Triangular Tessellation Documentation

by

J.A. Galt

Genwest Systems Inc

P. O. Box 397

Edmonds, WA 98020

April, 2011

Acknowledgement:

Partial funding for this report provided by NOAA's Office of Response & Restoration

Abstract

Triangular tessellation has the advantage that it is able to deal with arbitrary spatial data sets such as the bathymetric charts that are typically available and can be generalized with respect to multiply connected topological domains.

For more then three decades the NOAA Hazardous Materials group have been using triangular grids to interpolate and integrate pollutant transport equation as part of their modeling efforts. During that time a number of different approaches have the used depending on the computers available and demands of the particular response.

About 20 years ago a very fast, compact and robust algorithm was described by Knuth. This was implemented in a strict Delaunay sense for the Gnome Analyst program, but never included in the trajectory or hydrodynamic routines because it did not handle the constrained Delaunay problem. Thus the present suite of triangle algorithms used by HAZMAT is a mix that is limited by spatial resolution issues or less than optimal run times.

Recent advances in available software and a better understanding of the constrained Delaunay problem has lead to a more general formulation of Knuth's original algorithms and this report describes this and presents a number of examples related to HAZMAT's standard procedures. The author recommends its use in HAZMAT's modeling program.

Table of Contents

Introduction

For three and a half decades the HAZMAT modeling program has used arbitrarily spatial data to define realistically complex geophysical domains and distributions of oil particles modeled as Lagrangian Elements (LEs). This work started out on "personal computers" the size of an old telephone booth, then migrated to main frames over 300 baud terminals and phone lines. Many generations of DEC, Apple, IBM, CDC, and modern personnel computers have seen service in this effort. Software options have been equally diverse including machine languages, Pascal, Fortran, C, C++, Python, and Java. By the late seventies both functional and operational models were available and have been used since that time. These models covered both basic hydrodynamic and oil trajectory estimates and have been continually studied and improved by an active research effort, while maintaining a robust response capability. It may seem like the procedures of interpolating between arbitrary points using triangles is not that complex or difficult a problem. This is naïve. The constraints associated with multi-connected boundaries added to the arbitrarily placement of the data points will introduce many subtle and difficult problems to the mix. The progress of triangle routines in HAZMAT has represented a continual attack and slow stepwise improvement in the performance of the trajectory analysis routines. The major focus of this work is about the introduction of new analysis techniques first published by Knuth (1992), which could only be partially implemented due to software language limitations. In 2002 I wrote a small (C++) extension library that made the implementation of these procedures possible but various programmatic constrains halted these studies at the software testing phase. Now modern object oriented languages support in native code the features needed to implement Knuth's procedures. In addition I have added modifications to handle some of the complex boundary issues. These algorithms are fast and robust and should be migrated into the suite of HAZMAT trajectory routines.

Basics

In environmental sciences it is often necessary to refer to fields. These may be scalar fields such as temperature or topographic elevation or vector fields such as wind or currents. These are invariably thought of as defined over spatial domain and may also be extended through time. In functional form these would be written as:

$$T = T(x, y) \tag{1}$$

$$\vec{V} = \vec{V}(\vec{r}, t) \tag{2}$$

In reality these continuous fields are never available and the scalar or vector fields are instead specified at a discrete set of locations which are distributed is such a way that the fields can be inferred. Typical sets might be:

$$t_i \in T | t_i = f(x_i, y_i) \wedge x_i, y_i \in G \tag{3}$$

Where we mean that t is an element in the range of T and x,y are elements in the domain G, f is a relationship f:G=>T. Since the elements of the set T are finite and discrete the problem remains of how to infer the field and what sort of properties it should have. The rest of this work will be trying to address this problem.

For the moment we will only consider scalar fields. (Vector fields can always be broken into scalar components so this is really not a restriction.) We will assume that we have discrete set of values each of which is associated with a point in space. We will also assume that the function representing this association is single valued. No point in space can have more then one scalar value. So now the question is one of interpolation. How can we fill in the field values between points? There a clearly lots of ways we might go about this and considerations of computational ease and smoothness of the resulting field are obviously of interest. The computational ease is important because we may be dealing with many thousands of points of data. The smoothness of the resulting field is of interest because they ofter go into differential or integral equation so that C0, C1 or C2 continuity may be essential.

Perhaps the simplest interpolation scheme might be to consider any point in the field not exactly on a point that is know but close to it should have a value equal to the know point. This would be a “nearest neighbor” strategy. For every point in the field find the know point closest to it and set the field value to that. To make this work we just need to partition the field domain into non-overlapping sub domains such that each subdomain represents the points closer to a unique known value then any other. This can be done easily using geometric consideration as can be seen in Figure 1. For any distribution of points draw a line from a test point to all of the near by points leading to a multi-armed star like pattern. For each of these arms construct a perpendicular bisector mid way along it length. Now construct similar bisectors for all the other arms extending out from the test point. From all the bisectors pick the minimum polygon that surounds the

test point. This polygon gives the sub-domain that "belongs" to the test point. Points within it are closer to the test point then to any other point in the data set. Repeating this process for all of the points in the domain will define all of the required sub-domains. This partition that works for arbitrary sets of points and produces what looks a bit like a "turtle back" pattern is referred to as a Voronoi diagram. The individual polygon associated with each point is a "Thessian Polygon". Since it works for arbitrary sets and quick algorithms are available to calculate them a Voronoi diagram has a number of computational uses. One example might be that you require a regional integral of a field over a domain. The natural differential area associated with each of the know data set values would be the Thessian polygon areas from the Voronoi diagram.

$$A = \iint t\, dx\, dy \approx \sum (t_i (Varea)_i) \tag{4}$$

As another example: Point data might represent Lagrangian information. (Trees in a forest from photographic analysis, or tons of oil from a spill trajectory analysis.) But what is desired is Eulerian information. (Find the lumber potential is board feet/acer, average oil thickness per. Meter squared.) In these cases the appropriate dimension change requires a simple constant and division by the Thessian polygon or Voronoi area.

$$L_i(\,point\ data) \rightarrow \frac{cL_i}{(Varea)_i} \tag{5}$$

When considering a Voronoi diagram a obvious question comes up about the area associated with a data point that is on the outer edge of a cluster of data points. It potentially could be the closest point to the semi-infinite half place. This of course is a special case that can easily occur in real geophysical data sets and must be addressed as will be seen later in this discussion.

