T-3273

APPLICATION OF A MONOTONIC DATA STRUCTURE TO AN IRREGULAR SIMULATION GRID

by

Michael D. Kelly

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

T-3273

A thesis submitted to the Faculty and Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Mathematics).

Golden, Colorado

Date 8/21/86

Signed: Michael D. Kelly
Michael D. Kelly

Approved: Jean Bell
Dr. Jean Bell
Thesis Advisor

Golden, Colorado

Date 8\21\86

Ardel Boes
Dr. Ardel Boes
Professor and
Department Head
Mathematics

ABSTRACT

This thesis investigates a newly proposed dynamic data structure designed to address several problems which arise in certain types of large scale physical simulation problems. The particular simulation problems studied are in the area of Hydrodynamics and involve a "Free Lagrangian" numerical method, developed by M.J. Fritts and J.P. Boris (Fritts & Boris,78).

This thesis offers a fundamental solution to the following problem inherent to the above mentioned numerical method. The numerical method uses a triangular connectivity. Data partitioning of the triangles is necessary to take advantage of parallel processing or multiple memory levels, but such partitioning has been virtually impossible due to the fact that an inappropriate list data structure is being used (Bell & Patterson,85). The access pattern of a list is random, and therefore eliminates vectorization of numerical computations.

A vectorized "Nearest Neighbor" algorithm, developed by Jay P. Boris of the Naval Research Laboratory (Boris,85), which utilizes modern parallel processing computer architectures is introduced and analyzed with respect to the solution of problems stemming from use of

the list data structure.

The algorithm is a dynamic data structure known as a Monotonic Logical Grid (MLG). The MLG lends itself to vectorization, and is partitionable to take advantage of multi-processor environments. In addition the MLG uses indexes of main memory arrays and contiguous memory locations to reduce particle-to-particle relationship calculations and near neighbor search times.

The MLG's structure is derived from and contains spatial relationships similar to the relationships of the simulation grid it represents. Since particle movement may violate the laws that govern the structure of the MLG, algorithms used to update the MLG are also investigated. Finally, new algorithms which utilize the properties of the MLG to search for neighboring elements are introduced and analyzed.

TABLE OF CONTENTS

TABLE OF CONTENTS continued

TABLE OF CONTENTS continued

LIST OF FIGURES

LIST OF FIGURES continued

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Dr. Jean Bell for all the help that she offered during the development and writing of this thesis, and two good friends John Barkmier and Pat Quist for many good ideas in the development of this study.

Personally, I would like to thank my parents for making all that I have done at Colorado School of Mines possible and deep thanks to my wife Laurel for putting up with all that has gone into trying to finish this work.

INTRODUCTION

Simulation models in the area of transient hydrodynamics which contain free surfaces, fluid interfaces, and fluid boundaries are approached most readily by means of Lagrangian methods using a rectangular mesh (Fritts & Boris,78). However, Lagrangian methods have, in the past, been restricted to "well behaved" flows since point movement will, in time, distort the differencing mesh to the extent of inaccurate numerical calculation and deterioration of the numerical method being used. Physical phenomenon modeled include breaking waves (smooth waves which turn turbulent) and shear flow (the interface between two different fluids which are moving parallel to each other with different velocity magnitudes).

M.J. Fritts, J.P. Boris and W.P. Crowley have introduced a new meshing technique which uses triangles as general mesh connections. Certain geometric properties of the triangle make their use advantageous over polygons of larger order. One advantage is the relatively easy restructuring of the mesh after point movement has caused point crossing or disconnection. Another advantage is the ability of the triangle to cover a surface with less cusps

and local irregularities as well as handle exterior fluid borders, interior object borders, and fluid interfaces.

Unfortunately management of point data for the triangular Lagrangian mesh still lacks the efficiency and speed demanded by real life problems. The use of the triangular mesh is in conjunction with a finite difference numerical method, which means that information about neighboring points and neighboring triangles must be readily available for point information updates. Since the triangular mesh may at times be very irregular in composition, there is no clear cut mapping between the grid points and a data structure that will render data quickly and efficiently. For instance if we are considering simulating a flow in two dimensions, there is no apparent mapping between the triangles of the flow space and a two dimensional "array" which would contain information about the flow.

Consequently, a list data structure is used to store point information. The list data structure is applicable in this case from the viewpoint that point updating is not required to be sequential. Only information about points and triangles near the point in question need be present during updates. Unfortunately, searching this list for information about any given point or triangle is very time

consuming and for large simulation problems (10^7 - 10^3 points), is intolerable.

Jay P. Boris of the Naval Research Laboratory has developed an algorithm which proposes to eliminate the forgoing problem as well as utilize performance intensive machine attributes such as vectorization and parallel processing.

The algorithm is based on a data structure known as a Monotonic Logical Grid (MLG). This data structure bridges the gap between the tesselated flow space and its structural representation in the data base, as well as lending itself to vectorization of numerical operations and parallel processing. The MLG gives us the mapping necessary to go between the flow space and the data structure while still preserving the spatial relationships between points in the space. As will be demonstrated later, because of the definition of the MLG, point movement in space implies an actual physical movement of data in the data base, giving us a dynamic data structure.

The Free Lagrangian Grid uses a special finite difference numerical method as the underlying mathematical model. Adjacent cell computation in a regular finite difference method now becomes adjacent triangle (or totality of triangles around a point) computation. Spatial

relativity is inherent to the MLG data structure, therefore reducing search times and search lengths for adjacent or surrounding triangles quite substantially by localizing spatially related data.

We will investigate several different characteristics of the MLG, as well as different governing schemes, in order to optimize its performance. One such scheme is the method of determining and maintaining the partitioned data sets. This involves determining which triangle attribute (i.e. triangle center, one particular vertex, etc.) will determine the best partition of data for parallel processing. One characteristic of the MLG to be studied is the preservation of spatial relationships between points in the space with respect to the data present in the data base. Test results of MLG performance show that the spatial "nearness" of one point to another point in the space is reflected and very apparent in the data base.

This locality of data is then applied to reducing the amount of time spent and length of search conducted in order to determine adjacent and bordering triangles. Test results show that overall the MLG produces much reduced search lengths for neighboring and bordering triangle information.

Algorithms have also been developed, and will be explained in later chapters, which use the data locality of the MLG and certain geometric properties of triangles to conduct searches of the database to find triangles for point update computations. In these algorithms a comparison is made between a scheme using more main memory storage and a scheme using more CPU time requirements. Results show that the scheme using more main memory storage decreases the search times involved in finding adjacent triangles, whereas the scheme involving less storage obviously increases search times while decreasing main memory requirements.

Although the physical models developed in this study are quite fundamental and at times seem to be quite well behaved, the algorithms produced were developed with emphasis on independence of data partitioning relative to the intricate particulars of the flow equation being used. In other words when the algorithms were developed, special care was taken not to construct code which depended on the flows to be tested.

And, as fluid motion changes, certain telltale traits of the motion might demand that certain parameters being used to partition the data be changed. This change in parameters is handled efficiently by the algorithms

developed and thusly add to the flexibility of the MLG.

Overall test results obtained from this study support the use of the MLG as a data structure for highly transient hydrodynamic simulation problems that require preservation of locally related spatial information (e.g. finite difference numerical methods) in order to establish partitionable data sets which will take advantage of supercomputer attributes such as parallel processing and vectorization of computations.

Certain limits are put upon the tests conducted in this study. In the Free Lagrangian models, the reconstruction of a corrupted grid is very important in maintaining a reliable mathematical model. The algorithms developed for this study do not reconstruct a grid if triangle inversion (crossing of triangle sides) occurs, nor do they reconstruct triangles in order to maintain even grid resolution. The reason for this is that the algorithms for grid bookkeeping are complex and too difficult to incorporate into this study at the present time.

Therefore, all flows applied to the points of the space are restricted so as to produce small scale movement in the space for up to approximately 5-8 time steps.

CHAPTER 1 - THE FREE LAGRANGIAN MODEL

1.1 INTRODUCTION

In modeling highly transient fluid flow in hydrodynamics, Lagrangian methods have been extensively used and developed. The need for a dynamic meshing grid, a grid which actual follows fluid flow, as well as a method for simplification of the numerical equations (elimination of the term which accounts for fluid movement through the grid), were some of the factors in the motivation behind the Lagrangian grid.

Jay P. Boris, Marty J. Fritts, and W.P. Crowley have been conducting research in the development of meshing grids for use in these difficult hydrodynamic simulation problems. One particular result of their work which is the fundamental building block of this study is a meshing technique called the Lagrangian grid (Fritts & Boris,79).

1.2 THE LAGRANGIAN GRID

Lagrangian methods for solutions to fluid flow problems differ from conventional differencing methods in that the reference point of the observer changes between the two methods. In a conventional finite difference method using a "fixed grid" mesh, the observation space is tesselated into a regular two, or three dimensional

lattice with orthogonal connections between lattice points (fig. 1.1). Once the simulation has begun, points move through the mesh while the observer calculates point interactions relative to the the stationary meshing grid. Relationships between points of the space are related to the grid cell in which they happen to be at any given time in the simulation run.

In Lagrangian methods, the meshing grid is constructed using the actual points, or a subset of points, being monitored in the observation space as the grid cell connections (fig. 1.2). When point movement in the observation space takes place the meshing grid cells move and distort in conjunction with the observation points. Thus, the observers reference point moves relative to the points of the fluid flow.

As can be seen in figures 1.3 and 1.4, if fluid flow were to create the situation indicated, difficult grid resolution representation is forced on the fixed grid mesh. For example, point migration to one section of the grid would eventually cause more than one point to be present in a single grid cell. Because of the use of a finite difference numerical method the resolution of the grid would have to be increased over the entire grid as a

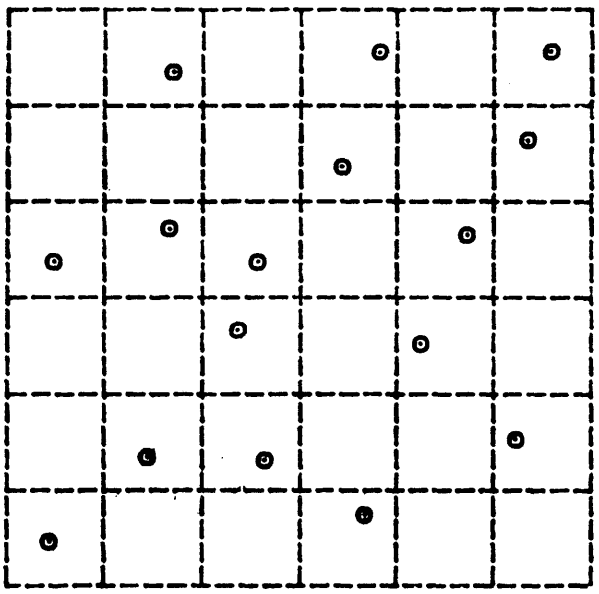


FIG. 1.1: Typical 2-D fixed lattice grid

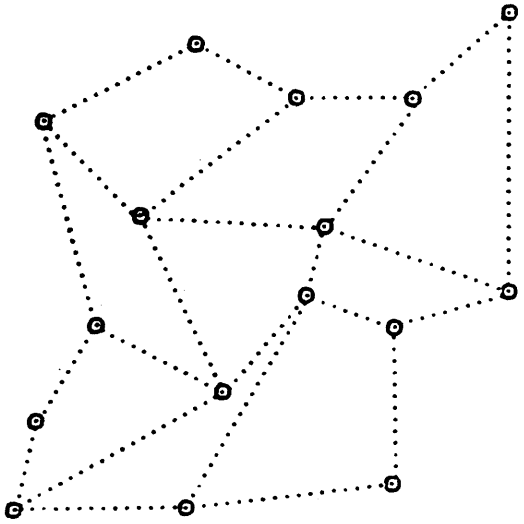


FIG. 1.2: Typical rectangular 2-D Lagrangian grid

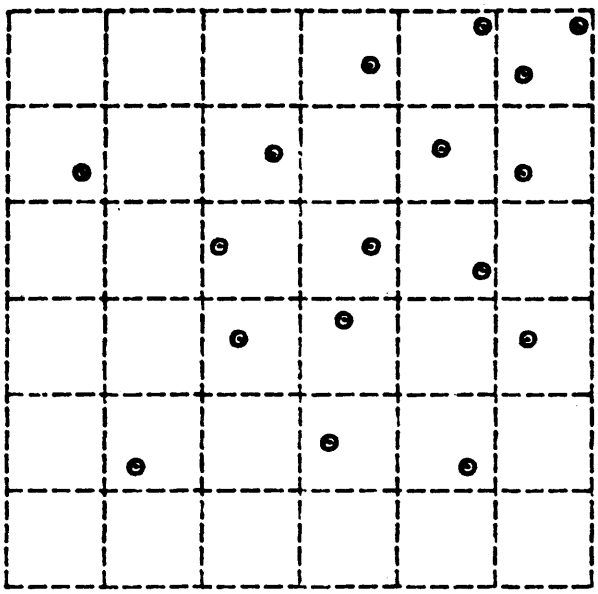


FIG 1.3: Point migration in a fixed lattice grid

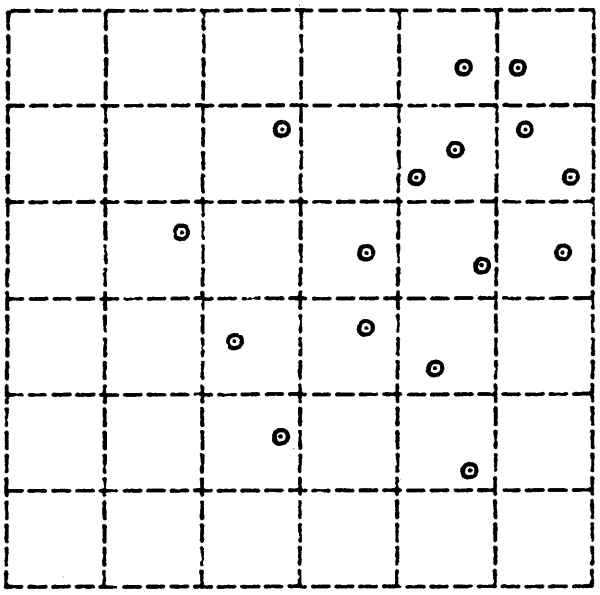


FIG. 1.4: High and low sectional point resolution

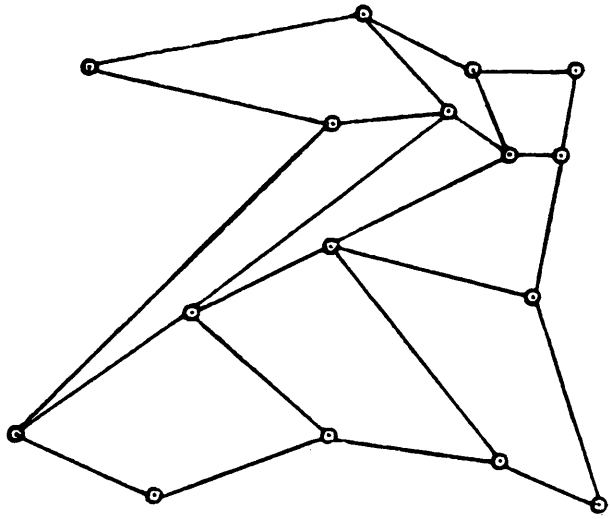


FIG. 1.5: Point migration in a rect. Lagrangian grid

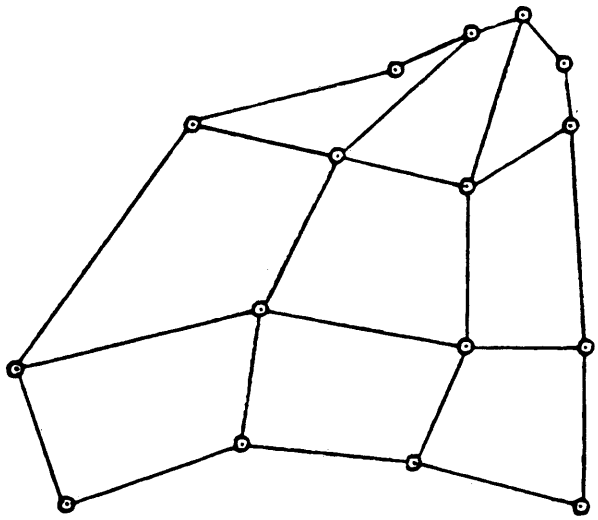


FIG. 1.6: Sectional point resolution in Lagrangian grid one particular area.

result of the need for grid resolution increase in this

Alternately, the Lagrangian grid conforms to the fluid flow (fig. 1.5 and 1.6) making grid resolution related to rectangular area. In reference to figures 1.5 and 1.6, large area in a certain rectangle indicates low resolution of points in that particular section of the grid. Thus, grid resolution is localized, meaning fine resolution in one area of the grid does not imply fine resolution throughout the grid. Therefore, representation of the grid resolution as given by the Lagrangian model need not suffer in one section of the grid because of point movement to another, as is the case in the use of the fixed meshing grid. This is not to say that the fixed grid method does not have its place in transient hydrodynamic simulation problems. If point movement throughout the space is well behaved and restricted to small incremental migrations over time then the fixed grid method would suffice as a meshing scheme. Unfortunately, the method breaks down when point movement is drastic and grid resolution is variable and quite diverse throughout the grid.

One obvious drawback to the use of a rectangular Lagrangian mesh is that over time, the mesh becomes

distorted to the extent that point motion could, and most likely would, cause crossing of rectangle sides (fig. 1.7). This situation renders the grid useless. Restructuring of the grid (i.e. reconnection or reconfiguration of points in order to "uncross or reconnect" rectangle sides) is a logical step in rectifying the problem.

This approach however, is difficult to achieve because

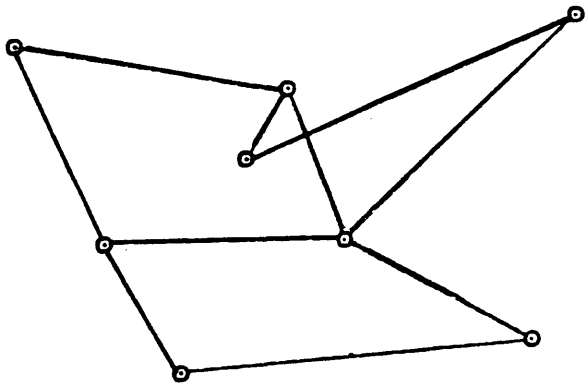


FIG. 1.7: Grid corruption due to point motion

of the topology of the rectangular mesh. The addition or deletion of grid points in hopes of correcting the misshaped grid may not be possible with the insertion or deletion of one or more points in the grid. In fact the determination of the number of points to add or delete, as well as the sides to add or delete, may be quite difficult to establish.

Another drawback to the use of a rectangular mesh is the representation of complex boundaries and structures. The topology of the rectangular mesh is such that irregularities may be present in boundary or free surface representations of the fluid, thereby presenting less than desirable curvature or resolution.

Finally, "Rectangular mesh approaches appear to suffer a serious 'even-odd' or computational-mode instability..." (Fritts & Boris,79) which necessitates a form of numerical dampening, in turn destroying the reversibility of the simulation.

1.3 THE FREE LAGRANGIAN GRID

Geometric properties of the triangle offer a path around the difficulties of grid restructuring and boundary representation inherent to a rectangular mesh. As a Lagrangian mesh distorts with point movement through time, restructuring of the grid is imperative, and a triangular mesh lends itself to relatively easy restructuring. Boris, Fritts, and Crowley are credited with the idea of a triangular Lagrangian mesh called a Free Lagrangian Grid (Fritts & Boris,79).

The Free Lagrangian grid offers an answer to the grid reconstruction problem. As the triangular mesh becomes

distorted, restructuring of grid points or the insertion/deletion of triangles and points becomes relatively easy. Insertion of just one point into the grid can be accomplished with the guarantee that a triangular mesh can be redefined. This argument involves three cases. Case one involves the insertion of a point outside the boundary established by the points in the space. All that is needed to create at least one triangle are the two nearest points to the inserted point. From there, more triangles may be created using points near the new point as long as no triangle sides are crossed in the creation (fig 1.8).

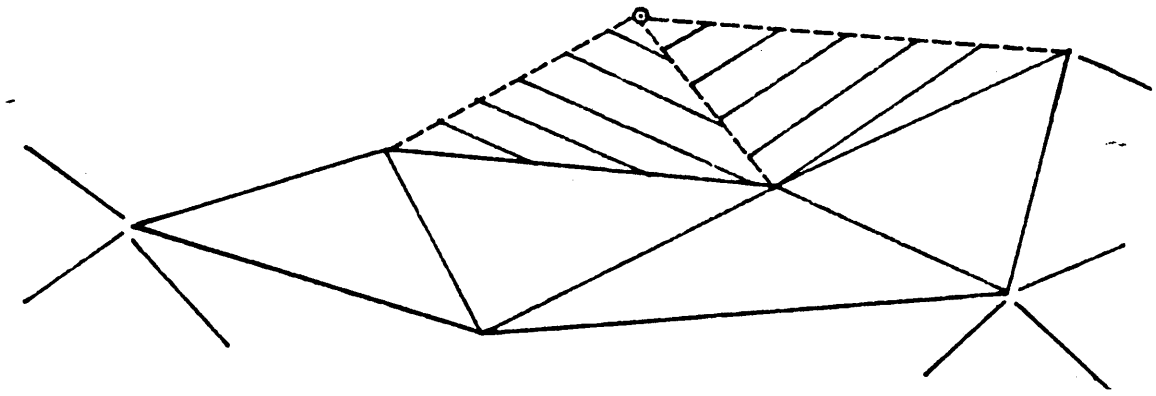


FIG. 1.8: Point insertion - boundary exterior

Case two and three are the basis of two grid update routines to be discussed shortly and involve the insertion

of a point onto an existing triangle side and insertion of a point on the interior of an existing triangle, respectively.

As the Free Lagrangian grid moves in time and becomes distorted the possibility of triangle inversion, or side intersection, becomes greater. As an example, this sort of situation is present when long narrow triangles border larger triangles. When this "mismatch" of triangle areas occurs the numerical method used becomes less accurate because of the difference in relative areas. One method of solution is a special reconnection algorithm (Fritts & Boris,79). In the algorithm it is noted that "every non-boundary line uniquely specifies its bordering triangles". Once the two bordering triangles are established, the two possible diagonals of the quadrilateral that is formed are computed and the shortest of the two are used as the new border line (fig. 1.9).

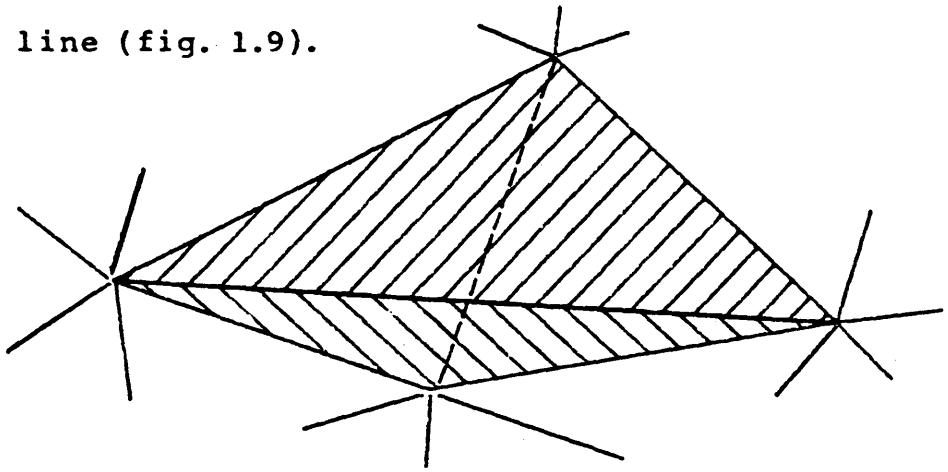


FIG. 1.9: Triangle reconstruction

If a change in grid resolution is needed, the insertion or deletion of points as well as reformulation of triangles in the grid will produce the desired results. Point insertion or deletion can be accomplished in at least two ways. First of all consider the case of increasing grid resolution. One method is called Triangle Side Bisection (Fritts & Boris,79),(fig. 1.10).

One obvious advantage to the side bisection insertion method is the increase in accuracy of curve representation at fluid boundaries and interfaces. This comes as a result of shorter line segment lengths which in turn produce finer curvature.

Insertion of points within the boundaries of a triangle is another method of grid resolution increase and is called Triangle Trisection (Fritts & Boris,79), (fig. 1.11).

As for point deletion from the grid (decrease in grid resolution), the procedures are the inverses of the insertion methods, be it triangle side bisection or triangle trisection.

As can be seen there are no ambiguous considerations to be made when restructuring a triangular mesh. If more triangles are needed in a section of the space in order to increase grid resolution, then point addition, and

consequently triangle addition, in that section will achieve the desired resolution. The same is true for a decrease in grid resolution, accomplished by point and triangle deletion.

Now that the grid configuration is set up, certain definitions concerning the grid for purposes of numerical computations must be made as well as an overview of the numerical method and the demands it makes on data management.

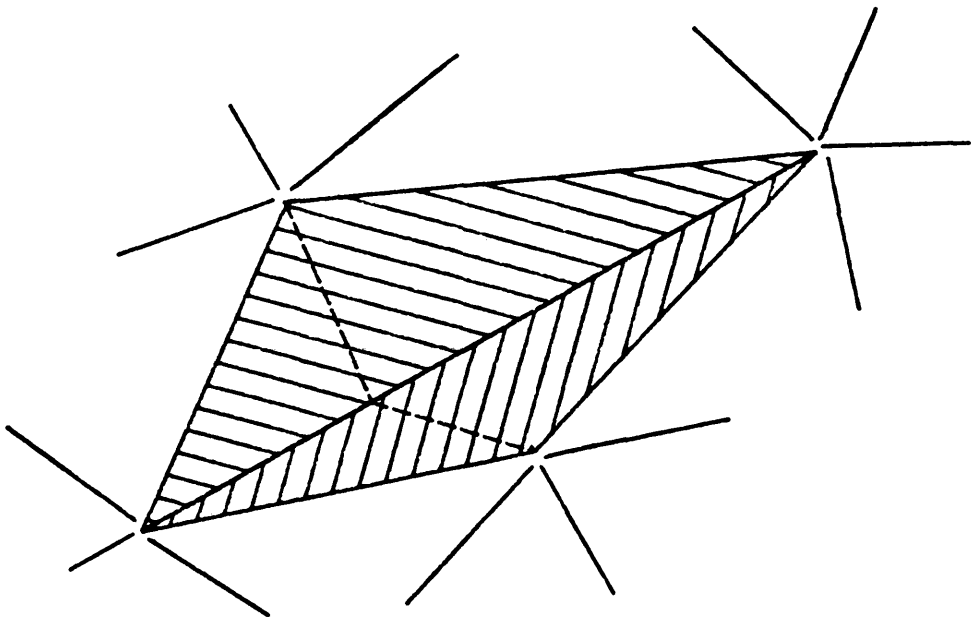


FIG. 1.10: Side bisection

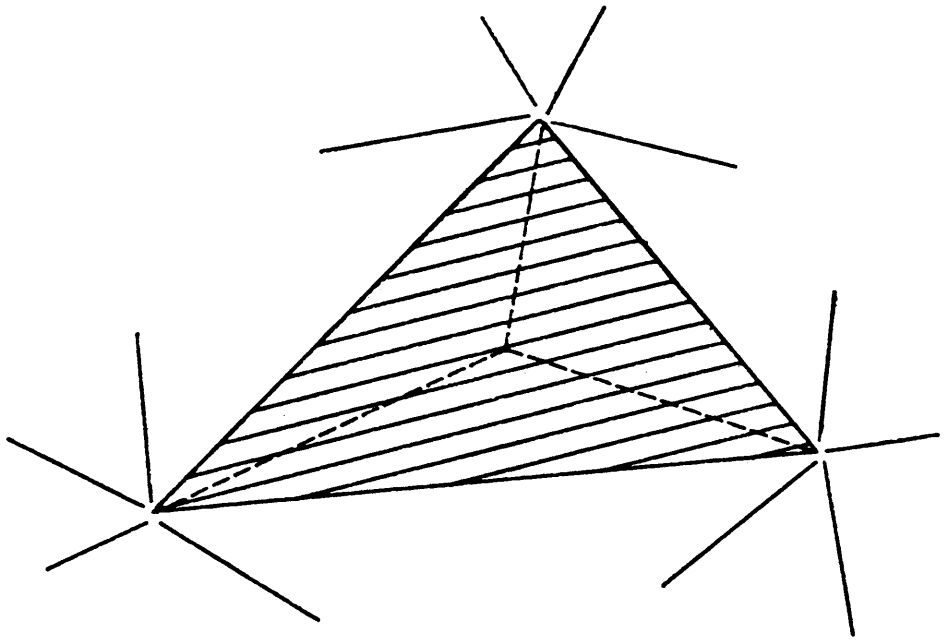


FIG. 1.11: Triangle trisection

1.4 THE NUMERICAL METHOD

One particular model built around simulation problems in transient hydrodynamics is a model called SPLISH which is currently being developed by Fritts and Boris. The numerical method used in this model is called an implicit finite difference method on a Lagrangian grid (Bell,82).

