Lecture Notes Presented at the VKI Lecture Series 1994-05. Revised February 1995.

ASPECTS OF UNSTRUCTURED GRIDS AND FINITE-VOLUME SOLVERS FOR THE EULER AND NAVIER-STOKES EQUATIONS

Timothy J. Barth
Advanced Algorithms and Applications Branch
NASA Ames Research Center
Moffett Field, CA 94035
barth@nas.nasa.gov, http://www.nas.nasa.gov/~barth

ASPECTS OF UNSTRUCTURED GRIDS AND FINITE-VOLUME SOLVERS FOR THE EULER AND NAVIER-STOKES EQUATIONS

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

Introduction

The intent of these notes is to discuss basic algorithms for unstructured mesh generation and fluid flow calculation. The volume of literature on these subjects has grown enormously in recent years. Consequently, we consider only a small subset of this new technology for the remaining notes. Hopefully, the remaining portion of these notes will convince the reader that the theory and application of numerical methods for unstructured meshes has improved significantly in recent years.

1.0 Preliminaries

1.1 Graphs and Meshes

Graph theory offers many valuable theoretical results which directly impact the design of efficient algorithms using unstructured grids. For purposes of the present discussion, only *simple* graphs which do not contain self loops or parallel edges will be considered. Results concerning simple graphs usually translate directly into results relevant to unstructured grids. Perhaps the most well known graph theoretic result is Euler's formula which relates the number of edges $n(e)$, vertices $n(v)$, and faces $n(f)$ of a polyhedron (see figure 1.1.0(a)):

$$n(f) = n(e) - n(v) + 2 \qquad (\text{Euler's formula}) \tag{1.1.0}$$

This polyhedron can be embedded in a plane by mapping one face to infinity. This makes the graph formula (1.1.0) applicable to 2-D unstructured meshes. In the example below, the face 1-2-3-4 has mapped to infinity to form the exterior (infinite) face. If all faces are numbered including the exterior face, then Euler's formula (1.1.0) remains valid.

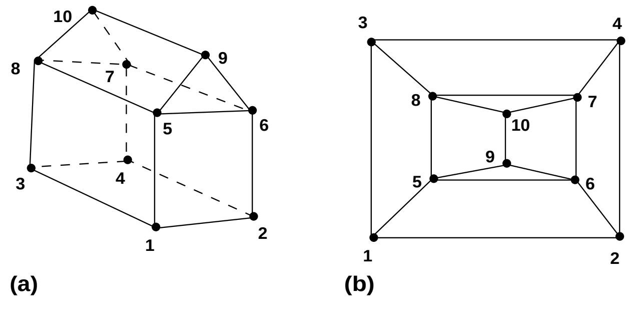


Figure 1.1.0 (a) 3-D Polyhedron, (b) 2-D Planar embedding

The infinite face can be eliminated by describing the outer boundary in terms of boundary edges which share exactly one interior face (interior edges share two). We also consider boundary edges which form simple closed polygons in the interior of the mesh. These polygons serve to describe possible objects embedded in the mesh (in this case, the polygon which they form is not counted as a face of the mesh). The number of these polygons is denoted by $n(h)$. The modified Euler's formula now reads

$$n(f) + n(v) = n(e) + 1 - n(h) \tag{1.1.1}$$

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

$$2n(e)_{interior} + n(e)_{bound} = \sum_{i=3}^{max\ d(f)} i\ n(f)_i \tag{1.1.2}$$

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

$$n(f)_3 = 2n(v) - n(e)_{bound} - 2 + 2n(h) \tag{1.1.3}$$

or

$$n(e) = 3n(v) - n(e)_{bound} - 3 + 3n(h). \tag{1.1.4}$$

This is a well known result for planar triangulations. (For brevity, we will sometimes use N to denote $n(v)$ in the remainder of these notes.) In many cases boundary edges are not explicitly given and the boundary is taken to be the convex hull of the vertices, the smallest convex polygon surrounding the vertices. (To obtain the convex hull in two dimensions, envision placing an elastic rubber band around the cloud of points. The final shape of this rubber band will be the convex hull.) *A key observation for planar meshes is the asymptotic linear storage requirement with respect to the number of vertices for arbitrary mesh arrangements.*

The Euler formula extends naturally to an arbitrary number of space dimensions. In this general setting, Euler's formula relates basic components (vertices, edges, faces, etc.) of higher dimensional polytopes. In computational geometry jargon, vertices, edges, and faces are all specific examples of "k-faces". A 0-face is simply a vertex, a 1-face corresponds to a edge, a 2-face corresponds to a *facet* (or simply a face), etc. A polytope $\mathcal{P}$ in $\mathbf{R}^d$ contains k-faces $\forall\ k\ \in \{-1, 0, 1, ..., d\}$. The -1-face denotes the *null set* face by standard convention. Let the number of k-faces contained in the polytope $\mathcal{P}$ be denoted by $N_k(\mathcal{P})$. For example, $N_0(\mathcal{P})$ would denote the number of vertices. Using this notation, we have the following relationships:

$$\begin{array}{ll} N_0 \equiv n(v)\ \ \text{(vertices)}, & N_1 \equiv n(e)\ \ \text{(edges)} \\ N_2 \equiv n(f)\ \ \text{(faces)}, & N_3 \equiv n(\phi)\ \ \text{(volumes)} \end{array}$$

By convention, there is exactly one null set contained in any polytope so that $N_{-1}(\mathcal{P}) = 1$. Using these results, we can succinctly state the general Euler formula for an arbitrary polytope in $\mathbf{R}^d$

$$\sum_{k=-1}^{d} (-1)^k N_k(\mathcal{P}) = 0 \quad (\text{Euler's Formula in } \mathbf{R}^{\mathrm{d}}) \tag{1.1.5}$$

On the surface of a polyhedron in 3-space, the standard Euler formula (1.1.0) is recovered since

$$-1 + N_0 - N_1 + N_2 - 1 = 0$$

or equivalently

$$n(f) + n(v) = n(e) + 2.$$

To obtain results relevant to three-dimensional unstructured grids, the Euler formula (1.1.5) is applied to a four-dimensional polytope.

$$n(f) + n(v) = n(e) + n(\phi) \tag{1.1.6}$$

This formula relates the number of vertices, edges, faces, and volumes $n(\phi)$ of a three-dimensional mesh. As in the two-dimensional case, this formula does not account for boundary effects because it is derived by looking at a single four-dimensional polytope. The example below demonstrates how to derive exact formulas including boundary terms for a tetrahedral mesh. Derivations valid for more general meshes in three dimensions are also possible.

<u>Example:</u> Derivation of Exact Euler Formula for 3-D Tetrahedral Mesh.

Consider the collection of volumes incident to a single vertex v_i and the polyhedron which describes the shell formed by these vertices. Let $F_b(v_i)$ and $N_b(v_i)$ denote the number of faces and vertices respectively of this polyhedron which actually lie on the boundary of the entire mesh. Also let $E(v_i)$ denote the total number of edges on the polyhedron surrounding v_i. Finally, let $d_\phi(v_i)$ and $d_e(v_i)$ denote the number of tetrahedral volumes and edges respectively that are incident to v_i. On this polyhedron, we have exact satisfaction of Euler's formula (1.1.0), i.e.

$$\overbrace{d_\phi(v_i) + F_b(v_i)}^{\text{polyhedral faces}} + \overbrace{d_e(v_i) + N_b(v_i)}^{\textit{vertices on polyhedron}} = E(v_i) + 2 \tag{1.1.7}$$

Note that this step assumes that the polyhedron is homeomorphic to a sphere (otherwise Euler's formula fails). In reality, this is not a severe assumption. (It would preclude a mesh consisting of two tetrahedra which touch at a single vertex.) On the polyhedron we also have that

$$2E(v_i) = 3\overbrace{(d_\phi(v_i) + F_b(v_i))}^{\text{polyhedral faces}}. \tag{1.1.8}$$

Combining (1.1.7) and (1.1.8) yields

$$d_e(v_i) = \frac{1}{2} d_\phi(v_i) + \frac{1}{2} F_b(v_i) - N_b(v_i) + 2. \tag{1.1.9}$$

Summing this equation over <u>all</u> vertices produces

$$\overline{d}_e n(v) = \frac{1}{2}\left(\overline{d}_\phi n(v) + \sum_i F_b(v_i)\right) - \sum_i N_b(v_i) \tag{1.1.10}$$

where $\overline{d}_e$ and $\overline{d}_\phi$ are the average vertex degrees with respect to edges and volumes. Since globally we have that

$$\overline{d}_e n(v) = 2n(e), \quad \overline{d}_\phi n(v) = 4n(\phi), \tag{1.1.11}$$

substitution of (1.1.11) into (1.1.10) reveals that

$$n(e) = n(\phi) + n(v) + \frac{1}{4}\sum F_b(v_i) - \frac{1}{2}\overbrace{\sum N_b(v_i)}^{n(v)_{bound}}. \tag{1.1.12}$$

Finally, note that $\sum F_b(v_i) = 3n(f)_{bound}$. Inserting this relationship into (1.1.12) yields

$$n(e) = n(\phi) + n(v) + \frac{3}{4}n(f)_{bound} - \frac{1}{2}n(v)_{bound} \tag{1.1.13}$$

Other equivalent formulas are easily obtained by combining this equation with the formula relating volume, faces, and boundary faces, i.e.

$$n(f) = 2n(\phi) + \frac{1}{2}n(f)_{bound} \tag{1.1.14}$$

An exact formula, similar to (1.1.6), is obtained by combining (1.1.14) and (1.1.13)

$$n(e) + n(\phi) = n(f) + n(v) + \frac{1}{4}n(f)_{bound} - \frac{1}{2}n(v)_{bound} \tag{1.1.15}$$

<u>1.2 Duality</u>

Consider a planar graph G of vertices, edges, and faces (cells). We define a dual graph G_{Dual} to be any graph with exhibits the following three properties: each vertex of G_{Dual} is associated with a face of G; each edge of G is associated with an edge of G_{Dual}; if an edge separates two faces, f_i and f_j of G then the associated dual edge connects two vertices of G_{Dual} associated with f_i and f_j. This duality plays an important role in some algorithms for solving conservation law equations.

