shock waves and other flowfield discontinuities are present. In this case, interpolation estimates can lead to poor mesh adaptations.

2.4 Conclusions

A wide variety of mesh generation techniques have been examined and there appear to be no clear cut winners in the grid generation race. Note that we have not discussed mesh generation in three dimensions. Mesh generation in three dimensions is much less developed. This is probably because the underlying theory in three dimensions is not as well understood. Indeed, the complexity of problems in three-dimensional computational geometry is much higher than in two dimensions. We hinted at this by giving the result of Klee. Simple operations such as edge swapping do not have 3-D counterparts. Three dimensional grid generation will no doubt be a challenging field for some time to come.

3.0 Finite-Volume Unstructured Mesh Solvers

In the remaining sections, we consider the numerical solution of the Euler equations on unstructured grids. We begin first by considering scalar conservation law equations in the integral, divergence, and weak form. We then show that the spatial discretization obtained from Galerkin's method applied to the weak form of the equations is identical to a finite volume discretization of the integral equations on the median dual. We then examine the discrete Laplacian operator. Although the continuous Laplacian operator has a maximum principle, we find in general that this is not true of the discrete Galerkin (Finite Volume) discretization on arbitrary meshes. To our relief, we do discover that meshes constructed via Delaunay triangulation do guarantee a discrete maximum principle. Upwind algorithms for scalar conservation law equations are then considered. We view these from the point of view of Godunov's method [23]. That is to say, we think of cell averaged quantities rather that pointwise values. This introduces the concept of reconstruction. Schemes for scalar equations which are based on high order reconstruction are then constructed. When discontinuities are present, monotonicity enforcement is required (at least approximately); the best way to do this is still an open question. Stability of the upwind algorithms for scalar equations is outlined. Again, we show that a discrete maximum principle can be invoked. We then turn to the Euler equations themselves. We review the schemes of Jameson and Mavriplis which are essentially Galerkin's method with added smoothing. Upwind schemes for the Euler equations are then considered. These schemes require the choice of "numerical" flux function. This function serves to correctly sort out waves entering and leaving the control volume. Several choices are available. We will only consider a few. Finally, we show some recent results obtained by high order reconstruction.

3.1 Integral, Divergence, and Weak Form of Scalar Conservation Law Equations

The choice of numerical methods used to solve a conservation law equation can be influenced by the form in which the equation is presented. A conservation law asserts that the rate of change of the total amount of a substance with density z in a fixed region Ω is equal to the flux $\mathbf{F}$ of the substance through the boundary $\partial\Omega$.

$$\frac{\partial}{\partial t}\int_\Omega z \; da + \int_{\partial\Omega} \mathbf{F}(z)\cdot\mathbf{n} \; dl = 0 \qquad \text{(integral form)}$$

A finite difference practitioner would apply the divergence theorem to this integral equation and let the area of Ω shrink to zero thus obtaining the divergence form of the equation.

$$\frac{\partial}{\partial t} z + \nabla\cdot\mathbf{F}(z) = 0 \qquad \text{(divergence form)}$$

The finite element practitioner constructs the divergence form then multiplies by an arbitrary test function ϕ and integrates by parts.

$$\frac{\partial}{\partial t}\int_\Omega \phi z \; da - \int_\Omega \nabla\phi\cdot\mathbf{F}(z) \; da + \int_{\partial\Omega} \phi\mathbf{F}(z)\cdot\mathbf{n} \; dl = 0 \qquad \text{(weak form)}$$

These three forms can produce seemingly different numerical schemes. In reality these methods are closely related. Some differences do appear in the handling of boundary conditions, solution discontinuities, and nonlinearities. When considering flows with discontinuities, the integral form appears advantageous since conservation of fluxes comes for free and the proper jump conditions are assured. At discontinuities, the divergence form of the equations implies satisfaction in the sense of distribution theory. Consequently, at discontinuities special care is needed to construct finite difference schemes which produce physically relevant solutions. Because the test functions have compact support, the weak form of the equations also guarantees satisfaction of the jump conditions over the extent of the support.

3.2 Galerkin and Finite Volume Discretizations

Although the integral and weak forms of the equations appear different, it is not difficult to show that they can produce indentical discretizations. In fact, we can show this for a Galerkin discretization (linear elements) of a general model equation with diffusion $\mu > 0$:

$$\frac{\partial}{\partial t} z + \nabla \cdot \mathbf{F}(z) = \nabla \cdot \mu \nabla z$$

Multiplying by a test function ϕ and integrating over Ω by parts produces the weak form of the equation.

$$\frac{\partial}{\partial t}\int_\Omega \phi z \, da - \int_\Omega \nabla\phi \cdot \mathbf{F}(z) \, da + \int_{\partial\Omega} \phi \, \mathbf{F}(z) \cdot \mathbf{n} \, dl = -\int_\Omega \mu \nabla\phi \cdot \nabla z \, da + \int_{\partial\Omega} \mu\phi \nabla z \cdot \mathbf{n} \, dl \quad (3.1)$$

In the finite element method, the entire domain is first divided into smaller elements. In this case the elements are triangles T_j, such that $\Omega = \cup T_j$, $T_j \cap T_l = \emptyset$, $l \neq j$. In Fig. 3.1a we show a representative vertex with adjacent neighbors. (To simplify the discussion in the remainder of these notes, we adopt the convention that the index "j" in subsequent equations refers to a global index of a mesh whereas the index "i" always refers to a local index. We sometimes use v_0 which is interpreted globally as v_j.) The linear variation of the solution in each triangle T_j can be locally expressed in terms of the three local nodal values of the solution, $u^h_{T_j,i}$, $i = 1,2,3$, and three element shape functions n_i, $i = 1,2,3$.

$$u^h(x,y)_{T_j} = \sum_{i=1}^{3} n_i(x,y) \; u^h_{T_j,i} \qquad \text{(local representation)}$$

Each element shape function n_i represents a linear surface which takes on unit value at v_i and vanishes at the other two vertices of the triangle as well as everywhere outside the triangle. The solution can also be expressed globally in terms of nodal values of the solution and global shape functions.

$$u^h(x,y) = \sum_{nodes} N_j(x,y) \; u^h_j \qquad \text{(global representation)}$$

In this form the global shape functions are piecewise linear pyramids which are formed from the union of all local shape functions with have unit value at v_j. These global shape functions also enjoy compact support, i.e. they vanish outside the region Ω_j formed from the union of all triangles incident to v_j. A global shape function for vertex v_j is shown in Fig. 3.1b.

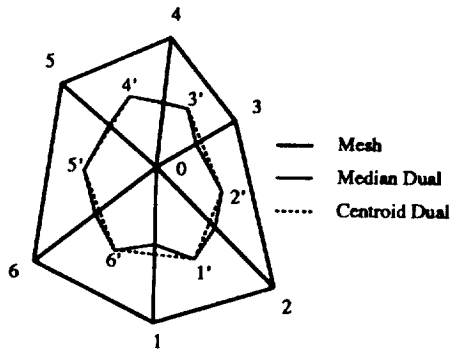

Figure 3.1a. Local mesh with centroid and median Duals.

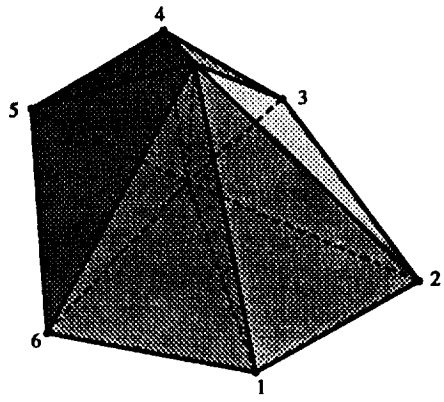

Figure 3.1b. Global shape function for vertex v_0 (not labelled).

The Galerkin finite element method assumes that the class of test functions is identical to the class of functions approximating the solution. The simplest test functions of this sort are the individual shape functions. To obtain a Galerkin discretization for a typical vertex v_j, we simply set $\phi^h = N_j$ and evaluate (3.1) in Ω_j. Since ϕ vanishes on $\partial\Omega_j$ we can simplify eqn. (3.1) and obtain the discrete Galerkin approximation:

$$\frac{\partial}{\partial t}\int_{\Omega_j} \phi^h z^h \, da - \int_{\Omega_j} \nabla\phi^h \cdot \mathbf{F}(z^h) \, da = -\int_{\Omega_j} \mu \nabla\phi^h \cdot \nabla z^h \, da \tag{3.2}$$

Before we evaluate eqn. (3.2), we need to introduce more notation concerning the geometry of Fig. 3.1a. In Fig. 3.2 we give the index and normal convention which will be used throughout these notes.

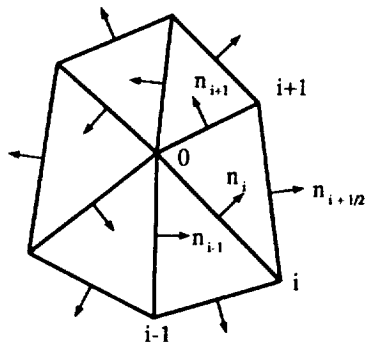

Figure 3.2. Vertex v_0 and adjacent neighbors.

We assume a periodic index i, i.e. $[1,2,...,d(v_0),1,2,...]$ and denote the triangle with vertices 0, i, and $i+1$ as triangle $T_{i+1/2}$. We also use this convention for other quantities such as areas and gradients which are computed in $T_{i+1/2}$. We also find it convenient to define normals for straight edges which are scaled by the length of the edge. These will be denoted by $\vec{\mathbf{n}}$. In this notation we have a simple formula for the gradient of the numerical solution in a triangle $T_{i+1/2}$

$$\nabla z^h_{i+1/2} = \frac{-1}{2A_{i+1/2}} \left(z^h_0 \vec{\mathbf{n}}_{i+1/2} + z^h_i \vec{\mathbf{n}}_{i+1} - z^h_{i+1} \vec{\mathbf{n}}_i \right) \tag{3.3}$$

Similarly, we can calculate the gradient of the test function in each triangle (replace z by ϕ in the previous formula with $\phi_0 = 1, \phi_i = 0, \phi_{i+1} = 0$).

$$\nabla \phi_{i+1/2} = \frac{-1}{2A_{i+1/2}} \vec{\mathbf{n}}_{i+1/2}$$

The discrete form of eqn. (3.2) is now written as

$$\frac{\partial}{\partial t} \int_{\Omega_j} \phi z^h \, da + \sum_{i=1}^{d(v_0)} \frac{\vec{\mathbf{n}}_{i+1/2}}{2A_{i+1/2}} \cdot \int_{T_{i+1/2}} \mathbf{F}(z^h) \, da = \sum_{i=1}^{d(v_0)} \frac{\vec{\mathbf{n}}_{i+1/2}}{2A_{i+1/2}} \cdot \int_{T_{i+1/2}} \mu \nabla z^h \, da \tag{3.4}$$

We can now evaluate the flux integral either by exact integration (when possible) or numerical quadrature. In this case we assume the latter.

$$\int_{T_{i+1/2}} \mathbf{F}(z^h) \, da = \frac{1}{3} A_{i+1/2} \left(\mathbf{F}(z^h_0) + \mathbf{F}(z^h_i) + \mathbf{F}(z^h_{i+1}) \right)$$

The diffusion term is also evaluated by numerical quadrature with ∇z^h constant in $T_{i+1/2}$.

$$\int_{T_{i+1/2}} \mu \nabla z^h \, da = A_{i+1/2} \, \overline{\mu}_{i+1/2} \nabla z^h_{i+1/2}$$

This simplifies (3.4) considerably.

$$\frac{\partial}{\partial t} \int_{\Omega_j} \phi z^h \, da + \sum_{i=1}^{d(v_0)} \frac{1}{6} \vec{\mathbf{n}}_{i+1/2} \cdot \left(\mathbf{F}(z^h_0) + \mathbf{F}(z^h_i) + \mathbf{F}(z^h_{i+1}) \right) = \sum_{i=1}^{d(v_0)} \frac{1}{2} \overline{\mu}_{i+1/2} \, \vec{\mathbf{n}}_{i+1/2} \cdot \nabla_{i+1/2} z^h \tag{3.5}$$

Let's recap what we have so far. Equation (3.5) represents a Galerkin discretization of the model equation assuming piecewise linear functions. Note that as far as the geometry is concerned, only the exterior normals of Ω_j appear. Conspicuously absent are the normal vectors for interior edges. This strengthens our confidence that we can show an equivalence to a finite volume discretization on *nonoverlapping* control volumes. To do this we need to exploit geometrical identities in order to show an equivalence to a consistent boundary integration in the finite volume method. Reference [21] points out that the flux term can

be manipulated using the identity $\sum_{i=1}^{d(v_0)} \vec{\mathbf{n}}_{i+1/2} = \mathbf{0}$ into a form in which the relevant geometry is any path connecting adjacent triangle centroids:

$$\begin{aligned}\sum_{i=1}^{d(v_0)} \frac{1}{6}\vec{\mathbf{n}}_{i+1/2}\cdot\left(\mathbf{F}(z_0^h)+\mathbf{F}(z_i^h)+\mathbf{F}(z_{i+1}^h)\right) &= \sum_{i=1}^{d(v_0)} \frac{1}{6}\left(\mathbf{F}(z_0^h)+\mathbf{F}(z_i^h)\right)\cdot\left(\vec{\mathbf{n}}_{i+1/2}+\vec{\mathbf{n}}_{i-1/2}\right)\\ &= \sum_{i=1}^{d(v_0)} \frac{1}{6}\left(\mathbf{F}(z_0^h)+\mathbf{F}(z_i^h)\right)\cdot\int_{\mathbf{R}_{i-1}}^{\mathbf{R}_{i+1}}\mathbf{n}\,dl\\ &= \sum_{i=1}^{d(v_0)} \frac{1}{2}\left(\mathbf{F}(z_0^h)+\mathbf{F}(z_i^h)\right)\cdot\int_{\frac{1}{3}(\mathbf{R}_0+\mathbf{R}_i+\mathbf{R}_{i-1})}^{\frac{1}{3}(\mathbf{R}_0+\mathbf{R}_i+\mathbf{R}_{i+1})}\mathbf{n}\,dl\end{aligned} \tag{3.6}$$