Examples of physical phenomenon modeled by SPLISH are breaking waves, shear flow (involving the interface between two fluids with parallel velocity vector fields but different magnitudes), Rayleigh-Taylor instabilities, droplet burning, and air flow over hydrofoils.

These different types of phenomena exhibit highly transient fluid flows in which the numerical representation can be quite difficult to model. Thus the Lagrangian grid was used in hopes of eliminating some of the computational problems which arise from these types of flows. SPLISH has certain data management requirements which will be quite useful in testing the proposed data structure to be studied within this report.

In actuality the Free Lagrangian grid is used in this particular model (SPLISH) because of its ability to represent fluid boundaries, interfaces, and object surfaces with higher accuracy than grids using polygons of higher order. With the types of fluid motion just mentioned, the need for change in resolution throughout the grid is also present and is handled sufficiently by the triangular grid.

The number of attributes concerning general fluid flow simulations which are of interest to the scientist vary from application to application. One such study may require knowledge about "ideal" fluid characteristics such as vertex velocities and triangle velocities, while another study may require magnetic fields, charges, currents, resistivities and plasma densities in order to model the phenomenon correctly (Bell,82). But as stated in

(Bell,82), the number of attributes to be used by a simulation model may vary between models but, the data management remains very similar.

The scale of the space being monitored also varies from problem to problem. In one situation the overall dimensions may be on the order of centimeters while in another the scale may be in thousands of meters. This slight inconvenience is handled by normalizing the variables so that scale is no longer a problem, "exactly the same behavior is seen on different scales for different problems" (Bell,82).

SPLISH also handles a variety of boundary conditions, from rigid bottom boundaries to periodic vertical boundaries to free surface boundaries at the top of the grid. Bottom boundaries may consist of ocean bottom topography in modeling ocean currents or it may be the earth's surface in modeling atmospheric patterns. The top boundaries may be, in the same models, the ocean surface or one of many levels of the atmosphere, respectively.

As of yet SPLISH is only a two dimensional model using cross-sectional planes of a three dimensional domain as the observation space. In the future the model is to be extended to three dimensions and revised in order to handle more complicated flows such as viscous and

compressible fluids and reactive flow such as burning elements or chemical reaction.

1.5 THE MATHEMATICAL METHOD

SPLISH uses a special finite difference numerical method in which differencing is done over triangular cells within the mesh. Traditional finite difference methods difference over a regular grid mesh. Thus the mathematical equations must be reformulated in terms of the Free Lagrangian representation of the space. There are three main equations in the Lagrangian formulation. They are conservation of mass, conservation of vorticity and Euler's laws of motion (Bell,82 , Fritts & Boris,79). As with most finite difference algorithms, future values to be calculated for one grid cell depend on the neighboring cells around it. Therefore, data management must provide a route to information about surrounding triangles.

In modeling highly transient hydrodynamics, the Free Lagrangian method has created the definition of a Vertex Cell. The vertex cell, used in vertex velocity calculations, is created by taking all triangles around a point and calculating all three side bisectors for each triangle. The intersection of these bisectors locates the triangle's centroid (assuming mass is constant throughout

the triangle) and divides the triangle into six equal-area sub-triangles. Once the surrounding triangles have been divided, two sub-triangles from each of the triangles surrounding the central point are combined to form the vertex cell. A typical vertex cell is shown in figure 1.12.

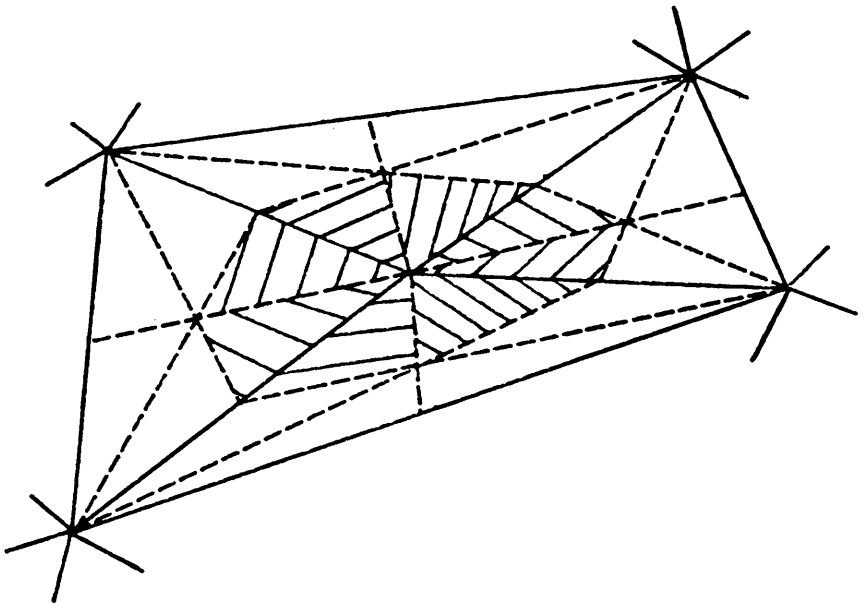


FIG. 1.12: Typical vertex cell

1.6 DATA MANAGEMENT REQUIREMENTS

The Lagrangian formulation of finite difference numerical methods requires the formulation of vertex cells for each vertex update. Therefore knowledge about

triangles that surround points of the space is one requirement put upon the data management process.

This particular numerical method also requires information about triangle areas and centroids. This requirement is handled readily by the fact that only information about triangle verticies is needed to calculate a triangle's area or centroid. In other words, to obtain any particular triangle's area or centroid one need only have the coordinates of the verticies of the triangle. Information about other triangles is not needed.

In the "rotation" part of the numerical method, triangles are checked to make sure that vorticity, or rotational flow, is conserved throughout the grid. In other words, all triangles must be constrained to conserve vorticity from one time step to another. This involves the determination of the triangles which surround any given triangle as well as which of the verticies are connected (Bell,82).

When it comes to the reconstruction of triangle sides and the insertion or deletion of grid points, knowledge about neighboring triangles is again a requirement before determining positions for alterations. For instance, if a triangle is determined to be too large (in terms of area), then its neighboring triangles must be determined in order

to implement either the side bisection insertion or the triangle trisection methods. Similarly, if resolution needs to be increased along a border or fluid interface, then information about which triangles lie on the border (relative position with respect to the overall grid) needs to be gathered along with the information about which side of the triangle actually lies along the border.

This list of data management requirements is in no way exhaustive. The purpose of the forgoing discussion is to present a common prerequisite of the candidate data management facilities. The common factor between the data management requirements is the accessibility of knowledge about neighboring triangles and neighboring points. A spatial relationship inherent to the data structure would give quick and efficient data retrieval and would meet the requirements of the Free Lagrangian Method.

Unfortunately, a very costly data structure in terms of access time and search lengths is presently being used. A list or linked list data structure is currently being used to store point and triangle information. There is no preservation of data locality in a list structure, therefore, when looking for spatially related data, searching must be conducted through the entire list.

A linked list may be the answer to this problem, but

updating of the list and pointers must be performed for all insertions and deletions of points in the space which come about from grid restructuring and resolution changes. This idea becomes very complicated when simultaneous deletions and additions are made and if the list of triangles or points is large, the time involved in updating the list becomes less desirable.

Because of the lack of data locality the hope of even slightly reducing the search lengths by using a partitioned subset of the total data set is abolished.

Arguments to the effect that the list data structure is used because point or vertex updating is not required to be sequential with respect to the point space are understandable, but this argument completely ignores the spatial relationships between the points of the space of which the Lagrangian numerical method requires information.

The problem of excessive access times as well as oversized search lengths to find needed information has hindered the numerical method to the point that a different data structure which will reduce the time spent searching and re-searching long lists must be found.

CHAPTER 2 - THE MONOTONIC LOGICAL GRID

2.1 INTRODUCTION

Jay P. Boris of the Naval Research Laboratory in Washington D.C. has introduced a data structure which eliminates major problems with the list data structure approaches used in the past. The data structure is called a Monotonic Logical Grid (MLG).

The MLG gives us preservation of spatial relationships within the data structure, therefore allowing the data management processes to obtain data quickly in order to service the computational routines being used.

In the MLG, searching begins at a record containing a triangle which is close to the triangle being searched upon, so the length of the search is reduced by the elimination of looking at unrelated data. Since spatially related data is localized in the MLG, search lengths are cut back because searching the entire list is no longer necessary. Therefore, independent subsets of the list are established by storing data in the MLG. The independence of these data sets, in turn, allows us to perform multiple processing.

One obvious difference in the "shapes" of the data

structures is that the MLG uses a data structure which is similar to the spatial data it contains. If the simulation model is run in a three dimensional space then the MLG takes the form of a three dimensional array. The same is true for two and even one dimensions.

As for the placement of the data into the data structure, the method is not random as in the case of the list structure. The placement, and subsequent storage, of data into the MLG is completely dependent on spatial attributes of the elements being stored.

2.2 DEFINITION OF A MONOTONIC LOGICAL GRID

Boris gives the following definition of an MLG (Boris,85):

> For N particles in three dimensions, the arrays of object locations, X(i,j,k), Y(i,j,k) and Z(i,j,k), constitute an MLG if and only if:
>
> $$X(i,j,k) \leq X(i+1,j,k) \text{ for } 1 \leq i \leq NX-1$$
> $$Y(i,j,k) \leq Y(i,j+1,k) \text{ for } 1 \leq j \leq NY-1 \qquad (1.0)$$
> $$Z(i,j,k) \leq Z(i,j,k+1) \text{ for } 1 \leq k \leq NZ-1$$
>
> where NX, NY and NZ are the number of points in the x, y and z directions respectively.

In other words, if the point positions are stored in three dimensional arrays, then all x directional vectors, X(i,j,k) for all $1 \leq i \leq NX$ with fixed j and k, would be monotone increasing with an increase in the index i. All y directional vectors Y(i,j,k) for all $1 \leq j \leq NY$ with fixed i and k, would be monotone increasing with an increase in the j index. And all z directional vectors, Z(i,j,k) for all $1 \leq k \leq NZ$ and fixed i and j, would be monotone increasing with an increase in the k index.

2.3 PROPERTIES OF THE MLG

Implicitly the definition for the MLG states that if the ordering defined by the inequalities 1.0 of section 2.2 is not present in the data structure the point information must be sorted into Monotonic Logical Order (MLO). Figure 2.4 of section 2.4 give us the guarantee that MLO is always possible. An algorithm for sorting the MLG into MLO is described in section 2.4.

But first, what is to be noted from the definition of the MLG is that relationships between points of the space are present in the MLG. If point A is directly next to point B in the space, then by virtue of the spatial coordinates of the two points the information concerning point A is stored directly next to the information

concerning point B . If a point is between point A and B in the space then the information about that point will reside in the MLG between the information about A and B. Also, Different point configurations will yield different MLG data structures. This can be seen in the situation that if a point C were to move past A and position itself lower and to the left of A, then the information about point C would have to "move" to a cell which was lower and to the left of information about point A in order to preserve its relative relationship to A and in order to conform to the inequalities (1.0). Thus sorting of MLG information is necessary.

Also note that the points of the space need not be aligned in such a regular pattern as the lattice example described above. If the points of the space were situated as pictured in figure 2.1 there still exists an MLG structure which would conform to the points of the space. One possible MLG configuration is shown in figure 2.2.

This MLG representation is not unique. Figure 2.3 is one more possible MLG structure which represents the same point space. Another look at the definition of the MLG will explain at least one situation which could create two possible MLG data structures for the same point configuration.

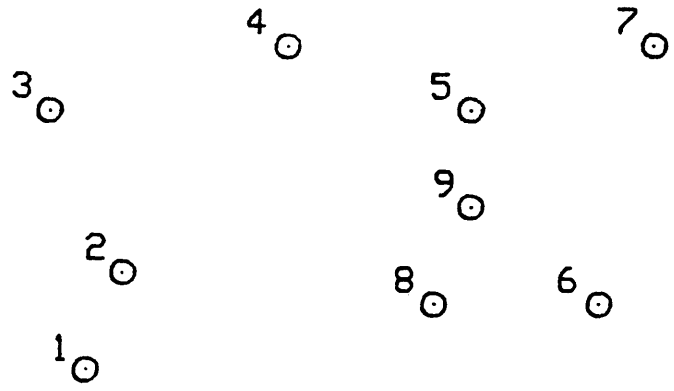


FIG. 2.1: Irregular point configuration

3	4	7
2	9	5
1	8	6

FIG. 2.2: First possible MLG representation of FIG. 2.1

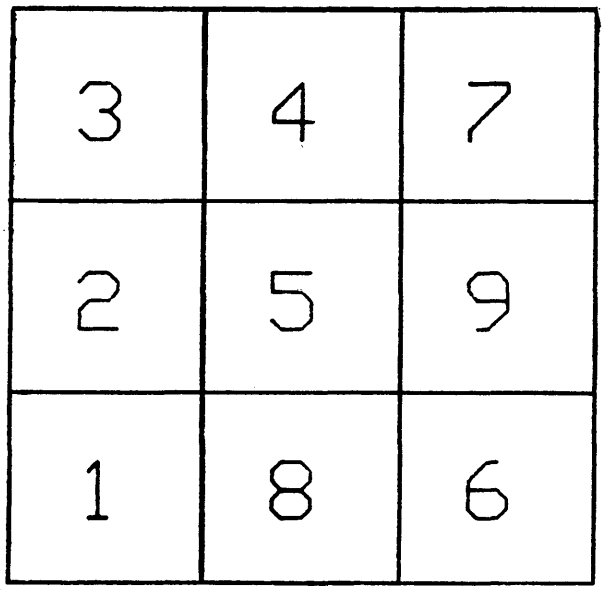


FIG. 2.3: Second possible MLG representation of FIG. 2.1

In the case of a two dimensional model let:

$$Y(i,j+1) = Y(i+1,j+1) \qquad (1)$$

$$Y(i,j-1) = Y(i+1,j-1) \qquad (2)$$

$$\text{and} \quad X(i,j) = X(i+1,j) \qquad (3)$$

for a fixed j such that $2 \leq j \leq NY-1$ and

for a fixed i such that $1 \leq i \leq NX-1$.

Note that the forgoing assumptions do not violate inequalities (1.0), so that we are actually looking at a

small section of an existing MLG structure. Now if we were to exchange information in the following manner:

X(i,j) with X(i+1,j) and Y(i,j) with Y(i+1,j)

then since (1)-(2) are given, monotonicity as given in inequalities (1.0) is not violated by the interchange Y(i,j) with Y(i+1,j). And since (3) is given, the interchange X(i,j) with X(i+1,j) does not violate monotonicity either. Thus we have two MLG structures representing one point space configuration. This situation does not pose any problems in distinguishing which structure to use. Actually, the presence of more than one possible data structure gives us the flexibility of optimizing MLG's to the particular data set being used. It is quite possible that one MLG configuration would yield better results (with respect to maximum index offsets) in vectorization and partitioning than would another MLG configuration of the same data.

When the number of points in the space becomes larger, so does the number of possible MLG configurations. The proof given above only involved a small section of an MLG. The existence of this situation in other sections of the MLG is quite probable, thus increasing the number of possible MLG data structures for the global data set. As a

result, optimization of MLG structures is a topic which is still under research and which will most likely advance the use of the MLG quite drastically.

S.G. Lambrakos and J.P. Boris have conducted work in the area of optimization of MLG structures (Lambrakos & Boris,85). In their work they have established and defined certain properties of the MLG which are used in determining proper MLG representation of data.

One such definition is of a Nearest Neighbor Template (NNT). The NNT is defined to be the section or partition of the MLG surrounding a particular target cell, "... a finite set of small index offsets in the MLG which correspond to the near neighbors in space." (Lambrakos & Boris,85). In their work testing was conducted concerning the size of the NNT and its relationship to the operational cost of the sorting and calculation routines of the MLG algorithm. It was found that search lengths are directly related to the size of the NNT. A large NNT will produce larger search lengths, while a smaller NNT will produce smaller search lengths.

2.4 AN ORDER N MLG SORTING ALGORITHM

Point movement in the grid necessitates sorting of the MLG in order to retain MLO. J.P. Boris has developed an

algorithm of order N for sorting the MLG into MLO (Boris,85).

If the motion of the points in the space is large (i.e. large numbers of points passing each other within the timestep) then the possibility of violating monotonicity (section 2.3 inequalities (1.0)) are quite high; as is the number of points that actually interchange relative positions. In the same instance, if point movement in the grid is small, the possibility of violations, along with the actual number of violations, decrease. We shall now follow this general observation in analyzing the algorithm offered by Boris.

Boris first offers a vector sort routine which scales as N log (N) (Boris,85). In the algorithm, all N object locations must be sorted into increasing Z order. Next, the first NY X NX object locations are to be sorted into increasing Y order while keeping the k index equal to 1.

Now within the first NY X NX objects which are sorted into increasing Y order, sort the first NX into increasing X order while keeping the j index equal to 1 and indexing these NX object locations such that i=1,...,NX. Once the first NX of the first NY X NX object locations are sorted then continue with the next NX objects, until all NY X NX objects are sorted accordingly.

The next step is to increment the k index to 2 and conduct the sorting in a similar manner on the next NY X NX object locations. This step is repeated until all NZ "planes" (each of size NX X NY) are sorted.

Two problems accompanying this algorithm are, its computational cost and the fact that if point motion is large in a certain section of the point domain then, certain cell information may have to cross a large section of the MLG in order to find its correct position (Boris,85) (fig. 2.4).

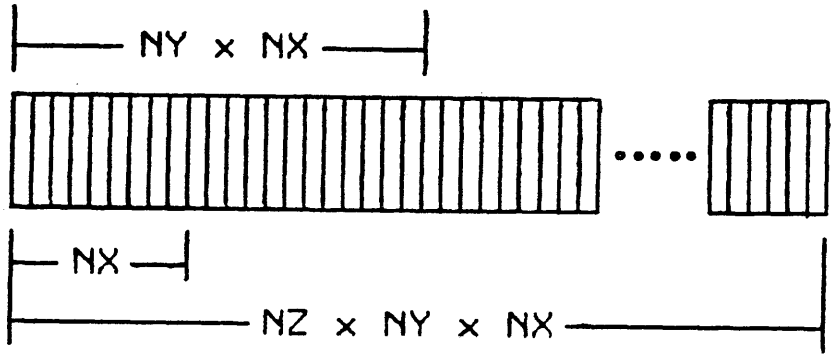


FIG. 2.4: Order N log N sort algorithm

As an alternative Boris offers another algorithm which scales as N. In the algorithm the concept of dimensionality is kept in the sort. In other words, if the simulation model is two dimensional, then sorting will be done on a two dimensional array (the MLG resembles a two

dimensional array for 2-D models). Each row in the MLG is defined to be an i directional vector, while each column is defined to be a j directional vector (fig 2.5).

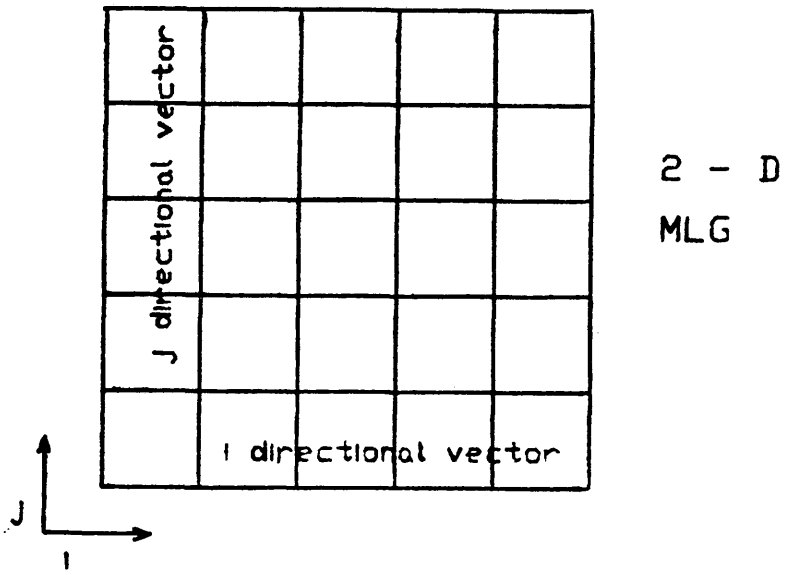


FIG. 2.5: i and j directional vectors used in sorting

The first step is to sort all i directional vectors into increasing X order. This sort does not rely on any relationships between different vectors (i.e. X(i,j,k) as related to $X(i,j_m,k_n)$ where $j \neq j_m$ and $k \neq k_n$). In other words while fixing both the j and k indexes sort all object locations ($1 \leq i \leq NX$) in increasing X order without regard to object locations in vectors around the current i directional vector. The same technique is used for all j and k directional vectors.

Boris also gives us an efficient algorithm for determining if consecutive array elements are out of order. The algorithm follows.

In order to determine if two consecutive elements are in increasing order, first calculate the sign of the difference of the the two object locations and assign it to the number s = 0.5 . Next calculate the weights w and (1-w) as w = s + 0.5 and (1-w) = s - 0.5 respectively. Now w = 1 and (1-w) = 0 when the two object locations are in increasing order while w = 0 and (1-w) = 1 when they are not.

Once the order of object locations has been determined then the following four operation statements will swap data if consecutive object locations in an i directional vector are out of order and will leave the information intact if they are not:

$$T(i,j,k) = w \times X(i,j,k)$$

$$U(i,j,k) = (1-w) \times X(i,j,k)$$

$$X(i,j,k) = T(i,j,k) + (1-w) \times X(i+1,j,k)$$

$$X(i+1,j,k) = w \times X(i+1,j,k) + U(i,j,k)$$

where T(i,j,k) and U(i,j,k) are temporary storage variables. This swap procedure can be conducted on all adjacent elements in the MLG for all i,j and k directional

vectors.

Underlying this sort algorithm is the technique of sweeping through the grid for each sort iteration. When, for instance, all i directional vectors are being sorted only adjacent elements are swapped. The technique of reiterating through the vector until all positions are in order is not done. One sweep is made through all i directional vectors exchanging only adjacent elements that are out of order. Next all j directional vectors are swept through and finally all k directional vectors. A count is kept of the number of actual element swaps. If this count is non-zero the entire sweep through all directions is repeated.

Further optimization of this sort routine can be achieved in determining local areas within the MLG grid which do not need sorting thereby avoiding many unneeded calculations. Bookkeeping involved with this idea, however, becomes quite sophisticated and may not return sufficient benefits to warrant the coding of the algorithms.

In looking at the relative motion of the points in the space with respect to the resultant amount of sorting required as a consequence of that motion, one observation made by Boris about the magnitude of the number of sort

iterations needed to establish MLO, "Almost all of the grid restructuring occasioned by particles passing each other occurs in the first two or three vectorized iterations." (Boris,85), is supported by test results conducted in this study. The number of sort iterations is generally low. The term generally is used because cases have arisen in which a noticeable increase in the number of sort iterations was recorded for several timesteps within particular test runs. This discrepancy can be explained when looking at the particular flow equations being used to perturb the points.

Sort iteration counts remain low, (5 - 10), in sections of consecutive timesteps until point motion triggers either minor global, or major localized point passing, which would in turn necessitate an increase in sort operation counts. Such situations will be presented in the result chapter of this study.

Originally the MLG was developed to monitor points moving about in a spatial domain. Data about the points was stored in the MLG and dynamically moved about in conjunction with the motion of the corresponding points.

This study will extend the model of the MLG to handle two dimensional figures (triangles). This is accomplished by characterizing the triangles with one particular point,

while storing information about the triangle as a whole in the MLG.

CHAPTER 3 - REPRESENTATION OF THE FREE LAGRANGIAN GRID USING THE MLG

3.1 INTRODUCTION

The MLG model introduced by J.P. Boris was developed for points moving about in space. This chapter discusses the extension of the point model to a 2-D figure model.

We will discuss storing point data in the MLG versus storing triangle data, as well the minimum amount of data which can be stored in the MLG.

3.2 POINT DATA VS. TRIANGLE DATA

The Free Lagrangian numerical method discussed earlier requires information about the triangles which surround points in the real space as well as information about adjacent triangles of the space.

If we were to store point data in the MLG, we would lose information about the vertices (points) which determine triangles of the space. For instance, if we were required to find all vertices of the triangle T_m, we would have to resort back to searching a list containing triangle data in order to relate the triangle T_m to its vertices, because the MLG does not contain any information about triangle-vertex relationships. This is exactly the problem we are trying to overcome.

On the other hand, storing information about triangles in the MLG allows us to relate point and triangle data directly. For example, if we need to determine the coordinates of all verticies of a triangle, we would simply use the triangle I.D. stored in the MLG as the index reference to the array containing triangle information. Once we have found the correct position in the triangle array we can use the vertex I.D.'s as index references to the point array.

3.3 MINIMUM MLG STORAGE REQUIREMENTS

The next step in the transformation of the MLG point model is to determine the minimum amount of data which needs to be stored. Since triangle data is being stored, the triangle I.D., in the least, must be stored. As will be seen in the next section, the coordinates of the "triangle processing attribute" must also be stored in order for us to sort the MLG. All other references to data can be made using the triangle I.D. as the array index.

Therefore, the minimum amount of data needed in the MLG is comprised of the triangle identification number, and the coordinates of the "triangle processing attribute". Data such as point coordinates and triangle vertex I.D.'s can be stored in arrays outside the MLG, and

referenced using the MLG triangle I.D. as the array index.