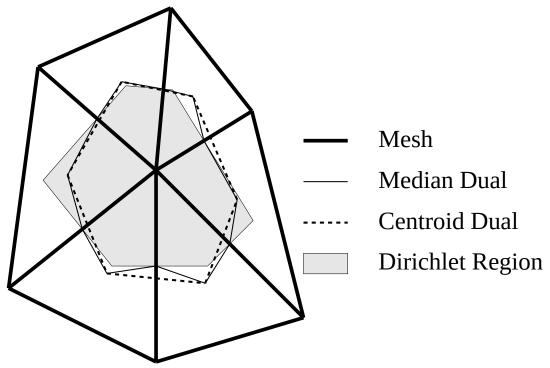


Figure 1.2.0 Several triangulation duals.

In figure 1.2.0 edges and faces about the central vertex are shown for duals formed from median segments, centroid segments, and by Dirichlet tessellation. (The Dirichlet tessellation of a set of points is a pattern of convex regions in the plane, each region being the portion of the plane closer to some given point P of the set of points than to any other point.) These geometric duals arise naturally for two-dimensional finite-volume schemes which solve conservation law equations. Schemes which use the cells of the mesh as control volumes are often called "cell-centered" schemes. Other "vertex" schemes use mesh duals constructed from median segments, Dirichlet regions, or centroid segments. In later sections, we will construct algorithms for solving the Euler and Navier-Stokes equations which use these geometric duals. The graph theoretic results of the previous section will provide sharp estimates of the computational work and storage requirements for these algorithms. The estimates obtained in two space dimensions are reasonably intuitive whereas results in three space dimensions can be nonintuitive and somewhat surprising. This is the power of graph theoretic and combinatoral analysis.

1.3 Data Structures

The choice of data structures used in representing unstructured grids varies considerably depending on the type of algorithmic operations to be performed. In this section, a few of the most common data structures will be discussed. The mesh is assumed to have a numbering of vertices, edges, faces, etc. In most cases, the physical coordinates are simply listed by vertex number. The "standard" finite element (FE) data structure lists connectivity of each element. For example in figure 1.3.0(a), a list of the three vertices of each triangle would be given. The FE structure extends naturally to three dimensions. The FE structure is used extensive in finite element solvers for solids and fluids.

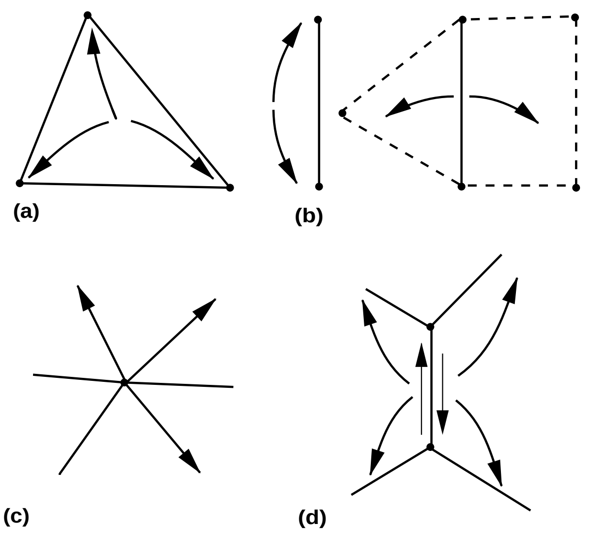


Figure 1.3.0 Data structures for planar graphs. (a) FE data structure, (b) Edge structure, (c) Out-degree structure, (d) Quad-edge structure.

For planar meshes, another typical structure is the *edge structure* (figure 1.3.0(b)) which lists connectivity of vertices and adjacent faces by storing a quadruple for each edge consisting of the origin and destination of the each edge as well as the two faces (cells) that share that edge. This structure allows easy traversal through a mesh which can be useful in certain grid generation algorithms. (This traversal is not easily done using the FE structure.) The extension to three dimensions is facewise (vertices of a face are given as well as the two neighboring volumes) and requires distinction between different face types.

A third data structure provides connectivity via *vertex lists* as shown in figure 1.3.0(c). The brute force approach is to list all adjacent neighbors for each vertex (usually as a linked-list). Many sparse matrix solver packages specify nonzeros of a matrix using row or column storage schemes which list all nonzero entries of a given row or column. For discretizations involving only adjacent neighbors of a mesh, this would be identical to specifying a vertex list. An alternative to specifying all adjacent neighbors is to direct edges of the mesh. In this case only those edges which point outward from a vertex are listed. In the next section, it will be shown that an *out-degree list* can be constructed for planar meshes by directing a graph such that no vertex has more than three outgoing edges. This is asymptotically optimal. The extension of the out-degree structure to three dimensions is not straightforward and algorithms for obtaining optimal edge direction for nonplanar graphs are still under development.

The last structure considered here is the *quad-edge structure* proposed by Guibas and Stolfi [GuiSt85], see figure 1.3.0(d). Each edge is stored as pair of directed edges. Each of the directed edges stores the edge origin and pointers to the next and previous directed edge of the region which lies to the left of the edge. The quad-edge structure is extremely useful in grid generation where distinctions between topological and geometrical aspects are sometimes crucial. The structure has been extended to three dimensional arrangements by Dobkin and Laslo [DobL89] and Brisson [Bri89].

2.0 Some Basic Graph Operations Important in CFD

Implementation of unstructured grid methods on differing computer architectures has stimulated research in exploiting properties and characterizations of planar and nonplanar graphs. For example in Hammond and Barth [HamB91], we exploited a recent theorem by Chrobak and Eppstein [ChroE91] concerning directed graphs with minimum out-degree. In this work an Euler equation solver was constructed on arbitrary triangulations. The directed graph of the triangulation was used as a means for optimal load balancing of computation on a SIMD parallel computer.

In this section, we review this result as well as present other basic graph operations that are particularly useful in CFD. Some of these algorithms are specialized to planar graphs (2-D meshes) while others are very general and apply in any number of space dimensions.

2.1 Planar Graphs with Minimum Out-Degree

Theorem: *Every planar graph has a 3-bounded orientation*, [ChoE91].

In other words, each edge of a planar graph can be assigned a direction such that the maximum number of edges pointing outward from any vertex is less than or equal to three. A constructive proof is given in [ChroE91] consisting of the following steps. The first step is to find a *reduceable* boundary vertex. A reduceable boundary vertex is any vertex on the boundary with incident exterior (boundary) edges that connect to at most two other boundary vertices and any number of interior edges. Chrobak and Eppstein prove that reduceable vertices can always be found for arbitrary planar graphs. (In fact, two reduceable vertices can always be found!) Once a reduceable vertex is found then the two edges connecting to the other boundary vertices are directed *outward* and the remaining edges are always directed *inward.* These directed edges are then removed from the graph, see Figures 2.1.0(a-j). The process is then repeated on the remaining graph until no more edges remain. The algorithm shown pictorially in figure 2.1.0 is summarized in the following steps:

Algorithm: Orient a Graph with maximum out-degree ≤ 3.

Step 1. Find reduceable boundary vertex.

Step 2. Direct exterior edges outward and interior edges inward.

Step 3. Remove directed edges from graph.

Step 4. If undirected edges remain, go to step 1

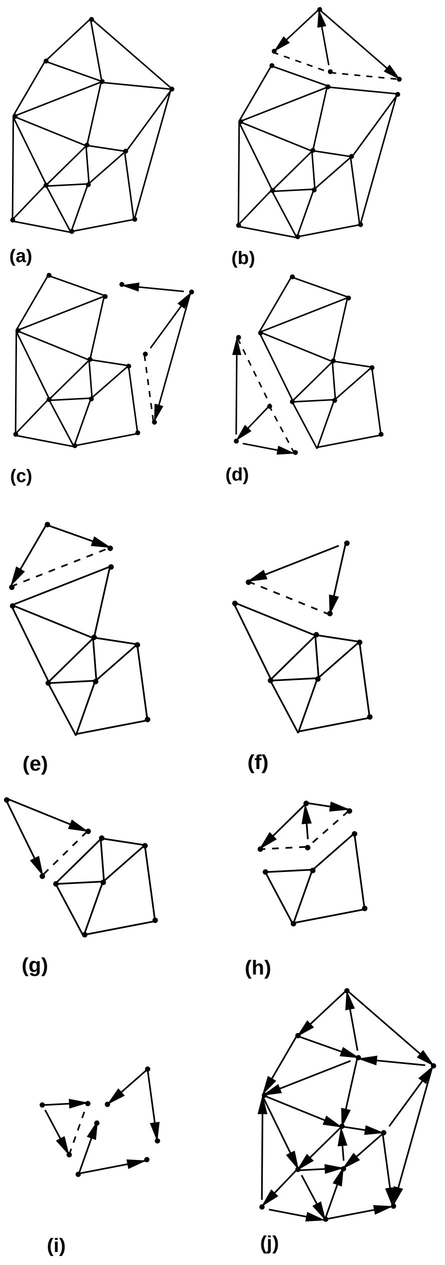


Figure 2.1.0 (a-i) Mesh orientation procedure with out-degree 3, (j) final oriented triangulation.