The diffusion term also simplifies.

$$\sum_{i=1}^{d(v_0)} \frac{1}{2}\vec{\mathbf{n}}_{i+1/2}\cdot\overline{\mu}_{i+1/2}\nabla_{i+1/2}z^h = \sum_{i=1}^{d(v_0)} \overline{\mu}_{i+1/2}\nabla_{i+1/2}z^h\cdot\int_{\frac{1}{2}(\mathbf{R}_0+\mathbf{R}_i)}^{\frac{1}{2}(\mathbf{R}_0+\mathbf{R}_{i+1})}\mathbf{n}\,dl \tag{3.7}$$

To obtain a single consistent path for the integrations appearing in eqns. (3.6) and (3.7) requires that the path pass through the centroid of each triangle and the midside of each interior edge. If we simply connect these points, we see that this is precisely the median dual of the mesh. This dual completely covers the domain (no holes), thus it represents a consistent and conservative finite volume discretization of the domain which is spatially equivalent to the Galerkin approximation. We can now write the scheme in a finite volume fashion ($\mathbf{R}'_{i+1/2} = \frac{1}{3}(\mathbf{R}_0+\mathbf{R}_i+\mathbf{R}_{i+1})$, $\mathbf{R}_i^m = \frac{1}{2}(\mathbf{R}_0+\mathbf{R}_i)$)

$$\frac{\partial}{\partial t}\phi^h z^h\,da + \sum_{i=1}^{d(v_0)}(\mathbf{H}\cdot\vec{\mathbf{n}})_i = 0$$

where $\mathbf{H}$ is the numerical flux of the finite volume discretization.

$$\begin{aligned}(\mathbf{H}\cdot\vec{\mathbf{n}})_i =& \frac{1}{2}(\mathbf{F}(z_0^h)+\mathbf{F}(z_i^h))\cdot\int_{\mathbf{R}'_{i+1/2}}^{\mathbf{R}'_{i+1/2}}\mathbf{n}\,dl\\ &-\overline{\mu}_{i-1/2}\nabla_{i-1/2}z^h\cdot\int_{\mathbf{R}'_{i-1/2}}^{\mathbf{R}_i^m}\mathbf{n}\,dl - \overline{\mu}_{i+1/2}\nabla_{i+1/2}z^h\cdot\int_{\mathbf{R}_i^m}^{\mathbf{R}'_{i+1/2}}\mathbf{n}\,dl\end{aligned} \tag{3.8}$$

Conclusion: *The spatial discretization produced by the Galerkin finite element scheme with linear elements has an equivalent finite volume discretization on nonoverlapping control volumes with bounding curves which pass through the centroid of triangles and midside of edges. One such set of control volumes satisfying these constraints is the median dual.*

We now need to ask if the time integrals produce identical "mass" matrices for the Galerkin finite element and finite volume schemes. The answer to this question is no. In

fact, these matrices are not the same in one space dimension. The Galerkin mass matrix for a simple 1-D mesh with uniform spacing produces a row of the mass matrix with the following weights:

$$\frac{\partial}{\partial t}\int_{\Omega_j} \phi^h z^h dx = \frac{\partial}{\partial t}\Delta x \frac{1}{6}(z_{j-1} + 4z_j + z_{j+1}) \qquad \text{(Finite Element)}$$

The finite volume scheme on "median" dual produces the following weights:

$$\frac{\partial}{\partial t}\int_{\Omega_j} z^h dx = \frac{\partial}{\partial t}\Delta x \frac{1}{8}(z_{j-1} + 6z_j + z_{j+1}) \qquad \text{(Finite Volume)}$$

Although the finite volume matrix gives better temporal stability, the finite element mass matrix is more accurate.

3.3 Conditions for a Discrete Maximum Principle for the Laplacian Operator.

In this section, we ask under what conditions the finite volume (Galerkin) discretization of the Laplacian $\nabla^2 z^h$ with linear elements possesses a discrete maximum principle, i.e. the value at an interior vertex does not exceed the minimum or maximum of all adjacent neighbors. In the paper by Ciarlet and Raviart [22], they state and prove that sufficient conditions for a discrete maximum principle for ∇^2 are that *all angles of the triangulation are* $\leq (\pi/2) - \epsilon$ *for fixed* ϵ. In this section we will show that this condition is overly restrictive. We prove a less restrictive geometric condition on the discretization: *meshes constructed by the method Delaunay triangulation guarantee a discrete maximum principle.* Although we are analyzing the Laplacian operator $\nabla \cdot \nabla z = \nabla^2 z^h$ with linear elements, we could assume the more general diffusion operator of the previous section, $\nabla \cdot \mu \nabla z^h$, without additional difficulties. Note that the discrete Laplacian is a linear operator depending only on immediate neigbors. We can write the discretization in terms of neighboring values of the solution (including z_0^h) and constant weights.

$$\nabla^2 z^h = \sum_{i=0}^{d(v_0)} w_i z_i^h \tag{3.9}$$

For constant weights, a maximum principle for arbitrary data is guaranteed if

$$\sum_{i=0}^{d(v_0)} w_i = 0, \quad \text{with } w_i \geq 0 \text{ for } i \geq 1 \quad \text{and} \quad w_0 < 0 \tag{3.10}$$

Since we have $\nabla^2 z^h = 0$, we can solve for z_0^h.

$$z_0^h = \frac{\sum_{i=1}^{d(v_0)} w_i \, z_i^h}{\sum_{i=1}^{d(v_0)} w_i} \tag{3.11}$$

This implies a maximum principle.

$$\min\left(z_1^h, z_2^h, ..., z_{d(v_0)}^h\right) \leq z_0^h \leq \max\left(z_1^h, z_2^h, ..., z_{d(v_0)}^h\right) \tag{3.12}$$

As we have seen before, the discrete form can be written as

$$\nabla^2 z^h = \frac{1}{A_{\Omega_j}} \sum_{i=1}^{d(v_0)} \nabla z_{i+1/2}^h \cdot \vec{\mathbf{n}}_{i+1/2} \tag{3.13}$$

We also have the expression for the gradient in each triangle (eqn. 3.3).

$$\nabla z_{i+1/2}^h = \frac{-1}{2A_{i+1/2}} \left(z_0^h \vec{\mathbf{n}}_{i+1/2} + z_i^h \vec{\mathbf{n}}_{i+1} - z_{i+1}^h \vec{\mathbf{n}}_i \right) \tag{3.14}$$

Putting these expressions together, we have the full discretization of the Laplacian.

$$\nabla^2 z^h = \frac{1}{2A_\Omega} \sum_{i=1}^{d(v_0)} z_0^h \frac{-(\vec{\mathbf{n}}_{i+1/2} \cdot \vec{\mathbf{n}}_{i+1/2})}{A_i} + z_i^h \left(\frac{(\vec{\mathbf{n}}_{i-1/2} \cdot \vec{\mathbf{n}}_{i-1})}{A_{i-1/2}} - \frac{(\vec{\mathbf{n}}_{i+1/2} \cdot \vec{\mathbf{n}}_{i+1})}{A_{i+1/2}} \right) \tag{3.15}$$

First, we need to prove that all coefficients sum to exactly zero. It should be clear that this condition is satisfied given that all quadratures integrate linear functions exactly. It should also be clear that the coefficient weight multiplying z_0^h is always negative. The only remaining task is to find conditions such that

$$\frac{(\vec{\mathbf{n}}_{i-1/2} \cdot \vec{\mathbf{n}}_{i-1})}{A_{i-1/2}} - \frac{(\vec{\mathbf{n}}_{i+1/2} \cdot \vec{\mathbf{n}}_{i+1})}{A_{i+1/2}} \geq 0 \tag{3.16}$$

After applying elementary vector and trigonometric identities, we find that this term is written most simply in terms of the tangent of the angles δ and γ as described by Figs. 3.2 and 3.3.

$$\frac{(\vec{\mathbf{n}}_{i-1/2} \cdot \vec{\mathbf{n}}_{i-1})}{A_{i-1/2}} - \frac{(\vec{\mathbf{n}}_{i+1/2} \cdot \vec{\mathbf{n}}_{i+1})}{A_{i+1/2}} = 2\left(\tan\delta + \tan\gamma\right)$$

We have that $\tan(\delta) + \tan(\gamma) = \tan(\delta+\gamma)(1 - \tan(\delta)\tan(\gamma))$. This implies that $\delta + \gamma \leq 180°$.

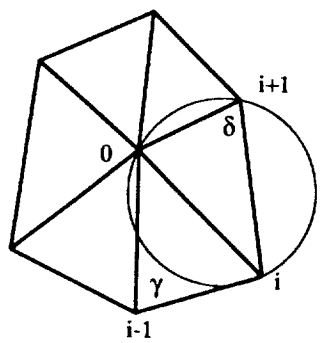

Figure 3.3. Circumcircle test for adjacent triangles.

If we consult equation (2.5), we see that this is precisely a characterization of the circumcircle test given in section 2 for Delaunay triangulation. These conditions are required of all interior edges of the mesh. Therefore, we are in a position to state the following conclusion:

Conclusion: *A maximum principle for the discrete Laplacian operator with linear elements is guaranteed if the triangulation is Delaunay.*

3.4 Upwind Algorithms for Scalar Conservation Law Equations.

In this section we consider upwind algorithms for scalar hyperbolic equations. In particular, we concentrate on upwind schemes based on Godunov's method [23] and defer the discussion of "upwind" schemes based on the Petrov-Galerkin formulations (SUPG, GLS) or the fluctuation decomposition method [24] to the lectures of Profs. Johnson and Deconinck, respectively.

The development presented here follows many of the ideas developed previously for structured meshes. For example, in the extension of Godunov's scheme to second order accuracy in one space dimension, van Leer [25] developed an advection scheme based on discontinuous piecewise linear distributions together with Lagrangian hydrodynamics. Colella and Woodward [26] and Woodward and Colella [27] further extended these ideas to include discontinuous piecewise parabolic approximations with Eulerian or Lagrangian hydrodynamics. Harten et. al. [28,29] later extended related schemes to arbitrary order and clarified the entire process. These techniques have been applied to multi-dimensional structured meshes via individual one-dimensional application along coordinate lines. This has proven to be a highly successful approximation but does not directly extend to unstructured meshes. In previous work, we have constructed schemes for unstructured meshes using discontinuous piecewise linear distributions of the solution in each cell (Barth and Jespersen [30]). In a recent paper (Barth and Frederickson [31]), we construct high order numerical schemes for unstructured meshes using a reconstruction algorithm of arbitrary order. Much of the discussion presented here is taken from these papers.

We begin with a general formulation of the finite volume upwind method using piecewise polynomial functions. Given the general form, we can then go back and look at the simpler cases, i.e. the first and second order upwind schemes. Hopefully, this will provide a clear prospective of the method.

General Upwind Formulation

In developing a general finite-volume scheme, we consider the integral form of a conservation law equation for some domain, Ω and its tessellation $\mathcal{T}(\Omega)$ comprised of cells, c_j, $\Omega = \cup c_j$, $c_k \cap c_j = \emptyset, \quad k \neq j$.

$$\frac{\partial}{\partial t} \int_{c_j} u \, da + \int_{\partial c_j} \mathbf{F}(u) \cdot \mathbf{n} \, dl = 0 \tag{3.17}$$

In the introduction to section 3.1, we gave the textbook definition of a conservation law equation, i.e. an equation that asserts that the rate of change of the total amount of a substance with density u in a fixed control volume is equal to the flux of the substance through the boundary. In order to motivate Godunov's method, we note that the total amount of the substance in a cell c_j is equal to the integral cell average of the substance density, $\overline{u}$, multiplied by the volume (area in 2-D) of c_j.

$$\int_{c_j} u \, da = \overline{u}_j A_j$$

In Godunov's method, we treat these cell averages as the fundamental unknowns. Of course if we were given an function and asked to calculate the integral average of this function in each cell, the result would be a piecewise constant function. It would appear that we have thrown away a great deal of information. The question is how much information can we recover (reconstruct) given the piecewise constant cell averages? This is an inverse problem. Given an averaging operator, we would like to construct its inverse. The amount of information we can reconstruct hinges critically on smoothness of the original function. Clearly at the level of the numerical scheme, we can only afford to think about functions of finite dimension, for example polynomials of degree k or less. We denote this class of functions by $\mathcal{P}_k$. If the original function is a member of this class, then we should expect that we can construct a unique inverse, that is to say we can perform an exact reconstruction. We refer to this as the property of "k-exactness". Unfortunately, the majority of functions of interest do not reside in $\mathcal{P}_k$. In this case, we seek a reconstruction operator which approximates these functions as well as possible in this restricted class of polynomials. We also choose to use C^{-1} discontinuous polynomials (of degree k or less) from cell to cell. We do this because we require that the finite dimensional reconstruction operator be of compact support. (In fact, we require that the entire scheme be of compact support.) This is a major complaint of finite element schemes (Galerkin and Petrov-Galerkin) using continuous elements. These schemes produce a nondiagonal "mass matrix" which implies global support. This is a holdout from elliptic equations. Clearly for hyperbolic equations this is physically and mathematically incorrect.