The surface represented by a Voronoi diagram can partition a spatial region spanned by an arbitrary set of data points and associate an area weight with each point but as an interpolation scheme it suffers from having a discontinuity as each of the polygon edges. If we think of the nearest neighbor interpolation each of the polygons has the height of its data point, but there is a step at each edge. It is only C0 continuous so it is integrable but has many places where derivatives are not defined. To move to an interpolation scheme with a higher level of continuity we can consider it topological dual "Delaunay triangles".

Delaunay triangles are generated by considering a one-to-one correspondence between

the points where individual polygons of a Voronoi diagrams meet (three will always meet together) and the three data points nearest to that junction. See Figure 2. The junction points of three Voronoi polygons will in fact fall at the centroid of the defined Delaunay triangle and thus a circle could be drawn (centered at the junction point) which will go through each of the three nearest neighbor points. In addition none of the other data points from the set will be inside that "nearest neighbor" circle. This in fact defines the set of Delaunay triangles. Formally we have:

$$j_i \in V \wedge t_i \in D | f(j_i) \rightarrow t_i \tag{6}$$

Where $_{it}$ is a junction point in the set V of Thessian polygons of a Voronoi diagram and t_i is a triangle in the set D of Delaunay triangles. f(v,) is then a bijective mapping of V onto D. Thus the sets contain exactly the same information and each can be derived from each other.

Now let consider an interpolation scheme that uses our same arbitrary spaced data points but in this case consider proximal neighborhoods in terms of Delaunay triangles instead of Vononoi diagrams. In this case all the space where we do not have data falls within a triangle defined by exactly three data points and no other know data is found within that triangle. This immediately suggest a local C1 continuous data representation of the form:

$$z(x, y) = ax + by + c \tag{7}$$

Where a,b and c are constants chosen so that (7) matches the known data points at the three vertices. In particular if triangle (i) has vertices values v1,v1 and v3:

$$\begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} v1 \\ v2 \\ v3 \end{bmatrix} \tag{8}$$

This gives a linear plate that is continuous over the triangle and since the edges will be straight lines that depend only on the know data values along that edge they will match the interpolated values from the neighboring triangle. Thus when all the Delaunay triangle plates are added together we will have a complete C1 continuous surface covering the convex hull defined by the distribution of the known data points. Now instead of the "turtle back" discontinuous interpolation surface we were dealing with using a Voronoi diagram and Thessian polygons we have a "geodesic dome" continuous interpolation surface representing the spatial domain. It is also significant that the Delaunay interpolation has no requirement to extend beyond the convex hull defined by

the data itself and thus is not subject to arbitrary decisions about guessing what the data is like beyond where it exists. Thus the undefined half-planes along the edge of the Voronoi diagram are no longer an issue.

It is often useful to define an interpolation function at the vertex or data point level rather then the triangle level. In particular (8) gives the interpolation surface in terms of the three vertices point data for a single triangle. An alternative might be to consider the multi-triangle region made up of all the triangles that hold a particular data point in common (See Figure 3) (Zienkiewicz,1971) and then come up with an interpolation function for that vertex or data value. For each of the these triangles we identify the central vertex as v0 at location x0,y0 and the remaining vertices v1 at x1,y1 and v2 at x2,y2. With these values we form the following component linear plate.

$$\begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_0 & y_0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{9}$$

This gives a function of the form:

$$z(x, y) = ax + by + c \tag{10}$$

Which equals 1 at the central vertex and 0 at the two outer vertices. Fitting functions of this form for each of the triangles shown in Figure 3 we have a piece-wise continuous interpolation representation of the domain that is influenced by the vertex or data point at v0. This composite is usually referred to as an "influence function" or "hat function" and is written as

$$\varphi_i(x, y) \cup a_{tri(j)} x + b_{tri(j)} y + c_{tri(j)} \tag{11}$$

Where the index "j" is over the sum of all the triangles that have vertex "i" in common. It is seen that the φ(x,y) functions are localized in space but not orthogonal. The first order linear interpolation can be written as:

$$z(x, y) = \sum_{i=1}^{nv} v(i) \varphi_i(x, y) \tag{12}$$

The φ(x,y) functions are little local representations of a particular vertex or data point and when they are all added together we get the geodesic dome shape that matches the data at all known points and varies linearly in the triangle areas between points. The interpolated surface covers the convex hull defined by the data and and does not assume any user estimated scaling information. The volume integral over the defined data region is easily calculated:

$$\int z(x,y)\,dx\,dy = \sum v_i \left(\sum \frac{Area_j}{3} \right) \tag{13}$$

Where "i" is a vertex value and "j" summed over the triangle areas that have that "i" as a common vertex. Not surprisingly the value given by (13) is the same value that would be given by (4) if the area of integration is confined to a portion of the convex hull defined by the data. (Thus avoiding the regions along the boundaries of the domain where the Voronoi diagram are not well defined.) Equation (13) also defines piece-wise continuous first derivatives so that terms in a 1st order partial differential equation can also be integrated over the interpolated region:

$$\int \frac{(\partial z(x,y))}{(\partial x)}\,dx\,dy = \sum v_i (a_j\, Area_j) \tag{14}$$

And:

$$\int \frac{(\partial z(x,y))}{(\partial y)}\,dx\,dy = \sum v_i (b_j\, Area_j) \tag{15}$$

Many of the partial differential equations that we wish to deal with are second order so it is worth while to see if we can integrate our interpolated surface's second derivatives. Since the linear plates that make up the geodesic dome surface are only first order and second derivates are zero in general as well as indeterminately at plate seams this appears to be an impossible problem. On the other hand we can assume that at some density of known data points the represented surface will approach a continuous (eventually approximating an analytic) representation. To proceed with this argument we can reconsider the representation of a generic field as seen in (12). In this form we see that the φ(x,y) functions represent a bases set that span the space of all possible geodesic domes given the spatial distribution of known points. The v(i)'s are the components of the φ(x,y) vector representation of the interpolated surface. Thus we could write a general geodesic dome surface for a given distribution of known points in vector form as:

$$S(x,y) = a_i \varphi_i \tag{16}$$

Where we will leave out the summation notation where it is obvious. With this vector notation it makes sense to define an inner product, for example:

$$\varphi_i \cdot \varphi_j = \iint \varphi_i(x,y)\,\varphi_j(x,y)\,dx\,dy \tag{17}$$

Looking back at Figure 3 we can see that equation (17) will evaluate to zero unless "i" and "j" are points within the same Delaunay triangle, so once again the φ(x,y) bases

functions are not orthogonal, but highly localized. With the introduction of shape functions the interpolation problem can be considered as the projection of data onto a spatially dependent linear vector space. Fields can be projected onto one another, differentiated, and defined as orthogonal, etc.