2.2 Graph Ordering Techniques

The particular ordering of solution unknowns can have a marked influence on the amount of computational effort and memory storage required to solve large sparse linear systems and eigenvalue problems. In many algorithms, the band width and profile characteristics of the matrix determines the amount of computation and memory required. Most meshes obtained from grid generation codes have very poor natural orderings.

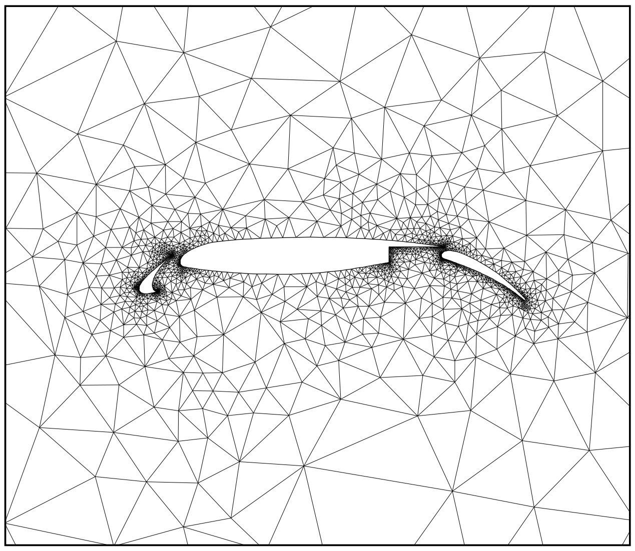


Figure 2.2.0 Typical Steiner triangulation about multi-component airfoil.

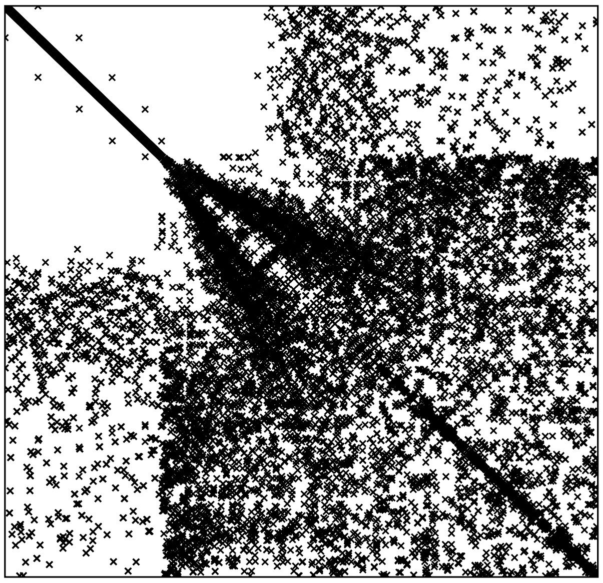


Figure 2.2.1 Nonzero entries of Laplacian matrix produced from natural ordering.

Figures 2.2.0 and 2.2.1 show a typical mesh generated about a multi-component airfoil and the nonzero entries associated with the "Laplacian" of the graph. The Laplacian of a graph would represent the nonzero entries due to a discretization which involves only adjacent neighbors of the mesh. Figure 2.2.1 indicates that the band width of the natural ordering is almost equal to the dimension of the matrix! In parallel computation, the ordering algorithms can be used as means for *partitioning* a mesh among processors of the computer. This will be addressed in the next section.

Several algorithms exist which construct new orderings by attempting to minimize the band width of a matrix or attempting to minimize the fill that occurs in the process matrix factorization. These algorithms usually rely on heuristics to obtain high efficiency, and do not usually obtain an optimum ordering. One example would be Rosen's algorithm [Ros68] which iterates on the ordering to minimize the maximum band width.

Algorithm: Graph ordering, Rosen.

Step 1. Determine band width and the defining index pair (i, j) with $(i < j)$
Step 2. Does their exist an exchange which increases i or decreases j so that the band width is reduced? If so, exchange and go to step 1
Step 3. Does their exist an exchange which increases i or decreases j so that the band width remains the same? If so, exchange and go to step 1

This algorithm produces very good orderings but can be very expensive for large matrices. A popular method which is much less expensive for large matrices is the Cuthill-McKee [CutM69] algorithm.

Algorithm: Graph ordering, Cuthill-McKee.

Step 1. Find vertex with lowest degree. This is the *root* vertex.
Step 2. Find all neighboring vertices connecting to the root by incident edges. Order them by increasing vertex degree. This forms level 1.
Step 3. Form level k by finding all neighboring vertices of level $k-1$ which have not been previously ordered. Order these new vertices by increasing vertex degree.
Step 4. If vertices remain, go to step 3

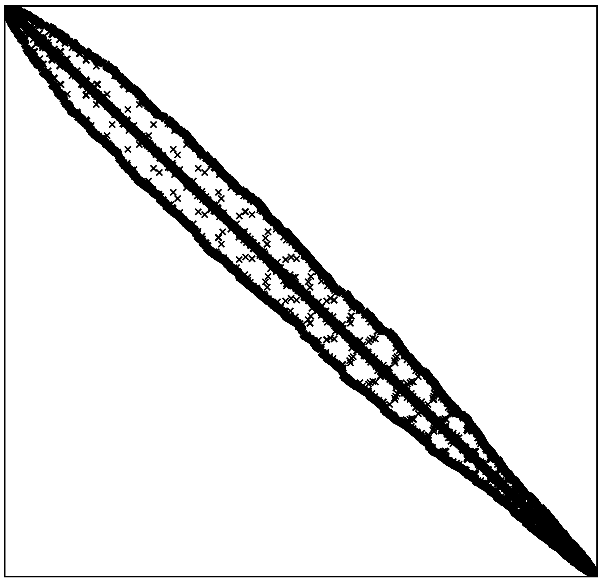


Figure 2.2.2 Nonzero entries of Laplacian matrix after Cuthill-McKee ordering.

The heuristics behind the Cuthill-McKee algorithm are very simple. In the graph of the mesh, neighboring vertices must have numberings which are near by, otherwise they will produce entries in the matrix with large band width. The idea of sorting elements among a given level is based on the heuristic that vertices with high degree should be given indices as large as possible so that they will be as close as possible to vertices of the *next* level generated. Figure 2.2.2 shows the dramatic improvement of the Cuthill-McKee ordering on the matrix shown in figure 2.2.1.

Studies of the Cuthill-McKee algorithm have shown that the fill characteristics of a matrix during L-U decomposition can be greatly reduced simply by reversing the ordering of the Cuthill-McKee algorithm, see George [Geo71]. This amounts to a renumbering given by

$$k \to n - k + 1 \tag{2.2.0}$$

where n is the size of the matrix. While this does not change the bandwidth of the matrix, it can dramatically reduce the fill that occurs in Cholesky or L-U matrix factorization when compared to the original Cuthill-McKee algorithm.

2.3 Graph Partitioning for Parallel Computing

An efficient partitioning of a mesh for distributed memory machines is one that ensures an even distribution of computational workload among the processors and minimizes the amount of time spent in interprocessor communications. The former requirement is termed *load balancing*. For if the load were not evenly distributed, some processors will have to sit idle at synchronization points waiting for other processors to catch up. The second requirement comes from the fact that communication between processors takes time and it is not always possible to hide this latency in data transfer. In our parallel implementation of a finite-volume flow solver on unstructured grids, data for the vertices that lie on the boundary between two processors is exchanged, hence requiring a bidirectional data-transfer. On some systems, a synchronous exchange of data can yield a higher performance than when done in an asynchronous fashion. To exploit this fact, edges of the communication graph are colored such that no vertex has more than one edge of a certain color incident upon it.

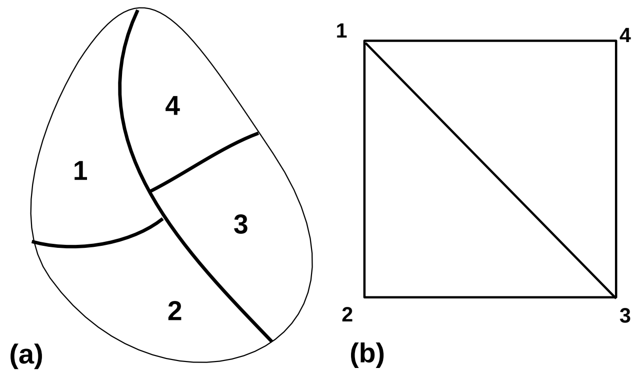


Figure 2.3.0 (a) Four partition mesh, (b) Communication graph.