In the extension of Godunov's method to high order, we consider distinct piecewise polynomials in each control volume (expanded about the centroid)

$$u^k(x,y) = \sum_{m+n \leq k} \alpha_{(m,n)} P_{(m,n)}(x - x_c, y - y_c) \tag{3.18}$$

where $P_{(m,n)}(x-x_c, y-y_c) = (x-x_c)^m (y-y_c)^n$. These are the reconstructed polynomials given cell averages of the numerical solution. Because these polynomials are discontinuous from cell to cell, along a cell boundary two distinct values of the solution can be obtained. To resolve this nonuniqueness, the flux is replaced by a "numerical flux function", $\overline{\mathbf{F}}(u_+, u_-, \mathbf{n})$, which when given two solution states produces a single unique flux. We choose to use numerical flux functions derived via mean value linearizations. As we will see, this simplifies the analysis and allows us to prove stability and monotonicity. The choice of flux function is subject to the general constraint that when both states are identical that the true flux is obtained.

$$\overline{f}(u, u, \mathbf{n}) = \mathbf{F}(u) \cdot \mathbf{n}$$

Approximating (3.17) by piecewise polynomials and numerical flux function, we obtain a finite dimensional approximation to the exact integral equation.

$$\frac{d}{dt}\int_{c_j} u^k \, da + \int_{\partial c_j} \overline{f}(u_+^k, u_-^k, \mathrm{n}) \, dl = 0 \tag{3.19}$$

Given the consistency condition for the numerical flux, it should be clear that (3.19) is exact whenever the exact solution is a polynomial of degree k or less, i.e. $u \in \mathcal{P}_k$. Since we are given the numerical cell averages, $\overline{u}_j^k$, we also require that the reconstruction operation preserve cell averages. Reguardless of what polynomial is reconstructed, we require that its cell average be exactly $\overline{u}_j^k$. We refer to this as "conservation of the mean".

The solution algorithm for (3.19) is a relatively standard procedure for extensions of Godunov's scheme in Eulerian coordinates [23-31]. The basic idea is to start with piecewise constant data in each cell with value equal to the integral cell average. We then reconstruct piecewise polynomials in each cell given cell averages of surrounding neighbors. Using these reconstructed polynomials, we can carry out the boundary integral in (3.19) to higher accuracy than that attainable with piecewise constant data. Once the boundary integral in (3.19) is carried out (either exactly or by numerical quadrature), the solution can be evolved in time. In most cases, we use standard techniques for integrating the resultant ODE equations, i.e. Euler implicit, Euler explicit, Runge-Kutta. Because we can interpret the time integral as an evolution of the cell average, the result of the evolution process is a new collection of cell averages. The process can then be repeated. The process can be summarized in the following steps:

(1) **Reconstruction in Each Cell**: Given integral cell averages in all cells, reconstruct piecewise polynomial coefficients $\alpha_{(m,n)}$ for use in equation (3.18). For solutions containing discontinuities we must consider monotonicity enforcement.

(2) **Flux Evaluation on Each Edge**: Consider each cell boundary, ∂c_j, to be a collection of edges from the mesh. Along each edge, perform a high order accurate flux quadrature.

(3) **Evolution in Each Cell**: Collect flux contributions in each cell and evolve in time using any time stepping scheme, i.e., Euler explicit, Euler implicit, Runge-Kutta, etc. The result of this process is once again cell averages.

By far, the most difficult of these steps is the polynomial reconstruction given cell averages. This is especially true for $k > 1$. For linear reconstruction, the process is slightly simpler because any piecewise linear function constructed about the centroid of the control volume has the correct cell average reguardless of the slope. In this case we need only worry about k-exactness. In the following paragraphs, we describe some design criteria for constructing a general reconstruction operator.

Reconstruction

The reconstruction operator serves as a finite-dimensional (possibly pseudo) inverse of the cell-averaging operator $\mathbf{A}$ whose j-th component $\mathbf{A}_j$ computes the cell average of the solution in c_j.

$$\overline{u}_j = \mathbf{A}_j u = \frac{1}{a_j}\int_{c_j} u(x,y) \, da$$

As we mentioned earlier, we usually require the following properties of the reconstruction operator:

(1)**Conservation of the mean**: Simply stated, given cell averages $\overline{u}$, we require that all polynomial reconstructions u^k have the correct cell average.

$$\text{if } \; u^k = \mathbf{R}^k \overline{u} \; \text{ then } \; \overline{u} = \mathbf{A} u^k$$

This means that $\mathbf{R}^k$ is a right inverse of the averaging operator $\mathbf{A}$.

$$\mathbf{A}\mathbf{R}^k = I \tag{3.21}$$

(2) **k-exactness**: We say that a reconstruction operator $\mathbf{R}^k$ is *k-exact* if $\mathbf{R}^k\mathbf{A}$ reconstructs polynomials of degree k or less exactly. We require,

$$\text{if } \; u \in \mathcal{P}_k \; \text{ and } \; \overline{u} = \mathbf{A}u, \; \text{ then } \; u^k = \mathbf{R}^k \overline{u} = u$$

In other words, $\mathbf{R}^k$ is a left-inverse of $\mathbf{A}$ restricted to the space of polynomials of degree at most k.

$$\mathbf{R}^k \mathbf{A} \Big|_{\mathcal{P}_k} = I \tag{3.22}$$

Note that the second property of k-exactness is not mandatory. We could envision reconstruction operators which are only of the correct order of accuracy. Schemes using k-exactness are easier to analyze and to assure high order accuracy. If we consider an interior edge of the mesh then two estimates for the solution can be obtained; one from each adjacent cell. Assuming smoothness, the property of k-exactness provides that the difference between these two values diminish with increasing k at a rate proportional to h^{k+1} were h is a maximum diameter of the two cells. In Fig. 3.4a we show a global quartic polynomial $u \in \mathcal{P}_4$ which has been averaged in each interval. In Fig. 3.4b we show a quadratic reconstruction $u^k \in \mathcal{P}_2$ given the cell averages of Fig. 3.4a. Close inspection of Fig. 3.4b reveals small jumps in the piecewise polynomials at interval boundaries. Because we are approximating by piecewise quadratics, the jumps are very small. These jumps would decrease even more for cubics and vanish altogether for quartic reconstruction. Property (1) requires that the area under each piecewise polynomial is exactly equal to the cell average.

Figure 3.4a. Cell averaging of quartic polynomial.

Figure 3.4b. Piecewise quadratic reconstruction.

Flux Evaluation

The task here is to evaluate the flux integral.

$$\int_{\partial c_j} \overline{f}(u_+^k, u_-^k, \mathbf{n})\, dl \tag{3.22}$$

We assume that the solution states u_+^k and u_-^k can be evaluated anywhere on an edge. As mentioned earlier, we prefer flux functions based on mean value linearizations, i.e. define $f(u, \mathbf{n}) = \mathbf{F}(u) \cdot \mathbf{n}$ then

$$\overline{f}(u_+, u_-, \mathbf{n}) = \frac{1}{2}(f(u_+, \mathbf{n}) + f(u_-, \mathbf{n})) - \frac{1}{2}|f(\tilde{u}, \mathbf{n})'|(u_+ - u_-) \tag{3.23}$$

where $f(u_+) - f(u_-) = f(\tilde{u}, \mathbf{n})'(u_+ - u_-)$ and $\tilde{u} = \theta u_- + (1-\theta)u_+$ for some $\theta \in [0,1]$. This last statement is merely an application of Taylor's formula with remainder. In practice, the flux integral (3.22) is never evaluated exactly, except when the data is piecewise constant. When piecewise linear functions are used, a midpoint quadrature formula is usually employed. This is used rather than the slightly more accurate trapezoidal quadrature because it requires only one flux evaluation per edge segment while the trapezoidal quadrature requires two. When considering schemes with reconstruction order k greater than one, we suggest in [31] that Gauss quadrature formulas be used. Recall that N point Gauss quadrature formulas integrate $2N-1$ polynomials exactly. These quadrature formulas give the highest accuracy for the lowest number of function evaluations. We find that for k-exact reconstruction, we need to use $N \geq (k+1)/2$ point Gauss quadrature formulas.

Duality of Schemes

The choice of control volumes is not unique. Even so, the various choices usually fall into one of two categories: control volumes formed from the mesh or control volumes formed from the dual of the mesh, see Fig. 3.1a. We can show in the general formulation that all geometrical information needed in Godunov's method is contained in the following calculation:

$$I_{(m,n)} = \int_c x^m \, y^n \, da, \quad \forall \, m+n \leq k \tag{3.24}$$

Note that $I_{(0,0)}$ is the area of the control volume, $I_{(1,0)}, I_{(0,1)}$ give the centroid coordinates (scaled by area), $I_{(1,1)}, I_{(2,2)}, I_{(1,2)}, I_{(2,1)}$ are the moments of inertia, etc. More importantly, this integral is only a function of the bounding curve, ∂c.

$$I_{(m,n)} = \int_{\partial c} \frac{1}{m+1} x^{m+1} y^n \, n_x \, dl = \int_{\partial c} \frac{1}{n+1} x^m y^{n+1} \, n_y \, dl = \int_{\partial c} \mathbf{G} \cdot \mathbf{n} \, dl \tag{3.25}$$

where $\mathbf{n} = (n_x, n_y)^T$ and $\mathbf{G}$ is any vector primitive such that $\nabla \cdot \mathbf{G} = x^m y^n$. All of these forms can be calculated exactly on any control volume with perimeter defined by straight line segments. If we ignore boundaries, given a geometrical description of each edge or dual edge, *all* geometrical information is known. All other information concerning the relationship between the mesh and dual mesh is determined, i.e. each vertex of the mesh corresponds to a cell of the dual, each cell of the mesh corresponds to a vertex of the dual, and so on. Again ignoring boundary edges, this is perfect duality.

Although working with cells of the mesh as control volumes seems like the most straightforward thing to do, it is not always the best strategy. If we consider a mesh of triangles, the number of cells of the mesh out number vertices by roughly two to one (see equation 2.3). This means than the number of control volumes required using cells rather than dual cells is also two to one. Consequently, using mesh duals offers some savings in terms of storage and computation for those quantities computed and stored at each control volume. Note that edges and dual edges are one-to-one (neglecting boundaries) so that variables stored by edge and computations performed by edge are roughly the same.

In section 3.2, control volumes formed from the median dual of the mesh naturally arose in the Galerkin-like finite volume discretization. Of course in the general finite volume method, we are free to choose other control volumes. Taking a look at Fig. 3.1a, we see that other control volumes are possible. The dual formed by connecting adjacent cell centroids looks appealing. The use of the Dirichlet regions also looks interesting. Several other possibilities exist, but only a few are of practical value. The most nagging problem in working with duals is that which occurs near boundaries. In Fig. 3.5a, we show a centroid dual for a mesh about a curved concave boundary. Because of the curvature, boundaries of the dual cells lie within the body. Note that this becomes more aggravated as the cells are refined in the normal direction. In this case, several layers of dual cells could penetrate into the body. The use of a median dual offers relief to this problem (see Fig. 3.5b). But because the edges of the dual are not straight, simple flux quadratures can give very poor results. We will take up this topic in later sections.

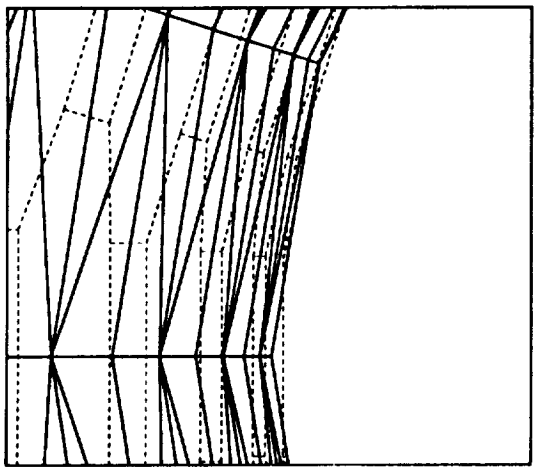

Figure 3.5a. Mesh (solid lines) and Centroid Dual (dashed lines).

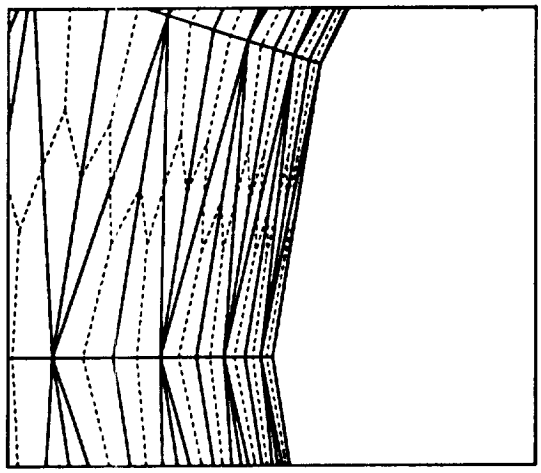

Figure 3.5b. Mesh (solid lines) and Median Dual (dashed lines).