The MLG takes the form of a four dimensional array. The first three dimensions are the i, j and k indices of the MLG cells, while dimension 4 contains the triangle data. Following are the data positions in the fourth dimension of the MLG cell at (i,j,k):

Dimension 4: Triangle attribute storage

- Position 1: x coordinate of triangle processing attribute
- Position 2: y coordinate of triangle processing attribute
- Position 3: z coordinate of triangle processing attribute
- Position 4: Triangle I.D. of triangle at (i,j,k)

In the initial development of this study the author stored triangle vertex I.D.'s in the MLG in addition to all of the above outlined data. It was thought that the triangle vertices were needed in the MLG when referencing the point array for point coordinates, but was later found not to be the case.

3.4 EXTENSION OF THE POINT MLG MODEL TO REPRESENTATION OF TRIANGLE DATA

We must now determine how to apply two dimensional figures (triangles) to a point MLG model. This is easily

accomplished by characterizing each triangle using one particular point of the triangle. For instance, we could characterize all triangles of the space by their centroids, or by the vertex which has the smallest x coordination.

We will call this characterization point the "triangle processing attribute". All sorting which was discussed in the preceding chapter will now be performed on the triangle processing attribute.

There are many possible triangle processing attributes which could be used. Only four were chosen. These four attributes will be discussed in detail in later chapters. For now the four triangle processing attributes are:

1. Triangle centroid
2. Triangle vertex with the smallest x coordinate
3. Average vertex coordinates
4. Midpoint of the longest triangle side bisector

CHAPTER 4 - MLG TEST PARAMETERS

4.1 INTRODUCTION

In order to test the MLG and its adaptation to a Free Lagrangian grid certain tests were developed that indicate the usefulness of the MLG in searching for adjacent and surrounding triangles. The following test procedures and parameters were developed in order to test key issues such as search times and lengths and partitionability of the data stored in the MLG.

The process of testing algorithms involves the establishment of execution parameters in accordance with each issue being examined. For instance, when examining the partitionability of data stored in the MLG, different flow equations (fundamental, non-conservative equations) are used to "move" data about in the MLG, therefore testing the sort and search algorithms in relation to locality of spatially related information in the MLG. In addition, both regular and irregular initial grid configurations are used in testing the MLG performance.

The reason for the varying initial grid configuration is to correlate data accessibility between different initial grid shapes, thus determining the difference between data partitioning using regular and irregular

initial grid constructs.

Different partitioning schemes have been developed in order to establish which triangle attributes make the best triangle processing attributes. The calculation of operation counts and the determination of maximum index offsets in searching algorithms will establish which attributes are optimal.

4.2 INITIAL GRID CONFIGURATIONS

Appendix A gives plots of all initial grid configurations used in the testing of the MLG. These initial grids are broken down into two main categories, regular and irregular grids.

Regular initial grid construction is performed in a predetermined manner using a special technique for placing points and triangles into a regular pattern. All points are initially placed at lattice positions in a two dimensional space. From here triangles are formed using these points as verticies. The pattern formed by these triangles is very regular and constant throughout the grid. Appendices A.1 and A.2 show examples of both a "symmetric" and a "slightly deviated" regular initial grids.

Irregular initial grid construction carries the

regular grid construction phase one step further by adding a random number of points to randomly determined pre-existing triangles in the space. The method of inserting more points is the Triangle Trisection method (section 1.3) and the number of triangles trisected is taken to be a predetermined percentage of space triangles present from the regular grid construction phase. In other words, if it is desired to tesselate 90% of the regular grid into an irregular pattern and if 100 triangles were constructed in the regular grid construction phase, then there will be 190 triangles present in the initial irregular grid after all construction is complete. Appendices A.3 and A.4 show examples of "symmetric" and "slightly deviated" irregular initial grids.

Initialization of the grid into either a regular or irregular configuration is done in order to compare sorting and searching results obtained from the use of both the regular and irregular schemes. Any differences in the results will suggest partiality by the MLG to initial grid configuration, thereby reducing the flexibility of the MLG.

"Symmetric" and "slightly deviated" initial grid options are used as an extra deviation to the initial grid construction. The amount of deviation of point placement

from the regular lattice grid positions is variable and can be changed from one execution to another. This added initial deviation produces different grids through the duration of executions and thusly eliminates the dependence between flows and processing attributes and the initial grid structure. For example, if consecutive executions use a "slightly deviated" irregular initial grid construction the correlation between the execution runs is not dependent on the use of the same initial grid.

4.3 PARTICLE MOVEMENT EQUATIONS

Movement of points and triangles in the point space causes the movement of data in the MLG. As was seen earlier, sorting of the MLG will preserve the data locality within the data structure, thus making data access quicker and more efficient.

In order to test the MLG and its dynamic representation of the point space, different particle movement equations are used to observe data movement in the MLG. Three types of particle movement were developed and tested.

The vector equations modeling these three flows are as follows:

1. Modified Uniform Strain Flow

$$P_{x(t+1)} = P_{x(0)}\ e^{(-bt)} + P_{x(t)}$$
$$P_{y(t+1)} = P_{y(0)}\ e^{(bt)} + P_{y(t)}$$

2. Modified Parabolic Flow in X

$$P_{x(t+1)} = P_{x(0)} + t(y_o - P_{y(t)})^2$$
$$P_{y(t+1)} = P_{y(0)}$$

where y_o is a constant.

3. Random Flow

$$P_{x(t+1)} = P_{x(t)} + r_1$$
$$P_{y(t+1)} = P_{y(t)} + r_2$$

where r_1 and r_2 are randomly generated, uniformly distributed, numbers in a predetermined interval.

$P_{x(t)}$ and $P_{y(t)}$ are the x and y components of the point coordinates at any time t. Appendix B shows all three particle movement results, B.1-B.4 picture the Uniform Strain flow, B.5-B.8 the Parabolic flow and B.9-B.12 the Random flow.

As can be seen from the plots of the point space and the triangular interconnections, the Parabolic and Random flows distort the grid sufficiently enough to warrant further investigation into the adaptability of the MLG to

these different flows. The plot of the Uniform Strain flow, however, shows that point motion governed by these vector equations does not produce enough distortion in the triangular connections to disrupt the MLG data to any large extent.

In fact, numerical results show that very little physical change occurs in the MLG as a result of Uniform Strain point motion in the space.

4.4 DATA PARTITIONING SCHEMES

Several triangle attributes have been tested in order to classify one or two of them as being the best processing attribute. First of all an explanation of the triangle processing attribute.

As stated above (section 3.3), positions 1, 2 and 3 of an MLG cell contain the x, y and z components of the triangle processing attribute, respectively. The triangle processing attribute is defined to be one certain "point" of the triangle on which all sorting will be done. Therefore if the triangle processing attribute is defined to be the triangle centroid, then the coordinates of the centroid will be stored in positions 1-3 of the MLG triangle attributes and the sorting of the MLG cells will use these three coordinates as the sort criterion. Thus

when sorting all i directional vectors the following comparison:

$$X(i,j,k) \leq X(i+1,j,k)$$

will take the form:

$$MLG(i,j,k,1) \leq MLG(i+1,j,k,1)$$

where MLG(i,j,k,n) is the cell at i,j,k in the MLG data structure. Likewise, sorting j and k directional vectors would involve the use of elements in positions 2 and 3 respectively.

Since there are many attributes of the triangle which could be used as processing attributes, only a few were chosen. These few were chosen so as to give different insight to the relative shape of the triangle they represent. For instance, if triangles in a particular point space are forced to become elongated (long and narrow) then the processing attribute defined to be the midpoint of the longest bisector would also give information about relative dimensions of the sides of the triangle. In other words, the point being used for the processing attribute will be somewhat skewed from the center towards the long end of the triangle, thereby representing the majority of the triangle as being farther

away from the center than would a processing attribute defined to be the triangle centroid. The triangle centroid in this instance would lead us to believe that the triangle is centralized more around its center point than is really the case.

Four triangle attributes were chosen to be used as processing attributes and they are:

1. Triangle centroid (assuming mass is constant)
2. Vertex with the smallest x coordinate
3. Midpoint of the longest side bisector
4. Average vertex coordinate value

Attribute 1 (triangle centroid) was chosen for the fact that the triangle could be generalized close to its center (fig. 4.1).

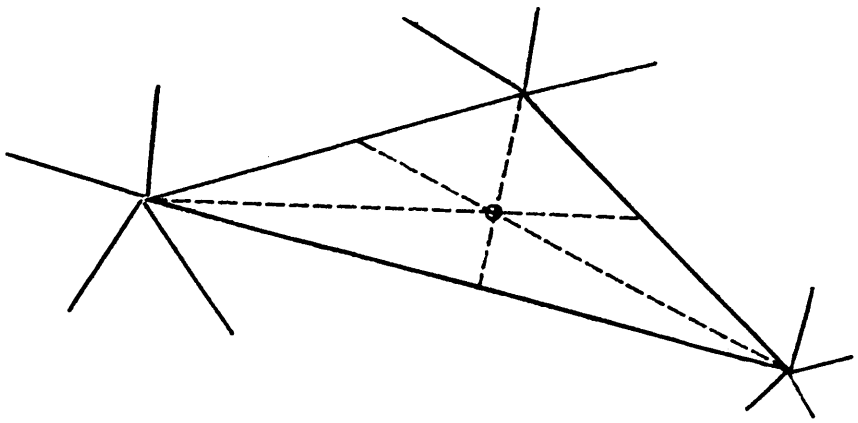


FIG. 4.1: Triangle centroid

If the flow is such that large differences in triangle areas are not present in the space then the triangle centroid would sufficiently represent the state of the triangle. As a matter of fact, this particular attribute works quite well in the processing of a wide variety of shapes and sizes of triangles. This choice of points has proven to be one of the best of the four, in most combinations of flows.

If flow is random and triangle distortion is difficult to predict then attribute 2 (vertex) should represent the space sufficiently (fig. 4.2).

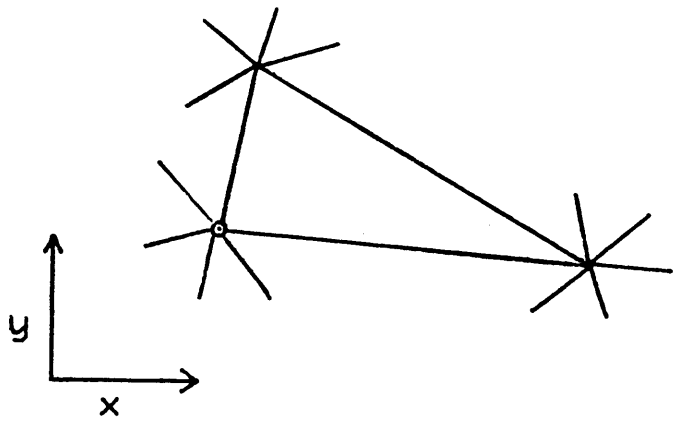


FIG. 4.2: Vertex containing the smallest x coordinate

However, as will be seen, this attribute causes a loss in the information about the general shape of the

triangle. Information about only one of the vertices does not give us a general description of the triangle. The triangle could be very elongated and all we will know is the position of the vertex with the smallest x coordinate. This attribute has proven to be the poorest of the four choices.

If point motion causes elongation of the space triangles then attribute 3 (longest bisector midpoint) will reflect the state of the triangle sufficiently (fig. 4.3), and testing shows that this is the case. However, if triangle shapes evolve to the point that triangle sides on the average are close in magnitude then extra work is being done calculating the midpoint, when the triangle centroid could be used with less computation.

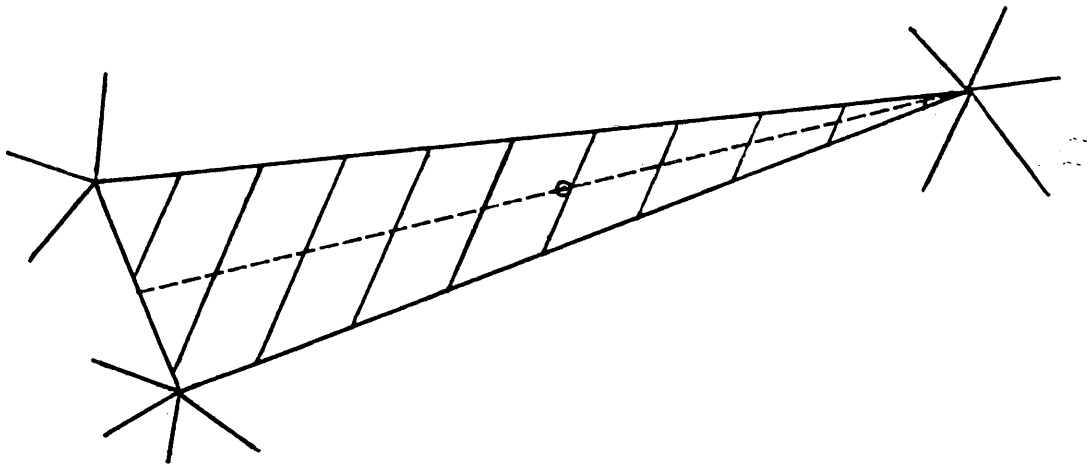


FIG 4.3: Midpoint of the longest side bisector

Finally, attribute 4 (average coordinate) was chosen to average out the vertex coordinates and give a mean representation of the triangle (fig. 4.4).

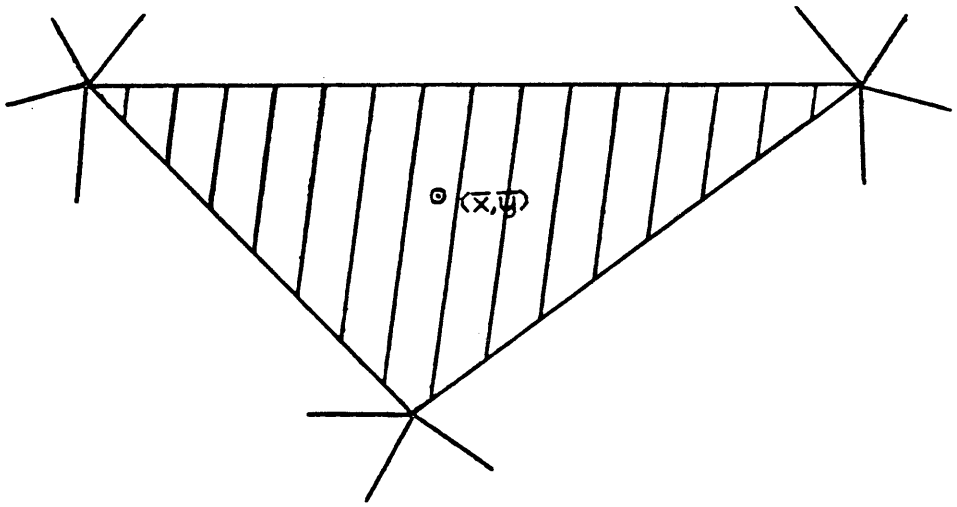


FIG. 4.4: Average vertex coordinates

This approach dampens differences in side lengths giving us an average positional location for the coordinates of the triangle. This type of attribute can be applied to triangles of varying shapes, giving similar MLG representation to various triangles of different shapes. This choice of attributes has shown to be quite good, and as will be shown later is one of two attributes that give similar good results.

4.5 MLG SEARCHING ALGORITHMS

Once the flow and attribute parameters have been set up the issue of determining how well the MLG performs with

respect to searching must be addressed. What search algorithms will test the MLG model as if it were being used in real time applications?

The Free Lagrangian model requires point and triangle mesh update procedures to use information about neighboring points and triangles. The vertex cell defined in chapter 1 requires information concerning all triangles around a point, while the triangle mesh bookkeeping algorithms require information concerning adjacent triangles.

This then will be the basis of developing searching algorithms which will aid in data access processes. In this chapter we will only develop the evaluation criterion for searching out adjacent and surrounding triangles. The next chapter will develop the actual algorithms used.

4.5.1 ADJACENT TRIANGLE SEARCH

First of all, we will develop the criterion for determining the accessibility of information about adjacent triangles.

The MLG used in this study is related to a two dimensional array. Let us define an "index offset of m" to be all MLG cells MLG(i,j,k,n) such that:

$$i_o - m \leq i \leq i_o + m$$
$$j_o - m \leq j \leq j_o + m$$
$$k_o - m \leq k \leq k_o + m$$

where i_o, j_o and k_o are the MLG indices of the central cell.

A typical index offset of 1 is pictured in figure 4.5. In the case that i_o, j_o or k_o is equal to 1 then the index offset of 1 would only include the central cell as the outermost cell in the direction of the MLG boundary (figure 4.6).

In searching for adjacent triangles a measurement will be made as to the magnitude of the index offset in order to find all adjacent triangles to the triangle contained in the central cell. Since the MLG offers us data locality, index offsets measure the relative closeness of spatial data within the MLG. We will then use this measurement in evaluating MLG performance in adjacent triangle searches

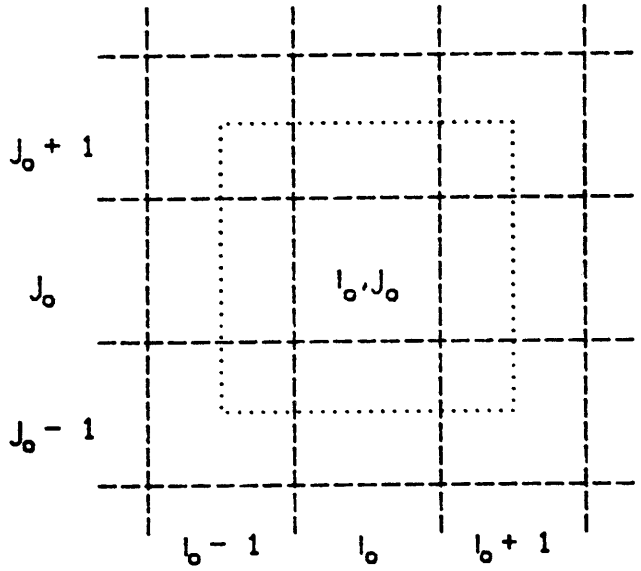


FIG. 4.5: Typical symmetric index offset of 1

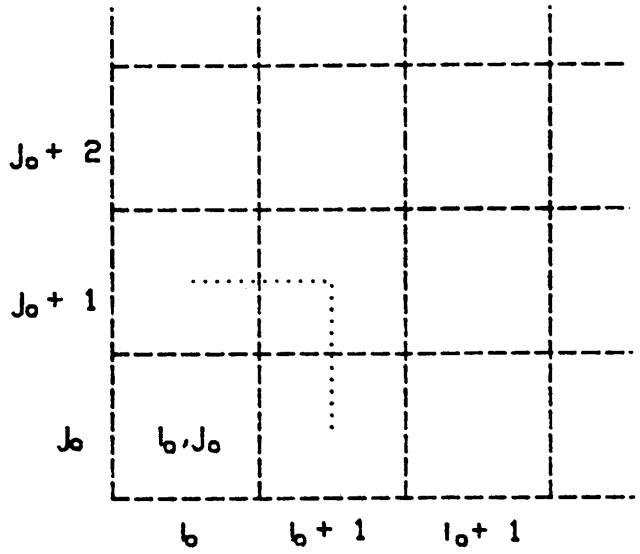


FIG. 4.6: Index offset of 1 at MLG boundaries

using different processing attributes as well as different flows. Once all adjacent triangles have been found for each MLG cell (triangle), an average index offset will be determined. This average, along with analysis of individual maximum and minimum cell index offsets and the variance of the offsets around their mean will be the basis of determining which processing attribute works best with certain point flow equations.

4.5.2 SURROUNDING TRIANGLE SEARCH

Now, we will define the procedures for the evaluation of algorithms which search for all triangles around a given point. These algorithms are more complex than their adjacent triangle counterparts, however segments of the adjacent triangle search carry over to the surrounding triangle searches, making this search algorithm somewhat related to the more fundamental adjacent triangle algorithm.

As for searching for all triangles around a given point, again a measurement of index offsets will be taken for each point. These index offsets will then be averaged over all points of the space. This average along with analysis of maximum offsets in the i, j and k directions and the variance of the offsets around their respective

means will be the basis of evaluation in this search.

The measuring of offsets in searching for triangles around a point differs from the adjacent triangle "index offset of m" definition given above. First of all, points are processed by stepping through the MLG, triangle by triangle, and processing all unprocessed points which appear as part of the present triangle. While stepping through each triangle, when points that have not been processed are encountered the triangle which contains them is recorded as being the "entry triangle" for that particular point. For instance, if the unprocessed point P_n is present in triangle T_m of the MLG then triangle T_m will be defined to be the "entry triangle" for point P_n. Since every triangle in the space has a unique MLG reference index, the i, j and k indices of the entry triangle then become the target cell indices. In other words, if triangle T_m has an MLG index reference of $MLG(i_o,j_o,k_o,n)$ then the target cell indices become i_o, j_o and k_o.

From this point a record is kept of the maximum index offset needed in all three of the i, j and k directions in order to find all triangles which share the point in question. This in turn is the measurement which we will use to evaluate the performance of the MLG in the case of

surrounding triangle searches. Figure 4.7 pictures a typical i index offset of 2 and j index offset of 1 for a point P_n contained in triangle T_m at the MLG index (i_o, j_o).

Each point will have associated with it a maximum i, j and k index offset, which suggests that index offsets may

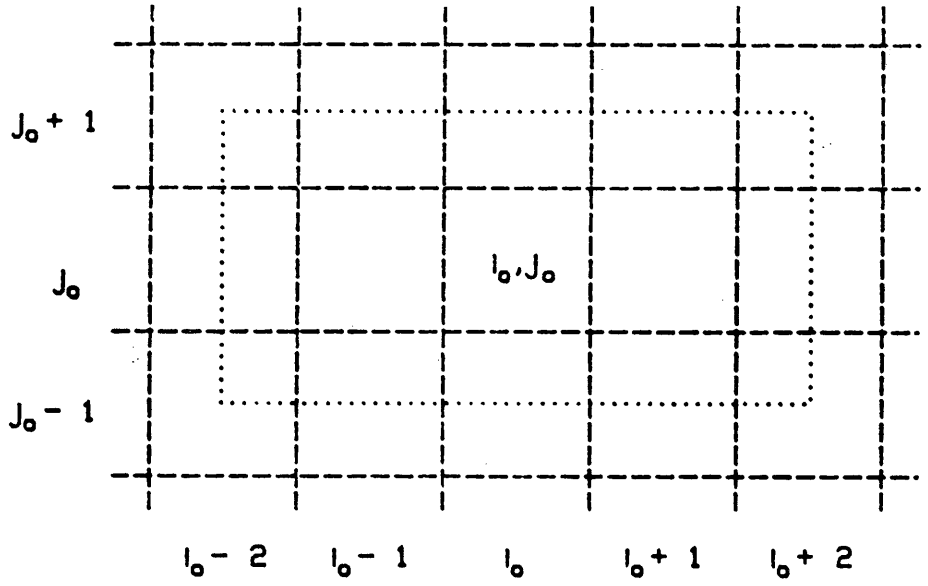


FIG. 4.7: i index offset of 2, j index offset of 1

not be symmetric in the i, j and k directions as is the case in searching for adjacent triangles. This difference in defining performance measurements around index offsets arises from the algorithm being used to perform the actual searching. This algorithm is to be developed in the next chapter.

4.6 SUBROUTINE CPU TIMINGS

One final evaluation tool has been incorporated in testing the performance of the MLG. Benchmark timings (CPU) have been established within all subroutines of the MLG test program. The actual CPU requirements of the sorting and searching subroutines will be used in comparisons between flows and between processing attributes in order to help in the narrowing down or simplification of establishing the optimal processing attributes with respect to the parameters used in this study. As will be seen in later chapters, time consumption figures for similar flows vary noticeably between uses of different processing attributes, suggesting that certain triangle attributes are truly superior to others.

Different combinations of flows, initial grid configurations, and searching techniques have been used in an extensive series of test runs of the MLG algorithm and the subroutines used to aid in testing the MLG performance. Future chapters will discuss the actual combinations of parameters used and will give results of the test runs, giving CPU timing comparisons, operation count comparisons and index offset comparisons.

CHAPTER 5 - ALGORITHMS

5.1 INTRODUCTION

In this chapter the algorithms used in testing the performance of the MLG will be developed. An overall view of all relevant algorithms used in the driver program MLG.FOR will be given. These algorithms consist of three general types.

Type one is a family of initialization procedures. Model parameters such as model size, time duration, time step size, plot and numerical data output confirmation, and certain flow constants are established by non-interactive initialization. Interactive input accounts for the input of flow type, type of searching to conduct, initial grid configuration and the particular processing attribute to be used in sorting.

Type two is a family of model execution algorithms. These algorithms are comprised of routines which, for each timestep, adjust the processing attribute of each MLG cell, sort the MLG, apply motion to the point space and search the space for either adjacent or surrounding triangles. These algorithms account for most of the execution run time of the driver program as a whole.

Finally, type three is the family of output

algorithms. These algorithms handle output of plot data, statistical data accumulated from searching algorithms and CPU timings obtained from the separate routines.

5.2 INITIALIZATION ALGORITHMS

The initialization family of algorithms can be further broken down into two sub groups, interactive and non-interactive.

Four execution parameters are read in from the user in the interactive section of program initialization. These four parameters are:

1. Initial grid configuration
2. Point flow equation
3. Triangle processing attribute
4. Type of searching to conduct on the data base

Non-interactive initialization routines consist of the initialization of equation parameters, model size, point and triangle arrays, and the initialization of output files.

5.2.1 INITIAL GRID CONSTRUCTION

In the initialization of the triangular grid, triangles are set up in a regular configuration (section 4.2 and appendices A.1 & A.2) . If the interactive

parameter for initial grid construction specifies the use of an irregular initial grid, the regular triangle space is then tesselated into an irregular configuration.

GRIDFRAC, is a non-interactive parameter which determines the maximum fractional part of the regular grid which will be irregularly tesselated in this step. The maximum of, GRIDFRAC X NUMTRI (where NUMTRI is the number of triangles in the space) and a randomly generated number in the interval (1 , NUMTRI), is taken to be the fraction of the initial regular grid which will receive additional tesselation.

This incorporation of randomness is used in order to produce random tesselation patterns and thus give us the facility to compare similar runs (with respect to processing attributes, point flows and searching schemes) on different initial grids. This can be useful in examining differences in a processing attribute's mean performance on dissimilar grids.

Additionally, the number of triangles will not change during execution of MLG.FOR . The reason for this is that the algorithms needed for the bookkeeping of insertions or deletions of triangles and points are too complex at this stage to incorporate. Therefore after initial grid construction, triangle and point interconnections are not

broken and the number of triangles as well as the number of points are kept constant. In keeping with this convention, point flow equations have been formulated such that point motion does not become extremely radical, therefore keeping the grid uncorupted for as many timesteps as possible.

5.3 MODEL EXECUTION ALGORITHMS

In this section we will discuss algorithms which are used in the actual model execution stage of the program. These algorithms consist of the adjustment of triangle processing attributes, sorting of the MLG into Monotonic Logical Order (MLO), searching the MLG for adjacent or surrounding triangles and the application of motion to the space points.

5.3.1 PROCESSING ATTRIBUTE ADJUSTMENTS

These particular algorithms, whose subroutine names consist of ADJCENT, ADJTRI, ADJAVG, and ADJLONG, are used to adjust any coordinates of the processing attributes which may have changed from the previous timestep due to point motion.

ADJCENT recalculates each triangle's new centroid using the updated point array.

Adjustment of the vertex with the smallest x

coordinate is performed by the subroutine ADJTRI. All vertices of the triangle are compared in order to determine the vertex which, after point motion, has the smallest x coordinate.