A communication graph is a graph in which the vertices are the processors and an edge connects two vertices if the two corresponding processors share an interprocessor boundary. The colors in the graph represent separate communication cycles. For instance, the mesh partitioned amongst four processors as shown in figure 2.3.0(a), would produce the communication graph shown in figure 2.3.0(b). The graph shown in figure 2.3.0(b) can be colored edgewise using three colors. In the first communication cycle, processors $(1,4)$ could perform a synchronous data exchange as would processors $(2,3)$. In the second communication cycle, processors $(1,2)$ and $(3,4)$ would exchange and in the third cycle, processors $(1,3)$ would exchange while processors 2 and 4 sit idle. Vizing's theorem proves that any graph of maximum vertex degree Δ (number of edges incident upon a vertex) can be colored using n colors such that $\Delta \leq n \leq \Delta + 1$. Hence, any operation that calls for every processor to exchange data with its neighbors will require n communication cycles.

The actual cost of communication can often be accurately modeled by the linear relationship:

$$Cost = \alpha + \beta m \tag{2.3.0}$$

where α is the time required to initiate a message, β is the rate of data-transfer between two processors and m is the message length. For n messages, the cost would be

$$Cost = \sum_n (\alpha + \beta m_n). \tag{2.3.1}$$

This cost can be reduced in two ways. One way is to reduce Δ thereby reducing n. The alternative is to reduce the individual message lengths. The bounds on n are $2 \leq N \leq P-1$ for $P \geq 3$ where P is the total number of processors. Figure 2.3.1(a) shows the partitioning of a mesh which reduces Δ, and 2.3.1(b) shows a partitioning which minimizes the message lengths. For the mesh in figure 2.3.1(a), $\Delta = 2$ while in figure 2.3.1(b), $\Delta = 3$. However, the average message length for the partitions shown in figure 2.3.1(b) is about half as much as that in figure 2.3.1(a).

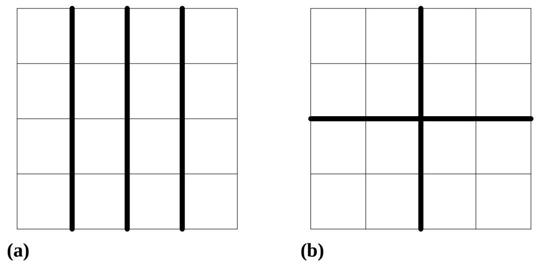


Figure 2.3.1 (a) Mesh partitioning with minimized Δ, (b) Mesh with minimizes message length.

In practice, it is difficult to partition an unstructured mesh while simultaneously minimizing the number and length of messages. In the following paragraphs, a few of the most popular partitioning algorithms which approximately accomplish this task will be discussed. All the algorithms discussed below: coordinate bisection, Cuthill-McKee, and spectral partitioning are evaluated in the paper by Venkatakrishnan, Simon, and Barth

[VenSB91]. This paper evaluates the partitioning techniques within the confines of an explicit, unstructured finite-volume Euler solver. Spectral partitioning has been extensively studied by Simon [Simon91] for other applications.

Note that for the particular applications that we have in mind (a finite-volume scheme with solution unknowns at vertices of the mesh), it makes sense to partition the domain such that the separators correspond to edges of the mesh. Also note that the partitioning algorithms all can be implemented recursively. The mesh is first divided into two sub-meshes of nearly equal size. Each of these sub-meshes is subdivided into two more sub-meshes and the process in repeated until the desired number of partitions P is obtained (P is a integer power of 2). Since we desire the separator of the partitions to coincide with edges of the mesh, the division of a sub-mesh into two pieces can be viewed as a 2-coloring of *faces* of the sub-mesh. For the Cuthill-McKee and spectral partitioning techniques, this amounts to supplying these algorithms with the *dual* of the graph for purposes of the 2-coloring. The balancing of each partition is usually done cellwise; although an edgewise balancing is more appropriate in the present applications. Due to the recursive nature of partitioning, the algorithms outlined below represents only a single step of the process.

Coordinate Bisection

In the coordinate bisection algorithm, face centroids are sorted either horizontally or vertically depending of the current level of the recursion.

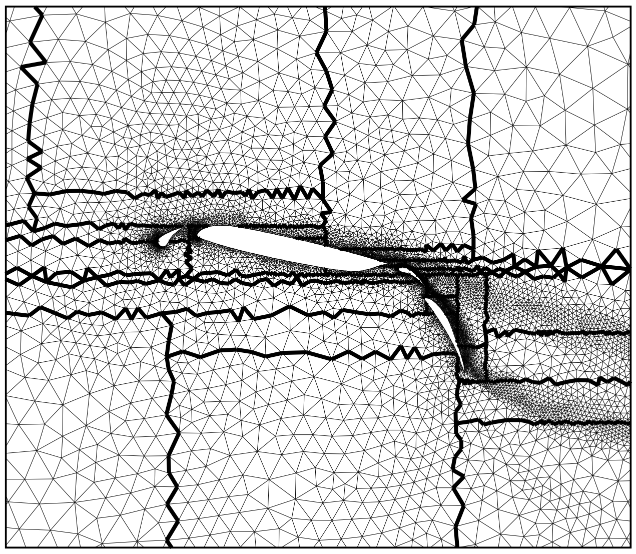


Figure 2.3.2 Coordinate bisection (16 partitions).

A separator is chosen which balances the number of faces. Faces are colored depending on which side of the separator they are located. The actual edges of the mesh corresponding to the separator are characterized as those edges which have adjacent faces of different color, see figure 2.3.2. This partitioning is very efficient to create but gives sub-optimal performance on parallel computations owing to the long message lengths than can routinely occur.

Cuthill-McKee

The Cuthill-McKee algorithm described earlier can also be used for recursive mesh partitioning. In this case, the Cuthill-McKee ordering is performed on the *dual* of the mesh graph. A separator is chosen either at the median of the ordering (which would balance the coloring of faces of the original mesh) or the separator is chosen at the level set boundary *closest* to the median as possible. This latter technique has the desired effect of reducing the number of disconnected sub-graphs that occur during the course of the partitioning. Figure 2.3.3 shows a Cuthill-McKee partitioning for the multi-component airfoil mesh. The Cuthill-McKee ordering tends to produce long boundaries because of the way that the ordering is propagated through a mesh. The maximum degree of the communication graph also tends to be higher using the Cuthill-McKee algorithm. The results shown in [VenSB91] for multi-component airfoil grids indicate a performance on parallel computations which is slightly worse than the coordinate bisection technique.

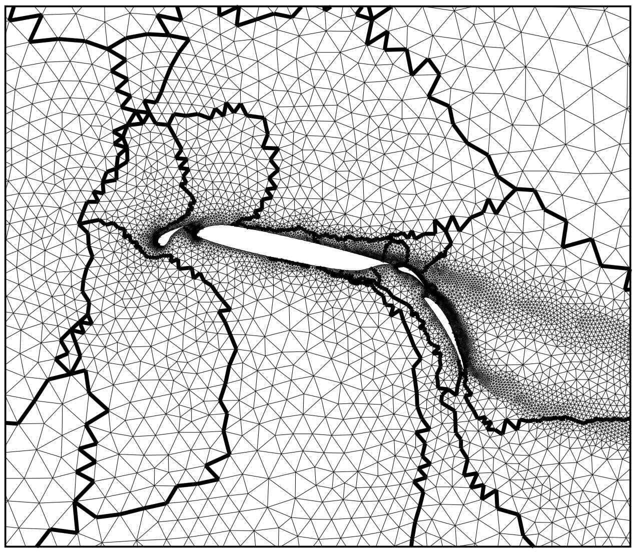


Figure 2.3.3 Cuthill-McKee partitioning of mesh.

Spectral Partitioning

The last partitioning considered is the spectral partitioning [PothSL90] which exploits properties of the Laplacian $\mathcal{L}$ of a graph (defined below). The algorithm consists of the following steps:

Algorithm: Spectral Partitioning.

Step 1. Calculate the matrix $\mathcal{L}$ associated with the Laplacian of the graph (dual graph in the present case).
Step 2. Calculate the eigenvalues and eigenvectors of $\mathcal{L}$.
Step 3. Order the eigenvalues by magnitude, $\lambda_1 \leq \lambda_2 \leq \lambda_3 ... \lambda_N$.
Step 4. Determine the smallest nonzero eigenvalue, λ_f and its associated eigenvector $\mathbf{x}_f$ (the Fiedler vector).
Step 5. Sort elements of the Fiedler vector.

Step 6. Choose a divisor at the median of the sorted list and 2-color vertices of the graph (or dual) which correspond to elements of the Fielder vector less than or greater than the median value.

The spectral partitioning of the multi- component airfoil is shown in figure 2.3.4. In [VenSB91] it was observed that superior performance was attained for parallel computations on the spectral partitioning than on the coordinate bisection or Cuthill-McKee partitionings. The cost of the spectral partitioning is high (even using a Lanczos algorithm to compute the eigenvalue problem). This cost has been reduced by the use of a multilevel Lanczos algorithm in [BarnS93].