We now consider some of the algorithmic details of constructing upwind schemes. To illustrate the methods, we will formulate these schemes on a control volume formed from the median dual as shown in Fig 3.6. This figure shows the control volume of interest c_j with neighboring control volumes c_i with local indexing $i = 1, 2, 3, ..., d(c_j)$. In the case of the median dual we have the added complexity of nonstraight edges which actually consist of two straight line segments.

We begin with the first order upwind scheme using piecewise constant approximations. Stability and monotonicity of the scheme will be proven. We then consider schemes which reconstruct piecewise linear polynomials. These schemes require slope limiting to insure monotonicity. This can be done in several different ways. We then present the technique described in [31] for obtaining high order accuracy using an arbitrary order k-exact reconstruction operator.

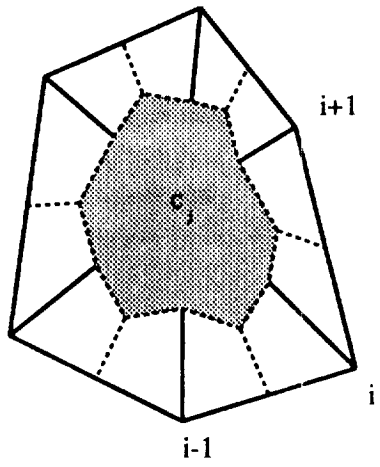

Figure 3.6 Typical mesh showing median control volumes with local indexing.

Piecewise Constant (k=0) Upwind Scheme

This is the simplest approximation in which the polynomial behavior in each cell, c_j, is a constant value equal to the cell average.

$$u^{k=0}(x,y) = \overline{u}_j \quad \text{for } u^k \in c_j \tag{3.28}$$

The flux quadrature then simplifies to the following form:

$$\overline{f}(u_+, u_-, \vec{\mathbf{n}}) = \overline{f}(\overline{u}_i, \overline{u}_0, \vec{\mathbf{n}}) = \frac{1}{2}(f(\overline{u}_0, \vec{\mathbf{n}}_i) + f(\overline{u}_i, \vec{\mathbf{n}}_i)) - \frac{1}{2}|f(\tilde{u}, \vec{\mathbf{n}}_i)'|(\overline{u}_i - \overline{u}_0) \tag{3.29}$$

In this formula, $\vec{\mathbf{n}}_i = \int_{e_i} \mathbf{n}\, dl$ is any path connecting endpoints of the edge of the control volume. From this we can write the entire scheme for c_j.

$$\frac{\partial}{\partial t}\int_{c_j} \overline{u}_0\, da + \sum_{i=1}^{d(c_j)} \frac{1}{2}\left(f(\overline{u}_0, \vec{\mathbf{n}})_i + f(\overline{u}_i, \vec{\mathbf{n}})_i\right) - \frac{1}{2}|f'(\tilde{u}, \vec{\mathbf{n}})_i|(\overline{u}_i - \overline{u}_0) = 0 \tag{3.30}$$

It is not difficult to prove stability and monotonicity of the scheme. Recall that the flux function was constructed such that

$$f(u_i, \vec{\mathbf{n}})_i - f(u_0, \vec{\mathbf{n}})_i = f'(\tilde{u}, \mathbf{n})_i (u_i - u_0), \quad \tilde{u} = \theta u_0 + (1-\theta)u_i, \quad \theta \in [0,1]$$

This allows us to regroup terms into the following form:

$$\frac{\partial}{\partial t}\int_{c_j} \overline{u}_0\, da + \sum_{i=1}^{d(c_j)} f(\overline{u}_0, \vec{\mathbf{n}})_i + \frac{1}{2}\left(f'(\tilde{u}, \vec{\mathbf{n}})_i - |f'(\tilde{u}, \vec{\mathbf{n}})_i|\right)(\overline{u}_i - \overline{u}_0) = 0 \tag{3.31}$$

For a closed control volume we have that $\sum_{i=1}^{d(c_j)} f(\overline{u}_0, \vec{\mathbf{n}})_i = 0$. Combining the remaining terms we obtain a final form for analysis:

$$\frac{\partial}{\partial t}\int_{c_j} \overline{u}_0\, da + \sum_{i=1}^{d(c_j)} \left(f'(\tilde{u}, \vec{\mathbf{n}})_i\right)^- (\overline{u}_i - \overline{u}_0) = 0 \tag{3.32}$$

First we verify the monotonicity of the scheme at steady state ($\frac{\partial}{\partial t}\int_c \overline{u}_0\, da = 0$) by solving for $\overline{u}_0$.

$$\overline{u}_0 = \frac{\sum_{i=1}^{d(c_j)} \left(f'(\tilde{u}, \vec{\mathbf{n}})_i\right)^- \overline{u}_i}{\sum_{i=1}^{d(c_j)} \left(f'(\tilde{u}, \vec{\mathbf{n}})_i\right)^-} = \sum_{i=1}^{d(c_j)} \alpha_i \overline{u}_i \tag{3.33}$$

From this we see that all weights α_i are positive and sum to unity. We can conclude that the scheme is monotone since $\overline{u}_0$ is a positive weighted average of all neighbors. This

implies that we have a maximum principle since $\overline{u}_0$ is bounded from above and below by $\overline{u}_{max}$ and $\overline{u}_{min}$. ($\overline{u}_{min} = min(\overline{u}_1, \overline{u}_2, ..., \overline{u}_{d(c)})$, $\overline{u}_{max} = max(\overline{u}_1, \overline{u}_2, ..., \overline{u}_{d(c)})$)

$$\overline{u}_{min} \leq \overline{u}_0 \leq \overline{u}_{max}$$

For explicit time stepping, a CFL-like condition is obtained for monotonicity. For Euler explicit time stepping we have the time approximation, $\frac{\partial}{\partial t}\frac{1}{A_c}\int_c \overline{u}_0 \, da \approx \frac{\overline{u}_0^{n+1} - \overline{u}_0^n}{\Delta t}$.

$$\overline{u}_0^{n+1} = \overline{u}_0^n - \frac{\Delta t}{a_c}\sum_{i=1}^{d(c_j)} \left(f'(\tilde{u}, \vec{\mathbf{n}})_i\right)^- (\overline{u}_i^n - \overline{u}_0^n) = \sum_{i=0}^{d(c_j)} \alpha_i \overline{u}_i^n \tag{3.34}$$

It should be clear that coefficients in (3.34) sum to unity. To demonstrate monotonicity, we again need to show positivity of coefficients. By inspection we have that $\alpha_i \geq 0 \;\; \forall \, i > 0$. To guarantee monotonicity, we require that $\alpha_0 \geq 0$.

$$\alpha_0 = 1 + \frac{\Delta t}{A_c}\sum_{i=1}^{d(c_j)} \left(f'(\overline{u}, \vec{\mathbf{n}})_i\right)^- \geq 0 \tag{3.35}$$

Thus we obtain the CFL-like condition which insures monotonicity and stability.

$$\Delta t \leq -\frac{A_c}{\sum_{i=1}^{d(c_j)} \left(f'(\tilde{u}, \vec{\mathbf{n}})_i\right)^-} \tag{3.36}$$

Note that in one dimension, this number corresponds to the conventional CFL number. In multiple space dimensions, we find that this number is sufficient but not necessary for stability. In practice somewhat larger timestep values may be used.

Conclusion: *The upwind algorithm (3.29) using piecewise constant data satisfies a discrete maximum principle for general unstructured meshes.*

Piecewise Linear (k=1) Upwind Schemes

The strategy here is to replace the assumption of piecewise constant data used in the first order upwind algorithm with the more accurate piecewise linear approximation. We will find that it is advantageous to consider these linear polynomials as being expanded about the centroid of the cell in which case conservation of the mean comes for free. In the case of linear reconstruction, we must take additional steps to insure monotonicity. The approach we pursue is the one suggested by van Leer [25] which has a simple geometric interpretation and extends to unstructured meshes. The only remaining piece of the puzzle is the flux quadrature. The flux quadrature suggested by the Galerkin discretization turns out to behave poorly for irregular meshes. We show some results for an improved quadrature.

We begin by assuming a piecewise linear reconstruction polynomial in each control volume c_j expanded about some as yet unknown origin (x_0, y_0).

$$u^k(x,y)_j = u_0 + \nabla u^k_{c_j} \cdot (\mathbf{R} - \mathbf{R_0}) \tag{3.37}$$

We now insist that this polynomial have the correct cell average in the control volume c_j.

$$\overline{u}_j A_j = \int_{c_j} u^k \, da = u_0 A_j + \nabla u^k_{c_j} \cdot \int_{c_j} (\mathbf{R} - \mathbf{R_0}) \, da$$

We see that this is satisfied when $u_0 = \overline{u}_j$ and $\mathbf{R_0}$ is the cell centroid, $\mathbf{R}_c$. (This is true for any gradient vector.) This has important implications. This means we can think of the cell average as a pointwise exact value of the linear function at the centroid of the cell. To obtain an estimate of the reconstructed gradient of u^k in c_j, we exploit this fact together with an approximate form of the exact Green-Gauss formula:

$$\int_\Omega \nabla u \, da = \oint u \, \mathbf{n} \, dl \tag{3.38}$$

The solution gradient for the cell c_j is estimated by computing the boundary integral of (3.38) for some path $\partial\Omega$ surrounding c_j.

$$\nabla u^k_{c_j} = \frac{1}{A_\Omega} \oint_{\partial\Omega} u^k \mathbf{n} \, dl \tag{3.39}$$

Note that this formula is exact whenever the solution behaves linearly in the region Ω. This technique is used in the algorithms proposed in [30,32,33,34]. We need only to determine the path and numerical quadrature for (3.39) to describe the linear reconstruction. The obvious path connects the centroids of all dual cells c_i which share an edge with c_j. This would form a closed loop surrounding c_j. We then consider the cell averages of these neighbors as pointwise values of the solution located at the centroids. A trapezoidal integration given these pointwise values guarantees that the calculation is exact whenever the solution varies linearly. This provides k-exactness.

When solution discontinuites and steep gradients as present, we must take steps to prevent oscillations from developing in the numerical solution. One way to do this was pioneered by van Leer [25] in the late 1970's. The basic idea is to take the reconstructed piecewise polynomials and enforce strict monotonicity in the reconstruction. We use monotonicity in this context to mean that the value of the reconstructed polynomial does not exceed the minimum and maximum of neighboring cell averages. Figure 3.7a plots a linear reconstruction of a quartic polynomial. Checking the interior cell reconstructions, we find that new extrema have been created. When a new extremum is located, we reduce the slope of the reconstruction until monotonicity is restored. This implies that when we are at a minimum or maximum the slope is reduced to zero, see Fig. 3.7b.

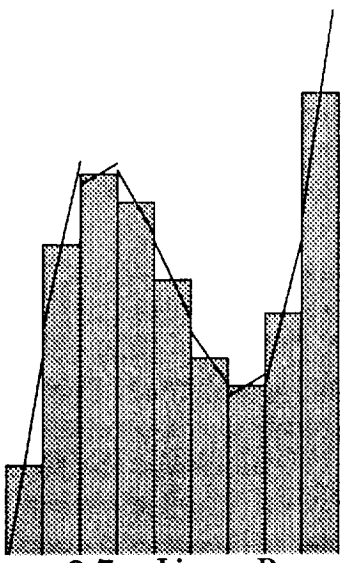

Figure 3.7a. Linear Reconstruction.

Figure 3.7b. Linear Reconstruction with monotone limiting.

The beauty of this method is that it generalizes to unstructured grids. In the case of piecewise linear functions, we consider a "limited" form of the piecewise linear distribution.

$$u^k(x,y)_j = \overline{u}_j + \Phi_j \nabla u^k_{c_j} \cdot (\mathbf{R} - \mathbf{R}_c) \tag{3.40}$$

The idea is to find the largest admissible Φ_j while invoking a monotonicity principle that values of the linearly reconstructed function must not exceed the maximum and minimum of neighboring centroid values (including the centroid value in c_j). To do this, first compute $u_j^{min} = \min(\overline{u}_j, \overline{u}_{neighbors})$ and $u_j^{max} = \max(\overline{u}_j, \overline{u}_{neighbors})$ then require that

$$u_j^{min} \leq u(x,y)_j^k \leq u_j^{max} \tag{3.41}$$

For linear reconstructions, extrema in $u(x,y)_j^k$ occur at the vertices of the control volume and sufficient conditions for (3.41) can be easily obtained. For each vertex of the cell compute $u_i = u^k(x_i,y_i)_j$, $i = 1, N_{c_j}$ to determine the limited value, ϕ_i, which satisfies (3.41):

$$\phi_i = \begin{cases} \min(1, \frac{u_j^{max} - \overline{u}_j}{u_i - \overline{u}_j}), & \text{if } u_i - \overline{u}_j > 0 \\ \min(1, \frac{u_j^{min} - \overline{u}_j}{u_i - \overline{u}_j}), & \text{if } u_i - \overline{u}_j < 0 \\ 1 & \text{if } u_i - \overline{u}_j = 0 \end{cases} \tag{3.42}$$

with $\Phi_j = \min(\phi_1, \phi_2, \phi_3, ..., \phi_{N_{c_j}})$.