ADJAVG is a subroutine used to adjust the average values of all vertices in a particular triangle. The coordinates of each triangle are gathered from the updated PT array and averaged in each dimension (x, y and z).

Similarly, the ADJLONG subroutine recalculates each side bisector of each triangle and determines the midpoint of the longest one.

5.3.2 MLG SORTING ALGORITHM

Basic concepts of the sort algorithm called MLGSORT were presented in section 2.4 of chapter 2. Appendix C.1 shows pseudo-code for MLGSORT. Actual FORTRAN code for MLGSORT is given appendix D.1

This algorithm incorporates a cut down version (with respect to iteration sweeps) of a bubble sort. Vectorized sweeps, as a result of the use of a red-black algorithm are made through all dimensions of the space, thus giving us the benefit of vectorized calculations and possible multiple processing since dimensional sorts are independent of vectors of the same dimension (section 2.4,

chapter 2).

5.3.3 MLG SEARCH ALGORITHMS

Two separate searching algorithms are to be presented. The first algorithm (adjacent triangle search) will be the building block for the second (surrounding triangle search). First of all let us discuss certain conventions and arrays which are utilized in both algorithms.

A matrix of dimension NUMPTS X 3 called PTSTAT, where NUMPTS is the number of points in the space, is utilized in determining the position of the triangle with respect to the boundaries of the triangular grid. This array contains three attributes pertaining to each point of the space. The attributes are stored in positions 1-3 of the second dimension while dimension 1 indexes the point I.D..

The particular attributes of the point relate to the position which the point holds relative to the boundary of points in the space. An entry of 1 in the first position of dimension 2 indicates that the point is interior to the point boundaries, an entry of 1 in position 2 indicates the point is considered to be a border point and an entry of 1 in position 3 indicates the point is a corner point.

An interior point is defined to be any point such that each triangle side which stems from it belongs to exactly

two triangles of the space (fig. 5.1).

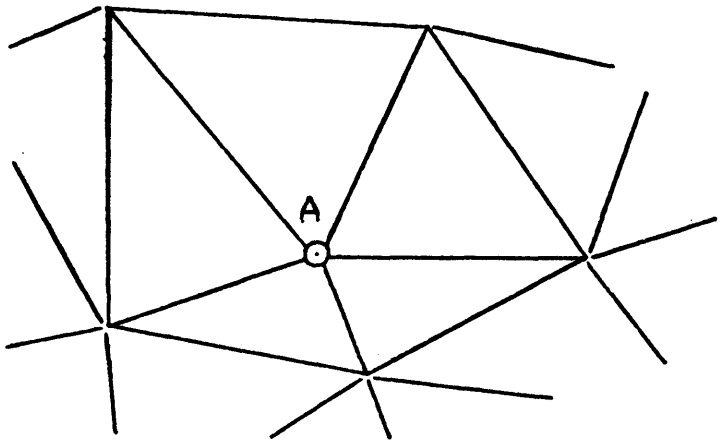


FIG.5.1: Interior point A

A corner point is defined to any point which has stemming from it only two triangle sides (fig. 5.2).

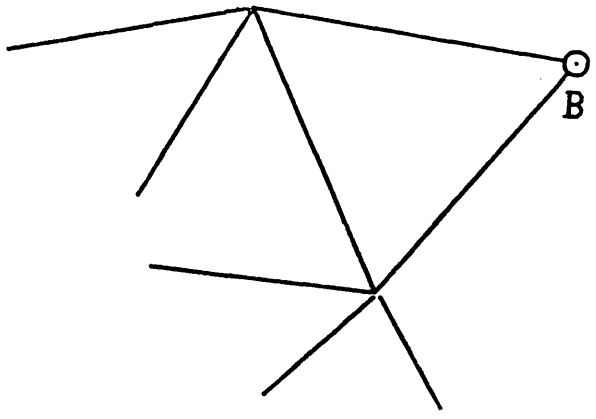


FIG. 5.2: Corner point B

All other points of the space are defined to be border points (fig 5.3).

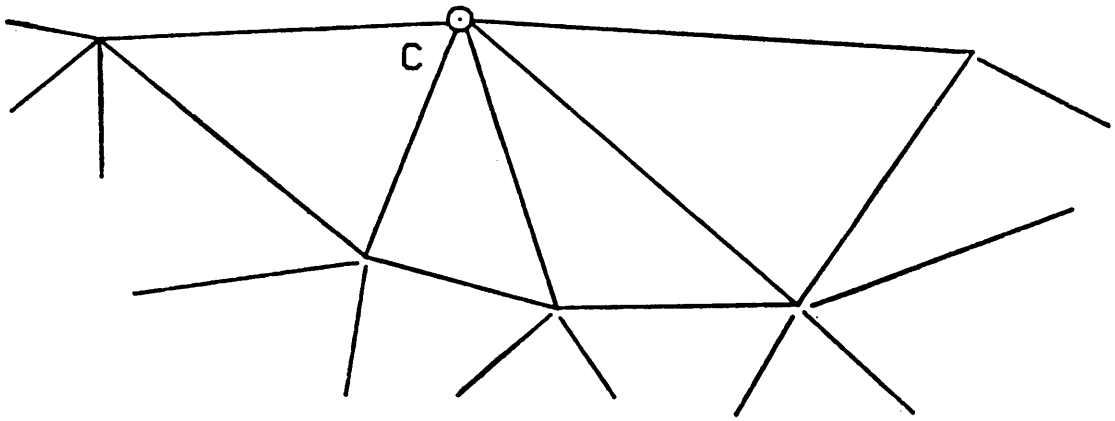


FIG. 5.3: Border point C

Also:

$$\sum_{i=1}^{3} \text{PTSTAT}(n,i) = 1 \ , \quad \text{for all } n \quad 1 \leq n \leq \text{NUMPTS}$$

The above summation equation is evident by the fact that it is impossible for a point to be defined in two ways. For example, if a point is defined to be an interior point then it is impossible for that same point to be defined as a boundary point.

The information obtained by summing the different positions of the PTSTAT matrix over all points (vertices)

of a particular triangle will tell us what type of a triangle we are dealing with.

There are also three definitions of triangles which are similar to the definitions of points. They are interior, border, and corner triangles. Following are the definitions for the classification of triangle types:

Let

$$\text{INTERIOR\_SUM} = \sum_{\text{(all vertices of Tm)}} \text{PTSTAT(i,1)}$$

$$\text{BORDER\_SUM} = \sum_{\text{(all vertices of Tm)}} \text{PTSTAT(i,2)}$$

$$\text{CORNER\_SUM} = \sum_{\text{(all vertices of Tm)}} \text{PTSTAT(i,3)}$$

then if

$$\text{INTERIOR\_SUM} = 3 \text{ and } \text{BORDER\_SUM} = 0 \text{ and } \text{CORNER\_SUM} = 0$$

then triangle T_m is defined to be an interior triangle. (1)

Also if

$$\text{INTERIOR\_SUM} = 2 \text{ and } \text{BORDER\_SUM} = 1 \text{ and } \text{CORNER\_SUM} = 0$$

then triangle T_m is defined to be an interior triangle. (2)

And if

INTERIOR_SUM = 1 and BORDER_SUM = 2 and CORNER_SUM = 0

then triangle T_m is defined to be a border triangle. (3)

And finally if

INTERIOR_SUM = 0 and BORDER_SUM = 2 and CORNER_SUM = 1

then triangle T_m is defined to be a corner triangle. (4)

5.3.3.1 ADJACENT TRIANGLE SEARCH

ADJSRCH is a subroutine which uses the data locality of the MLG in order to determine and calculate which triangles are adjacent to any given triangle. This subroutine, as well as the subroutine used to find surrounding triangles around points, utilize the "close" storage of data in the MLG with respect to neighboring triangles in the space. Also the concept of the "index offset" developed in section 4.5.1 will be utilized within.

The adjacent triangles to each of the triangles in the space are determined using a sequential access pattern looping over the MLG indices i, j and k. Since the MLG data structure physically resembles a three dimensional

matrix, the looping indices loop through the real space dimensions x, y, and z, with y being the inner most index, x the next level up and z the outer most index. The result of this order of looping produces update sweeps through all y directional vectors of the MLG.

Now for each triangle T_m being updated index offsets of consecutive magnitude (i.e. 1,2,3,...) are searched until all triangles which are adjacent to it are found. FINDADJ, a subroutine one level down from ADJSRCH, determines current index offset magnitudes of the search being conducted for triangle T_m, as well as the particular MLG index reference, i_o, j_o and k_o, of the triangle to be determined as either adjacent or not adjacent to triangle T_m. For each index offset, all triangles whose MLG index references fall within the bounds of the offset are checked to see if they are adjacent to triangle T_m. This is done by calling upon the subroutine ADJCONF which is yet another level below FINDADJ.

ADJCONF simply confirms or rejects the hypothesis of triangle T_m being adjacent to the triangle at the MLG index reference (i_o , j_o , k_o).

As for FINDADJ, this subroutine must also determine the number of adjacent triangles which it must look for while searching on each individual triangle, since no

record of adjacent triangles is being kept. This is where the PTSTAT matrix is utilized. We can use the previous definitions of interior, border and corner triangles to determine the number of adjacent triangles to search for. A corner triangle has exactly one adjacent triangle, while an interior triangle has exactly three adjacent triangles.

A border triangle, however, has the possibility of either two or three adjacent triangles. This non-unique determination causes considerable work in ADJSRCH and the subroutine to be discussed in the next section. The problem arises from the definition of the different types of triangles. A triangle which has interior, border and corner point counts of 1, 2 and 0 respectively, is defined to be a border triangle (definition (3) from above). However, it can also be classified as an interior triangle by the number of adjacent triangles around it. This triangle is in fact the only adjacent triangle to a corner triangle (fig. 5.4) and is called a "hidden interior triangle".

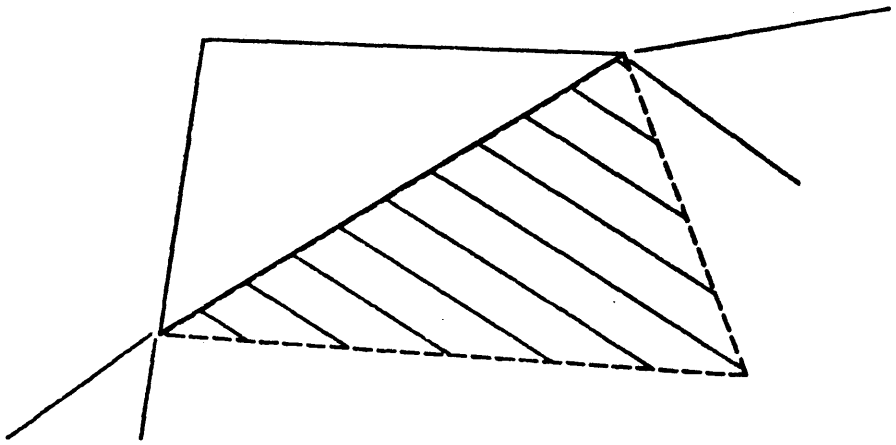


FIG. 5.4: Hidden interior triangle

Therefore additional work must be done to identify the triangles which are adjacent to corner triangles and then reprocess these triangles using the fact that there are really three adjacent triangles.

Appendices C.2 and C.3 give pseudo-code representations of ADJSRCH and FINDADJ, respectively. Actual FORTRAN code for these algorithms as well as ADJCONF are presented in appendices D.2 , D.3 and D.4 .

5.3.3.2 SURROUNDING TRIANGLE SEARCHES

The searches which involve finding all triangles around a given point and the subroutines which perform

this task are the subject of this discussion. The first subroutine to be discussed is called PNTSRCH, and is an extension of ADJSRCH, in that it uses the concepts that ADJSRCH uses and extends the searching to a more extensive level. This subroutine uses the subroutine FINDADJ which was described in section 5.3.3.1. The second of the two surrounding triangle search subroutines, which is called PNTSRCHF, does not use FINDADJ. Instead triangle I.D.'s of adjacent triangles are stored in the triangle data array and referenced whenever adjacent triangles are to be found.

In PNTSRCH, the MLG is searched sequentially as in the case of the adjacent triangle search, however our search does not directly involve triangles anymore. As we step through each triangle in the MLG we are looking at the points (vertices) of the triangle. In looking at one of the vertices, P_O, of the entry triangle (section 4.5.2, chapter 4), we need to find all triangles of the space which have this point as a vertex.

In the simplest case, if P_O is a corner point, it will only have one triangle surrounding it, namely the entry triangle. The number of surrounding triangles for interior or border points, however, is not nearly as easy to established. This problem arises from the fact that the

number of triangles which surround a point is variable.

By use of the subroutine FINDADJ, we can obtain all triangles which are adjacent to any given triangle.

Therefore, we begin the search by finding all triangles adjacent to the entry triangle of point P_o. We then must eliminate all triangles found to be adjacent to the entry triangle which do not contain P_o as a vertex. We must also eliminate all triangles which were found to contain P_o in earlier adjacent triangle searches (for the case of the first adjacent search the number of triangles found before the entry triangle will be zero). The resultant list of adjacent triangles will contain either the next triangle to perform the adjacent triangle search on, or the list will be empty. If the list is empty, searching is complete for P_o. If the list is not empty, we must find all adjacent triangles for the triangle remaining in the list and eliminate triangles as above. This process must be continued until the list of triangles, after elimination, is empty. This algorithm is pictured in figure 5.5 .

Problems arise with the use of this algorithm because of the circular path which the search takes around a point and its relationship to the entry triangle. Interior points do not experience this problem because the search

path around the point always starts and ends with adjacent triangles. Border point search paths do not start and end with adjacent triangles. If the entry triangle is not a border triangle then the search path will go around the point in one direction and shut off when the first involved border or corner triangle is encountered (fig. 5.6).

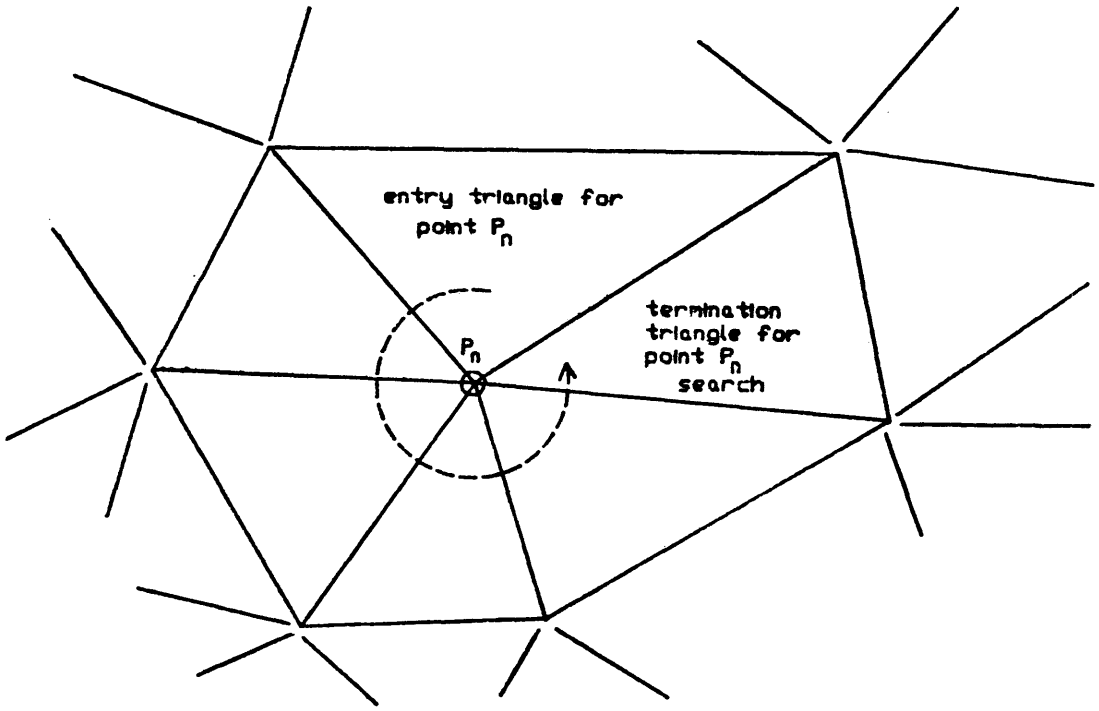


FIG. 5.5: Surrounding triangle search for interior point P_n

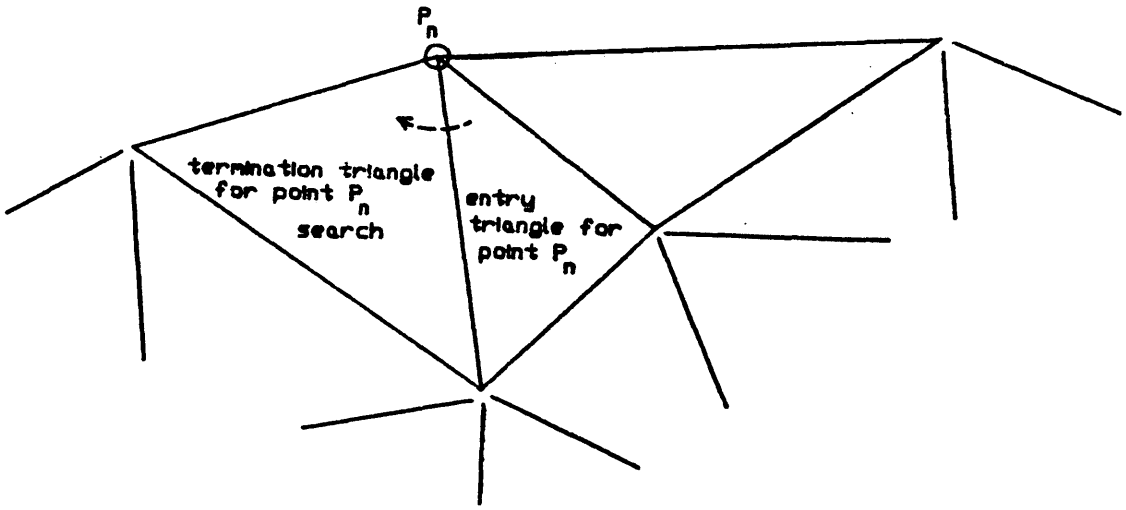


FIG. 5.6: Incomplete surrounding triangle search for border point P_n

In order to eliminate this problem, a check is made to determine if the entry triangle is a border or corner triangle. If it is, the search is conducted in the normal fashion. If the entry triangle is an interior triangle, then the MLG is searched using the fundamental "index offset" search until a border or corner triangle containing the border point is found. From this point the surrounding triangle search is conducted in the normal fashion (fig. 5.7).

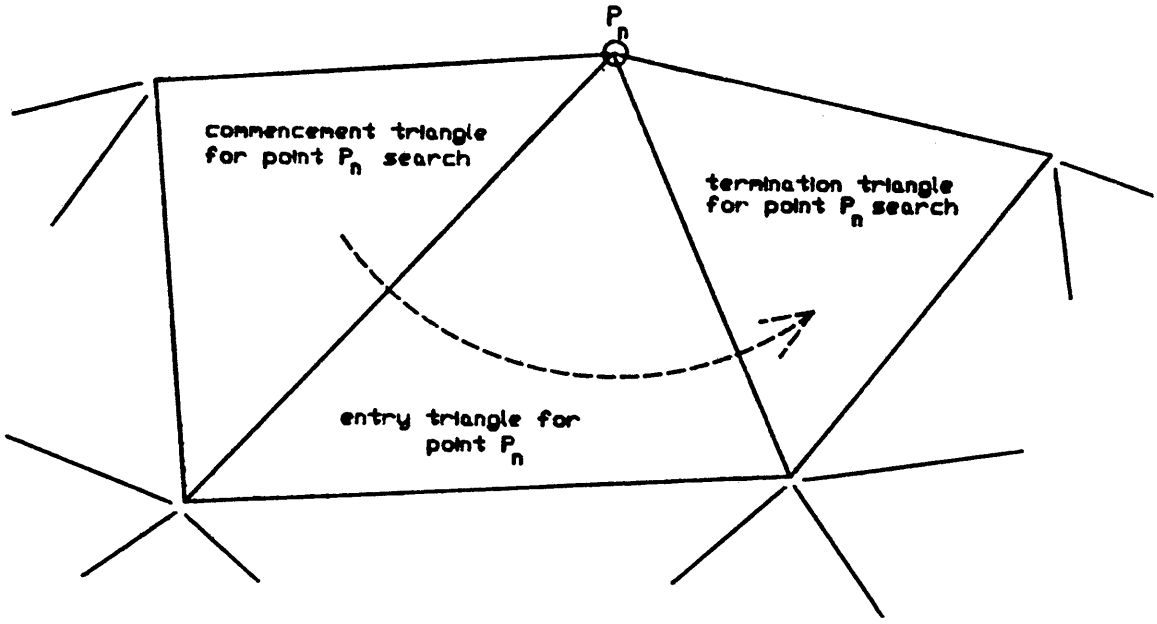


FIG. 5.7: Readjustment of commencement triangle for border point P_n search

Once again, as in the case of the adjacent triangle search, points being searched upon which come about through an entry triangle which is adjacent to a corner triangle (hidden interior triangle) must be reprocessed using the fact that the entry triangle in reality has three adjacent triangles.

Pseudo-code will not be given for this subroutine as it is very complex and would be very space consuming. Refer to appendix D.5 for the FORTRAN code of PNTSRCH.

Optimization on PNTSRCH has not yet been done. The

section of code which looks for the first occurrence of a corner or border triangle when updating a border point whose entry triangle is interior is an example of sections that would benefit from optimization. As was noted, the use of the rudimentary "index offset" search was incorporated to find the first qualifying triangle. However, this search does not keep track of triangles along the way which are involved with the point in question. Therefore when a corner or border triangle is found the whole process of finding surrounding triangles must be repeated.

In the models used for this study, model size was small enough to hide the time consumption of this section of code. If the model were to be increased to a sizable magnitude, the number of border points becomes small relative to the number of interior points but, time consumed in "reinventing the wheel" could be better spent processing more points. This is one area of improvement to be considered in future optimization.

The subroutine PNTSRCHF uses stored information about adjacent triangles instead of calling on the subroutine FINDADJ to calculate the adjacent triangles. This subroutine is used as a comparison between the methods of calculating adjacent triangles using the MLG (thus

reducing the amount of information being stored) and the method of storing all needed information about adjacent triangles (thus reducing calculation time). The triangle I.D.'s of all adjacent triangles are stored with each individual triangle's information in the triangle array. Thus producing large amounts of redundant data.

The PNTSRCH and PNTSRCHF algorithms are identical except when adjacent triangles are to be obtained. PNTSRCH uses FINDADJ to calculate the adjacent triangles, while PNTSRCHF references the triangle data array.

5.3.4 POINT MOTION

This section of the model execution family of algorithms applies the desired flow to the points of the space. As was described in section 4.3 of chapter 4, there are three possible flow equations, Modified Uniform Strain, Modified Parabolic flow in x and Random flow. Point motion is executed by a call to the subroutine MOVEPNT which in turn calls the individual point motion subroutines, STRAIN, PARABOL and RANDOM which are on the level below MOVEPNT. Calls to MOVEPNT are made at the end of each time step (time starts at t=0) so that execution of all sorting and searching is done at the start of the next time interval for the data of the previous timestep.

5.4 DATA OUTPUT ALGORITHMS

This particular family of algorithms produce output of data from program execution. This family is comprised of only one subroutine member which is called OUTDATA. Calls are made upon this subroutine passing a variable to distinguish the type of output desired.

There are three main categories of output, graphical, numerical partition, and statistical. Graphical output is used in the program DATAPLOT which plots triangle and point migration for all of the possible flows. Numerical partition data is used to watch the migration of data within the MLG. Finally, statistical output is generated for comparison testing which will be presented in chapter 6. The statistical data includes index offset values for different runs using both the adjacent triangle search (ADJSRCH) and the surrounding triangle search (PNTSRCH), as well as CPU timings for all subroutines (both initialization and model execution).

CHAPTER 6 - RESULTS AND CONCLUSIONS

6.1 INTRODUCTION

Results of the MLG performance in various test situations will be presented in this chapter along with conclusions that can be drawn from these results. Both results and conclusions will be intermixed throughout the chapter. An overview of the study and generalized conclusions will be given at the end along with future directions for the study of the MLG and its application to Lagrangian techniques. Results to be given in this chapter are as follows:

1. Use of the MLG is found to be superior with respect to search lengths when compared to the use of a list data structure.

2. No significant difference is detected between average offsets when regular and irregular initial grid configurations are used.

3. The computational cost of the MLG sort subroutine is N , the number of cells in the MLG.

4. Large sort sweep iteration counts are evident in the first two or three timesteps. After this point, counts decrease and remain relatively stable.

5. The centroid and the average coordinate attributes

are optimal processing attributes with respect to the four attributes tested in this study.

6. CPU reduction is evident when using PNTSRCHF as opposed to using PNTSRCH.

6.2 MLG vs. LIST

In this section we will discuss the advantages that the MLG holds over the List data structure in the area of search lengths.

The following definitions will be used in the foregoing discussion:

Define

N_{tot} - Total number of record searches in each of the searching algorithms.

N_{trs} - Number of triangle records searched for each individual point or triangle in each of the search algorithms.

N_t - Number of triangles present in the space.

N_p - Number of points present in the space.

In the case of the list data structure let us assume that the data structure is constructed of two, 2 dimensional arrays, namely the POINT and the TRIANGLE arrays. In the POINT array each record contains the three coordinates of the specific point while the TRIANGLE array

contains in each of its records the vertex (point) I.D.'s of its three vertices. The MLG data structure has been completely specified in earlier chapters and sections (section 3.3, chapter 3) and will not be explained at this time.

We now discuss the amount of searching required by both the list and MLG data structures in order to find all adjacent triangles around each triangle of the space.

In the case of use of the list data structure, for each triangle in the space T_m, the number of triangle records which must be searched is:

$$N_{trs} = N_t - 1 \qquad (1)$$

This is assuming that the number of adjacent triangles for each triangle is not known.

Using (1) from above, the total number of triangle record searches made in order to find the adjacent triangles to all triangles of the space is:

$$N_{tot} = N_t (N_t - 1)$$
$$= N_t^2 - N_t \qquad (2)$$

In using the MLG data structure, the maximum number of triangle records searched is calculated using the average of the maximum index offsets. Define O_m to be the maximum

index offset, so that the maximum number of triangle records searched in finding the adjacent triangles around triangle T_m is:

$$N_{trs} = ((2 \cdot O_m) + 1)^2 - 1 \qquad (3)$$

since the triangle T_m is not searched. And by (3), the maximum total number of triangle record searches conducted in order to find the adjacent triangles to all triangles of the space is:

$$\begin{aligned} N_{tot} &= N_t \ N_{trs} \\ &= N_t (((2 \cdot O_m) + 1)^2 - 1) \end{aligned} \qquad (4)$$

The use of the MLG will be advantageous only if:

$$N_t^2 - N_t \geq N_t (((2 \cdot O_m) + 1)^2 - 1)$$

or after simplification

$$(N_t^{1/2} - 1)/2 \geq O_m \qquad (5)$$

As an example, if 1000 triangles were present in the space, a maximum index offset of 15 would give us close to equal search lengths between the two data structures.