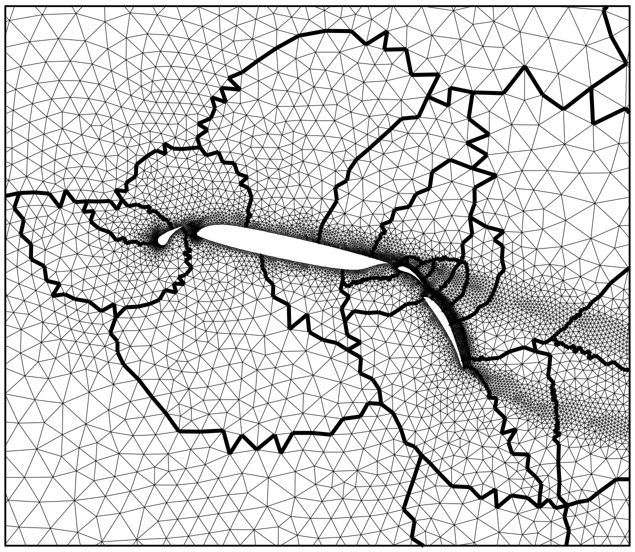


Figure 2.3.4 Spectral partitioning of multi- component airfoil.

The spectral partitioning exploits a peculiar property of the "second" eigenvalue of the Laplacian matrix associated with a graph. The Laplacian matrix of a graph is simply

$$\mathcal{L} = -\mathcal{D} + \mathcal{A}. \tag{2.3.2}$$

where $\mathcal{A}$ is the standard adjacency matrix

$$\mathcal{A}_{ij} = \begin{cases} 1 & e(v_i, v_j) \in \ G \\ 0 & \text{otherwise} \end{cases} \tag{2.3.2}$$

and $\mathcal{D}$ is a diagonal matrix with entries equal to the degree of each vertex, $\mathcal{D}_i = d(v_i)$. From this definition, it should be clear that rows of $\mathcal{L}$ each sum to zero. Define an N-vector, $\mathbf{s} = [1, 1, 1, ...]^T$. By construction we have that

$$\mathcal{L}\mathbf{s} = 0. \tag{2.3.3}$$

This means that at least one eigenvalue is zero with $\mathbf{s}$ as an eigenvector.

The objective of the spectral partitioning is to divide the mesh into two partitions of equal size such that the number of edges cut by the partition boundary is approximately minimized.

Technically speaking, the smallest nonzero eigenvalue need not be the second. Graphs with disconnected regions will have more than one zero eigenvalue depending on the number of disconnected regions. For purposes of discussion, we assume that disconnected regions are not present, i.e. that λ_2 is the relevant eigenmode.

Elements of the proof:

Define a partitioning vector which 2-colors the vertices

$$\mathbf{p} = [+1, -1, -1, +1, +1, ..., +1, -1]^T \tag{2.3.4}$$

depending on the sign of elements of $\mathbf{p}$ and the one-to-one correspondence with vertices of the graph, see for example figure 2.3.5. Balancing the number of vertices of each color amounts to the requirement that

$$\mathbf{s} \perp \mathbf{p} \tag{2.3.5}$$

where we have assumed an even number of vertices.

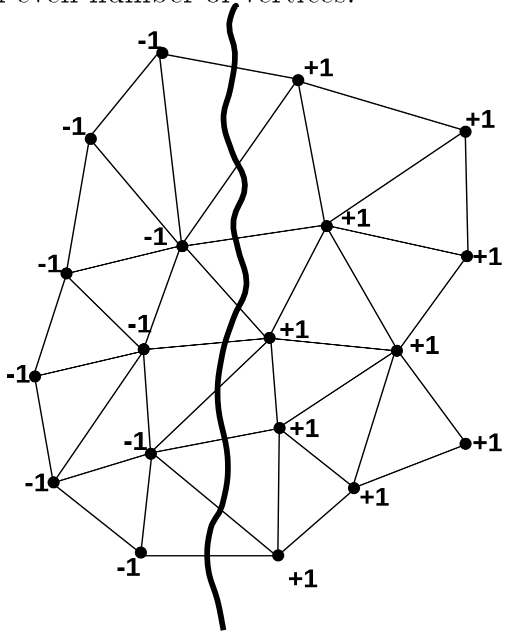


Figure 2.3.5 Arbitrary graph with 2-coloring showing separator and cut edges.

The key observation is that the number of cut edges, E_c, is precisely related to the L_1 norm of the Laplacian matrix multiplying the partitioning vector, i.e.

$$4E_c = \|\mathcal{L}\mathbf{p}\|_1 \tag{2.3.6}$$

which can be easily verified. The goal is to minimize cut edges. That is to find $\mathbf{p}$ which minimizes $\|\mathcal{L}\mathbf{p}\|_1$ subject to the constraints that $\|\mathbf{p}\|_1 = N$ and $\mathbf{s} \perp \mathbf{p}$. Since $\mathcal{L}$ is a real symmetric (positive semi-definite) matrix, it has a complete set of real eigenvectors which can be orthogonalized with each other. The next step of the proof would be to extend the domain of $\mathbf{p}$ to include real numbers (this introduces an inequality) and expand $\mathbf{p}$ in terms of the orthogonal eigenvectors.

$$\mathbf{p} = \sum_{i=1}^{n} c_i \mathbf{x}_i \tag{2.3.7}$$

By virtue of (2.3.5) we have that $\mathbf{x}_1 = \mathbf{s}$. It remains to be shown that $\|\mathcal{L}\mathbf{p}\|_1$ is minimized when $\mathbf{p} = \mathbf{p}' = n\mathbf{x}_2/\|\mathbf{x}_2\|_1$,i.e. when the Fiedler vector is used. Inserting this expression for $\mathbf{p}$ we have that

$$\|\mathcal{L}\mathbf{p}'\|_1 = n\lambda_2 \tag{2.3.8}$$

It is a simple matter to show that adding any other eigenvector component to $\mathbf{p}'$ while insisting that $\|\mathbf{p}\|_1 = N$ can only increase the L_1 norm. This would complete the proof. Figure 2.3.6 plots contours (level sets) of the Fiedler vector for the multi-component airfoil problem.

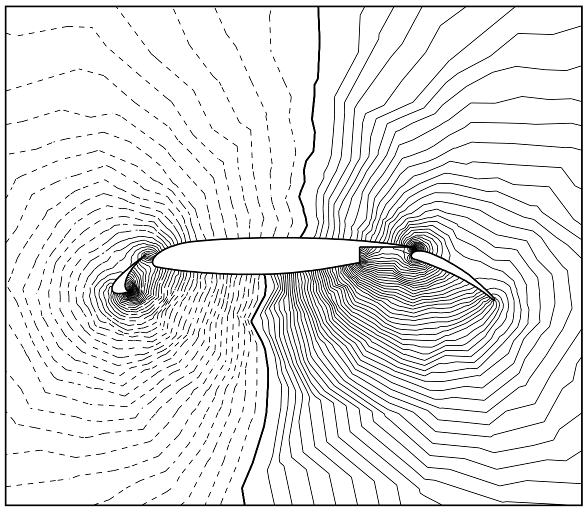


Figure 2.3.6. Contours of Fiedler Vector for Spectral Partitioning. Dashed lines are less than the median value.

3.0 Triangulation Methods

Although many algorithms exist for triangulating sites (points) in an arbitrary number of space dimensions, only a few have been used on practical problems. In particular, Delaunay triangulation has proven to be a very useful triangulation technique. This section will present some of the basic concepts surrounding Delaunay and related triangulations as well as discussing some of the most popular algorithms for constructing these triangulations.

3.1 Voronoi Diagrams and the Delaunay Triangulation

Recall the definition of the Dirichlet tessellation in a plane. The Dirichlet tessellation of a point set is the pattern of convex regions, each being closer to some point P in the point set than to any other point in the set. These Dirichlet regions are also called Voronoi regions [Vor07]. The edges of Voronoi polygons comprise the Voronoi diagram, see figure 3.1.0. The idea extends naturally to higher dimensions.

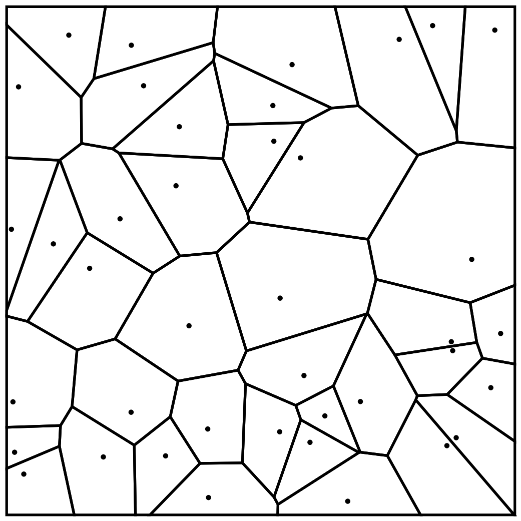


Figure 3.1.0 Voronoi diagram of 40 random sites.