The proper choice of flux quadrature for the median dual control volume requires careful thought. Recall that each edge of this control volume is actually composed of two straight line segments. The finite volume interpretation of the Galerkin discretization (see eqn. 3.8) suggests that we ignore the details of the shape of edge, i.e. compute a normal by connecting centroid values. Furthermore, in the Galerkin-like formulation, the solution is stored at the vertices of the mesh and the flux through the *ith* edge of the control volume is the arithmetic average of the fluxes calculated at the vertices v_i and v_j. In the spirit of

the present method, we would not take the average of the fluxes but rather extrapolate to a point half way between the vertices v_j and v_i and take the numerical flux of these states. This was suggested in Desideri [35]. We find that this does not always work particularly well for distorted meshes. This is especially true when performing Euler and Navier-Stokes equations at low Mach numbers. We prefer to split the numerical flux (3.23) into two parts:

$$\overline{f}(u_+, u_-, \mathbf{n}) = \underbrace{\frac{1}{2}(f(u_+, \mathbf{n}) + f(u_-, \mathbf{n}))}_{\text{Term 1}} - \underbrace{\frac{1}{2}|f(\tilde{u}, \mathbf{n})'|(u_+ - u_-)}_{\text{Term 2}} \tag{3.43}$$

We then compute the first term in two pieces by performing a midpoint quadrature on each straight line segment of the control volume edge. The second term is computed only once by extrapolating to the halfway point suggested by Desideri. This minimizes the additional computation involved. As we will see, this is especially true for the Euler equations where the second term in (3.43) dominates the calculation. For flows with discontinuities requiring monotonicity enforcement, we find that we have no choice but to compute the entire flux on each straight line segment.

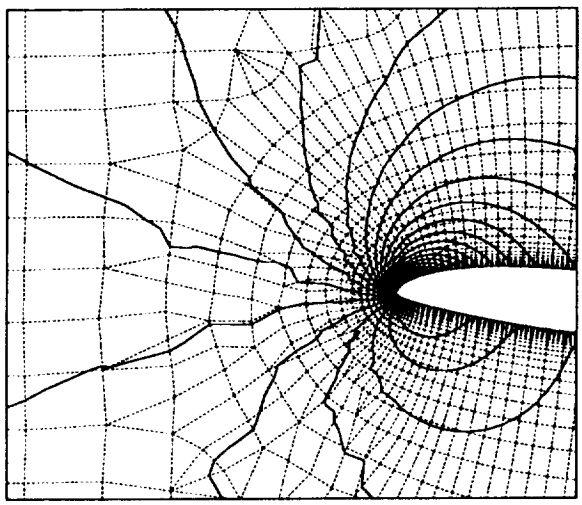

Figure 3.8a. One pt. quadrature formula.

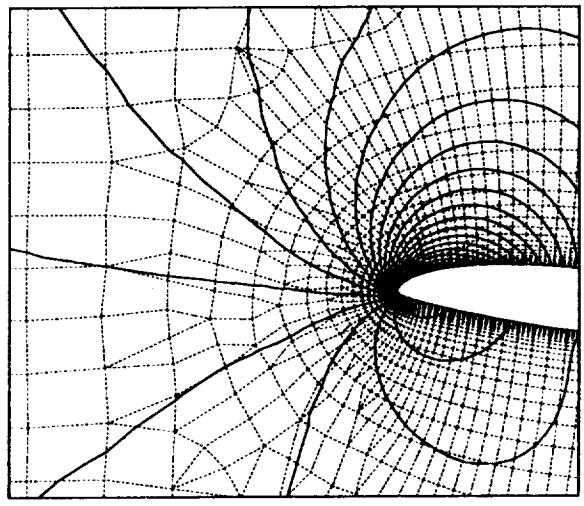

Figure 3.8b. Modified quadrature formula.

Figures 3.8a-b show pressure contours for a low Mach number calculation $M_\infty = .3$ using identical gradient reconstructions but different quadrature formulas. Figure 3.8a uses the formula suggested in [35] and Fig. 3.8b uses the new formula (3.43).

The combination of flux formula, reconstruction operator, and limiter function produces a scheme with excellent shock resolving characteristics. We can demonstrate this on a scalar nonlinear hyperbolic problem suggested by Struijs, Deconinck, and Coussement[24]. The equation is a multidimensional form of Burger's equation.

$$u_t + (u^2/2)_x + u_y = 0$$

We solve the equation in a square region $[0, 1.5] \times [0, 1.5]$ with boundary conditions: $u(x, 0) = 1.5 - 2x$, $x \leq 1$, $u(x, 0) = -.5$, $x > 1$, $u(0, y) = 1.5$, and $u(1.5, y) = -.5$.

In Figs. 3.8 and 3.9 we show carpet plots and contours of the solution on regular and irregular meshes.

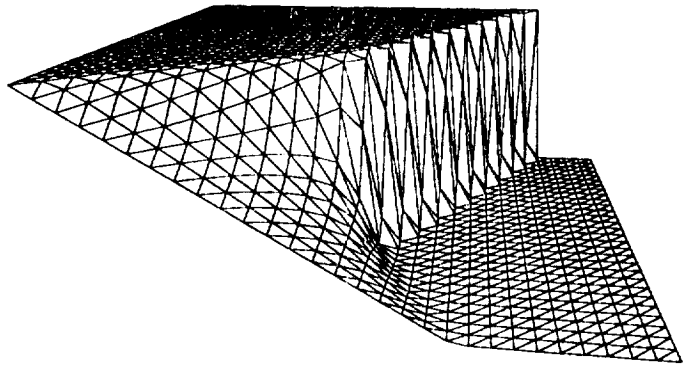

Figure 3.8a. Carpet plot of Burger's equation solution on regular mesh.

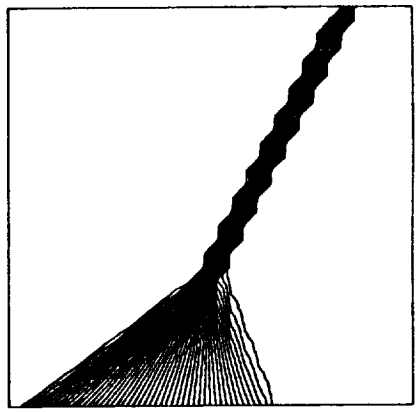

Figure 3.8b. Solution Contours.

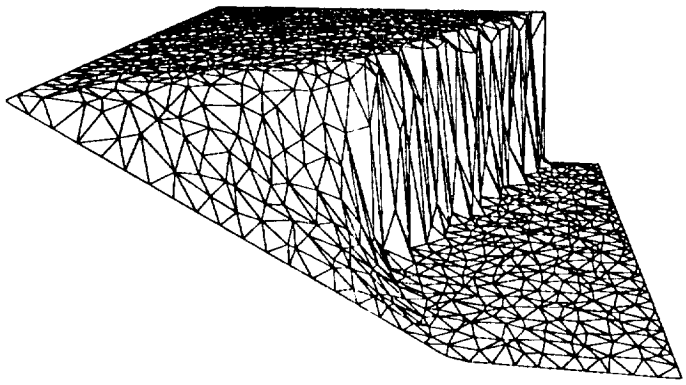

Figure 3.9a. Carpet plot of Burger's equation solution on irregular mesh.

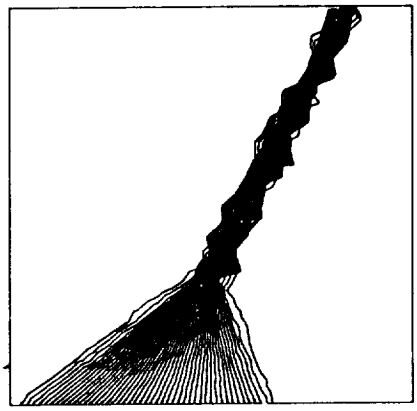

Figure 3.9b. Solution Contours.

Note that the carpet plots indicate that the numerical solution on both meshes is monotone. Even so, most people would prefer the solution on the regular mesh. This is an unavoidable consequence of irregular meshes. The only remedy appears to be mesh adaptation. We show similar results for the Euler equations on irregular meshes in a future section.

Arbitrary Order Reconstruction

In this section we give a brief account of the method we have developed in [31] for arbitrary order reconstruction. Upon first inspection, the use of high order reconstruction might appear to be an expensive proposition. In our case, we can optimize the efficiency of the reconstruction by precomputing as a *one time* preprocessing step the set of weights

$\mathbf{W}_j$ in each cell c_j with neighbor set $\mathcal{N}_{c_j}$ such that

$$\alpha_{(m,n)} = \sum_{i \in \mathcal{N}_{c_j}} W_{(m,n)i} \overline{u}_i \tag{3.44}$$

where $\alpha_{(m,n)}$ are the polynomial coefficents, see eqn. 3.18. This effectively reduces the problem of reconstruction to multiplication of predetermined weights and cell averages to obtain polynomial coefficients.

During the preprocessing to obtain the reconstruction weights $\mathbf{W_j}$ we assume a coordinate system with origin at the centroid of c_j to minimize roundoff errors. We then temporarily transform (rotate and scale) to another coordinate system $(\overline{x}, \overline{y})$ which is normalized to the cell $\mathbf{c}_j$

$$\begin{bmatrix} \overline{x} \\ \overline{y} \end{bmatrix} = \begin{bmatrix} \mathbf{D}_{1,1} & \mathbf{D}_{1,2} \\ \mathbf{D}_{2,1} & \mathbf{D}_{2,2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

with the matrix $\mathbf{D}$ chosen so that

$$\mathbf{A}_j(\overline{x}^2) = \mathbf{A}_j(\overline{y}^2) = 1$$

$$\mathbf{A}_j(\overline{x}\,\overline{y}) = \mathbf{A}_j(\overline{y}\,\overline{x}) = 0$$

We then temporarily represent polynomials on $\mathbf{c}_j$ using the polynomial basis functions $\overline{P} = [1, \overline{x}, \overline{y}, \overline{x}^2, \overline{x}\overline{y}, \overline{y}^2, \overline{x}^3, ...]$. Note that polynomials in this system are easily transformed to the standard cell-centroid basis

$$\overline{x}^m \overline{y}^n = \sum_{s+t \le k} \binom{m}{s} \binom{n}{t} \mathbf{D}_{1,1}^s \mathbf{D}_{1,2}^{m-s} \mathbf{D}_{2,1}^t \mathbf{D}_{2,2}^{n-t} x^{s+t} y^{m+n-s-t}$$

Since $0 \le s+t \le k$ and $0 \le m+n-s-t \le k$, we can reorder and rewrite in terms of the standard and transformed basis polynomials

$$\overline{P}_{(m,n)} = \sum_{s+t \le k} G_{m,n}^{s,t} P_{(s,t)} \tag{3.45}$$

We can guarantee satisfaction of conservation of the mean by introducing into the transformed coordinate system *zero mean* basis polynomials $\overline{P}^0$ in which all but the first have zero cell average, i.e $\overline{P}^0 = [1, \overline{x}, \overline{y}, \overline{x}^2 - 1, \overline{x}\,\overline{y}, \overline{y}^2 - 1, \overline{x}^3 - A_j(\overline{x}^3), ...]$. Note that using these polynomials requires a minor modification of (3.45) but retains the same form:

$$\overline{P}^0_{(m,n)} = \sum_{s+t \le k} \overline{G}_{m,n}^{s,t} P_{(s,t)} \tag{3.46}$$

Given this preparatory work, we are now ready to describe the formulation of the reconstruction algorithm.

Minimum Energy Reconstruction

We note that the set of cell neighbors $\mathcal{N}_j$ must contain at least $(k+1)(k+2)/2$ cells $\mathbf{c}_j$ if the reconstruction operator $\mathbf{R}_j^k$ is to be k-exact. That $(k+1)(k+2)/2$ cells is not sufficient in all situations is easily observed. If, for example, the cell-centers all lie on a single straight line one can find a linear function u such that $\mathbf{A}_j(u) = 0$ for every cell $\mathbf{c}_j$, which means that reconstruction of u is impossible. In other cases a k-exact reconstruction operator $\mathbf{R}_j^k$ may exist, but due to the geometry may be poorly conditioned.

Our approach is to work with a slightly larger support containing more than the minimum number of cells. In this case the operator $\mathbf{R}_j^k$ is likely to be nonunique, because various subsets would be able to support reconstruction operators of degree k. Although all would reproduce a polynomial of degree k exactly, if we disregard roundoff, they would differ in their treatment of non-polynomials, or of polynomials of degree higher than k. Any k-exact reconstruction operator $\mathbf{R}_j^k$ is a weighted average of these basic ones. Our approach is to choose the one of minimum Frobenius norm. This operator is optimal, in a certain sense, when the function we are reconstructing is not exactly a polynomial of degree k, but one that has been perturbed by the addition of Gaussian noise, for it minimizes the expected deviation from the unperturbed polynomial in a certain rather natural norm.