Now let us discuss searches for triangles surrounding the points of the space. In the case of the list data

structure, when finding all triangles which surround the point P_n, the number of triangle records searched is:

$$N_{trs} = N_t \qquad (6)$$

simply because the number of triangles which contain the point P_n as a vertex is unknown, thereby forcing us to search the entire triangle list.

By (6), the total number of triangle records searched in order to find the triangles which surround all points of the space is:

$$\begin{aligned} N_{tot} &= N_p \; N_{trs} \\ &= N_p \; N_t \end{aligned} \qquad (7)$$

In working with the MLG searching algorithm, PNTSRCH, in looking for surrounding triangles, since index offsets in the i and j directions may not be symmetrical, we must adjust our record search counts previously defined. Let us further define the following variables:

O_{mi} - Maximum index offset in the i direction.

O_{mj} - Maximum index offset in the j direction

The number of triangle record searches conducted to find all surrounding triangles around point P_n is

formulated to be:

$$N_{trs} = ((2 \cdot O_{mi}) + 1) \cdot ((2 \cdot O_{mj}) + 1) - 1 \quad (8)$$

since the entry triangle for point P_n is not searched. The total number of triangle record searches conducted in order to find the surrounding triangles for all points of the space is:

$$N_{tot} = N_p \cdot N_{trs}$$
$$= N_p \cdot ((((2 \cdot O_{mi}) + 1) \cdot ((2 \cdot O_{mj}) + 1) - 1) \quad (9)$$

Again, if the use of the MLG is to be to our advantage we must have (from (7) & (9)):

$$N_p \cdot N_t \geq N_p \cdot ((((2 \cdot O_{mi}) + 1) \cdot ((2 \cdot O_{mj}) + 1) - 1)$$

or after simplification

$$N_t/4 \geq (O_{mi} + 1/16) \cdot (O_{mj} + 1/16) \quad (10)$$

In the case of symmetrical offsets:

Given $O_{mi} = O_{mj} = O_m$

then $N_t/4 \geq (O_m + 1/16)^2$

or

$$(1/2) \cdot (n^{1/2} - 1/8) \geq O_m \quad (11)$$

The above operational count derivations using the MLG data structure depend entirely on the quantities O_m, O_{mi} and O_{mj}. If these index offsets are small enough, then the time and effort spent on the development of the MLG will be well worth it.

Rigorous mathematical bounds for the maximum index offsets for various types of flows and various types of processing attributes have not yet been determined. For now empirical proof will produce the ground work for a more stable definition of the exact numbers O_m, O_{mi} and O_{mj} on which the operational counts above depend.

One point needs to be made before we continue our discussion. The Uniform Strain flow equations presented in chapter 4 were found to be of little significance in testing the effectiveness of the MLG. This is evident when looking at plots of the point space through time. Shown in Table 6.1 are actual sort sweep iteration counts for a sample of 12 program executions using Uniform Strain point motion. It is obvious from these figures that data movement in the MLG is virtually non-existent after the first time step. Therefore, data pertaining to Uniform Strain point motion will not be incorporated into the results given below.

Therefore, there exist in this study two different types of fluid flows as well as four different types of processing attributes which can be used. A total of 24 programs runs were executed. The 24 runs were separated into 3 series. Within each series 8 runs were conducted using all possible combinations of flows and processing attributes. The mean offsets for each timestep were summed together to produce one entry for each execution per series. Results of these runs are presented in table 6.1 .

If we round the total column mean to the nearest integer and use it as the variable O_m in equation (5) we have:

$$(N_t^{1/2} - 1)/2 \geq 2 \qquad (12)$$

with $O_m = 2$

and for solving for N_t we obtain

$$N_t \geq 25$$

Table 6.1: Sort sweep iteration counts for 12 executions involving Uniform Strain Flow. (Three series of four executions)

Execution	t=0	t=1	t=2	t=3
Series 1				
Execution				
1	18	1	1	1
2	20	1	1	1
3	18	7	1	1
4	19	1	1	3
Series 2				
Execution				
1	20	1	1	1
2	19	1	1	1
3	19	1	1	2
4	19	1	1	1
Series 3				
Execution				
1	19	1	1	1
2	19	1	1	1
3	19	1	1	1
4	22	1	2	1

Table 6.2: Offset results of 24 program executions
with 4 timesteps per execution.
(Adjacent triangle search)
(Irregular initial grid configuration)

Table entries are the sums of average
mean offsets for each individual
execution with 4 timesteps/execution.
N1, N2 and N3 indicate number of total timesteps
per column. (one column corresponds to one series)

Flows and Attributes	N1=32	N2=32	N3=32	Attribute Mean
Random				
Centroid	6.523	6.344	6.000	1.572
Vertex	8.984	8.865	8.677	2.211
Avg. Coord.	6.094	6.245	6.478	1.568
Side Bisector	7.058	6.867	6.881	1.734
Parabolic				
Centroid	6.276	6.510	6.640	1.619
Vertex	9.122	9.126	8.864	2.259
Avg. Coord.	6.501	6.461	6.131	1.591
Side Bisector	7.163	7.310	7.467	1.828
Column totals	57.721	57.728	57.138	
Column means	1.804	1.804	1.786	

Total for all columns - 172.587
Total column mean - 1.798

NOTE: refer to table 6.3 for sample variances.

Therefore, if our model is using more than 25 triangles and the maximum index offset, O_m, is on the order of 2, the use of the MLG will be advantageous over the use of the list data structure. In reality the number of triangles will be much larger than 25. So the cost in search lengths is greatly decreased by use of the MLG in searching for adjacent triangles.

If we concentrate on comparing only the search lengths involved in using the MLG versus using the list, putting aside the cost of upkeep of the MLG, the MLG far out performs the list.

Using the fact that all initial grids in the experimental runs contained at least 760 triangles, the number of triangle record searches conducted to find all adjacent triangles in the space using the list data structure would be (from (2)):

$$N_{tot} = N_t^2 - N_t$$
$$= 576,840$$

Whereas, using (4) and an average maximum index offset of 2 for each triangle, we have (12):

$$N_{tot} = N_t \cdot (((2 \cdot O_m) + 1)^2 - 1)$$
$$= 18,240$$

which is approximately a 97% reduction in the list search length. This argument of course uses an average index offset in the place of a maximum index offset, but only a very small fraction of the triangles in these experimental runs deviated from this average. This is evident when examining the variances of mean offsets for a sample of individual executions (table 6.3).

Table 6.3: Variances about mean offsets for 8 program executions.(Adjacent triangle search) (4 timesteps for each execution)

Flows and Attributes	Mean Offset Variances t=0	t=1	t=2	t=3
Random				
Centroid	2.466	2.381	2.138	2.136
Vertex	1.378	0.956	0.737	0.722
Avg. Coord.	0.627	0.581	0.564	0.568
Longest Bis.	0.770	0.462	0.439	0.387
Parabolic				
Centroid	0.435	0.323	0.454	0.869
Vertex	2.478	0.726	0.855	1.232
Avg. Coord.	0.580	0.403	0.456	0.633
Longest Bis.	0.671	0.393	0.437	0.658

As for the search involving surrounding triangles around space points, average index offsets for both the i

and j directions were obtained through the same series of program executions only using the surrounding triangle search subroutine (table 6.4 & 6.4a).

Table 6.4: Offset results for 24 program executions with 4 time steps per execution.
(Surrounding triangle search)
(Irregular initial grid construction)

Table entries are the sums of i and j mean offsets for each individual execution. (4 timesteps/execution)

Flows and Attributes	N1=32		N2=32		N3=32	
	i	j	i	j	i	j
Random						
Centroid	4.112	4.145	4.152	4.180	4.154	4.172
Vertex	4.746	5.229	4.609	4.953	4.717	4.993
Avg. Coord.	4.026	4.204	4.214	4.186	4.116	4.302
Longest Bis.	4.354	4.237	4.468	4.330	4.398	4.508
Parabolic						
Centroid	4.400	4.171	4.531	4.297	4.570	4.201
Vertex	5.300	5.377	5.337	5.178	5.262	5.053
Avg. Coord.	4.559	4.161	4.596	4.218	4.469	4.197
Longest Bis.	4.513	4.371	4.499	4.420	4.725	4.356
Column Totals	36.010	35.895	36.406	35.762	36.411	35.782

i Total - 108.827
i Mean - 1.134

j Total - 107.439
j Mean - 1.119

Table 6.4a: Attribute Means from table 6.4

Table entries are the sums of i and j components of each row of table 6.4, divided by 12 (4 timesteps per series and a total of three series).

Flows and Attributes	Attribute means i	Attribute means j	std. dev. i	std. dev. j
Random				
Centroid	1.035	1.040	.0067	.0071
Vertex	1.173	1.265	.0351	.0514
Avg. Coord.	1.030	1.058	.0204	.0173
Side Bisector	1.102	1.090	.0175	.0341
Parabolic				
Centroid	1.125	1.106	.0935	.0240
Vertex	1.325	1.300	.1212	.0843
Avg. Coord.	1.135	1.048	.0914	.0153
Side Bisector	1.145	1.111	.0952	.0188

If we once again round up the mean i and j offsets obtained in table 6.4, we have for the variables O_{mi} and O_{mj}, the values 2 and 2 respectively.

If we use these variables in equation (10) we obtain:

$$N_t/4 \geq (33/16)^2 \simeq 4$$

and solving for N_t we get

$$N_t \geq 16$$

Thus, if the number of model triangles is larger than or equal 16 we will be reducing search lengths. If we use (7) and (9) and the fact that the minimum number of triangles present in these executions, N_t, is 730, and the minimum number of points present, N_p, is 406, we obtain, first of all, the total number of triangle record searches when using the list:

$$N_{tot} = 296,380 \text{ records}$$

while the number of triangle records searched in the MLG is:

$$N_{tot} = 9,744 \text{ records}$$

This represents again a 97% reduction in the list search length. And once again, the above argument relies on an average index offset obtained from experimentation. The variances of the individual executions are on the same order of magnitude as the variances shown in table 6.3, Therefore variances from the surrounding triangle search will not be given here.

What is to be noted from the above discussion is that overwhelming reductions in search lengths are accomplished by the incorporation of the MLG. Search length figures derived above prove this.

6.3 DIFFERENCES IN INITIAL GRID CONFIGURATIONS

The use of regular and irregular initial grid configurations was incorporated in this study so as to be a check on the independence of MLG performances with respect to initial grid construction. In other words, we would like to have a measure of the MLG's performance on a regular grid as well as an irregular grid so as to determine if the MLG's performance deteriorates with an increase of irregularity in the grid space. Table 6.5 gives sums of mean index offsets for 24 executions consisting of 3 series which use all combinations of flows and processing attributes. Table 6.5a gives the attribute (row) means for these executions. Each attribute mean is calculated by summing the individual i and j components of each row and then dividing each of these sums by 12 (3 series of 4 time steps each). A regular initial grid configuration was constructed and used in these executions.

Table 6.5: Offset results for 24 program executions with 4 time steps per execution.
(Surrounding triangle search)
(Regular initial grid construction)

Table entries are the sums of i and j mean offsets for each individual execution. (4 timesteps/execution)

Flows and Attributes	N1=32 i	N1=32 j	N2=32 i	N2=32 j	N3=32 i	N3=32 j
Random						
Centroid	4.000	4.131	4.000	4.089	4.000	4.338
Vertex	4.644	5.134	4.623	5.284	4.628	5.144
Avg. Coord.	4.000	4.267	4.006	4.307	4.000	4.235
Longest Bis.	4.201	4.333	4.243	4.391	4.239	4.274
Parabolic						
Centroid	4.375	4.233	4.348	4.208	4.463	4.250
Vertex	5.164	5.410	5.180	4.963	5.167	5.331
Avg. Coord.	4.377	4.305	4.311	4.200	4.323	4.161
Longest Bis.	4.430	4.419	4.488	4.560	4.493	4.409
Column Totals	35.191	36.232	35.199	36.002	35.313	36.142

i Total - 105.703
i Mean - 1.101

j Total - 108.376
j Mean - 1.129

Table 6.5a: Attribute Means from table 6.5

Table entries are the sums of i and j components of each row of table 6.5, divided by 12 (4 timesteps per series and a total of three series).

Flows and Attributes	Attribute means i	j	std. dev. i	j
Random				
Centroid	1.000	1.046	.0	.290
Vertex	1.157	1.296	.0100	.0975
Avg. Coord.	1.001	1.067	.0011	.0127
Side Bisector	1.057	1.083	.0088	.0477
Parabolic				
Centroid	1.098	1.057	.0980	.0309
Vertex	1.292	1.308	.1309	.1120
Avg. Coord.	1.084	1.055	.0699	.0231
Side Bisector	1.117	1.115	.0737	.0342

In examining the attribute means given in table 6.5a with the attribute means give in table 6.4a it can be seen that the differences in the means is very insignificant. Also when comparing standard deviations between attributes using regular and irregular initial grids it is noted that these standard deviations are very small.

As a sight conclusion from looking at these offset means and standard deviations, it can be seen that the

means of executions performed on regular initial grids do not seem to differ from mean of executions performed on irregular initial grids.

6.4 COST OF THE MLG SORT

In the above examination, we did not take into consideration the cost of upkeep on the MLG. The sorting routine which was described in earlier sections and chapters is an order N algorithm (N being the number of cells in the MLG), which is very cost efficient in the attempt to eliminate unneeded computations while still provide fast data access. Following is proof that the 2 dimensional MLG sort routine developed by J.P. Boris is of order N (Boris,85).

Let us make the following definitions:

Define N - number of cells in the MLG.

N_{ii} - number of MLG cell interchanges in all i directional vectors.

N_{ij} - number of MLG cell interchanges in all j directional vectors.

N_{ik} - number of MLG cell interchanges in all k directional vectors.

D_i MLG dimension in the i direction.

D_j MLG dimension in the j direction.

D_k MLG dimension in the k direction (D_k = 1, 2-Dimensional model).

Now

$$N_{ii} = (D_i - 1) \times (D_j) \times (D_k)$$

$$N_{ij} = (D_i) \times (D_j - 1) \times (D_k)$$

$$N_{ik} = (D_i) \times (D_j) \times (D_k - 1)$$

and for one iteration sweep through the MLG the total number of cell interchanges, N_{ti}, is:

$$N_{ti} = N_{ii} + N_{ij} + N_{ik}$$

and after simplification

$$N_{ti} = (3\ D_i\ D_j\ D_k) - (D_j\ D_k) - (D_i\ D_k) - (D_i\ D_j)$$

$$= (3\ N) - N\ (\ (1/D_i) + (1/D_j) + (1/D_k)\ ) \quad (13)$$

Now let $d = (1/D_i) + (1/D_j) + (1/D_k)$

so

$$N_{ti} = (3\ N) - (d\ N)$$

where $1 < d < 3$,

$d \to 1$ when D_i and D_j become large

and $d \to 3$ when D_i and $D_j \to 1$

As will be seen in the results of the next section,

the number of sweep iterations is relatively low throughout the program execution except for the first one or two time steps. This occurs because the initial sort of the MLG may require much data movement across the grid in order to find the proper residing place for the data. But for the most part, sweep iterations for each timestep are relatively low.

If we define N_{si} to be the average sweep iteration count for then the cost of the MLG sort subroutine is:

$$COST = N_{si} \quad ((3\ N) - (d\ N))$$

And as can be seen, this is an order N sorting algorithm.

6.5 SORT SWEEP ITERATION COUNTS

The number of sweeps which are made through the MLG while sorting, plays an important role in the evaluation of the MLG and the sorting techniques used on it. If large sort sweep iteration counts are common throughout program execution, the cost of maintaining the MLG becomes less desirable. Fortunately, sort sweep iteration counts are relatively low throughout program execution, with the exception of a few initial timesteps.

The same program executions performed in section 6.2

involving adjacent triangle searches produced data about sort sweep iteration counts for each of the individual executions. Table 6.6 summarizes the results of these runs.

Table 6.6: Sort sweep iteration count results of 24 program executions with 4 timesteps per execution.

Table entries are total execution sweep iteration counts for each execution
attribute mean = (N1 + N2 + N3)/12

Flows and Attributes	N1=32	N2=32	N3=32	Attribute Mean
Random				
Centroid	32	32	37	8.417
Vertex	37	34	35	8.833
Avg. Coord.	32	34	35	8.417
Longest Bis.	39	37	51	10.538
Parabolic				
Centroid	52	50	52	12.833
Vertex	48	47	46	11.750
Avg. Coord.	57	46	50	12.750
Longest Bis.	54	50	51	12.917
Column totals	351	330	357	
Column means	10.969	10.313	11.156	

Overall column total - 1038
Overall column mean - 10.813

The overall column mean from Table 6.6 is misleading.

This mean is averaging large counts which are present in the first one or two timesteps. Table 6.7 shows counts for the first and second timesteps for the same executions in Table 6.6 for only one series. As can be seen, the averages for these two timesteps are well above the overall mean count obtained in table 6.6, and in fact most of the mean for the two timesteps given in table 6.7 is contributed by the first timestep in all runs.

Table 6.7: Sort sweep iteration counts for 8 executions. (timesteps t=0 and t=1)

Table entries are sweep iteration counts for each indicated timestep

Flows and Attributes	t=0	t=1	row mean
Random			
Centroid	19	5	12.000
Vertex	20	7	13.500
Avg. Coord.	19	7	13.000
Longest Bis.	21	9	15.000
Parabolic			
Centroid	20	8	14.000
Vertex	20	8	14.000
Avg. Coord.	19	8	13.500
Longest Bis.	20	8	14.000
Column totals	158	60	
Column means	19.750	7.500	

As can be seen from tables 6.6 and 6.7, the numbers of

sort sweep iterations are large for the first one or two timesteps, but in as little as one timestep they decrease drastically

Presented in appendix E.1 and E.2 are computer generated results concerning program executions using random and parabolic point motion for 8 timesteps. In reference to appendix E.1 and E.2, it can be seen that after the first timestep for each of the flows, the sweep iteration counts decrease dramatically, and in general, are very well behaved. The parabolic data does present local increases in iteration counts, but this is explained by the build up of potential position changes in the timesteps previous, until enough point movement causes global point and data interchanges, thus increasing iteration counts.

6.6 CENTROID AND AVERAGE COORDINATE ATTRIBUTES

Different processing attributes produce different results when looking at average index offsets in searching for triangles. This section of the results will examine the four processing attributes of the study and show that the centroid and average coordinate attributes are superior to the remaining two.

Tables 6.8 and 6.9 give summaries of mean offsets for

adjacent and surrounding triangle searches which were given in tables 6.2 and 6.4a above.

Table 6.8: Mean attribute offsets for 24 program executions (adjacent triangle search)

Flows and Attributes	Attribute means
Random	
Centroid	1.572
Vertex	2.211
Avg. Coord.	1.568
Side Bisector	1.734
Parabolic	
Centroid	1.619
Vertex	2.259
Avg. Coord.	1.591
Side Bisector	1.828

Table 6.9: Mean attribute offsets for 24 program executions (surrounding triangle search)

Flows and Attributes	Attribute means i	j
Random		
Centroid	1.035	1.040
Vertex	1.173	1.265
Avg. Coord.	1.030	1.058
Side Bisector	1.102	1.090
Parabolic		
Centroid	1.125	1.106
Vertex	1.325	1.300
Avg. Coord.	1.135	1.048
Side Bisector	1.145	1.111

As can be seen by examining tables 6.8 and 6.9, the mean offsets for the centroid and average coordinate processing attributes are significantly lower than the mean offsets for the vertex attribute and the longest bisector attribute.

6.7 CPU REDUCTION USING PNTSRCHF

PNTSRCHF is a subroutine much like PNTSRCH, in that it also searches for surrounding triangles in the point space. The difference between the two subroutines is that in PNTSRCHF when searching for adjacent triangles, calls are not made to FINDADJ. Information about adjacent triangles is stored rather than derived as in the case of PNTSRCH. Therefore, three more arrays of dimension NUMTRI x 3 are needed to store information about these adjacent triangles. The time reduction accomplished by not relying on FINDADJ to calculate adjacent triangles is very apparent. Appendices E.3 and E.4 contain computer generated data showing, among other things, CPU time requirements for both PNTSRCHF (Analysis (stor)) and PNTSRCH (Analysis (srch)). The magnitude of the times are on the the order of 10-20 times greater for PNTSRCH than they are for PNTSRCHF. This characteristic is evident through all executions produced.

What this is saying is that the time difference between searching for adjacent triangles (in looking for surrounding triangles) and having the adjacent triangles already available to us is large in magnitude. However, the price we must pay in order to save this time is to require more main memory storage. Three additional arrays at NUMTRI elements per array requires a great deal of memory when the number of triangles in the space increases.

6.8 CONCLUSIONS

This study has conducted many different tests concerning the performance of a newly developed data structure. The MLG, which was originally developed for space point motion models has been extended to a 2 dimensional figure model for use in modeling transient hydrodynamic fluid flows.

Different types of triangle attributes have been tested in order to find one or two which will partition data into minimal independent data sets, therefore enabling the use of multiple processing techniques.

Two triangle attributes were found to partition data quite compactly with respect to the measure of index offsets. Therefore, the incorporation of multi-processing

techniques in the future is very possible.

As for the upkeep of the MLG, an efficient order N sort algorithm was introduced and tested. It was found to be relatively easy to adjust after point motion in the space had violated monotonicity. Also, as time progressed through the program executions, the MLG structure adapted itself to the data it contained to the extent that large numbers of sort iterations were not needed to keep the MLG functional.

Finally, the performance of the MLG under different initialization schemes was consistent, implying that flexibility in the MLG is evident.

Overall, the MLG performance studied and recorded in this report, suggests that this data structure will eliminate many data management problems which presently plague highly transient hydrodynamic simulation models

6.9 FUTURE DIRECTIONS

This study has established the MLG as a very good candidate for use as a data structure in highly transient hydrodynamic simulation problems. Some areas were, however not investigated and deserve to be mentioned. This study did not incorporate grid reconstruction after point motion corruption of grid connections. This is one area that is

strongly required in order to maintain a reliable mathematical model.

One more area worth expanding upon is the process of establishing a reliable maximum index offset requirement for data access. This study could only produce mean offsets to be used in place of a maximum. When the characteristics of the MLG become better known then the ability to nail down a concrete maximum index offset will follow. And once this absolute maximum can be established, data partitioning in order to utilize multi-processing computers will be obtained.

Optimization of search procedures as noted in section 5.3.3.2 is one more area where optimization will increase MLG performance.

Finally, the extension of the MLG to a three dimensional model must be accomplished in order to apply it to real life simulation problems.

REFERENCES CITED

Bell J.L., 1982, "Report on interview with Marty Fritts about Free Lagrangian models", personal interview - no publication, PP. 1-16.

Bell J.L., Patterson G.S., 1985, "Data Organization in Large Numerical Computations", to be published, PP. 11-12.

Boris J.P., 1985, "A Vectorized 'Nearest Neighbor' Algorithm of Order N using a Monotonic Logical Grid", NRL Memorandum Report 5570, PP. 9-27, Laboratory for Computational Physics, Naval Research Laboratory, Washington D.C. .

Fritts M.J., Boris J.P., 1979, "The Lagrangian Solution of Transient Problems in Hydrodynamics using a Triangular Mesh", Journal of Computational Physics, PP. 173-215, Naval Research Laboratory, Washington D.C. .

Lambrakos S.G, Boris J.P., 1985, "Geometric Properties of the Monotonic Logical Grid Algorithm for Near Neighbor Calculations", technical report, PP. 1-12.

APPENDIX A

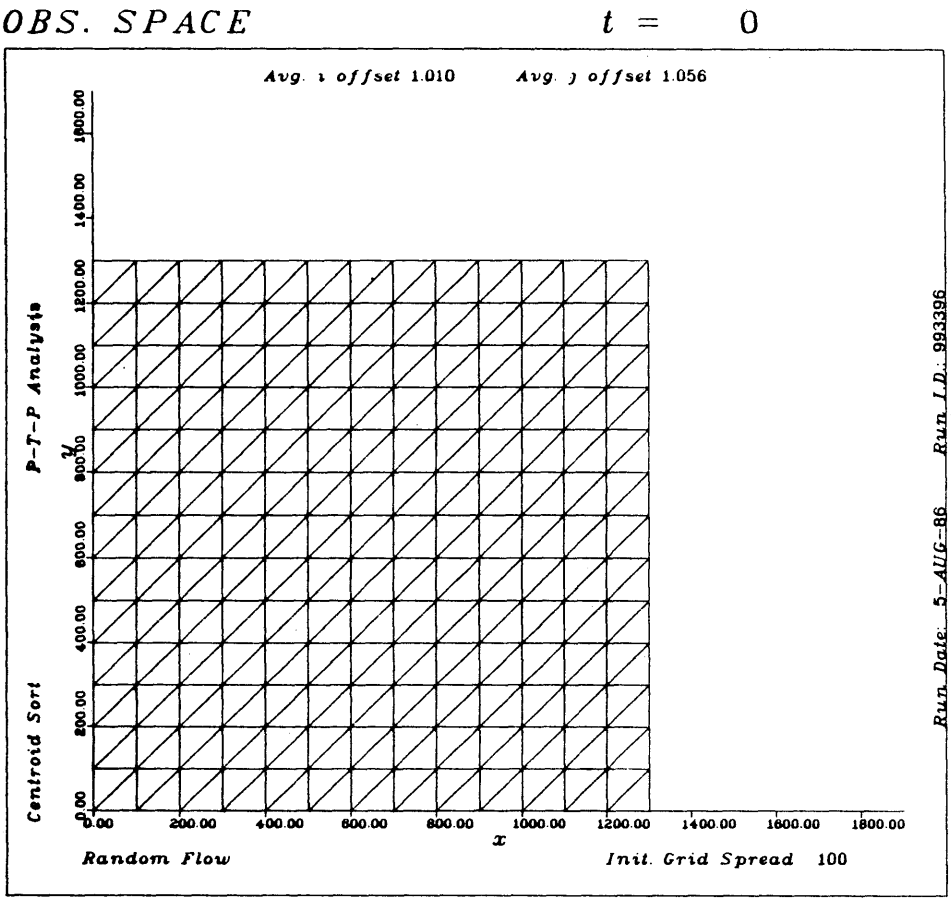
This appendix contains computer generated plots of initial grid configurations constructed by the program MLG. There are two main types of possible initial grids, regular and irregular. Each of these types are broken down into two subgroups, symmetrical and slightly deviated. The plots consist of only triangles, however implicitly these triangles are created from the points of the space. Therefore a point exists (but is not plotted) at each of the triangle vertices.