Voronoi diagrams have a rich mathematical theory. *The Voronoi diagram is believed to be one of the most fundamental constructs defined by discrete data.* Voronoi diagrams have been independently discovered in a wide variety of disciplines. Computational geometricians have a keen interest in Voronoi diagrams. It is well known that Voronoi diagrams are related to convex hulls via stereographic projection. Point location in a Voronoi diagram can be performed in $O(\log(n))$ time with $O(n)$ storage for n regions. This is useful in solving post-office or related problems in optimal time. Another example of the Voronoi diagram which occurs in the natural sciences can be visualized by placing crystal "seeds" at random sites in 3-space. Let the crystals grow at the same rate in all directions. When two crystals collide simply stop their growth. The crystal formed for each site would represent that volume of space which is closer to that site than to any other site. This would effectively construct a Voronoi diagram. We now consider the role of Voronoi diagrams in Delaunay triangulation [Del34].

Definition: The Delaunay triangulation of a point set is defined as the dual of the Voronoi diagram of the set.

The Delaunay triangulation in two space dimensions is formed by connecting two points if and only if their Voronoi regions have a common border segment. If no four or more points are cocircular, then we have that *vertices of the Voronoi are circumcenters of the triangles.* This is true because vertices of the Voronoi represent locations that are equidistant to three (or more) sites. Also note that from the definition of duality, edges of the Voronoi are in one-to-one correspondence to edges of the Delaunay triangulation (ignoring boundaries). Because edges of the Voronoi diagram are the locus of points equidistant to two sites, each edge of the Voronoi diagram is perpendicular to the corresponding edge of the Delaunay triangulation. This duality extends to three dimensions in a straightforward way. The

Delaunay triangulation possesses several alternate characterizations and many properties of importance. Unfortunately, not all of the two dimensional characterizations have three-dimensional extensions. To avoid confusion, properties and algorithms for construction of two dimensional Delaunay triangulations will be considered first. The remainer of this section will then discuss the three-dimensional Delaunay triangulation.

3.2 Properties of 2-D Delaunay Triangulations

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

(2) *The circumcircle criteria.* A triangulation of $N \geq 2$ sites is Delaunay if and only if the circumcircle of every interior triangle is point-free. For if this was not true, the Voronoi regions associated with the dual would not be convex and the Dirichlet tessellation would be invalid. Related to the circumcircle criteria is the *incircle* test for four points as shown in figures 3.2.0-3.2.1.

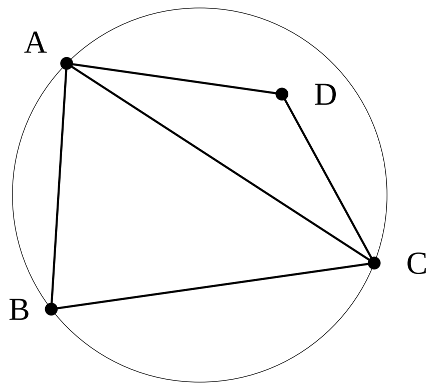


Figure 3.2.0 Incircle test for $\triangle ABC$ and D (true).

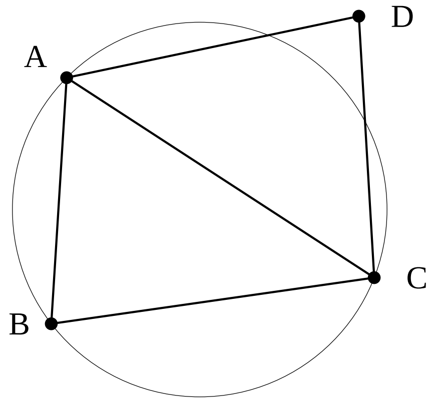


Figure 3.2.1 Incircle test for $\triangle ABC$ and D (false).

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

$$\angle ABC + \angle CDA = \begin{cases} < 180^\circ & \text{incircle false} \\ \phantom{<} 180^\circ & \text{A,B,C,D cocirc} \\ > 180^\circ & \text{incircle true} \end{cases} \tag{3.2.0}$$

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

(3)*Edge circle property.* A triangulation of sites is Delaunay if and only if there exists *some* circle passing through the endpoints of each and every edge which is point-free. This characterization is very useful because it also provides a mechanism for defining a *constrained* Delaunay triangulation where certain edges are prescribed apriori. A triangulation of sites is a constrained Delaunay triangulation if for each and every edge of the mesh there exists some circle passing through its endpoints containing no other site in the triangulation which is *visible* to the edge. In figure 3.2.2, site d is not visible to the segment a-c because of the constrained edge a-b.

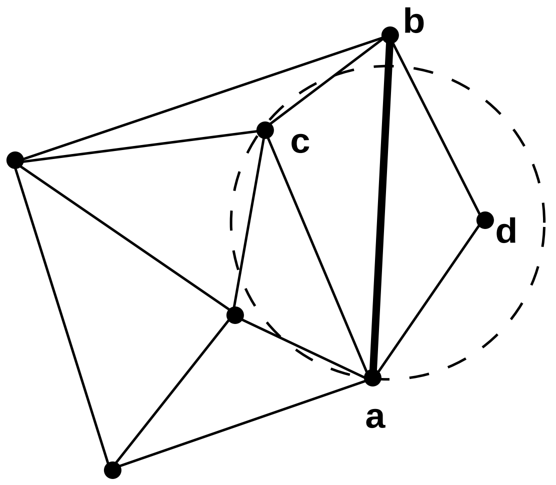


Figure 3.2.2 Constrained Delaunay triangulation. Site d is not visible to a-c due to constrained segment a-b.

(4)*Equiangularity property.* Delaunay triangulation maximizes the minimum angle of the triangulation. For this reason Delaunay triangulation often called the MaxMin triangulation. This property is also locally true for all adjacent triangle pairs which form a convex quadrilateral. This is the basis for the local edge swapping algorithm of Lawson [Law77] described below.

(5)*Minimum Containment Circle.* A recent result by Rajan [Raj91] shows that the Delaunay triangulation minimizes the maximum containment circle over the entire triangulation. The containment circle is defined as the smallest circle enclosing the three vertices of a triangle. This is identical to the circumcircle for acute triangles and a circle with diameter equal to the longest side of the triangle for obtuse triangles (see figure 3.2.3). This property extends to n dimensions. Unfortunately, the result does not hold lexicographically.

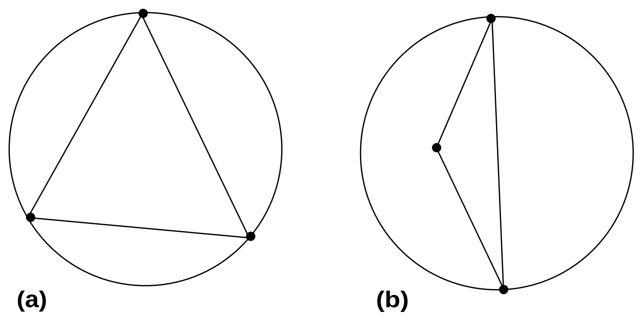


Figure 3.2.3 Containment circles for acute and obtuse triangles.

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

(7)*Minimal roughness.* The Delaunay triangulation is a minimal roughness triangulation for arbitrary sets of scattered data, [Rippa90]. Given arbitrary data f_i at all vertices of the mesh and a triangulation of these points, a unique piecewise linear interpolating surface can be constructed. The Delaunay triangulation has the property that of all triangulations it minimizes the roughness of this surface as measured by the following Sobolev semi-norm:

$$\int_T \left[\left(\frac{\partial f}{\partial x} \right)^2 + \left(\frac{\partial f}{\partial y} \right)^2 \right] \, dx \, dy \tag{3.2}$$

This is a interesting result as it does not depend on the actual form of the data. This also indicates that Delaunay triangulation approximates well those functions which minimize this Sobolev norm. One example would be the harmonic functions satisfying Laplace's equation with suitable boundary conditions which minimize exactly this norm. In a later section, we will prove that a Delaunay triangulation guarantees a maximum principle for the discrete Laplacian approximation (with linear elements).

3.3 Algorithms for 2-D Delaunay Triangulation

We now consider several techniques for Delaunay triangulation in two dimensions. These methods were chosen because they perform optimally in rather different situations. The discussion of the 2-D algorithms is organized as follows:

(a) Divide and Conquer Algorithm

(b) Incremental Insertion Algorithms

(i) Bowyer algorithm

(ii) Watson algorithm

(iii) Green and Sibson algorithm

(iv) Randomized Algorithms

(c) Global Edge Swapping (Lawson)

3.3a Divide-and-Conquer Algorithm

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

Algorithm: Delaunay Triangulation of Point Set Via Divide-and-Conquer

Step 1. Partition T into two subsets T_L and T_R of nearly equal size.

Step 2. Delaunay triangulate T_L and T_R recursively.

Step 3. Merge T_L and T_R into a single Delaunay triangulation.

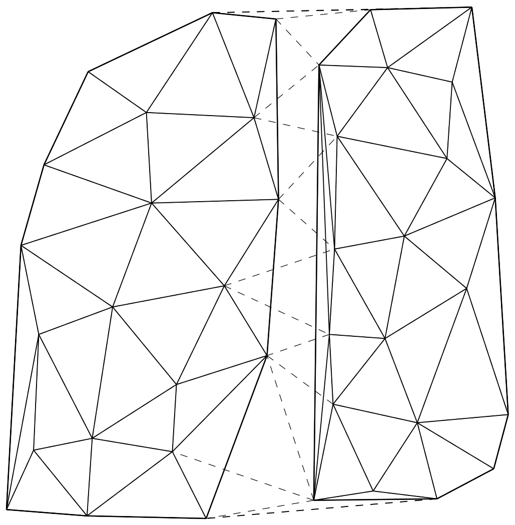


Figure 3.3.0 Triangulated subdivisions.