As we begin the formulation of the reconstruction preprocessing algorithm, the reader is reminded that the task at hand is to calculate the weights $\mathbf{W}_j$ for each cell c_j which when applied via (3.44) produce piecewise polynomial approximations. We begin by first rewriting the piecewise polynomial (3.18) for cell c_j in terms of the reconstruction weights (3.44)

$$u^k(x,y) = \sum_{m+n\le k} P_{(m,n)} \sum_{i\in\mathcal{N}_{c_j}} W_{(m,n)i}\overline{u}_i \tag{3.47a}$$

or equivalently

$$u^k(x,y) = \sum_{i\in\mathcal{N}_{c_j}} \overline{u}_i \sum_{m+n\le k} W_{(m,n)i}P_{(m,n)} \tag{3.47b}$$

Polynomials of degree k or less are equivalently represented in the transformed coordinate system using zero mean polynomials

$$u^k(x,y) = \sum_{i\in\mathcal{N}_{c_j}} \overline{u}_i \sum_{m+n\le k} W'_{(m,n)i}\overline{P}^0_{(m,n)} \tag{3.48}$$

Using (3.46) we can relate weights in the transformed system to weights in the original system

$$W_{(s,t)i} = \sum_{m+n\le k} \overline{G}^{s,t}_{m,n} W'_{(m,n),i} \tag{3.49}$$

We satisfy k-exactness by requiring that (3.48) is satisfied for all linear combinations of $\overline{P}^0_{(s,t)}(x,y)$ such that $s+t\le k$. In particular, if $u^k(x,y) = \overline{P}^0_{(s,t)}(x,y)$ for some $s+t\le k$ then

$$\overline{P}^0_{(s,t)}(x,y) = \sum_{m+n\le k} \overline{P}^0_{(m,n)} \sum_{i\in\mathcal{N}_{c_j}} W'_{(m,n)i}A_i(\overline{P}^0_{(s,t)})$$

This is satisfied if for all $s+t, m+n \leq k$

$$\sum_{i \in \mathcal{N}_{c_j}} W'_{(m,n)i} A_i(\overline{P}^0_{(s,t)}) = \delta^{st}_{mn}$$

Transforming basis polynomials back to the original coordinate system we have

$$\sum_{i \in \mathcal{N}_{c_j}} W'_{(m,n)i} \sum_{u+v \leq k} \overline{G}^{u,v}_{s,t} A_j(P_{(u,v)}) = \delta^{st}_{mn} \tag{3.50}$$

This can be locally rewritten in matrix form as

$$\mathbf{W}'_j \mathbf{A}'_j = \mathbf{I} \tag{3.51a}$$

and transformed in terms of the standard basis weights via

$$\mathbf{W}_j = \mathbf{G}\mathbf{W}'_j \tag{3.51b}$$

Note that $\mathbf{W}'_j$ is a $(k+1)(k+2)/2$ by $\mathcal{N}_j$ matrix and $\mathbf{A}'_j$ has dimensions $\mathcal{N}_j$ by $(k+1)(k+2)/2$. To solve (3.51a) in the optimum sense describe above, we perform an $\mathbf{L}_j\mathbf{Q}_j$ decomposition of $\mathbf{A}'_j$ where the orthogonal matrix $\mathbf{Q}_j$ and the lower triangular matrix $\mathbf{L}_j$ have been constructed using a modified Gram-Schmidt algorithm (or a sequence of Householder reflections) see ref. [36]. The weights $\mathbf{W}'_j$ are then given by

$$\mathbf{W}'_j = \mathbf{Q}^*_j \mathbf{L}^{-1}_j$$

Applying (3.49) we transform these weights to the standard centroid basis and the preprocessing step is complete.

We now show a few results presented earlier in ref. [31]. The first calculation involves the reconstruction of a sixth order polynomial with random normalized coefficients which has been cell averaged onto a random mesh. Figures 3.10a-b show a sample mesh and the absolute L_2 error of the reconstruction for various meshes and reconstruction degree.

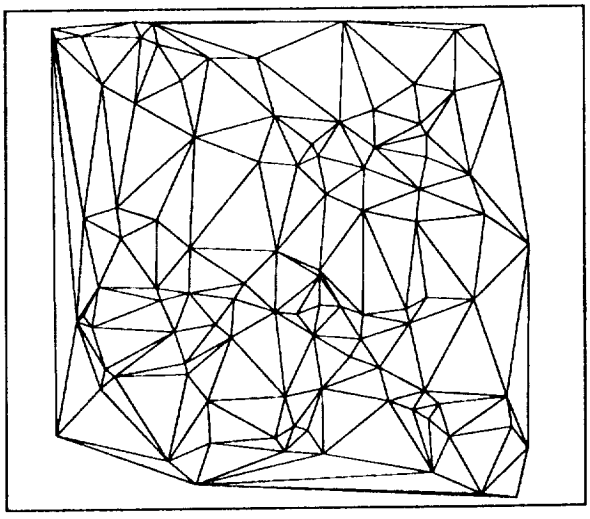

Figure 3.10a. Random mesh.

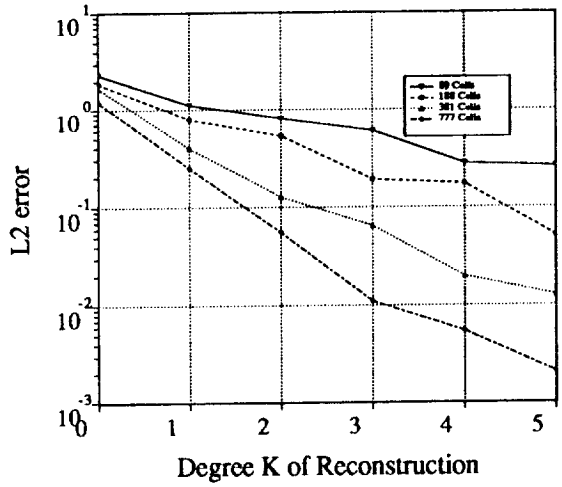

Figure 3.10b. L_2 error of reconstruction.

We also have tested the reconstruction algorithm on more realistic problems. In Figs. 3.11a-c, we show a mesh and reconstructions (linear and quadratic) of a cell averaged density field corresponding to a Ringleb flow, an exact hodograph solution of the gasdynamic equations, see [37].

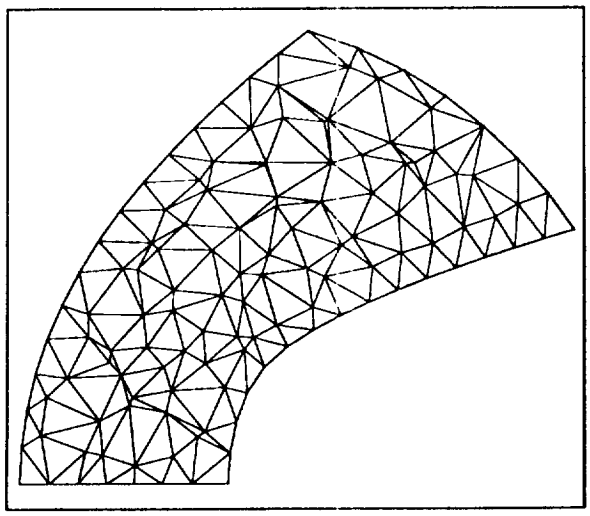

Figure 3.11a. Randomized mesh for Ringleb Flow.

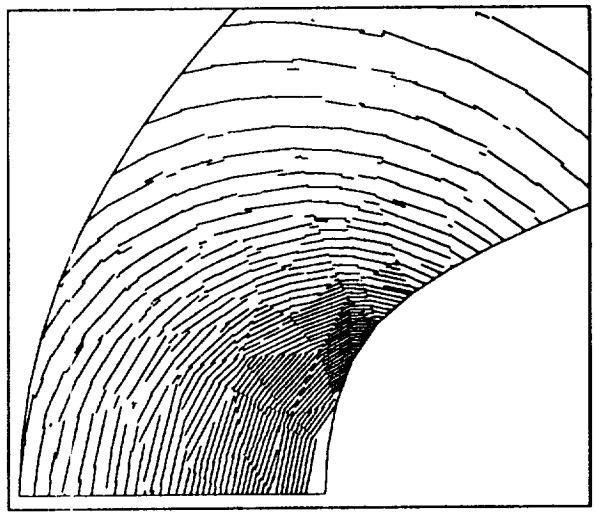

Figure 3.11b. Piecewise linear reconstruction of Ringleb flow.

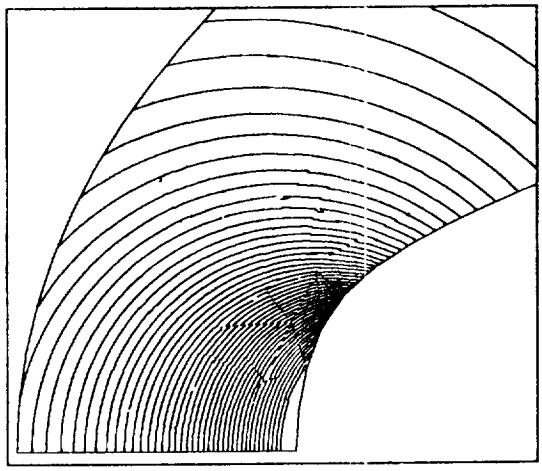

Figure 3.11c. Piecewise quadratic reconstruction of Ringleb flow.

The reader should note that the use of piecewise contours gives a crude visual critique as to how well the solution is represented by the piecewise polynomials. The improvement from linear to quadratic is dramatic in the case of Ringleb flow. In an upcoming section, we will show actual numerical solutions computed using this reconstruction operator. Complete details are given in ref. [31].

3.5 Upwind Finite Volume Solvers for the Euler Equations

In this section we consider the extension of upwind schemes for scalar hyperbolic equations discussed in section 3.4 to the Euler equations of gasdynamics. As we will see, the changes are relatively minor modifications as most of the work has already been done in designing the scalar scheme. We begin by writing the two-dimensional Euler Equations in integral form. In Ω we write the physical laws of conservation of mass, momentum, and energy.

Conservation of Mass

$$\frac{\partial}{\partial t}\int_{\Omega} \rho \, da + \int_{\partial\Omega} \rho(\mathbf{V}\cdot\mathbf{n}) \, dl = 0 \tag{3.52}$$

Conservation of Momentum

$$\frac{\partial}{\partial t}\int_{\Omega} \rho\mathbf{V} \, da + \int_{\partial\Omega} \rho\mathbf{V}(\mathbf{V}\cdot\mathbf{n}) \, dl + \int_{\partial\Omega} p\mathbf{n} \, dl = 0 \tag{3.53}$$

Conservation of Energy

$$\frac{\partial}{\partial t}\int_{\Omega} E \, da + \int_{\partial\Omega} (E+p)(\mathbf{V}\cdot\mathbf{n}) \, dl = 0 \tag{3.54}$$

In these equations $\rho, \mathbf{V}, p$, and E are the density, velocity, pressure, and total energy of the fluid. The system is closed by introducing a thermodynamical equation of state for a perfect gas:

$$p = (\gamma - 1)(E - \frac{1}{2}\rho(\mathbf{V}\cdot\mathbf{V})) \tag{3.55}$$

These equations can be written in a more compact vector equation:

$$\frac{\partial}{\partial t}\int_{\Omega} \mathbf{u} \, da + \int_{\partial\Omega} \mathbf{F}(\mathbf{u})\cdot\mathbf{n} \, dl = 0 \tag{3.56}$$

with

$$\mathbf{u} = \begin{pmatrix} \rho \\ \rho\mathbf{V} \\ E \end{pmatrix}, \quad \mathbf{F}(\mathbf{u})\cdot\mathbf{n} = \begin{pmatrix} \rho(\mathbf{V}\cdot\mathbf{n}) \\ \rho\mathbf{V}(\mathbf{V}\cdot\mathbf{n}) + p\mathbf{n} \\ (E+p)(\mathbf{V}\cdot\mathbf{n}) \end{pmatrix}$$

The formulation for systems of equations requires two modifications. The first concerns the flux function. We now need to consider a vector flux function rather than a scalar flux function. In our applications, we choose the Roe flux [45] because of it's simplicity. This flux is of exactly the same form as equation 3.23 except that quantities are now vectors and matrices. The second modification concerns the reconstruction. We now need to reconstruct several quantities. Although we are free to choose any complete set

of variables that we like (primitive variables, entropy variables, etc.), we must in general assure that the total amount of mass, momentum, and energy is conserved in each cell during the reconstruction. In other words, we still need conservation of the mean in terms of the conserved variables. For steady state applications this is not exactly true, the spatial conservation is guaranteed because of the integral formulation and the final solution is only dependent on how accurately we can reconstruct the solution, i.e. the degree of k-exactness. In Figs. 3.12a-e we show a numerical solution for Euler flow over a NACA 0012 airfoil at transonic speeds ($M_\infty = .80, \alpha = 1.25°$) on a semi-random mesh using the system extension of the schemes described in section 3.4.

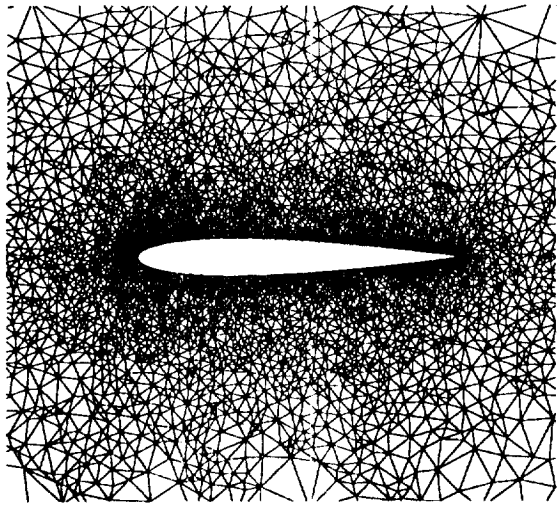

Figure 3.12a. Close up of Mesh.