We note any data defined on a set of points can be envisioned as a field and represented as a continuous vector using "shape functions" dependent only on the Delaunay triangulation of those points. In particular we could define an unknown field by the unknown components of its projection onto the shape function space. As an example we might consider two empirical orthogonal functions defined from a time series of point set data.

$$E_1 = k_n^1 \cdot \varphi_n \tag{18}$$

$$E_2 = k_m^2 \cdot \varphi_m \tag{19}$$

Since these are known to be orthogonal we immediately have a constraint on the k's:

$$E_1 \cdot E_2 = \left[k_n^1 k_m^2\right]\left[\varphi_n \cdot \varphi_m\right] = 0 \tag{20}$$

The rows of the k's matrix are orthogonal to the columns for the φ's matrix, which are known, banded and sparse.

Moving to a more general case we will consider an unknown field (Ψ) constrained by a linear partial differential equation represented by the operator L().

$$L(\Psi) = 0 \tag{21}$$

And assume that Ψ can be approximated as an unknown vector in the now familiar φ space of our Delaunay triangles.

$$\Psi'(x, y) = s_i \varphi_i \tag{22}$$

Note that Ψ' will only be an approximation of Ψ since we have no reason to believe that our interpolation space will be complete enough to match all of the possible solutions that could be represented by (21). Keeping this in mind and substituting (22) into (21) will give:

$$L(s_i \varphi_i) = \epsilon(x, y) \tag{23}$$

Where ε(x,y) is the error field due to the geodesic dome's possible mismatch to the true solution. Obviously we would like to find a solution to the s_i's that will minimize ε(x,y) and in some sense lead to (22) being a good solution. One common option is to use the Galerkin method (Zienkiewicz,1971)(Oden and Carey,1983). Other options to minimize (22) are common and lead to a whole class of what are known as "weak" solutions (LeVeque,1992). This approach sets the inner product of each of the shape functions (φ's) with the error ε(x,y) equal to zero.

$$\varphi_i \cdot L(s_j \varphi_j) = \left[\varphi_i \cdot L(\varphi_j)\right] s_j = 0 \tag{24}$$

As we can see this lead to a matrix equation where the matrix is defined in terms of the bases shape functions and the linear operator on the same set. We have turned a partial differential equation into a linear algebra problem represented by the spatial distribution of our data space. In simple terms we force the error to be orthogonal to our geodesic dome function space. This means that no changes in any of the s_i component amplitudes of the φ's can make the error smaller. Note that this does not say that the error is "small" or bounded in a classical δ-ε proof sort of way. It does however say that given the spatial distribution of the Delaunay triangles and the assumption of a geodesic dome interpolation in the φ space, this is as good as we can get. It is a small, but significant leap of faith to expect that as the point density goes up and the Delaunay triangles become smaller the error term will approach zero at least in a integral least-squares sense. We will always make this assumption.

As a final example in this section we will consider a linear operator of 2nd order and the geodesic field approximating it solution. This will get us back to the problem of how to define second derivatives on a φ space based on Delaunay triangles. Given:

$$L(\Psi) = \frac{\partial^2 \Psi}{\partial x^2} + \frac{\partial^2 \Psi}{\partial y^2} = 0 \tag{25}$$

And assuming our field approximation and a Galerkin constraint we have:

$$\left[\varphi_i \cdot \frac{\partial^2 \varphi_j}{\partial x^2} + \varphi_i \cdot \frac{\partial^2 \varphi_j}{\partial y^2}\right] s_j = 0 \tag{26}$$

To handle the 2nd derivatives in (26) we can carry out an integration by parts over the x,y domain. From the first term we get:

$$\varphi_i \cdot \frac{\partial^2 \varphi_j}{\partial x^2} = \iint \left[-\left(\frac{\partial \varphi_i}{\partial x} \frac{\partial \varphi_j}{\partial x}\right) + \frac{\partial}{\partial x}\left(\varphi_i \frac{\partial \varphi_j}{\partial x}\right) dx \right] dy = \iint \left[-\left(\frac{\partial \varphi_i}{\partial x} \frac{\partial \varphi_j}{\partial x}\right) \right] dx\, dy \tag{27}$$

Which simplifies to:

$$\varphi_i \cdot \frac{\partial^2 \varphi_j}{\partial x^2} = \iint \left[-\left(\frac{\partial \varphi_i}{\partial x} \frac{\partial \varphi_j}{\partial x} \right) \right] dx\, dy + \int \left[\varphi_i \frac{\partial \varphi_j}{\partial x} \right]_{x1}^{x2} dy \tag{28}$$

Going through the same procedure for the second term in (26) gives:

$$\varphi_i \cdot \frac{\partial^2 \varphi_j}{\partial y^2} = \iint \left[-\left(\frac{\partial \varphi_i}{\partial y} \frac{\partial \varphi_j}{\partial y} \right) \right] dx\, dy + \int \left[\varphi_i \frac{\partial \varphi_j}{\partial y} \right]_{y1}^{y2} dx \tag{29}$$

Add (28) and (29) and using Stokes theorem in the plane to combine the single integrals we get

$$\left[\varphi_i \cdot \frac{\partial^2 \varphi_j}{\partial x^2} + \varphi_i \cdot \frac{\partial^2 \varphi_j}{\partial y^2} \right] s_j = \left[\iint \left[\left(\frac{\partial \varphi_i}{\partial y} \frac{\partial \varphi_j}{\partial y} \right) + \left(\frac{\partial \varphi_i}{\partial y} \frac{\partial \varphi_j}{\partial y} \right) \right] dx\, dy + \oint \varphi_i \frac{\partial \varphi_j}{\partial \vec{\boldsymbol{n}}} \cdot d\vec{\boldsymbol{s}} \right] s_j = 0 \tag{30}$$

We see that the problem of 2nd order terms in a linear operator can be finessed into into a 1st order term plus a constraint from the boundary of the domain. The boundary term is just the product of the field times its normal derivative integrated along the boundary. In many cases this product evaluates to zero (which is usually referred to as the "natural boundary conditions") and the line integral can be dropped. The reduced solution with the field or its normal derivative set to zero along the boundary is referred to as a "weak solution" (Zienkiewicz,1971)(LeVeque,1992).

All of the hydrodynamic routines used in CATS are based on formulations starting with (24). The concepts described in this section have numerous applications familiar to all of us through our spill response activity.