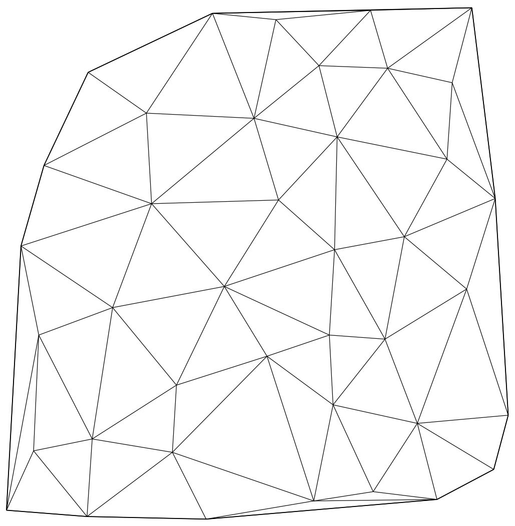


Figure 3.3.1 Triangulation after merge.

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

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

(2) *Vertices with minimum (maximum) y value in the left and right triangulations always connect as cross edges.* This provides an initial guess for determining the lower common tangent (LCT) of the two triangulations (dashed line in fig. 3.3.2).

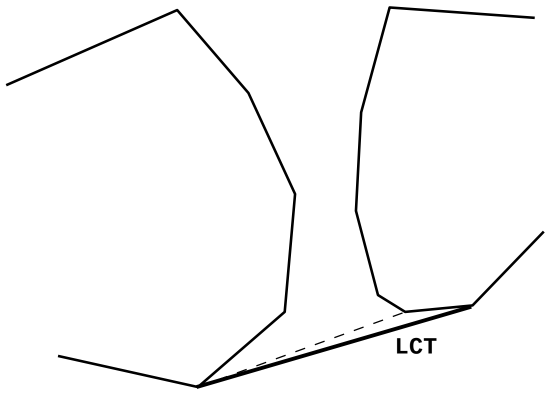


Figure 3.3.2 Calculation of the LCT given an initial guess.

Given a starting guess for the LCT, a traversal algorithm finds the LCT with overall time $O(N)$.

Given these properties and a LCT for the two triangulations, we now outline the "rising bubble" [GuiS85] merge algorithm. This algorithm produces cross edges in ascending y-order. The LCT forms the initial cross edge for the rising bubble algorithm. More generally consider the situation in which we have a cross edge between A and B and all edges incident to the points A and B with endpoints above the half plane formed by a line passing through $A - B$, see figure 3.3.3.

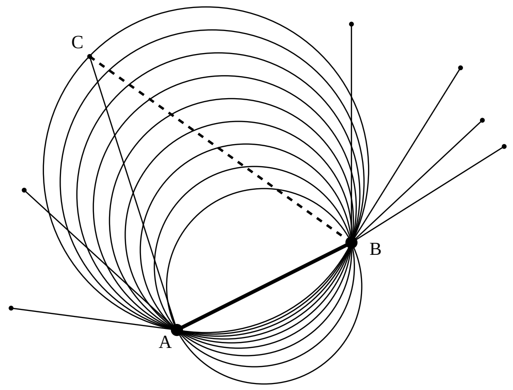


Figure 3.3.3 Circle of increasing radius in rising bubble algorithm.

This figure depicts a continuously transformed circle of increasing radius passing through the points A and B. Eventually the circle increases in size and encounters a point C from the left or right triangulation (in this case, point C is in the left triangulation). A new cross edge (dashed line in figure 3.3.3) is then formed by connecting this point to a vertex of $A - B$ in the other half triangulation. Given the new cross edge, the process can then be repeated and terminates when the top of the two meshes is reached. The deletion of $L - L$ or $R - R$ edges can take place during or after the addition of the cross edges. Properly implemented, the merge can be carried out in linear time, $O(N)$. Denoting $T(N)$ as the total running time, step 2 is completed in approximately $2T(N/2)$. Thus the total running time is described by the recurrence $T(N) = 2T(N/2) + O(N) = O(N \log N)$.

3.3b Incremental Insertion Algorithms

For simplicity, assume that the site to be added lies within a bounding polygon of the existing triangulation. If we desire a triangulation from a new set of sites, three initial phantom points can always be added which define a triangle large enough to enclose all points to be inserted. In addition, interior boundaries are usually temporarily ignored for purposes of the Delaunay triangulation. After completing the triangulation, spurious edges are then deleted as a postprocessing step. Incremental insertion algorithms begin by inserting a new site into an existing Delaunay triangulation. This introduces the task of point location in the triangulation. Some incremental algorithms require knowing which triangle the new site falls within. Other algorithms require knowing *any* triangle whose circumcircle contains the new site. In either case, two extremes arise in this reguard. Typical mesh adaptation and refinement algorithms determine the particular cell for site insertion as part of the mesh adaptation algorithm, thereby reducing the burden of point location. In the other extreme, initial triangulations of randomly distributed sites usually require advanced searching techniques for point location to achieve asymptotically optimal complexity $O(N \log N)$. Search algorithms based on quad-tree and split-tree data structures work extremely well in this case. Alternatively, search techniques based on graph traversal or "walking" algorithms are frequently used because of their simplicity. These methods work extremely well when successively added points are close together. The basic idea is start at the location in the mesh of the previously inserted point and move one edge (or cell) at a time in the general direction of the newly added point. In the worst case, each point insertion requires $O(N)$ traversals. This would result in a worst case overall complexity $O(N^2)$. For randomly distributed points, the average point insertion requires $O(N^{\frac{1}{2}})$ traversals which gives an overall complexity $O(N^{\frac{3}{2}})$.

The second issue in complexity analysis concerns the number of structural changes required when each site is inserted. Unfortunately this is *highly* dependent on the order in which sites are inserted. In fact poor orderings can give rise to worst case situations requiring $O(N^2)$ structural changes. In section 3.3c we consider algorithms which randomize the *order* in which sites are presented for insertion. When coupled with the Green-Sibson technique, the resulting algorithm has expected $N \log N$ performance. As we will see, the technique also suggests a new randomized data structure for proximity query which we later exploit in a multigrid method for solving differential equations.

Bowyer's algorithm [Bow81]

The basic idea in Bowyer's algorithm is to insert a new site into an existing Voronoi diagram (for example site Q in figure 3.3.4), determine its territory (dashed line in figure 3.3.4), delete any edges completely contained in the territory, then add new edges and reconfigure existing edges in the diagram. The following is Bowyer's algorithm essentially as presented by Bowyer:

Algorithm: Incremental Delaunay triangulation, Bowyer [Bow81].

Step 1. Insert new point (site) Q into the Voronoi diagram.

Step 2. Find any existing vertex in the Voronoi diagram closer to the new point than to its forming points. This vertex will be deleted in the new Voronoi diagram.

Step 3. Perform tree search to find remaining set of deletable vertices $\mathcal{V}$ that are closer to the new point than to their forming points. (In figure 3.3.4 this would be the set $\{v_3, v_4, v_5\}$)

Step 4. Find the set $\mathcal{P}$ of forming points corresponding to the deletable vertices. In figure 3.3.4, this would be the set $\{p_2, p_3, p_4, p_5, p_7\}$.

Step 5. Delete edges of the Voronoi which can be described by pairs of vertices in the set $\mathcal{V}$ if both forming points of the edges to be deleted are contained in $\mathcal{P}$

Step 6. Calculate the new vertices of the Voronoi, compute their forming points and neighboring vertices, and update the Voronoi data structure.

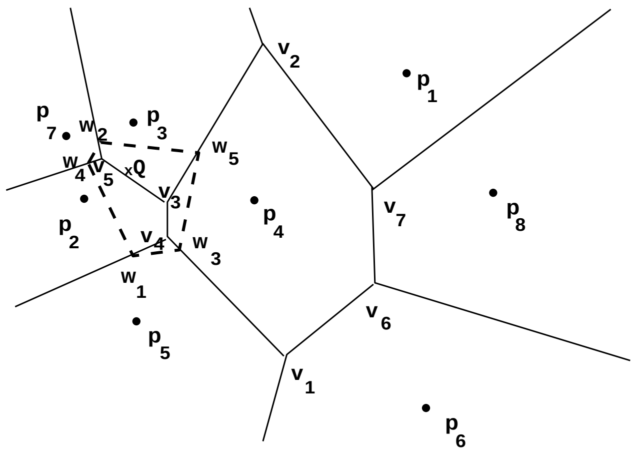


Figure 3.3.4 Voronoi diagram modified by Bowyer.

Implementational details and suggested data structures are given in the paper by Bowyer.