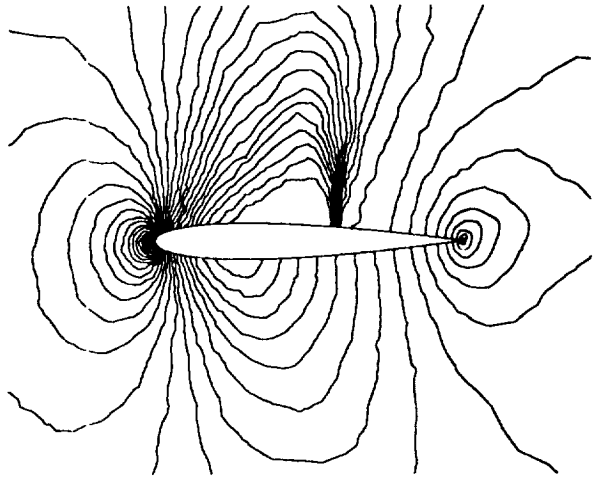

Figure 3.12b. Mach contours. First order upwind scheme.

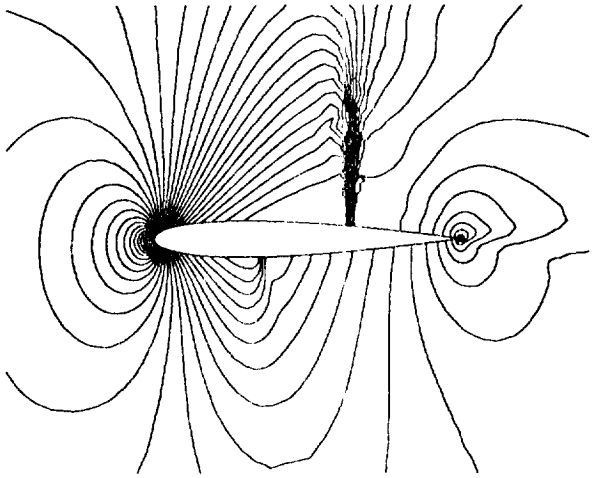

Figure 3.12c. Mach contours. Linear reconstruction scheme.

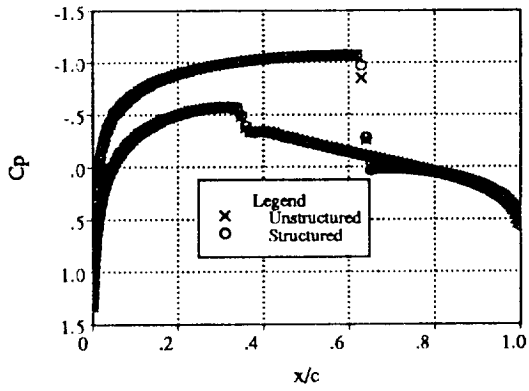

Figure 3.12d. Pressure coefficient distribution on surface.

Note that although the mesh is very irregular, smooth parts of the flowfield yield smooth solution contours. Also note the monotone resolution of the shock waves.

We previously mentioned the importance of using accurate flux quadrature formulas. In fact for k-exact reconstruction, we suggest N point Gauss quadratures with $N \geq (k+1)/2$. In Figs. 3.13a-b we demonstrate this importance by plotting density contours for a numerical calculation of the Ringleb flow (previously described) using quadratic reconstruction $k = 2$. Our formula suggests that two point quadratures should be used in this case. Figure 3.13a shows contours for a calculation using one point Gauss quadrature and Fig. 3.13b shows contours for a calculation using two point quadratures. The improvement in Fig. 3.13b is dramatic. Increasing the number of quadrature points to three leaves the solution unchanged.

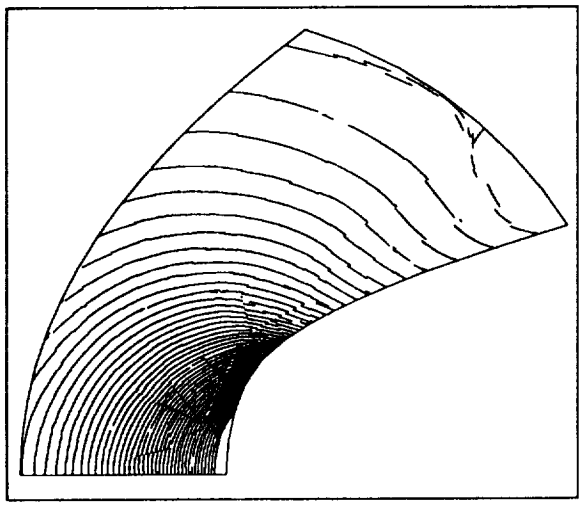

Figure 3.13a. Density Contours. $(k = 2, N = 1)$

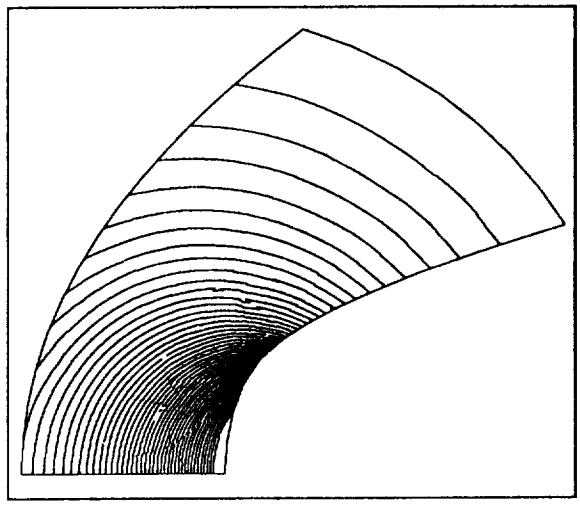

Figure 3.13b. Density Contours. $(k = 2, N = 2)$

3.6 Conclusions

The current state of technology in unstructured mesh solvers is rapidly changing. Now that a number of spatial discretizations have been developed, it is time to focus attention on solution strategies, i.e. efficient timestepping and iteration methods. This is probably a more challenging problem and will no doubt require some major breakthroughs to give unstructured grids universal appeal in the CFD community.

4.0 Mesh Transfer Algorithms

In this section we describe a particularly elegant technique for conservatively transferring cell average information from one mesh to another with high spatial accuracy. This algorithm is particularly appropriate when used in conjunction with flow solver schemes based on Godunov's method.

Given a mesh and solution, the task at hand is to transfer the solution information to a new mesh. We assume a worst case situation in which the old and new mesh (T and T^*) are not regular subsets or supersets of each other, see Fig. 4.0 for example.

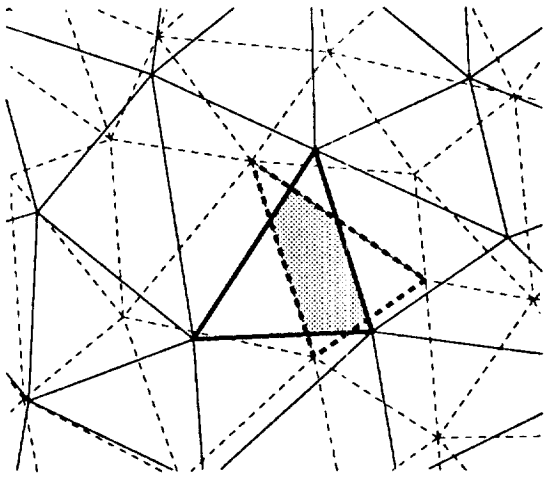

Figure 4.0. Disjoint meshes (solid lines are old mesh, dashed lines are new mesh).

This situation frequently arises from moving mesh adaptations, multigrid (or multiple grid) solvers, etc. In order that the transfer process be conservative we require that

$$\int_{\Omega(T)} u \, da = \int_{\Omega(T^*)} u \, da$$

From the standpoint of the schemes described in previous sections, we only require the accurate calculation of cell averages on T^*. In the simplest approach, we would assume that piecewise constant functions (the cell averages) are given on T. From Fig. 4.0, it should be clear that in general each cell of the new mesh receives contributions from several cells of the old mesh. The amount added to a new cell would be the area weighted average of all contributing cells. For example, in Fig. 4.0 consider the two highlighted cells, T_i and T_j^*, in the meshes T and T^* respectively. The amount added to T_j^* from T_i is the piecewise constant value of the function in T_i weighted by the ratio of the shaded area to the total area of T_i. A naive approach would be to do this on a cell by cell basis; in each case finding the fractional areas of all contributing cells. This would be exceedingly cumbersome. Moreover, the assumption of piecewise constant representations on the old mesh leads to a very inaccurate (but conservative) representation on the new mesh. If many grid transfers are required this introduces excessive diffusion into the solution. To obtain higher accuracy the reconstructed piecewise polynomials should be used. In this case the integral average in all fractional areas must be computed. Once again we can simplify this task tremendously by converting these area integrals to simple line integrals. This approach is not new; the algorithm was first reported by Dukowicz [48] and refined by Ramshaw [46,47]. The basic idea is to construct a primitive vector function $\mathbf{H}$ such that

$$\nabla \cdot \mathbf{H} = u \qquad \forall \, u \in \mathcal{P}_k$$

where $k = 0, 1$ for the algorithms of Dukowicz and Ramshaw. The integral average in each new cell c_i^* is the sum of the integral average of all fragments c_{f_j} in T interior to c_i^*. These area integrals then simplify by use of the divergence theorem:

$$\begin{aligned}\int_{c_i^*} u\, da &= \sum_{c_{f_i} \in c_i^*} \int_{c_{f_i}} u\, da = \sum_{c_{f_i} \in c_i^*} \int_{c_{f_i}} \nabla \cdot \mathbf{H}\, da \\ &= \sum_{c_{f_i} \in c_i^*} \int_{\partial c_{f_i}} \mathbf{H} \cdot \mathbf{n}\, dl\end{aligned} \tag{4.1}$$

In the case of linear functions $u(\mathbf{R}) = u_0 + (\nabla u)_0 \cdot \mathbf{R}$, a simple primitive function given in [47] is

$$\mathbf{H} = \frac{1}{2} u_0 \mathbf{R} + \frac{1}{3}((\nabla u)_0 \cdot \mathbf{R})\mathbf{R} \tag{4.2}$$

This idea extends naturally to higher order functions. In order to carry out the line integrals (4.1) each edge of T and T^* is further subdivided into segments delimited by intersections with edges of T^* and T respectively. Each segment of T lies interior to a single cell of T^*. Likewise, each segment of T^* lies interior to a single cell of T. Thus for each segment of T (T^*) we record the cell location in T^* (T) which contains the segment. The transfer process is carried out by integrating (4.1) for all segments of T and T^*. Segments of T separate two adjacent piecewise polynomials. First primitive functions are constructed for each piecewise polynomial. Then both are integrated (with properly oriented normal) from which the result contributes to the cell average of the single cell in T^*. Segments of T^* require a single piecewise polynomial from T. After constructing the primitive and integrating along an segment, the result contributes (positively or negatively depending on the orientation of the normal) to the two cells of T^* adjacent to the segment. Dividing the results in each cell of T^* by the cell area, the calculation is complete.

5. References.

1. Finkel, R. A., and Bentley, J. L., "Quad Trees, A Data Structure For Retrieval of Composite Keys", Acta Informatica, Vol. 4, No. 1, 1974.
2. Knuth, D. L., **The Art of Computer Programming**, Vol. 3, Addison Wesley, 1973.
3. Bowyer, A., "Computing Dirichlet Tessellations", The Computer Journal, Vol. 24, No. 2, 1981, pp. 162—166.
4. Green, P. J. and Sibson, R., "Computing the Dirichlet Tesselation in the Plane", The Computer Journal, Vol. 21, No. 2, 1977, pp. 168—173.
5. Guibas, L. and Stolfi, J., "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams", ACM Transactions on Graphics, Vol. 4, No. 2, April 1985, pp. 74–123.
6. Preparata, F.P. and Shamos, M.I., **Computational Geometry**, Springer-Verlag, 1985.
7. Baker, T. J., "Three Dimensional Mesh Generation by Triangulation of Arbitrary Point Sets", AIAA 87-1123-CP, 1987.
8. Lawson, C. L., "Software for C^1 Surface Interpolation", Mathematical Software III, (Ed., John R. Rice), Academic Press, New York, 1977.