A Short History of HAZMAT Triangles

In the mid-seventies before the actual formation of HAZMAT the modeling group at PMEL (Which eventually got moved into HAZMAT after it grew out of the OCEAP program..) realized that a triangular grid system was the only easily automated procedure that could use available arbitrarily-spaced bathymetric data to define geophysically complex domains. (Rasterized grids required additional interpolation schemes and resulted in saw tooth edges which caused hydrodynamic problems and conformal mapping was labor intensive and hard to generalize.) Our first Delaunay triangulation scheme was run on a personnel computer. It was a PDP-8 (about the size of a phone booth) and the personnel property of a NOAA Corp officer who was working

in the group. The algorithm started with a collection of points and drew lines from each point to all the other points. It then considered each pair of lines and if they crossed it discarded the longest of the pair. It was an $O(N^4)$ routine and for a couple of hundred points ran in about 4 days.

I tried to speed things up and by sequentially adding points and only considering lines generated in the new triangles (iterating on neighboring triangles) and came up with a $O(N^2)$ method and since we had migrated to the use of a remote access CDC mainframe in Boulder this ran fast enough to be operational. The major problem with this algorithm was that adding the boundaries to the convex hull required a reorientation of the triangles making them non-Delaunay. This turns out to be a decidedly non-trivial problem. Glen and I wrote dozens of routines each expected to finally solve the "triangle" problem, and with Debbie watched them fail (usually in the middle of the night during a response) because of some unanticipated special case. Eventually the routine got better and we were able to have an algorithm that was $O(N^{1.5})$ and crashed infrequently enough to be useful.

We then transitioned away from a remote mainframe back to desk top computers. In Pascal the triangle routine ran on an APPLE-II for a couple of hundred points in 3 hours. Glen went into the algorithm and took it apart rewriting it in machine language for the Motorola chip and dropped the time to 3 minutes. This made us operational again. This also took us into a later IBM-9000 which also used the same Motorola chip.

The next generation of triangle algorithms came when I ran into some finite element pieces that Boeing was using for flow code development. This algorithm started by putting all of the boundary segments into a list. Then for each segment in the list the interior point making the most "equilateral" triangle was used to create a new triangle. The original side of this triangle was deleted from the list and the and the two new sides were added into the list. Processing continued until the list was empty. Anyone watching the generation of triangles in the present version of CATS will recognize this algorithm at work.

The implementation of Knuth's algorithms were first tested in the early 90's and although they were extremely fast and nearly bullet proof they suffered from two drawbacks. The first was that (I thought – which turned out to be wrong) they could not easily be modified to handle boundaries (non-Delaunay constraints) and secondly that scaling of the coordinate space was required to deal with integer word lengths that were standardly available at the time. As a result of these limitations the DagTri routines

were only used in GNOME-Analyst and due to the boundary problems occasionally produced Splot contours under islands. The restrictions on Knuth's methods are now gone and based on their efficiency a re-evaluation of all the HAZMAT algorithms is in order. The actual tessellation algorithm that implements Knuth's method is composed of a few separate method and data structors that work with each other and are easily represented as object oriented classes. The following sections describe these classes as implemented in the DagTri algorithm.

Incircle Predicate

The fundamental algorithm used in DagTri is Incircle and it is implemented as a Java class. The class actually evaluates two related, but independent boolean predicate depending on the parameters given to its creator. All of the Incircle class objects methods and variables are "static". It is a function used to evaluate the predicates.

CounterClockWise (CCW) predicate

Given any three points in the plane v1(x,y),v2(x,y), and v3(x,y) where the x,y coordinate system if right handed. (See Figure 4.) Consider a line segment from v1 to v2 seen in the figure as vector A. If v3 is in the left half plane of the extension of the vector A then the CCW predicate returns true, otherwise it returns false. Thus if the vertices v1,v2 and v3 are presented in an order that appears to rotate in a counter-clock-wise fashion we have CCW=true. The algorithm evaluates the sign of the following determinate:

$$sign(det)=\begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix}; + \rightarrow true; - \rightarrow false \tag{31}$$

The magnitude of the determinate in (31) is twice the area of the triangle formed by the three vertices. The meaning of the sign is easily seen by a change of coordinates where we move the origin to (x_1,y_1). We then have:

$$sign(det)=\begin{bmatrix} 0 & 0 & 1 \\ (x_2-x_1) & (y_2-y_1) & 1 \\ (x_3-x_1) & (y_3-y_1) & 1 \end{bmatrix}; + \rightarrow true; - \rightarrow false \tag{32}$$

And we see that the sign of the determinate is the same as the vector cross product of

vectors A and B:

$$sign(det) = sign(A \times B) = sign((x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)) \quad (33)$$

And the sign is determined by the familiar "right hand rule". So far this seems like a straight forward algorithm and in an ideal world given any three vertices in the plane we expect CCW to return either true or false. Given infinite precision and no duplicate points this would be almost always true, yet it is the case where CCW returns zero (the points are co-linear or duplicate) that would typically require a great deal error checking code to figure out. On the other hand our application zero returns from CCW could be interpreted as either true of false (they are very close anyway) as long as it was absolutely guaranteed that the answers are consistent and repeatable. This leads to the idea of simplicity (Edelsbrunner and Mucke, 1988) (Edelsbrunner,1992). To implement "simplicity" we require:

- Points have some ordering relationship such that for all v1 and v2 either v1(index)<v2(index) or v1(index)>v2(index). In our case vertex coordinates are stored in an array and the array index serves for ordering.
- The mathematical evaluation of the predicate (33) is exact. In our case we will use Latitude and Longitude in micro degrees (corresponding to a displacement on the earth of < 11.1 cm) so any place on earth will be 10 decimal digits or less. Since the CCW predicate is 2nd order in coordinate values we must have 20 decimal digits available for arithmetic values.
- That the evaluation of the predicate can be represented by a Taylor series expansion around ambiguous value in terms of the virtual displacements for the independent variables.

We now proceed by defining the predicate as the a property of a function as in (33) and then expand it as a Taylor series in "virtual displacements".

$$sign(det) = f(x1, y1, x2, y2) = f_0 + \frac{\partial f}{\partial x1} dx1 + \frac{\partial f}{\partial y1} dy1 + \frac{\partial f}{\partial x2} dx2 + \frac{\partial f}{\partial y2} dy2 \quad (34)$$

If f_0 evaluates to non-zero we have our answer. If not we order the points noting that every time we switch rows in the original matrix order (31) we toggle the sign in the CCW predicate. We then check the sign in the 2nd term in the Taylor series, etc. The first non-zero term is return as the CCW predicate. If all of the terms in the expansion are zero it means that all three vertices have the same x,y coordinates (highly degenerate case) and we assume a positive value. We now always get a simple answer from CCW for any three vertices the answer will always evaluate to true or false, there are no

exceptions.