Each of the four possible initial grids are presented in the following subsections:

SUBSECTION #	INITIAL GRID CONFIGURATION
A.1	Regular symmetric
A.2	Regular deviated
A.3	Irregular symmetric
A.4	Irregular deviated

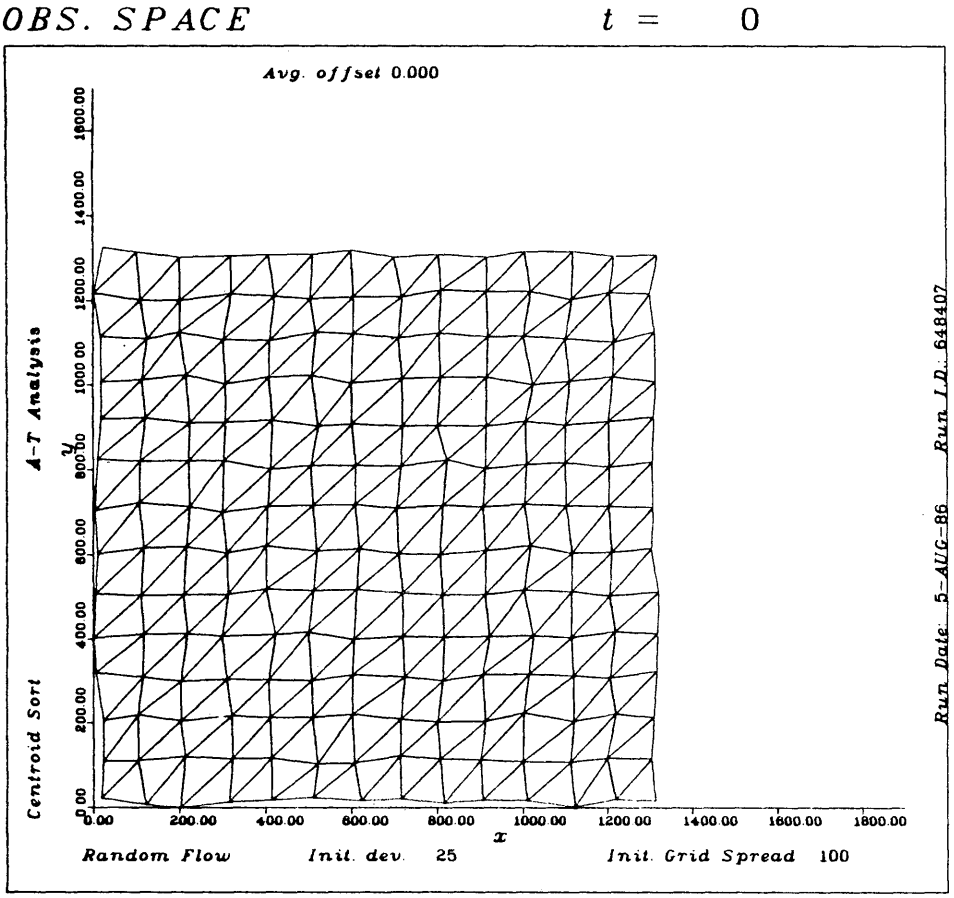
SUBSECTION A.1 - REGULAR SYMMETRIC

Following is a regular symmetric initial grid containing 196 points and 338 triangles.



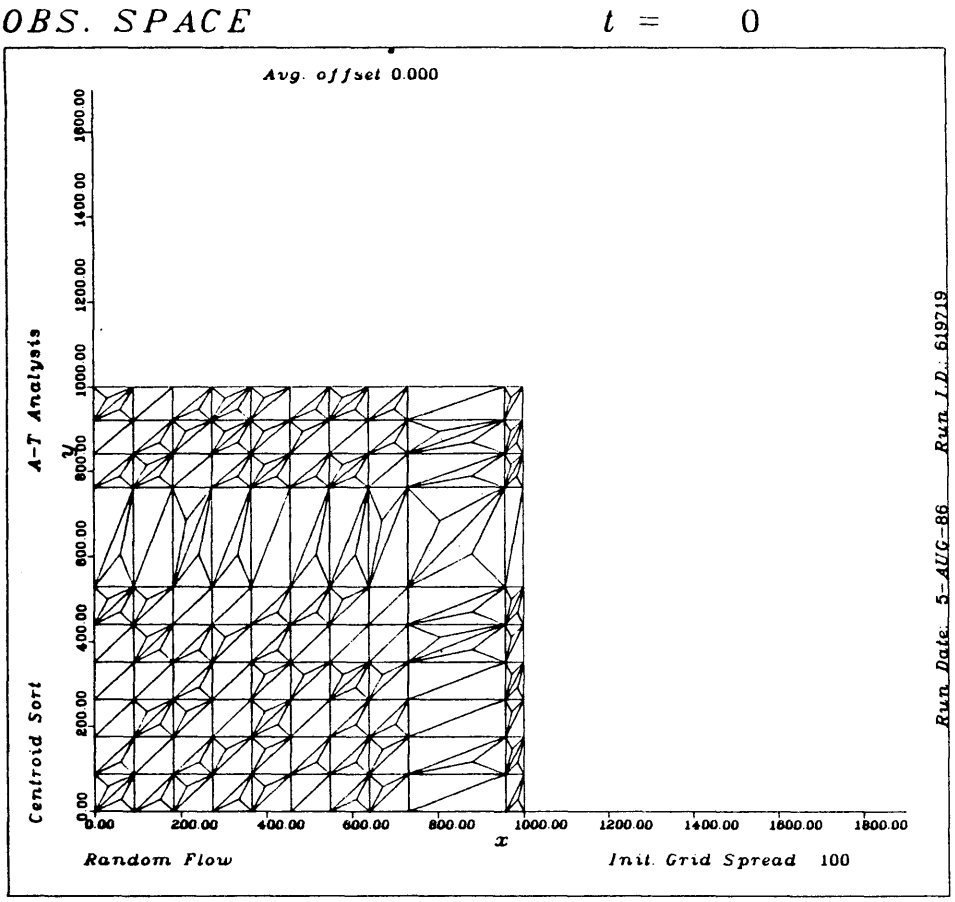
SUBSECTION A.2 - REGULAR DEVIATED

Following is a regular deviated initial grid containing 196 points and 338 triangles.



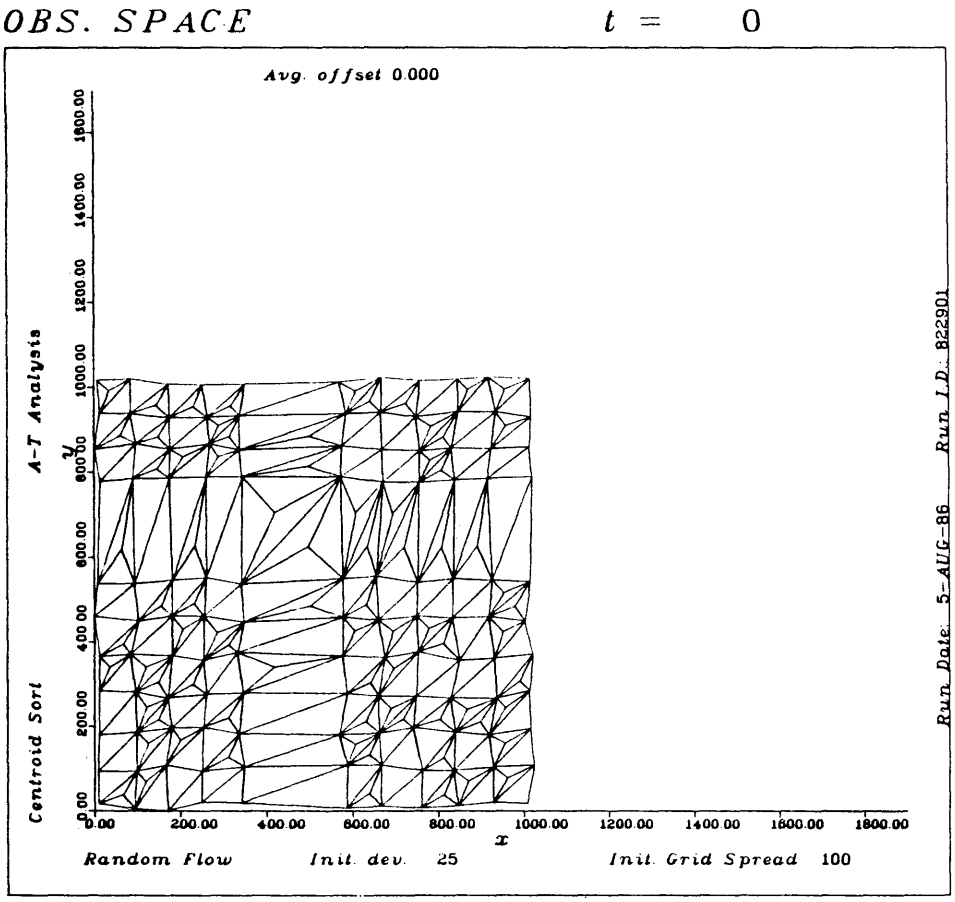
SUBSECTION A.3 - IRREGULAR SYMMETRIC

Following is an irregular symmetric initial grid containing 216 points and 390 triangles.



SUBSECTION A.4 - IRREGULAR DEVIATED

Following is an irregular deviated initial grid containing 200 points and 358 triangles.



APPENDIX B

Appendix B contains computer generated plots which show point (and triangle) motion whithin the space using three different flow equations, Modified Uniform Strain flow, Modified Parabolic flow in x and Random flow. Plots consist of only triangles, but implicitly points are present (but not plotted) at all triangle vertices.

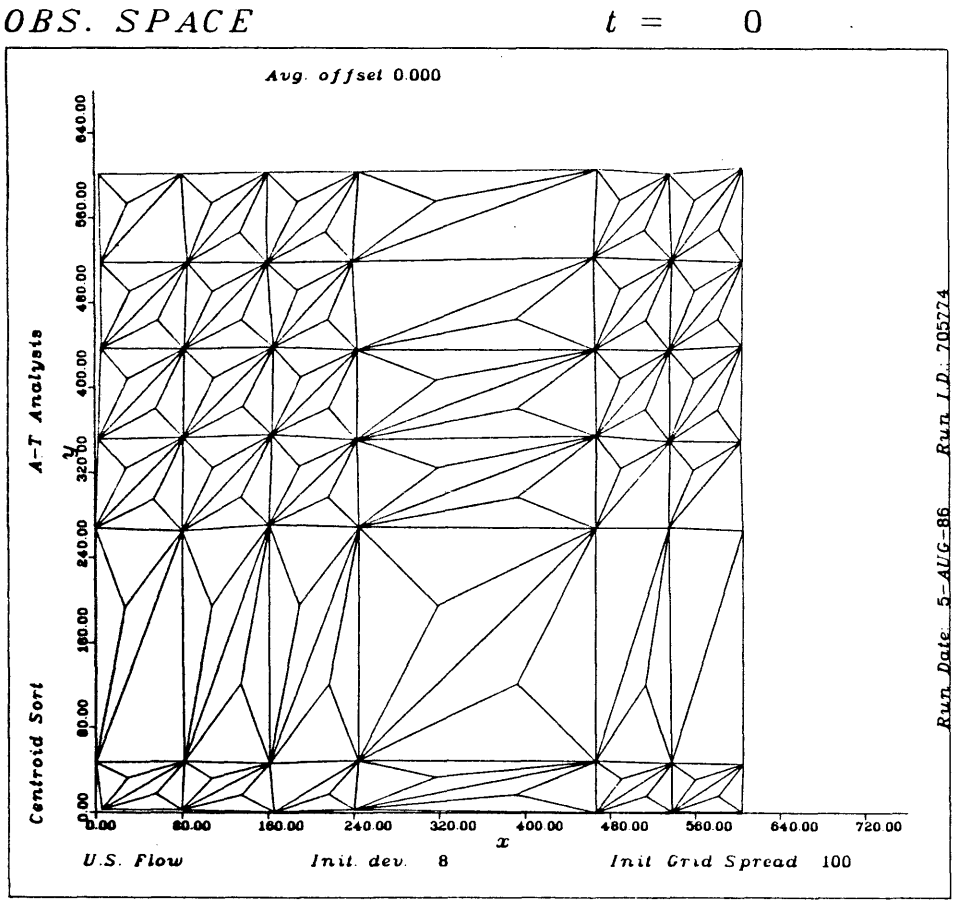
All flows are plotted over three timesteps not including time t=0 (time t=0 is the initial grid stage). The time increment for all flows is 5 units.

The following subsections contain plots of the individual flows:

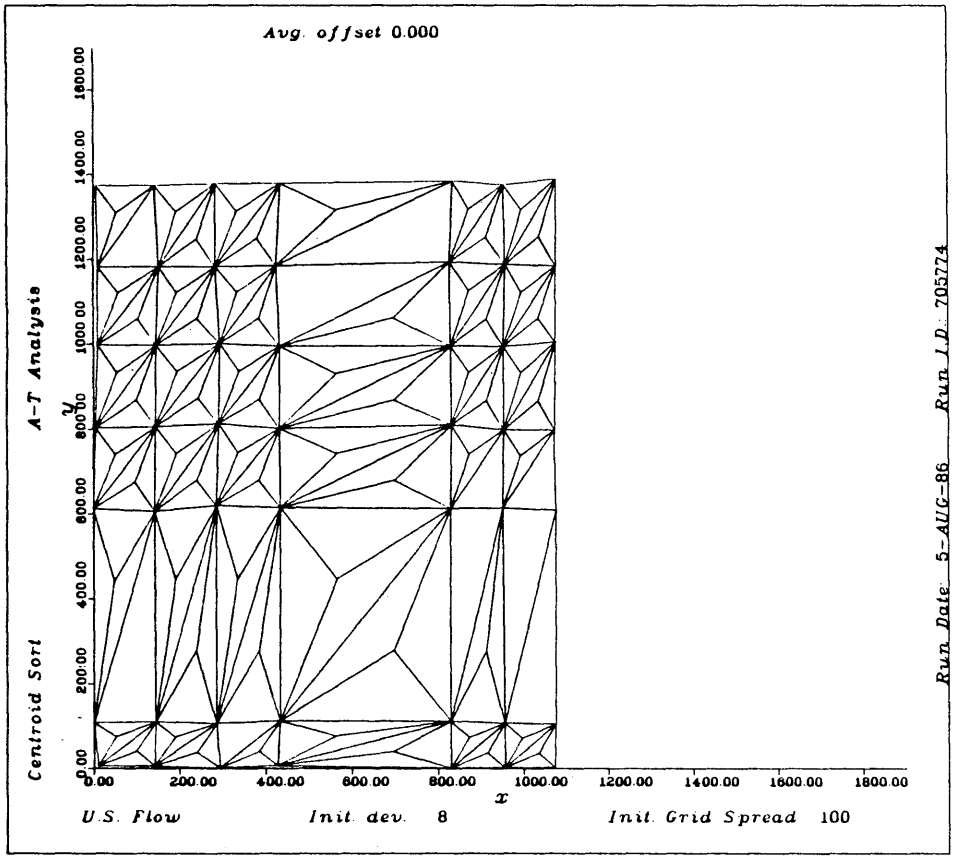
SUBSECTION #	FLOW
B.1-B.4	Uniform Strain
B.5-B.8	Parabolic in x
B.9-B.12	Random

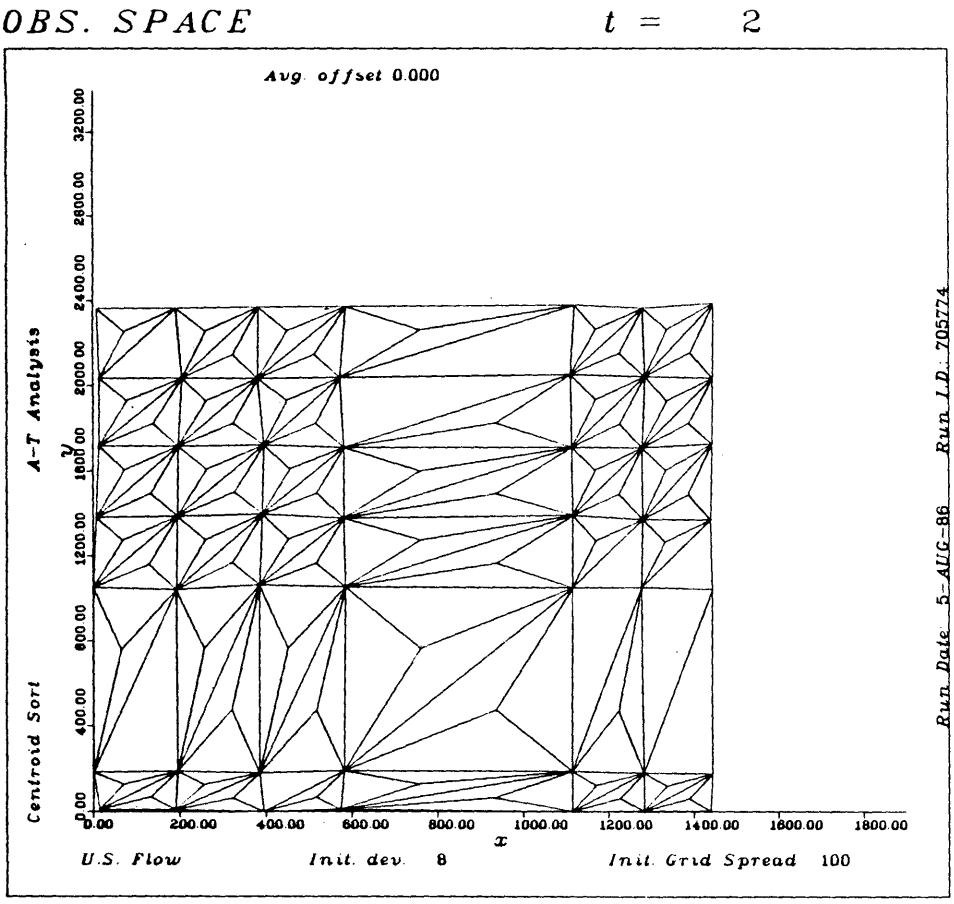
SUBSECTION B.1-B.4 - UNIFORM STRAIN FLOW

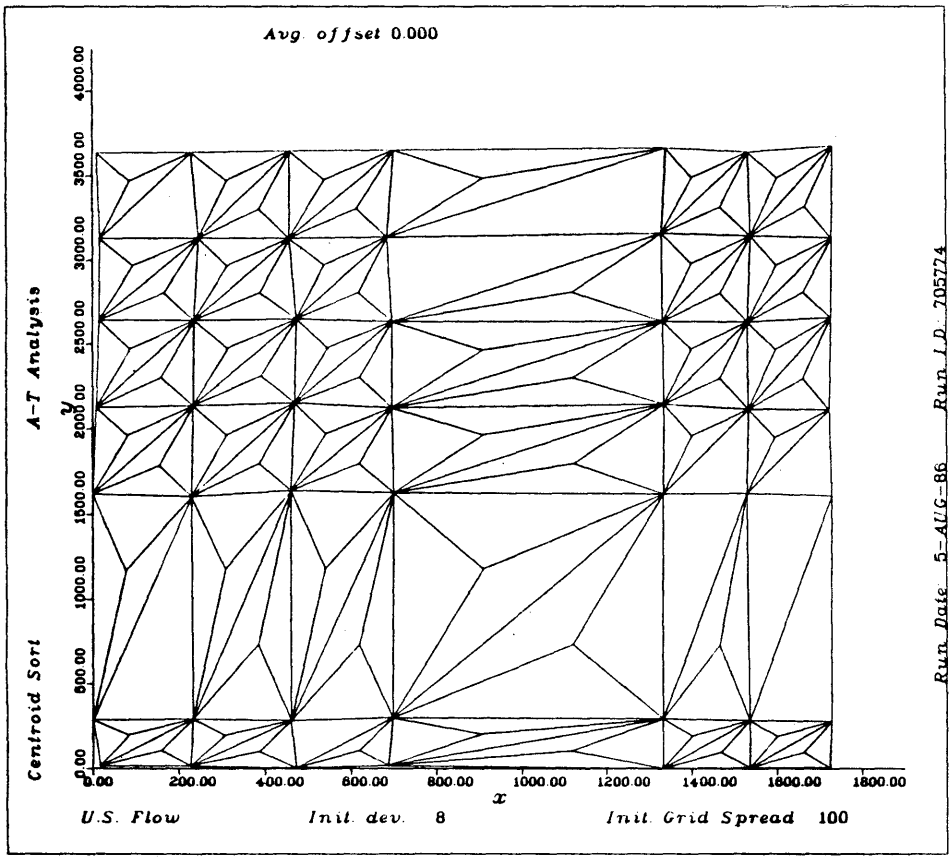
Following are computer generated plots for Uniform Strain flow used in MLG. Four plots are present including time t=0. There are 196 triangles and 111 points present in each plot frame.



OBS. SPACE $t = 1$

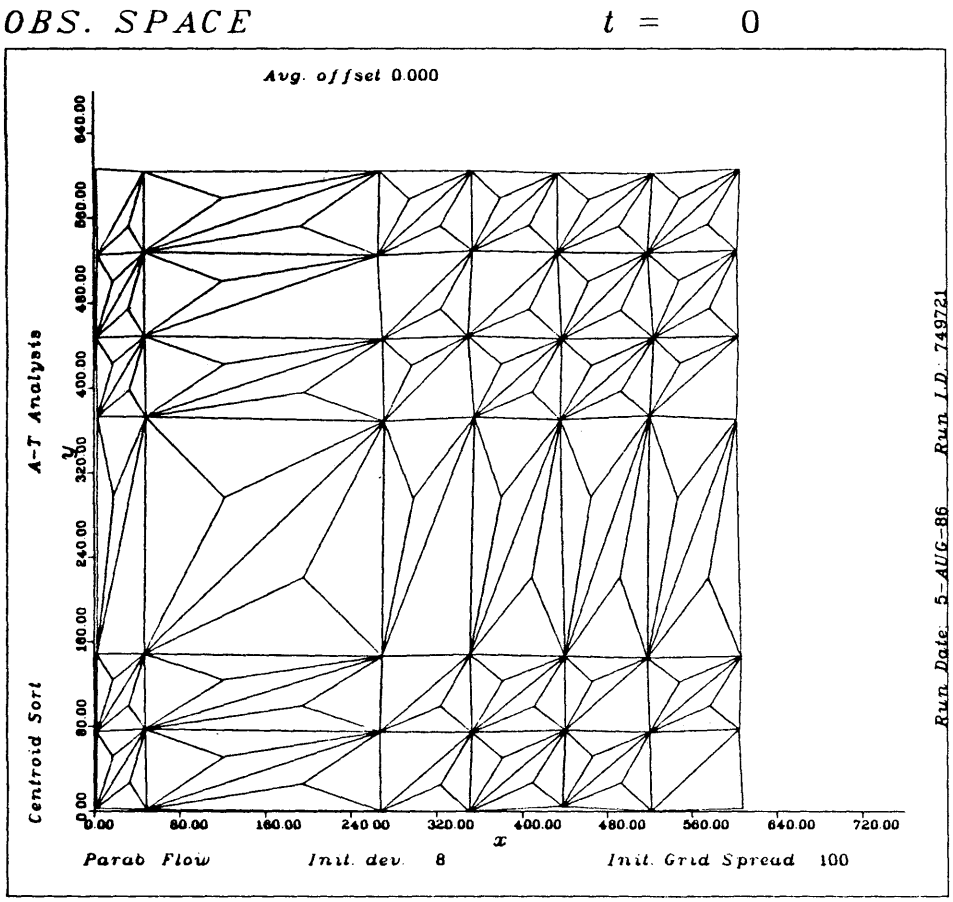

Avg. offset 0.000
A-T Analysis
Centroid Sort
Run Date: 5-AUG-86
Run I.D.: 705774
y
x
0.00
200.00
400.00
600.00
800.00
1000.00
1200.00
1400.00
1600.00
1800.00
U.S. Flow
Init. dev. 8
Init. Grid Spread 100


OBS. SPACE
t = 2
Avg. offset 0.000
A-T Analysis
Centroid Sort
y
x
0.00
200.00
400.00
600.00
800.00
1000.00
1200.00
1400.00
1600.00
1800.00
2000.00
2200.00
2400.00
2600.00
2800.00
3000.00
3200.00
Run Date 5-AUG-86
Run I.D.: 705774
U.S. Flow
Init. dev. 8
Init. Grid Spread 100

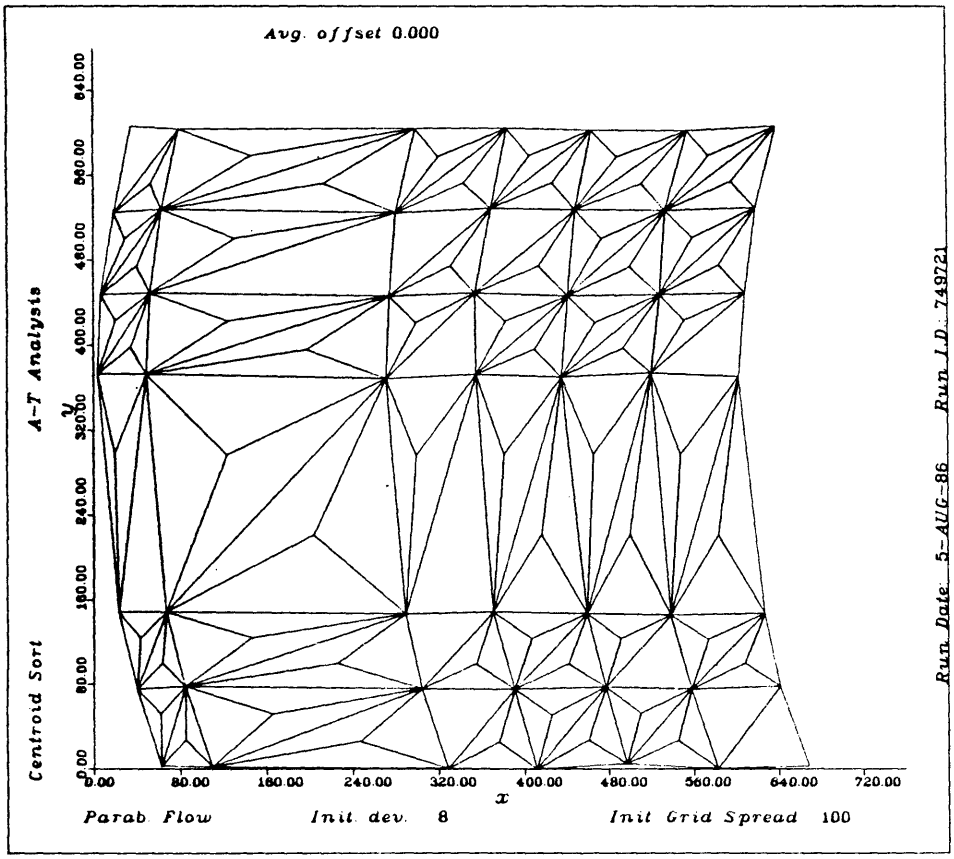

OBS. SPACE t = 3
Avg offset 0.000
A-T Analysis
Centroid Sort
Run Date 5-AUG-86
Run I.D. 705774
y
x
0.00
200.00
400.00
600.00
800.00
1000.00
1200.00
1400.00
1600.00
1800.00
2000.00
2500.00
3000.00
3500.00
4000.00
U.S. Flow
Init. dev. 8
Init. Grid Spread 100

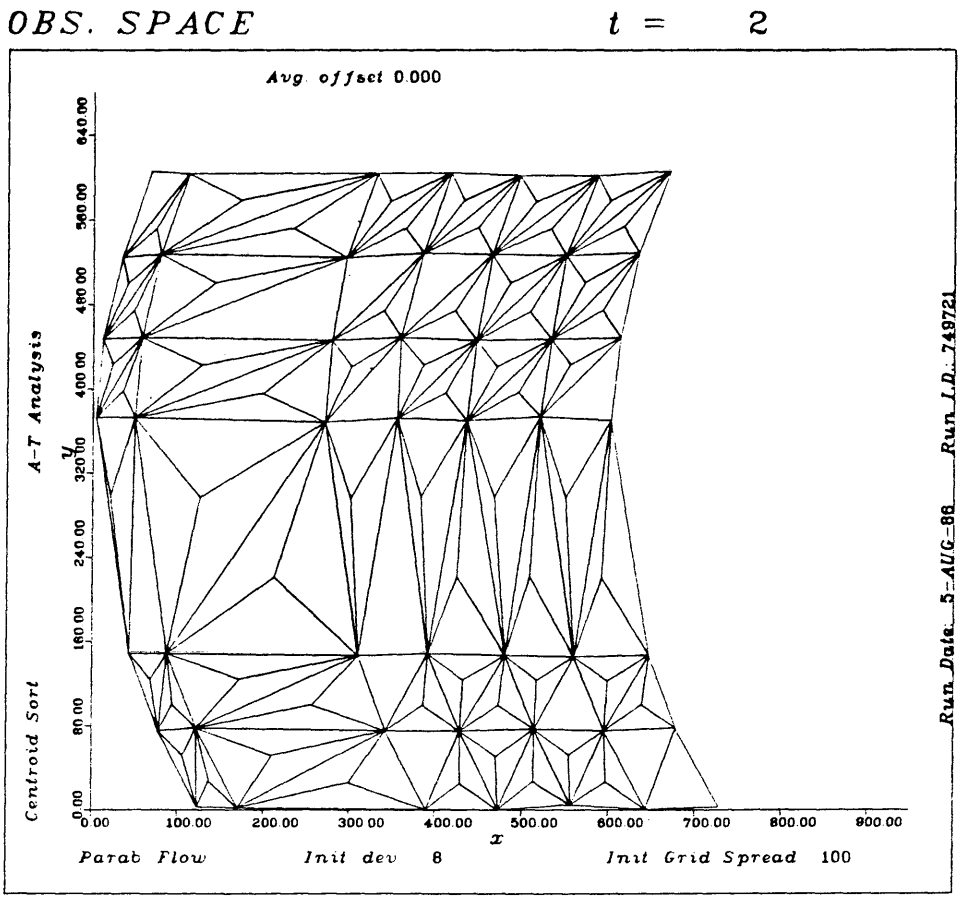
SUBSECTION B.5-B.8 - PARABOLIC FLOW

Following are computer generated plots for Parabolic flow used in MLG. Four plots are present including time t=0. There are 190 triangles and 108 points present in each plot frame.