9. Rippa, S., "Minimal Roughness Property of the Delaunay Triangulation", Phd Thesis in preparation, School of Mathematical Sciences, Tel-Aviv University, 1989.
10. Babuška, I., and Aziz, A. K., "On the Angle Condition in the Finite Element Method", SIAM J. Numer. Anal., Vol. 13, No. 2, 1976.
11. Löhner, R. and Parikh, P, "Generation of Three-Dimensional Unstructured Grids by the Advancing Front Method",AIAA Paper 88-0515, Reno, Nevada, January, 1988.
12. Edelsbrunner, H., **Algorithms in Combinatorial Geometry**, Springer-Verlag, 1987.
13. Klee, V., "On the Complexity of d-dimensional Voronoi diagrams", Archiv der Mathematik, Vol. 34, 1980.
14. Gilbert, P.N., "New Results on Planar Triangulations", Tech. Rep. ACT-15, Coord. Sci. Lab., University of Illinois at Urbana, July 1979.
15. Nira, D., Levin, D., Rippa, S.,"Data Dependent Triangulations for Piecewise Linear Interpolation", Manuscript, School of Mathematical Sciences, Tel-Aviv University, 1989.
16. Nira, D., Levin, D., Rippa, S.,"Algorithms for the Construction of Data Dependent Triangulations", Manuscript, School of Mathematical Sciences, Tel-Aviv University, 1989.
17. Merriam, M. M.,"A Fast Robust Algorithm for Delauny Triangulation", unpublished manuscript, CFD Branch, NASA Ames, 1990.
18. Carey, G. F., and Oden, J. T., **Finite Elements**, Prendice Hall, New Jersey, 1983.
19. Ciarlet, P.G., **The Finite Element Method for Elliptic Problems**, (Ed. Lions, Papanicolaou, and Rockafellar), North Holland/Elsevier, 1978.
20. Ciarlet, P.G., Raviart, P.-A., "The Combined Effect of Curved Boundaries and Numerical Integration in Isoparametric Finite Element Methods", **The Mathematical Foundations of the Finite Element Method with Application to Partial Differential Equations**, (Ed., A.K. Aziz), Academic Press, New York, 1972, pp. 409-474.
21. Roe, P.L.,"Error Estimates for Cell-Vertex Solutions of the Compressible Euler Equations", ICASE report 87-6, 1987.
22. Ciarlet, P.G., Raviart, P.-A., "Maximum Principle and Uniform Convergence for the Finite Element Method", Comp. Meth. in Appl. Mech. and Eng., Vol. 2., 1973, pp. 17-31.
23. Godunov, S. K., "A Finite Difference Method for the Numerical Computation of Discontinuous Solutions of the Equations of Fluid Dynamics", Mat. Sb., Vol. 47, 1959.
24. Struijs, R., Deconinck, H., "Multidimensional Upwind Schemes for the Euler Equations Using Fluctuation Distribution on a Grid Consisting of Triangles", 8th GAMM Conference on Numerical Methods in Fluid Mechanics, Delft University of Technology, September 27-29, 1989.
25. Van Leer, B., "Towards the Ultimate Conservative Difference Schemes V. A Second Order Sequel to Godunov's Method", J. Comp. Phys., Vol. 32, 1979.
26. Colella, P., Woodward, P., "The Piecewise Parabolic Method for Gas-Dynamical Simulations", J. Comp. Phys., Vol. 54, 1984.
27. Woodward, P., Colella, P., "The Numerical Simulation of Two-Dimensioal Fluid Flow with Strong Shocks" J. Comp. Phys., Vol. 54, 1984.
28. Harten, A. , Osher, S., "Uniformly High-Order Accurate Non-oscillatory Schemes, I.," MRC Technical Summary Report 2823, 1985.

29. Harten, A., Engquist, B., Osher, S.,Chakravarthy, "Uniformly High Order Accurate Essentially Non - Oscillatory Schemes III, ICASE report 86-22, 1986.
30. Barth, T. J., and Jespersen, D. C., "The Design and Application of Upwind Scehemes on Unstructured Meshes", AIAA-89-0366, Jan. 9-12, 1989.
31. Barth, T. J., and Frederickson, P. O., "Higher Order Solution of the Euler Equations on Unstructured Grids Using Quadratic Reconstruction", AIAA-90-0013, Jan. 8-11, 1990.
32. Struijs, R., Vankeirsblick, and Deconinck, H., "An Adaptive Grid Polygonal Finite Volume Method for the Compressible Flow Equations",AIAA-89-1959-CP, 1989.
33. Slack, D. C., Walters, R. W., and Löhner, R., "An Interactive Adaptive Remeshing Algorithm for the Two-Dimensional Euler Equations", AIAA-90-0331, Jan. 8-11, 1990.
34. Whitaker, D. L., Slack, D. C., and Walters, R. W., "Solution Algorithms for the Two-Dimensional Euler Equations on Unstructured Meshes",AIAA-90-0697.
35. Desideri, J. A., and Dervieux, A., "Compressible Flow Solvers Using Unstructured Grids", VKI Lecture Series 1988-05, March 7-11, 1988, pp. 1—115.
36. Börk, Å, "Least Squares Methods", To appear in **Handbook of Numerical Analysis**, Vol. 1: Solutions of Eqns. in R^n, (Ed. Ciarlet, Lions), Elsevier.
37. Chiocchia, G., " Exact Solutions to Transonic and Supersonic Flows", AGARD Advisory Report AR-211,1985.
38. Jameson, A. and Mavriplis, D., "Finite Volume Solution of the Two-Dimensional Euler Equations on a Regular Triangular Mesh", AIAA paper 85-0435, January 1985.
39. Mavriplis, D., "Multigrid Solution of the Euler Equations on Unstructured and Adaptive Meshes", ICASE Report No. 87-53.
40. Mavriplis, D. and Jameson, A., "Multigrid Solution of the Two-Dimensional Euler Equations on Unstructured Triangular Meshes', AIAA paper 87-0353, January 1987.
41. Morgan K., Peraire J.,, "Finite Elements for Compressible Flows", VKI Lecture Notes, 1988.
42. Lohner, R., Morgan K., and Peraire J., "Finite Elements for Compressible Flow", Numerical Methods for Fluid Dynamics (K.W. Morton and M.J. Baines eds.) Oxford University Press, 1986.
43. Shakib, F., "Finite Element Analysis of the Compressible Euler and Navier-Stokes Equations", PhD Thesis, Department of Mechanical Engineering, Stanford University, 1988.
44. Rostand, P. and Stoufflet, B., "A Numerical Scheme for Computing Hypersonic Viscous Flows on Unstructured Meshes", Second International Conference on Hyperbolic Problems, Aachen, W. Germany, 1988.
45. Roe, P.L.,"Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes", J. Comput. Phys., Vol 43, 1981.
46. Ramshaw, John D., "Conservative Rezoning Algorithm for Generalized Two-Dimensional Meshes", J. Comp. Phys., Vol. 59, 1985.
47. Ramshaw, John D., "Simplified Second-Order Rezoning Algorithm for Generalized Two-Dimensional Meshed",J. Comp. Phys., Vol. 67, 1986.
48. Dukowicz, J. K., "Conservative Rezoning Algorithm for General Quadrilateral Meshes" J. Comp. Phys., Vol. 54, 1984.

Appendix A – Fortran Implementation

Although tree searches are inherently recursive, they can be implemented in a non-recursive language such as FORTRAN. FORTRAN code for a split tree is listed here. The subroutine sptree2 sets up the tree, the function clpnt2 searches through it. The non-standard features "do while" and "enddo" , as well as a non-standard comment syntax, are used for clarity and brevity. If necessary, all these features can be translated into standard FORTRAN.

```
c
c  This routine generates a roughly balanced, alternating
c  direction, binary tree for two dimensional data in the array xy.
c  The tree consists of the permutation list perm
c  and the array of pointers, The pointer array is sized
c  here for a worst case.  It would usually be much smaller
      subroutine sptree2(xy,dnpt,npt,pointer,perm)
      integer levels, buckets
      parameter (levels = 50, buckets = 5000)
      integer dnpt, npt, pointer(buckets,7), perm(dnpt)
      real*8 xy(dnpt,2)
c     Local variables
      real*8 sum,avg,divval
      integer p,p2,save,npntr,lpntr,nodes,dir,first,last,pnode,
     >     child1,child2,divnode,nmin,levptr,level
      nmin = 18     ! theoretically independent of npts
c     first pointer
      pointer(1,1) = 1   !first point in section (permuted list)
      pointer(1,2) = npt !last point in section  (permuted list)
      pointer(1,3) = npt !number of points in section
      pointer(1,4) = 1   !direction for next split, 1 = x, 2 = y
c     initialize permutation list
      do p = 1,npt
         perm(p) = p
      enddo
c     main loop
      npntr = 1
      lpntr = 1
      levptr = 1    ! last bucket on this level
      level  = 1
      do while (npntr .le. lpntr )
         nodes = pointer(npntr,3)
c     don't split buckets that can easily be searched
c     don't exceed the dimensions of pointer
c     don't exceed the dimensions in the retrieval routine
        if ((nodes .gt. nmin) .and.
     >       (lpntr+2 .le. buckets) .and.
     >       (level .lt. levels))        then
```

```
c     decide whether to split in x or y
                  dir = pointer(npntr,4)
c     decide where to split (compute average)
                  first = pointer(npntr,1)
                  last  = pointer(npntr,2)
                  divnode = perm(first)
                  divval = xy(divnode,dir)
                  sum = 0
                  do p = first,last
                     pnode = perm(p)
                     sum = sum+ xy(pnode,dir)
                     if (xy(pnode,dir) .lt. divval) then
                        divval = xy(pnode,dir)
                        divnode = pnode
                     endif
                  enddo
c     avg is a rough approximation for the median
                  avg = sum/nodes
c     sort into two piles
                  p2 = first-1
                  do p = first,last
                     pnode = perm(p)
                     if (xy(pnode,dir) .le. avg) then
                        p2 = p2 +1
                        save = perm(p2)
                        perm(p2) = perm(p)
                        perm(p) = save
                        if (xy(pnode,dir) .gt. divval) then
                           divnode = pnode
                           divval = xy(pnode,dir)
                        endif
                     endif
                  enddo
c     update pointers
                  child1 = lpntr+1
                  child2 = lpntr+2
                  pointer(npntr,5)= child1
                  pointer(npntr,6)= child2
                  pointer(npntr,7)= divnode
                  pointer(child1,1) = first
                  pointer(child1,2) = p2
                  pointer(child1,3) = p2-first +1
                  pointer(child1,4) = 3 - dir !toggles between 1 and 2
                  pointer(child2,1) = p2+1
                  pointer(child2,2) = last
```

```
                pointer(child2,3) = last - p2
                pointer(child2,4) = 3 - dir !toggles between 1 and 2
                lpntr = lpntr+2
          else
             pointer(npntr,5) = 0   !signal that this block isn't split
          endif
c     next pointer
          if (npntr .eq. levptr) then
             level = level + 1
             levptr = lpntr
          endif
          npntr = npntr+1
       enddo
       print *,'tree consists of ',lpntr,' buckets',
     >        ' on ',level-1, ' levels'
       print *,'No bucket contains more than ',nmin, ' nodes.'
       return
       end

c      Returns the index of the point closest to x,y
c      among those which satisfy (userfun(p) .eq. .true.)
c      Points farther away than sqrt(dsq) are ignored, unless
c      dsq is negative, in which case this test is skipped.
c
c      If no points are found, the value -1 is returned.
c      This can happen if dist is small or userfun is restrictive
c
       integerfunction clpnt2(xy2,xy,dnpt,userfun,dsq,pointer,perm)
       integer levels, buckets
       parameter (levels = 50, buckets = 5000)
       integer dnpt,pointer(buckets,7),perm(dnpt),dbg
       real*8 xy2(2),dsq,xy(dnpt,2)
c      Don't forget!  Declare userfun external in calling program
       logical userfun
       external userfun
c      Local variables
       real*8 xydist(levels,2,2),mindist,xnode,ynode,distsq,ave,dsq2
       integer pout,level,branch(levels),point(levels,2)
       integer pnum,nodes,first,last,dir,child1,child2,p
       integer pnode,nearer,farther,avep
       logical nopoint
       pout = -1          !the node number in question (not permuted)
       level = 1          !avoid over running arrays by counting levels
       branch(1) = 2      !the whole is the second branch on level 1
       point(1,2) = 1     ! which starts at pointer number 1.
```

```
      nopoint = (dsq .lt. 0)
      xydist(level,branch(level),1) = 0
      xydist(level,branch(level),2) = 0
      dsq2 = dsq
      do while(level .ge. 1)
c     Tree pruning occurs here
         mindist =  xydist(level,branch(level),1) +
     >              xydist(level,branch(level),2)
         if (mindist .lt. dsq2 .or. nopoint) then
            pnum = point(level,branch(level))
            child1 = pointer(pnum,5)
            nodes = pointer(pnum,3)
c     If there are no more children, search exhaustively
            if (child1 .eq. 0)then
               first = pointer(pnum,1)
               last  = pointer(pnum,2)
               do p = first,last
c     compute distance
                  pnode = perm(p)
                  xnode = xy(pnode,1)
                  ynode = xy(pnode,2)
                  distsq = (xnode-xy2(1))**2 + (ynode-xy2(2))**2
c     compare with dsq
c     see if it passes userfun
                  if (nopoint .or. distsq .lt. dsq2) then
                     if (userfun(pnode)) then
                        pout = pnode
                        nopoint = .false.
                        dsq2 = distsq
                     endif
                  endif
               enddo
               branch(level) = branch(level) + 1
            else
c     decide which child to try first
               dir     = pointer(pnum,4)
               child1  = pointer(pnum,5)
               child2  = pointer(pnum,6)
               avep    = pointer(pnum,7)
               ave     = xy(avep,dir)
               xydist(level+1,1,1) = xydist(level,branch(level),1)
               xydist(level+1,1,2) = xydist(level,branch(level),2)
               xydist(level+1,2,1) = xydist(level,branch(level),1)
               xydist(level+1,2,2) = xydist(level,branch(level),2)
               xydist(level+1,2,dir) = (xy2(dir)-ave)**2
```

```
                  if (xy2(dir) .le. ave) then
                     nearer  = child1
                     farther = child2
                  else
                     nearer  = child2
                     farther = child1
                  endif
                  level          = level + 1
                  point(level,1) = nearer
                  point(level,2) = farther
                  branch(level) = 1
               endif
            else
c               branch(level) = branch(level)+1
               branch(level) = 3
            endif
c     decide which pointer to try next
            do while (branch(level) .gt. 2 .and. level .ge. 2)
               level          = level -1
               branch(level) = branch(level) + 1
            enddo
            if (level .eq. 1 .and. branch(level) .gt. 2) level = 0
         enddo
         clpnt2 = pout
         return
         end
```