Figure 5 shows test output from the CCW predicate algorithm. The BLACK point is v1 and the YELLOW point is v2. Then random points are generated and plotted, GREEN if CCW is true and RED if CCW is false. Time test on a small lap top indicate about 1,000,000 evaluations per second for large numbers of points and 10 digit coordinate values.

At this point it might not be obvious how the CCW predicate is necessary or even useful in the triangular tessellation problem, so as a simple example consider the line cross problem.(Sedgewick,1992) Given two lines defined by their end vertices: line 1 (with v1 and v2), line2 (with v3 and v4). Then:

- boolean check1=CCW(v1,v2,v3)
- boolean check2=CCW(v1,v2,v4)

If check1=check2 then the lines can't intersect. If check1!=check2 then:

- boolean check3=CCW(v3,v4,v1)
- boolean check4=CCW(v3,v4,v2)

If check3=check4 the lines still don't cross, otherwise then do.

Incircle Predicate

As was mentioned above a Delaunay triangle is defined for a set of vertices as made up of three vertices such that a circle whose origin is at the centroid and drawn through the three vertices contains no other vertices from the set. The Incircle predicate takes four vertices as input variables. The first three are assumed to form a triangle with points in counter-clocks-wise order. The forth point is a test vertex. If the test vertex falls within the circle then it returns true, otherwise false. To see how this relates back to Incircle see Figure 6 . For any pair of adjacent triangles if there common side does not divide them into a Delanuay pair the four points entered into Incircle is counter-clocks-wise order will return true. Reorienting the common side will then create a Delaunay pair.

The algorithm to evaluate Incircle is given in Knuth (1992) and is expressed as the sign of a determinate.

$$Incircle = (sign)\, det \begin{bmatrix} x_p & y_p & (x_p^2 + y_p^2) & 1 \\ x_q & y_q & (x_q^2 + y_q^2) & 1 \\ x_r & y_r & (x_r^2 + y_r^2) & 1 \\ x_s & y_s & (x_s^2 + y_s^2) & 1 \end{bmatrix} \quad (35)$$

This is treated in the same way that the CCW predicate was. In this case the functional form is a bit more complicated. The predicate evaluation will require 4th order in coordinate values so invoking "simplicity" will require exact 40 digit decimal arithmetic to operate at the micro degree level of geophysical resolution.

The Incircle class can be run with test data and the results are seen in Figure 7 . In this case 3 vertices were randomly picked so that they resulted in CCW being true BLACK,YELLOW, and BLUE. Then random test points were selected and if Incircle returned true they were plotted in GREEN and if false in RED. Large numbers of vertices were evaluated at about 250,000 cases per second on a small lap top.

Incircle Class Public Interface

The Incircle class actually processes both the CCW and Incircle predicates. It includes a static public class constant INF and only one static public method (Incircle.test). Calls can be made to:

- boolean ccw= Incircle.test(int Incircle.INF,int q,int r,int s,long[] x,long[] y)
- boolean inCir=Incircle.test(int p,int q,int r,int s,long[] x,long[] y)

In these calls p,q,r and s are index values into long integer arrays (x and y) that hold the latitude and longitude coordinates in micro degrees (projected onto a flat earth – assumed cartesian)

If the first argument is the defined constant Incircle.INF then q,r and s are taken as arguments to CCW and the CCW predicate is returned.

If the first argument is an proper index into the x[] and y[] arrays (-1<p<#of_vertices) then q,r and s are also assumed to be index values and the Incircle predicate is returned.

Arcs and Nodes

The tessellation procedure that DagTri uses builds several data structures that describe both the building process as it incrementally add vertices and the resulting topology

once the process is completed. The relatively complex nature of these structures contribute to the speed of the algorithm and apparent complexity. With out a good understanding of these structures the algorithm is difficult to follow and appears somewhat mystical.

Arc

An arc a single directional side of a triangle. It is represented as an object of class Arc. The object has three data elements each of which is an int index.

- Arc.vert is an int variable that gives the index of the vertex in a ccw direction along the side represented by the arc.
- Arc.next is an int variable that gives the index to the arc for the next triangle side in a ccw direction.
- Arc.inst is an int variable that is an index to the Node within the Dag tree structure that points back to and locates this triangular.

The three arcs that define a triangle are shown in Figure 8. The reference back to the Node will be explained later, but for the present we can note that all three arcs that are associated with a single triangle point to the same Node. The next variable makes the arcs of a triangle cyclic, that is: A=((A.next).next).next. Another important feature of the collection of Arcs is that they are stored in a array of type arc[]. Each triangle is represented by three arcs (not necessarily stored sequentially) but with the "next" variable they can be accessed like a linked-list for fast navigation. There is another topological relationship between arcs that is also important. Along each side of the triangle there is an adjacent triangle with its associated three arcs, also in ccw order (for that triangle). Thus each triangle side is represented by two arcs that point in opposite directions (both ccw in their respective triangles). When ever a new side is created two new arcs are added to the arcs array. They are always stored such that one is indexed from the beginning of the array and the other is indexed from the end of the array. Any modifications of the arcs during the DagTri building process must maintain this relationship. This relationship makes it possible to also navigate quickly from any triangle to its neighbors and serves as another special kind of linked-list. A special function (mate(arc (k)) in the DatData class will return the index of the arc of the neighbor.

Arc Public Interface

The public interface to the Arc class is particular simple and it is really not likely that

host users would ever need to use Arcs directly. None the less there are two public creators for the class:

- Arc myArc= new Arc() which creates a empty arc (place holder)with all variable fields zero.
- Arc myArc= new Arc(int vert,int next,int inst) which creates an arc with the values specified in the parameter list.

Nodes

Nodes are the data structures that are used in the DAG tree that is built while the tessellation process is developed. It is basically a binary tree and supports ISOS (in-side-out-side) determination for the decision of where a test point goes into the structure and is largely responsible for the overall algorithms speed. This structure is a class object and has 4 attribute fields. The meaning of these fields depend on whether the node is a “directional - binary node” or a “terminal - leaf node”.

For a directional node the attribute fields have the following meanings:

- Node.p is an int index to the 1st vertex of a dividing line
- Node.q is an int index to the 2nd vertex of a dividing line (together these two vertices separate the coordinate space into two half planes defined by CCW)
- Node.ln is an int index to a Node associated with the left (CCW) half plane determined by the first two attributes.
- Node.rn is an int index to a Node associated with the right (!CCW) half plane determined by the first two attributes.