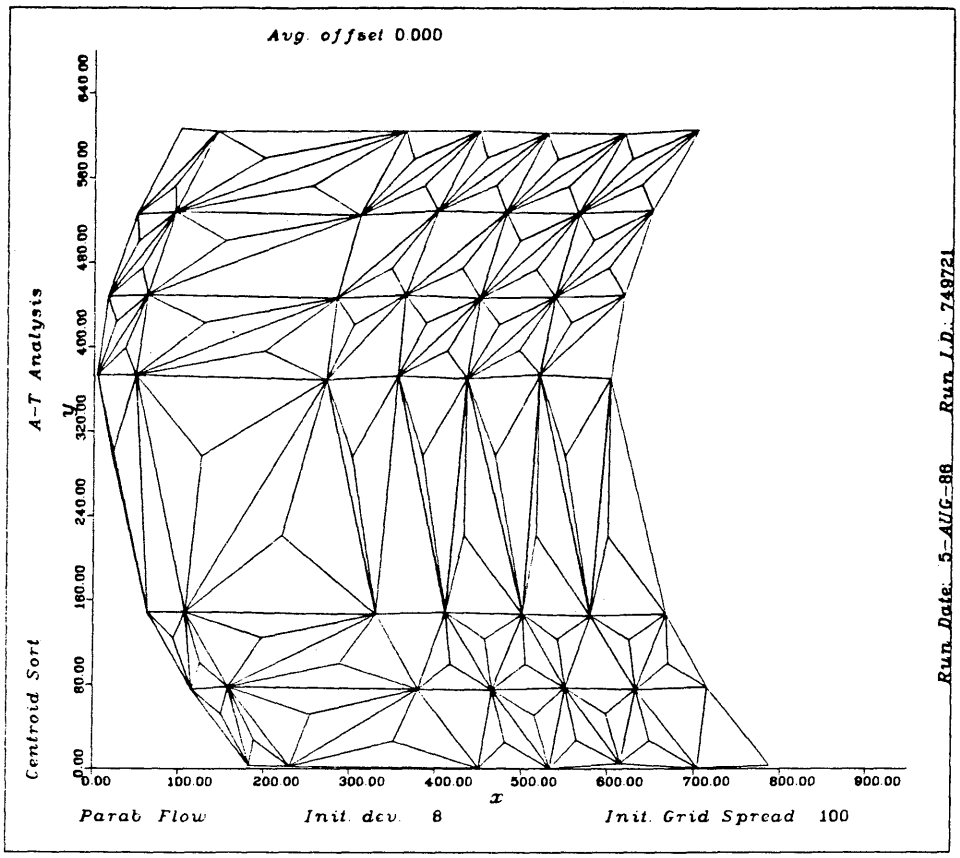


OBS. SPACE t = 1


Avg. offset 0.000
A-T Analysis
Centroid Sort
Run Date: 5-AUG-86 Run I.D.: 749721
0.00
80.00
160.00
240.00
320.00
400.00
480.00
560.00
640.00
720.00
y
x
Parab Flow
Init dev. 8
Init Grid Spread 100

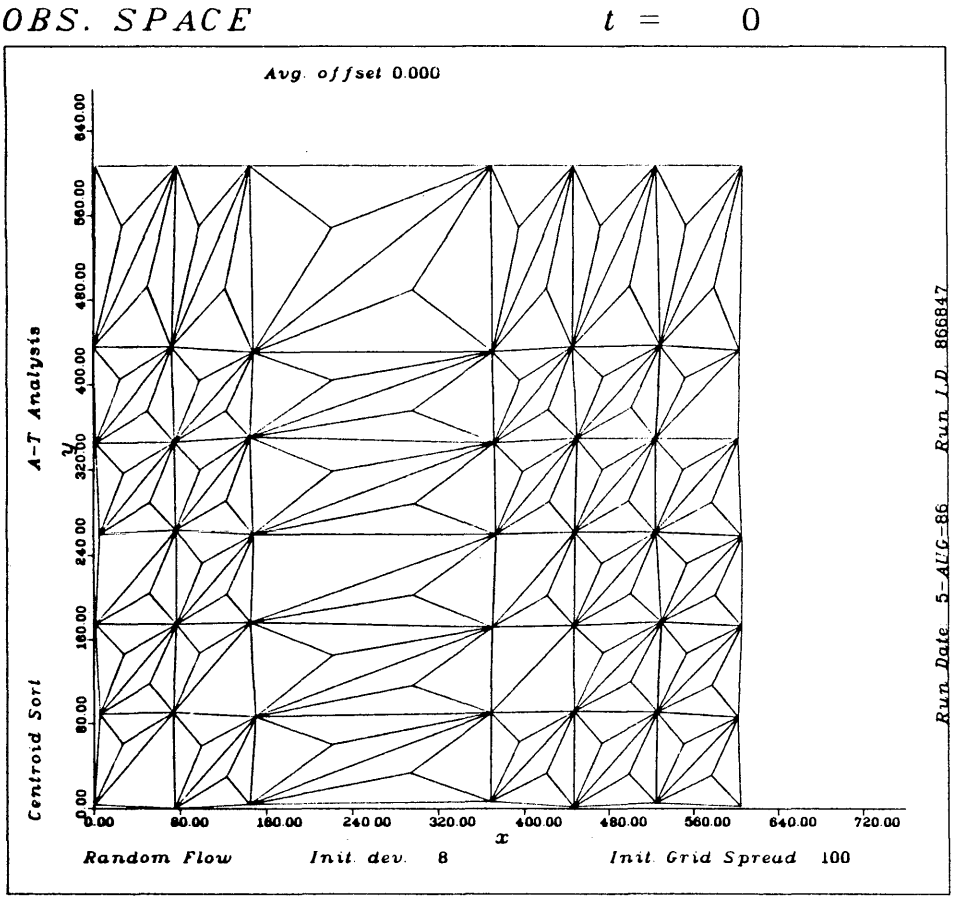

OBS. SPACE
t = 2
Avg offset 0.000
A-T Analysis
Centroid Sort
y
x
0.00
80.00
160.00
240.00
320.00
400.00
480.00
560.00
640.00
0.00
100.00
200.00
300.00
400.00
500.00
600.00
700.00
800.00
900.00
Parab Flow
Init dev 8
Init Grid Spread 100
Run Date 5-AUG-88
Run I.D. 749721

OBS. SPACE $t =$ 3

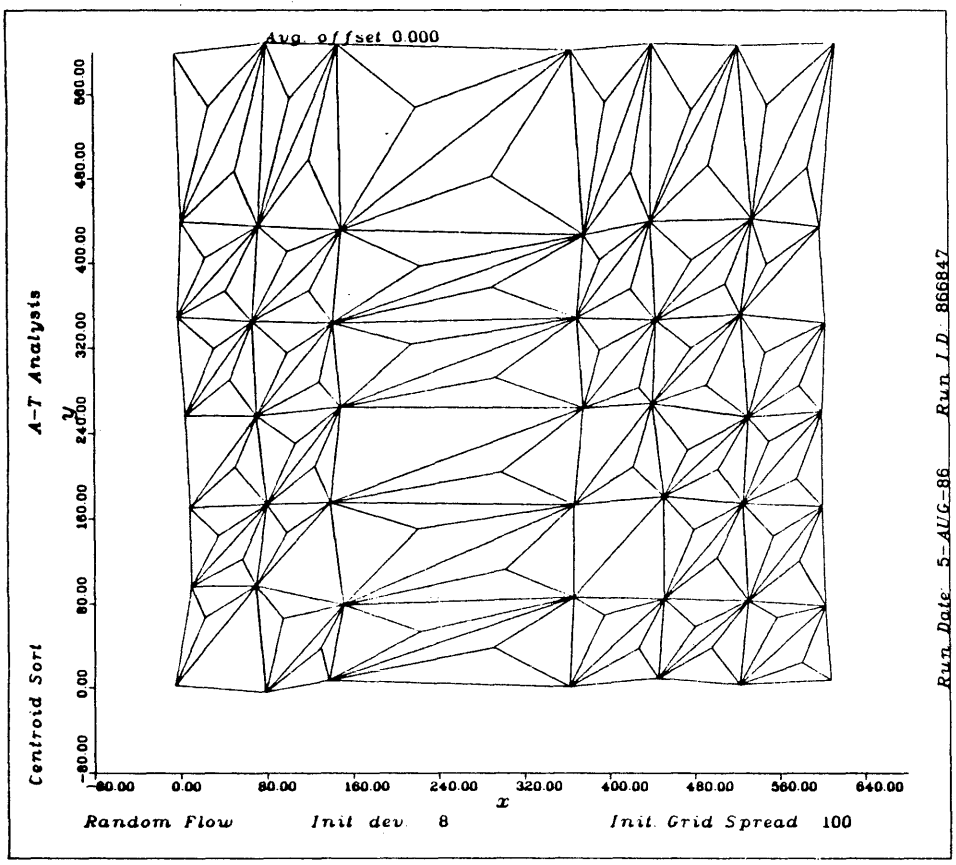

Avg. offset 0.000
A-T Analysis
Centroid Sort
y
0.00
80.00
160.00
240.00
320.00
400.00
480.00
560.00
640.00
0.00
100.00
200.00
300.00
400.00
500.00
600.00
700.00
800.00
900.00
x
Parab Flow
Init. dev. 8
Init. Grid Spread 100
Run Date: 5-AUG-88
Run I.D.: 749721

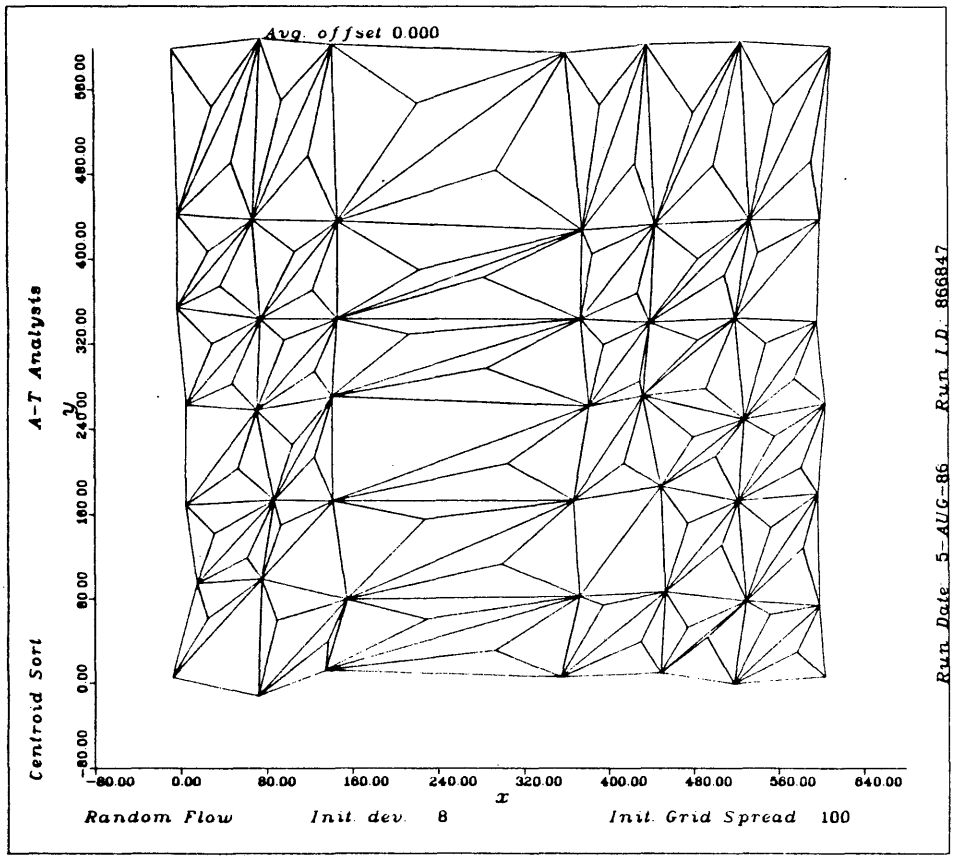
SUBSECTION B.9-B.12 - RANDOM FLOW

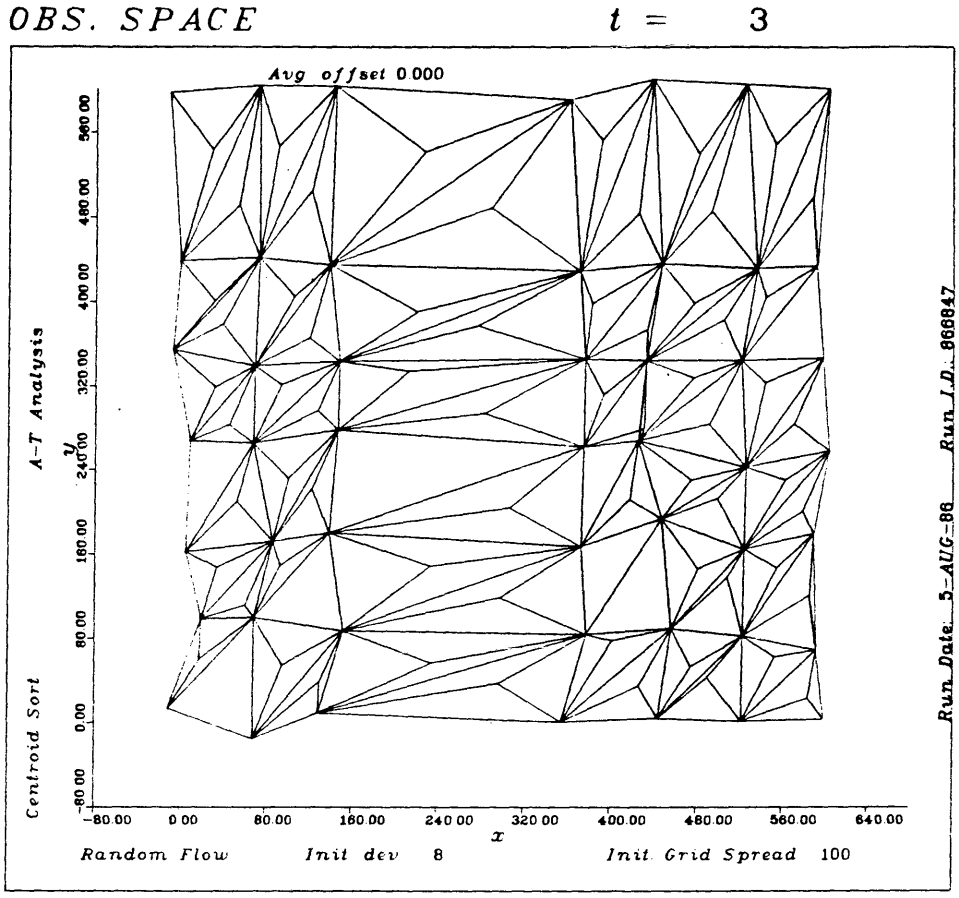
Following are computer generated plots for Random flow used in MLG. Four plots are present including time t=0. There are 200 triangles and 113 points present in each plot frame.



OBS. SPACE t = 1


Avg. offset 0.000
A-T Analysis
Centroid Sort
y
x
-80.00
0.00
80.00
160.00
240.00
320.00
400.00
480.00
560.00
640.00
Random Flow
Init dev 8
Init. Grid Spread 100
Run Date: 5-AUG-86
Run I.D. 866847


OBS. SPACE
t = 2
Avg. offset 0.000
A-T Analysis
Centroid Sort
Run Date 5-AUG-86
Run I.D. 866847
y
x
-80.00
0.00
80.00
160.00
240.00
320.00
400.00
480.00
560.00
640.00
Random Flow
Init. dev. 8
Init. Grid Spread 100


OBS. SPACE
t = 3
Avg offset 0.000
A-T Analysis
Centroid Sort
Run Date 5-AUG-86
Run I.D. 866847
-80.00
0.00
80.00
160.00
240.00
320.00
400.00
480.00
560.00
640.00
y
x
Random Flow
Init dev 8
Init Grid Spread 100

APPENDIX C

This appendix contains pseudo-code for three subroutines of the driver program MLG. The appendix is divided into three subsections which are:

SUBSECTION #	SUBROUTINE PSEUDO-CODE
C.1	MLGSORT
C.2	ADJSRCH
C.3	FINDADJ

SUBSECTION C.1 - PSEUDO-CODE FOR MLGSORT

Following is the pseudo-code for the subroutine MLGSORT which is called upon by the driver program MLG each timestep in order to sort the MLG into MLO. The pseudo-code consists of three sections corresponding to the three directional vector sweeps that must be performed.

```
    Program MLGSORT

--- section 1 ---
--- this section is done for all models being used ---

---loop through all i directional vectors ---
    k = 1
    do while k ≠ MLG dimension in k
         j = 1
         do while j ≠ MLG dimension in j
              l = 1
              do while l ≠ 2
                   i = 1
                   do while i ≠ MLG (dimension in i) - 1
                        if ( x coordinates of MLG array
                             elements at indexes (i,j,k) &
                         (i+1,j,k) are out of order ) then
                             switch MLG cell contents
```

```
                        end if
                        i = i + 1
                    end do  (i)
                    l = l + 2
                end do  (l)
                j = j + 1
            end do  (j)
            k = k + 1
        end do (k)
--- check to see if any swaps were made ---
        if ( model is one dimensional and no MLG cell switches
            are made ) then
            done
          else if ( model is one dimensional and at least one
                    switch was made ) then
                    go back to section 1, k loop
                    and re-execute looping
        end if

--- section 2 ---
--- this section is done if the model is at least 2-D ---

--- loop through all j directional vectors ---
        k = 1
        do while k ≠ MLG dimension in k
            i = 1
```

```
          do while i ≠ MLG dimension in i

               l = 1

               do while l ≠ 2

                    j = 1

                    do while j ≠ MLG (dimension in j) - 1

                         if ( y coordinates of MLG array
                              elements at indexes  (i,j,k)
                                        &
                           (i,j+1,k)  are  out of  order  )

                         then

                              switch MLG cell contents

                         end if

                         j = j + 1

                    end do  (j)

                    l = l + 2

               end do  (l)

               i = i + 1

          end do  (i)

          k = k + 1

     end do (k)

--- check to see any swaps were made ---

     if ( model is two dimensional and no MLG cell switches
          were made ) then

          done

       else if ( model is two dimensional and at least one
                 switch was made ) then
```

```
                go back to section 1, k loop and re-
               execute looping

    end if

--- section 3 ---
--- this section is reached only if model is 3-d ---

--- loop through all k directional vectors ---

    i = 1

    do while i ≠ MLG dimension in i

         j = 1

         do while j ≠ MLG dimension in j

              l = 1

              do while l ≠ 2

                   k = 1

                   do while k ≠ MLG (dimension in k) - 1

                        if ( z coordinates of MLG array
                             elements at indexes  (i,j,k)
                                        &
                           (i,j,k+1)  are  out of order  )

                          then

                              switch MLG cell contents

                        end if

                        k = k + 1

                   end do  (k)

                   l = l + 2

              end do  (l)
```

```
            j = j + 1

        end do  (j)

        i = i + 1

    end do (i)

   --- check to see if any swaps were made ---

    if  (  model  is three dimensional  and  no  MLG  cell
           switches were made ) then

        done

      else

        go  back  to  section 1,  k loop  and  re-execute
looping

    end if

    Program end  (MLGSORT)
```

SUBSECTION C.2 - PSEUDO-CODE FOR ADJSRCH

The following pseudo-code represents the subroutine ADJSRCH which is called upon by the driver program MLG. This subroutine searches the MLG for adjacent triangles. ADJSRCH steps through each triangle of the MLG, all the while determining the adjacent triangles for each of the triangles it encounters. ADJSRCH calls upon the subroutine FINDADJ to determine if triangles which it finds are truely adjacent to the triangle that it is presently working on. When hidden interior triangles are encountered, the MLG inedex of the triangle adjacent to the hidden interior triangle is recorded (in FINDADJ) and then reprocessed in section 2.

```
    Program ADJSRCH

--- section 1 ---
--- looping through MLG in a sequential pattern ---
--- (all j directional vectors) ---

--- j is the inner most loop therefore producing ---
--- update  sequences  sweeping  through  j ---
--- directional vectors ---

    TRICOUNT = 0

    k = 1

    do while k ≠ MLG dimension in k
```

```
          i = 1

          do while i ≠ MLG dimension in i

               j = 1

               do while j ≠ MLG dimension in j

                    TRICOUNT = TRICOUNT + 1

          --- don't process "ghost" triangles ---

                    if ( TRICOUNT ≠ NUMTRI ) then

          --- find adjacent triangles to triangle
              ALPHA(i , j , k) = T_m using FINDADJ.
              FINDADJ will keep track of corner
              triangles for later reprocessing
              (section 2) ---

                         call FINDADJ ( T_m )

                    end if

                    j = j + 1

               end do  (j)

               i  = i + 1

          end do  (i)

          j  = j + 1

     end do  (j)

--- section 2 ---

--- Reprocess hidden interior triangles. All hidden   ---
--- triangles and their MLG indices were recorded     ---
--- previously in FINDADJ.                            ---


     do while ( there are still corner triangles to be
                  reprocessed )
```

```
    --- find the adjacent triangle to the current    ---
    --- corner triangle. This will be a hidden       ---
    --- interior triangle                            ---

        call FINDADJ ( T_m )

    --- now reprocess the triangle using 3 as the    ---
    --- number of adjacent triangles to look for     ---

        call FINDADJ ( T_m )

end do  (more corner triangles)


Program end  (ADJSRCH)
```

SUBSECTION C.3 - PSEUDO-CODE FOR FINDADJ

This is the final subsection of appendix C. It contains the pseudo-code for FINDADJ. This subroutine calculates the number of adjacent triangles to search for and records the MLG indices for corner triangles to be used by both ADJSRCH and PNTSRCH in reprocessing hidden interior triangles.

```
    Program FINDADJ

    NUMBER_CORNERS = 0

--- this series of conditionals involve determining ---
--- if the general triangle is corner, border or    ---
--- interior. Summations are used (not given here)  ---
--- to determine trh triangle position              ---

    if ( triangle T_m is an interior triangle ) then

         NUMBER_ADJ = 3

      else if ( triangle T_m is a border triangle ) then

                NUMBER_ADJ = 2

             else

    --- triangle must be a corner triangle ---

                NUMBER_ADJ = 1

    --- record MLG indices for future processing ---

                store MLG indices of triangle

                NUMBER_CORNERS = NUMBER_CORNERS + 1

           end if
```

```
     end if

 --- reset NUMBER_ADJ to 3 if we are reprocessing ---
 --- hidden interior triangles                    ---

     if ( triangle T_m is a hidden interior triangle ) then

          NUMBER_ADJ = 3

     end if

 --- start with index offset of one for the search ---

     TOPROW = min ( (j index of T_m) + 1, MLG dim. in j )

     BOTROW = max ( (j index of T_m) - 1, 1 )

     RIGCOL = min ( (i index of T_m) + 1, MLG dim. in i )

     LEFCOL = max ( (i index of T_m) - 1, 1 )

     FOUND_COUNT = 0

 --- start searching the offset for adjacent triangles ---

     do while FOUND_COUNT ≠ NUMBER_ADJ

          ROW = BOTROW

          do while ROW ≠ TOPROW

               COL = LEFCOL

               do while COL ≠ RIGCOL

                    call ADJCONF

                    if  (  these triangles are  adjacent  )
                      then

                         FOUND_COUNT = FOUND_COUNT + 1

                         store triangle I.D. of triangle
                         found
```

```
                    end if
                    COL = COL + 1
                end do (COL)
                ROW = ROW + 1
            end do (ROW)
--- increase the offset by one for the next offset search
    (if the next offset is needed)                    ---
        TOPROW = min ( TOPROW + 1 , MLG dimension in j )
        BOTROW = max ( BOTROW - 1 , 1 )
        RIGCOL = min ( RIGCOL + 1 , MLG dimension in i )
        LEFCOL = max ( LEFCOL - 1 , 1 )
    end do (FOUND_COUNT ≠ NUMBER_ADJ)
    Program end  (FINDADJ)
```

APPENDIX D

This appendix is subdivided into 5 subsections. Presented in each subsection is the FORTRAN code for the various subroutines developed in chapter 5.

All code in this study was written on a VAX 8600 machine. Coding was carefully done so as to make it as portable as possible. Complete portability of any program, however, is harder to acheive as the complexity of algorithms increase. Segments of code which are not portable will be noted if they occur. Also, most of the code for compiling statistics on performance have been edited out of these subroutines in order to reduce the size of the code printed out. Most of the code which compiles the results is simple counting and summing code, while timing code is VAX dependent.