Watson's algorithm [Wat81]

Implementation of the Watson algorithm is relatively straightforward. The first step is to insert a new site into an existing Delaunay triangulation and to find *any* triangle (the root) such that the new site lies interior to that triangles circumcircle. Starting at the root, a tree search is performed to find all triangles with circumcircle containing the new site. This is accomplished by recursively checking triangle neighbors. (The resulting set of deletable triangles violating the circumcircle criteria is independent of the starting root.) Removal of the deletable triangles exposes a polygonal cavity surrounding site Q with all the vertices of the polygon visible to site Q. The interior of the cavity is then retriangulated by connecting the vertices of the polygon to site Q, see figure 3.3.5(b). This completes the algorithm. A thorough account of Watson's algorithm is given by Baker [Bak89] where he considers issues associated with constrained triangulations.

Algorithm: Incremental Delaunay triangulation, Watson [Wat81].

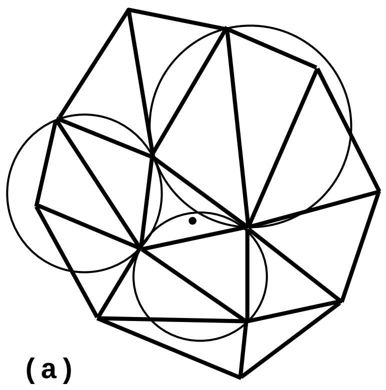
Step 1. Insert new site Q into existing Delaunay triangulation.

Step 2. Find any triangle with circumcircle containing site Q.

Step 3. Perform tree search to find remaining set of deletable triangles with circumcircle containing site Q.

Step 4. Construct list of edges associated with deletable triangles. Delete all edges from the list that appear more that once.

Step 5. Connect remaining edges to site Q and update Delaunay data structure.



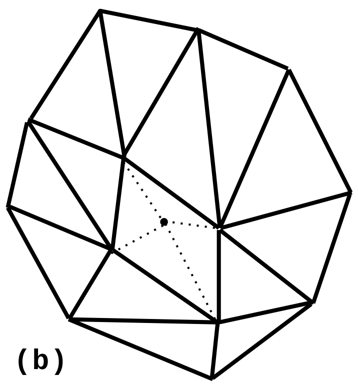
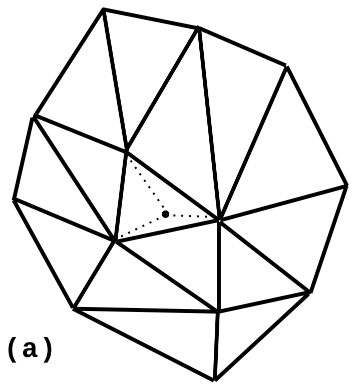


Figure 3.3.5 (a) Delaunay triangulation with site added. (b) Triangulation after deletion of invalid edges and reconnection.

Green and Sibson algorithm [GreS77]

The algorithm of Green and Sibson is very similar to the Watson algorithm. The primary difference is the use of local edge transformations (swapping) to reconfigure the triangulation. The first step is location, i.e. find the triangle containing point Q. Once this is done, three edges are then created connecting Q to the vertices of this triangle as shown in figure 3.3.6(a).



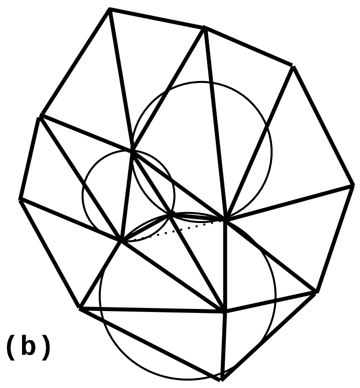


Figure 3.3.6 (a) Insertion of new vertex, (b) Swapping of suspect edge.

If the point falls on an edge, then the edge is deleted and four edges are created connecting to vertices of the newly created quadrilateral. Using the circumcircle criteria it can be shown that the newly created edges (3 or 4) are automatically Delaunay. Unfortunately, some of the original edges are now incorrect. We need to somehow find all "suspect" edges which could possibly fail the circle test. Given that this can be done (described below), each suspect edge is viewed as a diagonal of the quadrilateral formed from the two adjacent triangles. The circumcircle test is applied to either one of the two adjacent triangles of the quadrilateral. If the fourth point of the quadrilateral is interior to this circumcircle, the suspect edge is then swapped as shown in figure 3.3.6(b), two more edges then become suspect. At any given time we can immediately identify all suspect edges. To do this, first consider the subset of all triangles which share Q as a vertex. One can guarantee at all times that all initial edges incident to Q are Delaunay and any edge made incident to Q

by swapping must be Delaunay. Therefore, we need only consider the remaining edges of this subset which form a polygon about Q as suspect and subject to the incircle test. The process terminates when all suspect edges pass the circumcircle test.

The algorithm can be summarized as follows:

Algorithm: Incremental Delaunay Triangulation, Green and Sibson [GreS77]

Step 1. Locate existing cell enclosing point Q.

Step 2. Insert site and connect to 3 or 4 surrounding vertices.

Step 3. Identify suspect edges.

Step 4. Perform edge swapping of all suspect edges failing the incircle test.

Step 5. Identify new suspect edges.

Step 6. If new suspect edges have been created, go to step 3.

The Green and Sibson algorithm can be implemented using standard recursive programming techniques. The heart of the algorithm is the recursive procedure which would take the following form for the configuration shown in figure 3.3.7:

```
procedure swap[ $v_q$, $v_1$, $v_2$, $v_3$, $edges$]
if($incircle[v_q,v_1,v_2,v_3]$ = TRUE)then
    call reconfig_edges[$v_q$, $v_1$, $v_2$, $v_3$, $edges$]
    call swap[$v_q$, $v_1$, $v_4$, $v_2$, $edges$]
    call swap[$v_q$, $v_2$, $v_5$, $v_3$, $edges$]
endif
endprocedure
```

This example illustrates an important point. The nature of Delaunay triangulation guarantees that any edges swapped incident to Q will be final edges of the Delaunay triangulation. This means that we need only consider *forward propagation* in the recursive procedure. In a later section, we will consider incremental insertion and edge swapping for generating non-Delaunay triangulations based on other swapping criteria. This algorithm can also be programmed recursively but requires *backward propagation* in the recursive implementation. For the Delaunay triangulation algorithm, the insertion algorithm would simplify to the following three steps:

Recursive Algorithm: Incremental Delaunay Triangulation, Green and Sibson

Step 1. Locate existing cell enclosing point Q.

Step 2. Insert site and connect to surrounding vertices.

Step 3. Perform recursive edge swapping on newly formed cells (3 or 4).

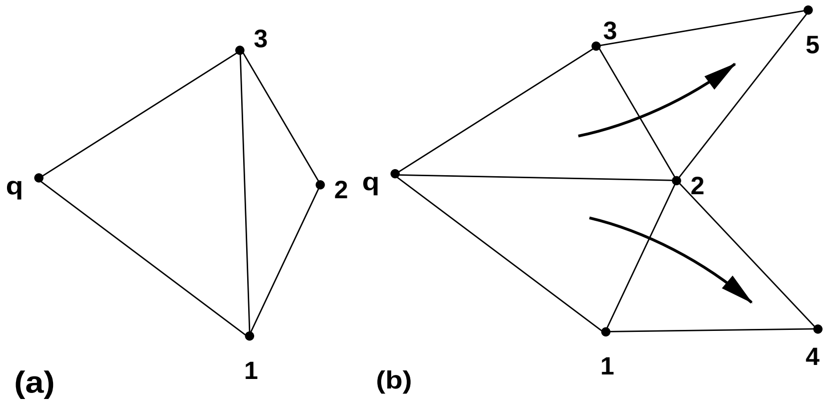


Figure 3.3.7 Edge swapping with forward propagation.

3.3c Randomized Algorithms [GuiKS92]

As mentioned in the introduction to incremental insertion methods, special configurations of sites can lead to $O(N^2)$ structural changes. Figure 3.3.8 shows one such example. This quadratic behavior can be eliminated with extremely high probability by randomizing the order in which sites are triangulated.

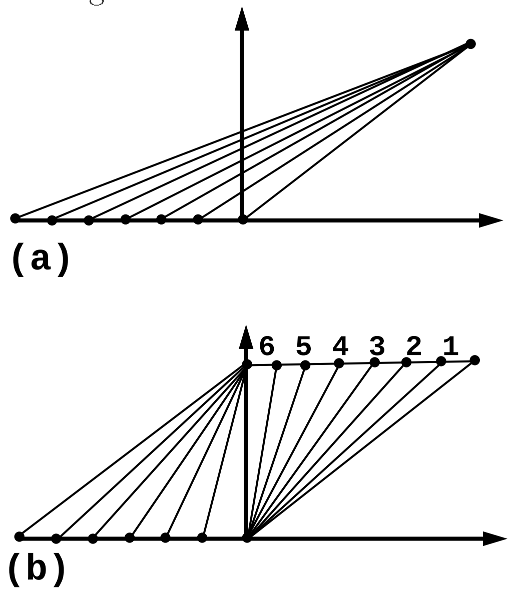


Figure 3.3.8 (a) Initial triangulation, (b) final triangulation. Difficult case which produces a quadratic number of structural changes when sites $1-6$ are added from right to left.

Define the scope of a triangle $w(\triangle)$ to be the number of sites inside the circumcircle of $\triangle$. Let T_j be the number of triangles of scope j.

$$Probability[\triangle \text{ appears as Delaunay}] = \frac{3!j!}{(j+3)!}$$