For terminal nodes the attribute fields have the following meanings:

- Node.p is the defined constant Incircle.INF as a flag to identify this as a terminal node
- Node.q is an int index to an Arc that references a particular triangle.
- Node.ln is not used.
- Node.lr is not used.
- Node.area is a double and holds the Voronoi area (if it is set)

There is a small undocumented subtlety associated with the q field in terminal nodes. If it references an interior triangle then it can point to any of the three arcs that define the triangle and of course all of the arc's inst field will point back to this terminal node. If on the other hand the q field references a triangle outside of the data sets convex hull

then one of the arcs will have a vert field that references Incircle.INF (a vertex at infinity) and the node's q field must reference that arc rather then one of the other two. All the triangle arcs still point back to the same terminal node. In the algorithm the arc's vert field is used as a flag in a branch structure. This is the sort of detail that makes the code fast, but difficult to follow.

The number of Nodes used in DagTri is not known until run time so they should be stored in a dynamic-collection structure. In this version an ArrayList<Node> is used. We can see that the directional nodes form a binary tree (whose depth will be ~ O(lnN)) which can be navigated using CCW. In addition the terminal nodes are part of a two way linked-list between the individual triangles and the arcs. The relationship between Arcs and Nodes is shown in Figure 9. The DagTri topology is tightly bound together be these two structures via a series of linked-lists. These details all contribute to the making the algorithm both fast and compact.

Nodes Public Interface

The public interface to the Node class is again very simple and it is really not likely that host users would ever need to use Nodes directly. None the less there is a public creator for the class:

- Node myNode= new Node(int p,int q,int ln,int rn)

DagData

The DagData class is a general “blackboard” for the DagTri tessellation routine. It contains the array structures that define coordinate data and a number of additional parameters that describe boundaries, bounding box and scaling data to support graphics and auxiliary fields for interpolation. In addition it contains utility methods and internal classes for interpolation and contour generation. The DagData class is a work in progress and will ultimately be the repository of most of the utilities associated with this package.

DagData Parameters

The class parameters for DagData are:

- vertex point data

    - DagData.nv is an int which is the number of vertices used in the tessellation.
    - DagData.x is a long[] array of x-coordinates for vertex data (in micro-degrees).
    - DagData.y is a long[] array of z-coordinates for vertex data (in micro-degrees).
    - DagData.z is a double[] array of z-values associated with vertex data.
    - DagData.vA is a double[] array of areas associated with the Voronoi diagram or Thessian polygon for the vertices.
- boundary data
    - DagData.bSeg is an int[] array that defines the last vertex of each of the segments of the vertex list that represent boundary segments given ccw order (standard HAZMAT topologicl convention)
    - DagData.arc is an Arc[] array that contains all of the arc structures maintaining the arc and mate(arc) relationships.
    - DagData.nodes is an ArrayList<Node> dynamic array of nodes that contains the DAG structure.
- scaling data
    - DagData.xmax is a long that gives the maximum x range of the vertex data.
    - DagData.xmin is a long that gives the minimum x range of the vertex data.
    - DagData.ymax is a long that gives the maximum y range of the vertex data.
    - DagData.ymin is a long that gives the minimum y range of the vertex data.
    - DagData.scale is a double that will scale the vertex data into a unit box
- auxiliary data structures for internal calculations
    - DagData.AreaSet is a boolean flag true if the Voronoi areas have been set.
    - DagData.bEdges is an ArrayList<Edge> dynamic array of boundary segment Edges.
    - DagDat.triNodes is an ArrayList<Integer> dynamic array that gives a quick link to the terminal nodes in the DAG tree.

DagData Public Interface

The public interface has a number of constructor that initialize the basic structors:

- DagData myDagData= new DagData(long[] x,long[] y) basic and set up the minimum structure ready for DagTri to use.
- DagData myDagData= new DagData(long[] x,long[],double[] z) sets up the basic structure and add an addition field (nominally depth) defined on the vertices.
- DagData myDagData= new DagData(long[] x,long[],int[] bSeg) sets up the basic

structure with the additional structures needed to enforce a multiply connected boundary on the tessellation.

- DagData myDagData= new DagData(long[] x,long[],double[] z,int[] bSeg) this constructor obviously extends the previous ones to set up the entire initialization.

Once the tessellation is complete an additional method can be called that performs several utility on the data.

- MyDagData.setArea_vA_AndTriISOS() as the name implies this method does a number of tasks. First of all it goes through the DAG tree and identifies all of the terminal nodes which represent triangles inside the convex hull of the data and calculates their area and stores the value (square micro-degrees) in the area field of the node. (It also adds the nodes index to the triNode list to facilitate future iterations on triangles.) It then checks to see if a bSeg has been defined. If the answer is yes then all triangles which are outside the boundary are assigned the negative of their area (thus any triangle with a negative area is outside the domain). Finally all of the positive area contributions to the Voronoi diagram are summed and entered in the vA array.

A helper Interpolater Class is provided that is created with a test point

- myInter= myDagData.new Interpolater(long x, long y)

The class has one method which will interpolate the value of a field of point data defined on the vertices in myDagData.

- double value= myInter.eval(double[] field) will return the value of the field interpolated to the point x,y

An additional helper ContourLine Class is provided that creates four ArrayLists<Long> sX,sY eX,eY which are the starting and ending line segments that make up a contour line using a field defined on the vertex points. Its only methods are a creator and a simple plot. It is used as follows:

- ContourLine C1= myDagData.new ContourLine(double[] field, double value) this will create a ContourLine object for the field defined on the vertex points of myDagData.
- C1.plot(Draw graphicWnd, Color penColor).

The following code fragment contours Gaussian plume that extends over a 100 kilometer box and which was then randomly sampled 50,000 times over the area of the box.

```
Draw wnd= new Draw("contour");
double val=1.0;
int i=0;
while(val>=1e-12){
        ContourLine C1= dat.new ContourLine(z,val);
        i++;
        if(i<=3)C1.plot(wnd, Draw.RED);
        if((i>3)&&(i<=6))C1.plot(wnd, Draw.GREEN);
        if(i>6)C1.plot(wnd, Draw.BLUE);
        val*=0.1;
}
```

The output from this is seen in Figure 10.

DagTri

Basic Algorithm

The DagTri class is the actual heart of the triangle tessellation program and directly implements the algorithm (Knuth,1992). This classes methods and utilities from DagData are what a host object or program will general be dealing with. The actual algorithm directly follows the procedures out lined by Knuth, section 18, p73-82. In Knuth's notes the algorithm is broken down into a number of steps and these are included as comments in the code.