The subsections are as follows:

SUBSECTION #	SUBROUTINE
D.1	MLGSORT
D.2	ADJSRCH
D.3	FINDADJ
D.4	ADJCONF
D.5	PNTSRCH

SUBSECTION D.1 - CODE FOR MLGSORT

Following is the code for the subroutine MLGSORT. The code has incorporated with it the capability of expansion to three dimensions, although the models being used in this study only deal with two dimensions. As can be seen, all looping is coded to handle sorting in all three dimensions. When the expansion of model dimensionality is completed, this subroutine will be readily executably as is.

```
      subroutine MLGSORT

* Sorting of resultant movements from previous movement
* subroutines to obtain monotonic logical order.

* This sort incorporates a RED-BLACK algorithm in order
* to vectorize the code. This subroutine is used
* no matter which attribute the user chooses to
* characterize the triangle with.

* In MLGSORT, the grid is "sorted" first in the x
* direction,
* secondly in the y direction, and then thirdly in the z
* direction. Each "sort" consists of only interchanging
* consecutive array elements (not a complete sort). After
* each full sweep through all directions, it is checked to
* see if a swap was made. If a swap occured, the sweep
* must be re-executed.

* Determination of number of exchanges to be made

        NUMEXCHG=NGRIDY*NGRIDZ*(NGRIDX-1)
     :  +NGRIDX*NGRIDY*(NGRIDZ-1)+NGRIDX*NGRIDZ*(NGRIDY-1)

158   FLAG=0

*==== SECTION 1 ====
```

```
*          Sweep through all vectors in first direction
*          This applies to 1,2, or 3 dimensional models


      do 210 k=1,NGRIDZ

        do 200 j=1,NGRIDY

          do 180,1=1,2

            do 175 i=1,NGRIDX-1,2

*  numerical determination if cells are out of order.
*  FLAG is increased by one every time MLG cells are
*  in order.

              DIFF=ALPHA(i+1,j,k,4)-ALPHA(i,j,k,4)
              W=sign(0.5,DIFF) + 0.5
              FLAG=FLAG+W
              COMP=1-W

*  swap MLG cell contents if they are out of order

              do 160 n=1,NUMPAR
                TEMP1=W*ALPHA(i,j,k,n)
                TEMP2=COMP*ALPHA(i,j,k,n)
                ALPHA(i,j,k,n)=TEMP1+COMP*ALPHA(i+1,j,k,n)
                ALPHA(i+1,j,k,n)=W*ALPHA(i+1,j,k,n)+TEMP2
160           continue

175         continue

180       continue

200     continue

210   continue

* If it is a one dimensional model, make sure grid is
* sorted. If the grid is not sorted, repeat the sweep,
* else
* return to calling program. If it is a two dimensional
* model, sort in next direction, (direction of NGRIDY)
```

```
       if(NDIM.eq.1.and.FLAG.eq.NUMEXCHG) then
         return
        else
         if(NDIM.eq.1.and.FLAG.lt.NUMEXCHG) then
           go to 158
         end if
       end if


*==== SECTION 2 ====

*            If model is 2-D then sweep through
*            all vectors of the second dimension

      do 410 k=1,NGRIDZ

        do 400 i=1,NGRIDX

          do 360 l=1,2

            do 350 j=1,NGRIDY-1,2

*  numerical determination if cells are out of order.
*  FLAG is increased by one every time MLG cells are
*  in order.

              DIFF=ALPHA(i,j+1,k,5)-ALPHA(i,j,k,5)
              W=sign(0.5,DIFF) + 0.5
              FLAG=FLAG+W
              COMP=1-W
              SWAPCNT(TIME)=SWAPCNT(TIME)+COMP

*  swap MLG cell contents if they are out of order

              do 345 n=1,NUMPAR
                TEMP1=W*ALPHA(i,j,k,n)
                TEMP2=COMP*ALPHA(i,j,k,n)
                ALPHA(i,j,k,n)=TEMP1+COMP*ALPHA(i,j+1,k,n)
                ALPHA(i,j+1,k,n)=W*ALPHA(i,j+1,k,n)+TEMP2
345           continue

350         continue

360       continue

400     continue
```

```
410   continue

* If it is a two dimensional model, make sure grid is
* sorted. If the grid is not sorted, repeat the sweep,
* else
* return to calling program. If it is a three dimensional
* model, sort in next direction.

      if(NDIM.eq.2.and.FLAG.eq.NUMEXCHG) then
        return
       else
        if(NDIM.eq.2.and.FLAG.lt.NUMEXCHG) then
          go to 158
       end if
      end if


*==== SECTION 3 ====

*          If model is 3-D then sweep through
*          all vectors in the third direction

      do 440 i=1,NGRIDX

        do 430 j=1,NGRIDY

          do 425 l=1,2

            do 420 k=1,NGRIDZ-1,2

*  numerical determination if cells are out of order.
*  FLAG is increased by one every time MLG cells are
*  in order.

              DIFF=ALPHA(i,j,k+1,6)-ALPHA(i,j,k,6)
              W=sign(0.5,DIFF) + 0.5
              FLAG=FLAG+W
              COMP=1-W
              SWAPCNT(TIME)=SWAPCNT(TIME)+COMP

*  swap MLG cell contents if they are out of order

              do 415 n=1,NUMPAR
                TEMP1=W*ALPHA(i,j,k,n)
                TEMP2=COMP*ALPHA(i,j,k,n)
                ALPHA(i,j,k,n)=TEMP1+COMP*ALPHA(i,j,k+1,n)
```

```
                ALPHA(i,j,k+1,n)=W*ALPHA(i,j,k+1,n)+TEMP2
415           continue

420         continue

425       continue

430     continue

440   continue


* if a swap was made in either i,j, or k dir. then the
* grid must be re-checked. (i.e. loop to the top of the
*  sort process)

      if(FLAG.lt.NUMEXCHG) then
        go to 158
      end if

      return
      end
```

SUBSECTION D.2 - CODE FOR ADJSRCH

The following code is the subroutine ADJSRCH. The code will run strictly with two dimensional models. The extension to three dimensions will cause much added code and code complexity.

```
      subroutine ADJSRCH

* This subroutine searches the MLG to find adjacent
* triangles for all triangles of the space.

       logical NUMFLG,CORFLG
       integer CURLIST(3,4)

       NUMFLG=.true.       !calculate # adj tri's
       CORFLG=.true.       !calculate all corner tri's
       TRICNT=0

       do 300 k=1,NGRIDZ

         do 200 i=1,NGRIDX

           do 100 j=1,NGRIDY

             TRICNT=TRICNT+1
             if(TRICNT.le.NUMTRI) then

* call to FINADJ in order to find adjacent triangles of
* the triangle at i,j and k indices of the MLG

               call
     :           FINDADJ(i,j,k,NADJ,CURLIST,NUMFLG,CORFLG)

             end if

100        continue

200      continue

300    continue
```

```
* === section 2 ===
* reprocess hidden interior triangles

      do 400 l=1,NUMCOR

        i=CORNER(l,2)    !corner is a common block array
        j=CORNER(l,3)    !containing corner triangle info.
        k=CORNER(l,4)    !CORNER is filled in FINDADJ.
        NUMFLG=.false.
        CORFLG=.false.
        NADJ=1

* find the adjacent triangle to a corner triangle

        call FINDADJ(i,j,k,NADJ,CURLIST,NUMFLG,CORFLG)

        i=CURLIST(NADJ,2)
        j=CURLIST(NADJ,3)
        k=CURLIST(NADJ,4)

        NUMFLG=.false.
        CORFLG=.false.    !# of adj. set to 3 for hidden
        NADJ=3            !interior triangles.

* find all three adjacent triangles to the hidden interior
* triangle

        call FINDADJ(i,j,k,NADJ,CURLIST,NUMFLG,CORFLG)

400   continue

      return
      end
```

SUBSECTION D.3 - CODE FOR FINDADJ

The code for FINDADJ follows and is strictly a two dimensional code. In order to upgrade this subroutine to three dimensions, a good deal of work would be required. However, in two dimensional models this code is very efficient in determining triangle position with respect to the grid boundaries and in actually obtaining the triangle I.D.'s of the adjacent triangles.

```
      subroutine FINDADJ(I,J,K,N,CUR,NFLG,CFLG)

* This subroutine finds all adjacent tri's to the current
* triangle. The first step is to calculate the number of
* adjacent triangles to look for and then to look until
* the  number of triangles found equals the analytical
* number calculated.

      logical LOG1,LOG2,LOG3,YORN,NFLG,CFLG
      integer CUR(3,4),ICNT,BCNT,CCNT,LEFTCOL
      integer  BOTROW,RIGHTCOL,TOPROW

* summation of point position counters if NFLG in calling
* program is set to true then the number of adj triangles
* will be calculated. If CFLG is set to true in the
* calling program the corner triangles will be
*  accumulated.

      if(NFLG) then   !calculate the number of adj tri

        ICNT=0
        BCNT=0
        CCNT=0

        do 100 l=1,3

          ICNT=ICNT+PTSTAT(ALPHA(I,J,K,l),1) !sum int.
          BCNT=BCNT+PTSTAT(ALPHA(I,J,K,l),2)  !bor. &
```

```
            CCNT=CCNT+PTSTAT(ALPHA(I,J,K,1),3)  !cor. pnts.

100       continue

* determination of triangle position (border, interior, or
*  corner)

          LOG1=(ICNT.eq.3.or.(ICNT.eq.2.and.BCNT.eq.1))
          LOG2=(CCNT.eq.1.and.BCNT.eq.2)
          LOG3=(BCNT.eq.2.and.ICNT.eq.1)

          if(LOG1) then   !interior tri

            N=3

           else

            if(LOG2) then   !corner tri
              N=1

*  check to see if current triangle is in the list of
*   corner triangles.

               if(CFLG) then
                 NM=0
                 do 200 l=1,NUMCOR
                   if(int(ALPHA(I,J,K,NUMPAR)).eq.CORNER(1,1))
     :               NM=1
200              continue

                 if(NM.eq.0) then  !cor. tri not in list
                   NUMCOR=NUMCOR+1
                   CORNER(NUMCOR,1)=int(ALPHA(I,J,K,NUMPAR))
                   CORNER(NUMCOR,2)=I
                   CORNER(NUMCOR,3)=J
                   CORNER(NUMCOR,4)=K
                 end if

               end if

            else

               N=2   ! border triangle

             end if  !(end of corner tri)!

          end if  !(end of interior tri)!
```

```
      end if  !(end of calculation of # adj tri)!


* initialize the first index offset

      TOPROW=min0(J+1,NGRIDY)
      BOTROW=max0(J-1,1)
      LEFTCOL=max0(I-1,1)
      RIGHTCOL=min0(I+1,NGRIDX)

      do 500  l=1,max0(NGRIDX-1,NGRIDY-1,NGRIDZ-1)

***     searching the offset for adjacent triangles

        do 400 COL=LEFTCOL,RIGHTCOL

          do 300 ROW=BOTROW,TOPROW

            BOUND=ALPHA(COL,ROW,K,NUMPAR)
            if((ROW.ne.J.or.COL.ne.I).and.
     :           BOUND.le.NUMTRI) then

              call ADJCONF(I,J,K,YORN) !confirm sub.

             if(YORN) then
               BORDCNT=BORDCNT+1
               CUR(BORDCNT,1)=int(ALPHA(COL,ROW,K,NUMPAR))
               CUR(BORDCNT,2)=COL
               CUR(BORDCNT,3)=ROW
               CUR(BORDCNT,4)=K
              end if

            end if

300       continue

400     continue

***     check to see if search should continue

        if(BORDCNT.eq.N) then
          return
        end if

***     adjust search boundaries for next offset
```

```
          TOPROW=min0(TOPROW+1,NGRIDY)
          BOTROW=max0(BOTROW-1,1)
          LEFTCOL=max0(LEFTCOL-1,1)
          RIGHTCOL=min0(RIGHTCOL+1,NGRIDX)

500       continue

          end
```

NOTE: The matrix PTSTAT is defined to be a BYTE integer data type which is an 8 bit representation. This variable declaration is VAX dependent and is not standard to fortran.

SUBSECTION D.4 - CODE FOR ADJCONF

This code simply confirms if two triangles are adjacent to one another. The MLG indices corresponding to the triangle being tested are passed as arguments while the indicies of the general triangle are passed in a common block. The logical variable YORN is set to true if the two triangle are adjacent or set to false otherwise.

```
      subroutine ADJCONF(i,j,k,YORN)

* This is a subroutine to determine if two triangles are
* adjacent. The MLG indices of the general triangle are
* passed using a common block while the indices of the
* triangle being tested are passed as aguments.

      logical YORN
      integer i,j,k

* looping through vertices of the triangle to find the
* first pair of vertices that match

      YORN=.false.
      do 1000 l=1,3

        do 900 m=1,3

          if(ALPHA(COL,ROW,k,m).eq.ALPHA(i,j,k,l)) then

* a pair of vertices were found so look for one more in
*  the vertices that remain.

            do 800 n=l+1,3

             do 700 nl=1,3

               if(ALPHA(COL,ROW,k,nl)
     :           .eq.ALPHA(i,j,k,n)) then
                YORN=.true.                !YORN is true
                 return                     !if adjacent
```

```
                end if
 700            continue
 800          continue
            end if
 900      continue
 1000   continue
        return
        end
```

SUBSECTION D.5 - CODE FOR PNTSRCH

The following code finds all triangles surrounding a given point.The code is separated into 2 sections. Section 1 processes information in the normal fashion while section 2 reprocesses hidden interior triangles and the points which are its vertices. The coding for section 2 will not be given since it resembles the code of section 1. The important issue is the code which determines all surrounding triangles.

```
      subroutine PNTSRCH

* This subroutine determines all triangles which surround
* the points in the space. This subroutine is comprised of
* two parts. The first part processes all points in the
* space in the same manner. The second part processes all
* points that are contained in hidden interior triangles.
* A hidden interior triangle is a triangle which is
* adjacent to a corner triangle and which is classified as
* a border triangle when in fact it is an interior
* triangle with three adjacent triangles.

      integer CURLIST(3,4),POINT,TRICNT,ORIGID,CURTRI
      integer TRIID,SUM,TOPROW,BOTROW,LEFTCOL,RIGHTCOL
      integer USEDLIST(50)
      logical NUMFLG,CORFLG

*** zeroing of point processed flag

      do 100 i=1,NUMPTS
        PT(i,4)=0
100   continue
```

```
*=== SECTION 1 ===

*** start of search loop

      NUMCOR=0
      CORFLG=.true.          !calculate # adj tri's
      NUMFLG=.true.          !and corner tri's in this
      TRICNT=0               !section

* looping through the MLG indices

      do 2200 k=1,NGRIDZ

        do 2100 i=1,NGRIDX

          do 2000 j=1,NGRIDY

            TRICNT=TRICNT+1
            if(TRICNT.le.NUMTRI) then

              ORIGID=int(ALPHA(i,j,k,NUMPAR))

              do 1900 l=1,3

              POINT=int(ALPHA(i,j,k,l))
              if(PT(POINT,4).eq.0) then    !POINT not
                PT(POINT,4)=1.0      !processed yet
                NUMUSED=0

*** diff. process is to be done if POINT is a border
*** point. Must make sure to enter a border point by a
*** border or corner triangle triangle.

                  if(PTSTAT(POINT,2).eq.1) then

                    SUM=0
                    do 400 mn=1,3

                      if(SUM.lt.2) then

***       set initial expansion borders

                        TOPROW=min0(j+1,NGRIDY)
                        BOTROW=max0(j-1,1)
                        LEFTCOL=max0(i-1,1)
                        RIGHTCOL=min0(i+1,NGRIDX)
```

```
*** searching the expansion for first border
*** triangles

                        do 900 mm=1,
     :                   max0(NGRIDX-1,NGRIDY-1,NGRIDZ-1)

                          do 800 COL=LEFTCOL,RIGHTCOL

                            do 700 ROW=BOTROW,TOPROW

                              BOUND=ALPHA(COL,ROW,k,NUMPAR)

                              if((ROW.ne.j.or.COL.ne.i).and.
     :                               BOUND.le.NUMTRI) then
                                do 600 m=1,3
                                  if(TRI(ALPHA(COL,ROW,k,
     :                                 NUMPAR),m).eq.POINT)
     :                              then

* sum up PTSTAT values for triangle vertices to determine
* triangle type

                                    SUM=0
                                    do 500 mn=1,3
                                      SUM=SUM +
     :                                 PTSTAT(ALPHA(COL,
     :                                  ROW,k,mn),2)
 500                                  continue

                                    if(SUM.eq.2) then
                                      IFIND=COL
                                      JFIND=ROW
                                      KFIND=k
                                      CURTRI=int(ALPHA(COL,
     :                                         ROW,k,NUMPAR))
                                      goto 10
                                    end if   !(border tri)

                                  end if   !(same POINT)

 600                              continue

                                end if   !(not ORIGID tri)
```

```
 700                      continue   !(loop on ROW)

 800                      continue   !(loop on COL)

 ***     increase expansion if no border triangle found

                          TOPROW=min0(TOPROW+1,NGRIDY)
                          BOTROW=max0(BOTROW-1,1)
                          LEFTCOL=max0(LEFTCOL-1,1)
                          RIGHTCOL=min0(RIGHTCOL+1,NGRIDX)

 900                    continue  !(loop on expansions)

                       else  !originating triangle is bord

                         IFIND=i
                         JFIND=j
                         KFIND=k
                         CURTRI=ORIGID

                       end if

                      else  !POINT is not border point

                            IFIND=i
                            JFIND=j
                            KFIND=k
                            CURTRI=ORIGID

                     end if !(end POINT id. .bord,inter*)

* initialize the current triangle list

 10                   do 1100 ll=1,3
                        do 1000 mm=1,4
                          CURLIST(ll,mm)=0
 1000                   continue
 1100                 continue

* find all adjacent triangles to the current triangle

                      call  FINDADJ(IFIND,JFIND,KFIND,
     :                       NADJ,CURLIST,NUMFLG,CORFLG)

* illiminate any adjacent tri.'s that are in USEDLIST

                    do 1400 ll=1,3
```

```
                    do 1300 mm=1,NUMUSED
                      if(CURLIST(11,1).eq.
     :                     USEDLIST(mm)) then
                        do 1200 nn=1,4
                          CURLIST(11,nn)=0
 1200                     continue
                       end if
 1300                  continue
 1400                continue

 * find triangle with side in "common" with CURTRI
 * if no triangles are found then search is complete

                    do 1800 11=1,NADJ
                      TRIID=CURLIST(11,1)
                      if(TRIID.ne.0) then
                        if(TRI(TRIID,mm).eq.POINT) then
                          IFIND=CURLIST(11,2)
                          JFIND=CURLIST(11,3)
                          KFIND=CURLIST(11,4)

                          USEDLIST(NUMUSED+1)=CURTRI
                          CURTRI=TRIID
                          NUMUSED=NUMUSED+1

                          goto 10
                        end if

                      end if

 1800               continue

               end if   !(end processing POINT)

 1900           continue   !(end pro. points in ORIGID tri)

            end if   !(end processing tri's in MLG)

 2000     continue   !(end j looping)

 2100    continue   !(end i looping)

 2200   continue   !(end k looping)
```

```
*=== SECTION 2 ===
*     code not to be given
*=================
```

NOTE: Section 2 code resembles the code of the first section except for the fact that a different subset of the MLG triangles are being processed, the hidden interior triangles.

APPENDIX E

In this appendix computer generated output for 4 executions of MLG.for are given. They are divided into 4 subsections and are as follows:

SUBSECTION	DATA
E.1	Random flow data
E.2	Parabolic flow data
E.3	CPU time data
E.4	CPU time data

SUBSECTION E.1

Computer generated data for Parabolic Flow over 8 timesteps is given on the following page. Data includes swap iteration counts for the sorting algorithm, maximum index offsets, mean index offsets and variances about these means.

==========
Partition Analysis
==========

Randomly disturb. init. grid
Parabolic Flow
Centroid Determined Partition
Point-to-Triangle Analysis
Large triangle initial grid
Initial point grid dimensions 20 X 20 X 1
Initial MLG dimensions 31 X 31 X 1

Initial # of grid points = 505
Initial # of triangles = 932

---

	time	0	1	2	3	4	5	6	7	8
Number of grid points present	=	505	505	505	505	505	505	505	505	505
Number of triangles present	=	932	932	932	932	932	932	932	932	932
Number of swaps to correct MLG	=	10024	439	289	319	236	212	403	369	227
Number of swap iterations to correct MLG	=	19	8	9	10	9	8	15	14	11
Avg. # of tri. searched per point	=	20.22	18.79	21.41	29.02	36.69	46.48	56.08	58.26	70.24
Maximum i directional offset	=	2	3	4	6	6	7	9	9	10
Number of points at max. i offset	=	21	5	1	2	2	4	1	4	8
% of points at max. i offset	=	0.042	0.010	0.002	0.004	0.004	0.008	0.002	0.008	0.016
Maximum j directional offset	=	4	4	4	4	4	3	3	3	3
Number of points at max. j offset	=	2	2	2	2	2	2	5	2	2
% of points at max. j offset	=	0.004	0.004	0.004	0.004	0.004	0.004	0.010	0.004	0.004
Maximum k directional offset	=	0	0	0	0	0	0	0	0	0
Number of points at max. k offset	=	505	505	505	505	505	505	505	505	505
% of points at max. k offset	=	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Mean i directional offset	=	1.042	1.085	1.170	1.299	1.451	1.566	1.774	1.788	2.032
Variance about the mean	=	0.040	0.098	0.241	0.591	0.899	1.448	2.159	2.330	3.364
Mean j directional offset	=	1.069	1.046	1.048	1.042	1.042	1.038	1.059	1.044	1.059
Variance about the mean	=	0.108	0.075	0.081	0.064	0.064	0.044	0.076	0.050	0.064
Mean k directional offset	=	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Variance about the mean	=	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

SUBSECTION E.2

Computer generated data for Random flow over 8 timesteps is given. Data includes swap iteration counts for the sorting algorithm, maximum index offsets, mean index offsets and variances about these means.

==========
Partition Analysis
==========

Randomly disturb. init. grid
Random Flow
Centroid Determined Partition
Point-to-Triangle Analysis
Large triangle initial grid
Initial point grid dimensions 20 X 20 X 1
Initial MLG dimensions 31 X 31 X 1

Initial # of grid points = 505
Initial # of triangles = 932

---

	time	0	1	2	3	4	5	6	7	8
Number of grid points present	=	505	505	505	505	505	505	505	505	505
Number of triangles present	=	932	932	932	932	932	932	932	932	932
Number of swaps to correct MLG	=	10187	106	26	30	27	29	30	16	8
Number of swap iterations to correct MLG	=	20	5	3	4	4	4	3	3	2
Avg. # of tri. searched per point	=	18.87	17.62	17.59	17.41	17.09	17.01	16.76	16.64	16.48
Maximum i directional offset	=	2	2	2	2	2	2	2	2	2
Number of points at max. i offset	=	21	28	28	25	26	28	22	22	21
% of points at max. i offset	=	0.042	0.055	0.055	0.050	0.051	0.055	0.044	0.044	0.042
Maximum j directional offset	=	4	4	4	4	4	4	4	4	4
Number of points at max. j offset	=	2	2	2	2	2	2	2	2	2
% of points at max. j offset	=	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004
Maximum k directional offset	=	0	0	0	0	0	0	0	0	0
Number of points at max. k offset	=	505	505	505	505	505	505	505	505	505
% of points at max. k offset	=	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Mean i directional offset	=	1.042	1.055	1.055	1.050	1.051	1.055	1.044	1.044	1.042
Variance about the mean	=	0.040	0.052	0.052	0.047	0.049	0.052	0.042	0.042	0.040
Mean j directional offset	=	1.073	1.057	1.057	1.059	1.057	1.061	1.055	1.057	1.057
Variance about the mean	=	0.116	0.094	0.094	0.096	0.094	0.097	0.092	0.094	0.094
Mean k directional offset	=	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Variance about the mean	=	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

SUBSECTION E.3

Computer generated CPU data for Parabolic flow over 8 timesteps is given.

```
==========
CPU Time Analysis
==========

Randomly disturb. init. grid
Parabolic Flow
Centroid Determined Partition
Point-to-triangle Analysis
Large triangle initial grid
Initial point grid dimensions  20 X  20 X   1
Initial MLG dimensions  31 X  31 X   1

-----------------------------------
Initial # of grid points       =     505
Initial # of triangles analyzed =     932
-----------------------------------

All times in secs. / Time frac. are %

time                0          1          2          3          4          5          6          7          8
-----------------------------------------------------------------------------------------------------------
Pnt. Initial.    0.060      (time frac: .027)
Tri. Initial.    0.100      (time frac: .037)
MLG Initial.     0.030      (time frac: .011)

MLG Adjust.      0.370      0.080      0.070      0.070      0.070      0.070      0.070      0.070      0.000
Time frac.       0.026      0.052      0.040      0.031      0.028      0.022      0.016      0.015      0.015

MLG Sort         1.510      0.640      0.720      0.800      0.720      0.640      0.720      0.720      0.000
Time frac.       0.586      0.416      0.412      0.359      0.288      0.202      0.169      0.168      0.164

Analysis (srch)  0.780      0.690      0.810      0.230      0.690      0.730      0.980      0.710      0.000
Time frac.       0.292      0.448      0.468      0.552      0.635      0.735      0.884      0.718      0.728

Analysis (stor)  0.120      0.120      0.130      0.130      0.120      0.130      0.130      0.120      0.000
Time frac.       0.045      0.078      0.075      0.058      0.048      0.041      0.029      0.029      0.025

Pnt. Movement    0.000      0.010      0.000      0.000      0.010      0.000      0.010      0.000      0.000
Time frac.       0.000      0.006      0.000      0.000      0.004      0.000      0.002      0.000      0.000

----

Avg. MLG Adjust          0.072
Avg. MLG Sort            0.914
Avg. Analysis (srch)     1.994
Avg. Analysis (stor)     0.126
Avg. Pnt. Movement       0.003

----

Overall run CPU time          28.190
```

SUBSECTION E.4

Computer generated CPU data for Random flow over 8 timesteps is given.

```
=================
CPU Time Analysis
=================

Randomly disturb. init. grid
Random Flow
Centroid Determined Partition
Point-to-triangle Analysis
Large triangle Initial grid
Initial point grid dimensions  20 X  20 X   1
Initial MLG dimensions  31 X  31 X  1

-----------------------------------------
Initial # of grid points          =    505
Initial # of triangles analyzed   =    932
-----------------------------------------

All times in secs. , Time frac. are %
```

time	0	1	2	3	4	5	6	7	8
Pnt. Initial.	0.070	(time frac. .025)							
Tri. Initial.	0.080	(time frac. .029)							
MLG Initial.	0.020	(time frac. .007)							
MLG Adjust.	0.070	0.070	0.080	0.070	0.070	0.070	0.080	0.070	0.070
Time frac.	0.025	0.053	0.068	0.056	0.057	0.057	0.070	0.063	0.067
MLG Sort	1.600	0.400	0.230	0.330	0.320	0.320	0.240	0.240	0.160
Time frac.	0.582	0.303	0.197	0.266	0.262	0.262	0.211	0.214	0.152
Analysis (srch)	0.690	0.640	0.650	0.630	0.620	0.620	0.600	0.590	0.600
Time frac.	0.251	0.485	0.556	0.508	0.508	0.508	0.526	0.527	0.571
Analysis (stor)	0.120	0.120	0.120	0.120	0.120	0.120	0.120	0.130	0.120
Time frac.	0.044	0.091	0.103	0.097	0.098	0.098	0.105	0.116	0.114
Pnt. Movement	0.100	0.090	0.090	0.090	0.090	0.090	0.100	0.090	0.100
Time frac.	0.036	0.068	0.077	0.073	0.074	0.074	0.088	0.080	0.095

```
----

Avg. MLG Adjust          0.072
Avg. MLG Sort            0.427
Avg. Analysis (srch)     0.627
Avg. Analysis (stor)     0.121
Avg. Pnt. Movement       0.093

----

Overall run CPU time        12.230
```