The first step in the tessellation is to create a DagData class that contains as a minimum two long[] arrays of the geophysical point data that is going to be the bases for the Delauney triangles. This is then passed to the DagTri constructor which will carry out the initialization using the first two data points. For example:

- long[] x,y
- DagData mydagDat= new DagData(x,y)
- DagTri mydagTri= new DagTri(mydagDat)

At this stage the constructor has set up the Node and Arc structures shown in Figure 11. There are 3 Nodes and 6 Arcs. (Described in Knuth's notes in eq. 18.1)

- The zero node identifies the line segment from V0 to V1. Points CCW from this

line will lead the N1 and points !CCW will lead to N2

- Terminal node N1 which has a reference to arc a2 which is part of triangle a2 – a3 – a1 which all have “inst” back to N1.
- Terminal node N2 which has a reference to arc b3 which is part of triangle b3 – b1 – b2 which all have “inst” back to N2
- a1 – b1, a2 – b2, and a3 – b3 are all mates and stored indexed to opposite ends of the arcs array.
- Since the triangles referenced by N1 and N2 have vertices at INF the arc that is referenced in these Nodes is the one out of the triangle that contains the INF vertex as its “vert” field.

There needs to be a bit of discussion as to where the vertex at INF is. At first blush the plot in Figure 11 does not seem to make sense. The two identified triangles have vertex angles that add up to 440 degrees. The plot can't possibly be on a plane surface. It is not. The plot should be thought of a very small segment of a sphere and the point at INF is at the antipode. (half the circumference of the earth in any direction) The triangles are “spherical triangles” whose vertex angles will sum to something between 440 – 180 degrees. The area we are tessellating is “small” enough so that we assume the coordinates are cartesian and plane geometry ideas work fine, but in the larger sense we should think of this place as a sphere whenever we need to go to INF. When plotting any arc that references INF we simply extend the line of the arc before it.

This leaves us with a DagTri object initialized and ready to go. We are ready to enter the algorithm proper.

What Knuth identifies as Step T1: is to pick the next vertex for addition to the tessellation and using the DAG tree of Nodes and the test coordinates x,y identify which triangle it falls in. This is done with the DagTri method getTerminalNode() by the following call:

- int vertexIndex = next point in x[], y[] to be added
- Node targetNode=mydagTri.getTerminalNode(vertexIndex)

The node that is returned identifies the triangle that contains the point where the test vertex must be place.

Step T2: examines the target node and prepares additional place holder Arcs and Nodes. There will be 6 new arcs (one triangle will be replaced by three – net gain of 6 arcs) and 3 new terminal nodes for the triangles created and the old terminal node will have to be modified to reflect that it is no longer a terminal node. (Described in Knuth's notes in eq. 18.2). Two points were used in the initialization (Figure 11) which creates the two half planes that are made up of triangles with a vertex at INF.

The third point (no matter where it falls will be in one of these "wedge" triangles so the algorithm will go through a somewhat specialized loop and end up with one real triangle made up of the first three vertices (3 arcs) and one vertex at INF which adds three more triangle (3 arc each). It is easier to understand the actual algorithm if we pick up the options after the forth point has been added and the situation is as depicted in Figure 12 .

It is worth studying this Node and Arc structure for a moment and making sure you understand all the various links that tie them together. Notice that N1 is a "fossil" link in the sense that it references a connection between vertex 0 and vertex 2 which no longer represents and Arc but none the less still bifurcates the plane for a Ln(n) search for text point locations. The creation of fossil links will turn out to be of critical concern later in this study when we consider adding boundary constraints to the DagTri algorithms.

When adding the fifth point (vertex 4) and all subsequent points there are three different cases for determining what new arrangement of the Arc and Node structure will be required. These are shown in Figure 13 and Knuth's algorithm responds according to which case is represented by the new text point.

Case 1: For this case the point to be added falls with in an existing triangle with in the existing convex hull (bounding triangle). Three new triangles will be formulated to replace this bounding triangle In Knuth's notes this is clearly shown in the figure shown on page 76 and described in the algorithm as step T3.

Case 2: In this case the point to be added fall is a wedge triangle (one which has a vertex at INF) and is thus outside of the existing convex hull. When creating the three new replacement triangles two will have vertices at INF and the third will be a "normal" triangle and form an attachment to the existing convex hull, that can replace the existing

convex hull. In Knuth's algorithm this is described in the first section of step T4 which then branches out of the step.

Case 3: In this case the point to be added also falls within a wedge triangle, but it differs from case 2 in that the new "normal" triangle adds to the existing convex hull, but leaves the "convex" condition broken in the sense that following the boundary of the union of all triangles that do not have a vertex at INF the sequence of vertices is not uniformly CCW. To repair the convex hull requires a "flip" operation (or multiples thereof). The flip operation is a fundament part of the Delaunay tessellation process and will be described below. (The new convex hull repair is indicated in Figure 13 -Case 3 but the dotted line.) The convex hull repair algorithm is described in the second half of Knuth's step T4.

The new set of triangle forms a complete tessellation of the sequential points added so far and also form a convex hull such that the boundary segments traversed in a counter-clock-wise direction all satisfy the CCW predicate. The tessellation may not be Delaunay however, because we have not checked the Incircle predicate for any of the new triangles that were added during step T3 or T4 of the algorithm. To check this potential problem we examine the sides of each of the triangles we just created along with its adjacent triangle (mate) and see if it satisfies the Incircle predicate for the opposite vertices. (Refer to Figure 6 for Incircle condition and Figure 9 for the Arc-Node topology notation). If this is the case we need to "flip" the triangles. In Knuth's algorithm this is described in step T5.

This basically completes the algorithm for the addition of all the points after the first two which were add during the initialization. The Knuth's entire algorithm can be described as follows:

- Initialize using **DagTri** creator which uses vertices 0 and 1.
- For vertex 3 through (number of vertices) call:
  - Call **getTerminalNode** to find where the put the new vertex. (step T1)
  - Call **setNewEdges** (steps T2 through T5)

Flip

We have had several occasions to mention the “flip” operation as part of the Delaunay tessellation process. Given a convex hull repeated local application of the Incircle predicate and “flip” operation will always lead to a Delaunay tessellation in n-dimensions. (Edelsbrunner,2001) Virtually all Delaunay generation schemes are based on this fact. The “flip” algorithm used in Knuth's routines is specialized to take full advantage of the Arc-Node data structures and is easily understood by looking at Figure 14 . Both the Nodes of the two adjacent triangles and all of the Arcs are reused with a minor relabeling and the existing “mate” structure is unchanged. This means that the two adjacent triangles can be reoriented and none of the fast indexing linked-lists are broken.

Sacred Sides

Thus far we have defined a way to sequentially create a Delaunay tessellation of an arbitrary set of point data. The union of these triangles will be a signally connected convex hull. This is just what we want for contouring an isolated dependent variable field, where the data defines its own spatial scales. (We have no need to impose a grid on that field which will introduce a new scale – likely having nothing to do with the process.) In contrast to this situation there are many cases where we wish to use arbitrary spaced data (for example depth values) to define a complex but realistic geophysical basin shape used for hydrodynamic computations. In general this sort of domain will not be signally connected, or convex. To solve this problem we need to answer the question: Can we create a Delaunay tessellation for a domain that is not convex, or signally connected? The short answer is no, but almost. In grid generation schemes this is usually referred to as the “constrained Delaunay” problem.

A naïve approach is to first construct the Delaunay tessellation and then go back and see if it is possible rebuild the boundary so that it is appropriately “con-convex” and/or multiply-connected. This turns out to be a computational nightmare and going back to the “short history section” above we abandoned this approach decades ago in favor of a slower but reliable “build from the boundary” approach. A careful analysis of the steps in the Knuth algorithm shows that we can modify it slightly and obtain the same results. The key to this rests on a theorem that states that the within a convex domain “incircle

and flip" procedure produces a "locally Delaunay" tessellation. (More importantly, if you break the rule locally – it does not propagate through out the domain) (Edelsbrunner,2001). This allows us to selectively apply the "flip" operation, making sure that we preserve the convex condition but do not allow required boundary segments ("sacred sides") to be replaced.

In the above description of the DagTri algorithm we note that the "flip" method was called at two different places. The first was in Step T4 and at this stage in the algorithm it always involved a triangle that had at least one vertex at INF. The reason for these "flips" were to repair the convex hull that was broken by the addition a new vertex (outside of the previously defined convex hull). Obviously we do not want to skip this process because without a convex hull the local Delaunay theorem does not hold and the algorithm comes to grief in a hurry. (Crash.) The second place in the DagTri algorithm where the "flip" method is call is in Step T5. In this case an internal triangle has been subdivided by the addition of a new vertex and the flip is triggered by an Incircle check indicating a locally "non-Delaunay" condition. We want these to take place when ever possible, it they don't remove a "sacred side". If the flip would remove a "sacred side" we do not allow it and locally allow the tessellation to be "non Delaunay".

To see how this works we have Figure 15 where ten randomly spaced vertices that tessellated in normal DagTri routine and that the same set of points is re-tessellated with the first to vertices making up a sacred side. This is a somewhat extreme "non Delaunay" case but it does indicate that the constrained boundary condition will be forced onto the algorithm. For a slightly more realistic example Figure 16 shows an example where an irregular domain with an island is presented as a collection of sacred sides. The upper portion of the figures shows the domain boundary. The central panel of the figure shows the complete convex hull tessellation that preserves the "sacred" boundary segments. And the lower panel shows the interior of the complex domain with the external triangles marked and not drawn.

Within the DagDat data structure and DagTri algorithm several modifications have been made to accommodate the introduction of "sacred sides". If the boundary segment flag is set (this happens automatically if the bSeg form of the constructor is called) then the algorithm checks at the appropriate point in Step T5 to make sure the flip is not removing a sacred side. The second algorithm modification is that all triangles are either inside or outside of the boundary and for triangles outside of the domain the area is

flagged with a negative value. In addition the triangle area is available through the terminal nodes (a modification of the original Knuth algorithm) so that a test point can be checked for inside/outside of the global domain in $\ln(N_{bathmetry})$ time which has numerous uses in trajectory analysis (for example beaching).

DagData Boundary Seed

As has been pointed out, one of the useful applications of Delaunay tessellation is to create a continuous representation of a z-field (suitable for contouring and/or integration) where the data is given over a set of arbitrary points in 2-dimensions. The strength of this approach is that the spatial scales of the data are not aliased by scales introduced during some externally imposed gridding procedure. This works easily with isolated clusters of z-field data such as LEs , but problems are introduced if the bounding region for the data is not a convex hull. This is the case if the z-field is constrained by non-convex (possibly multiply-connected) region as would be expected in a realistic geophysical domain. In this case the boundary points are clearly part of the problem and the 2-dimensional description of its points should be part of the tessellation procedure. The neglect of merging the LE point data with the map boundary data was a short coming in the Gnome Analyst routings and lead to strange behavior when contouring near and under islands and subsequently did not conserve mass in the resulting representations. With the introduction of "simplicity" and exact big integer math this problem can be corrected by a straight forward three step process.

Step 1- boundary seed

In the preceding section we described the concept of constraining the Delaunay tessellation process by introducing "sacred sides" to limit the "flip" operation in cases where a boundary segment might require a locally "non-Delaunay" representation. Using standard topology representation for listing boundary co-ordinates (CCW order) and bSeg array (to indicate the end vertices of each of the multiply connected segments) it is easy to carry out a DagData and DagTri process that creates general convex hull tessellation with the defined boundary segments embedded as triangle sides. At this point we can also calculate the area of each triangle and assign it to the terminal Node structure with the convention that triangles inside the boundary will have positive areas and those outside have negative areas. It should be noted that the DagData initialization needs to know the maximum number of vertices expected during the tessellation so the

number of LEs that are anticipated in the second step of this processes should be padded onto the nv-value. Failure to do this will lead to a run time error of "array index out of bounds". On the assumption that this boundary representation will be used as a seed for a number of tessellations representing LE distribution for various times, but on the same background map it may be appropriate the save the DagData and DagTri objects to an output file. The upper panel of Figure 17 shows a simple case of an irregular domain with a single island.

Step 2 – boundary seed

Starting with DagData and DagTri either generated by step 1 or read in from a save file we can now add LEs sequentially and in the standard DagTri procedure with the following exception. As each point is considered for addition the "find terminal Node" step will return the triangle that the point fall within. An area check will indicate a global inside/outside check. If the enclosing triangle has a negative area the LE is beached (outside of the domain) and should be flagged as such, but NOT used in the tessellation. In this way we will end up with a final tessellation of the boundary data and the LEs which fall within the domain. No triangle outside the original boundary will be partitioned and subsequently the global inside/outside check will remain unbroken and ln(N) fast. The central panel of Figure 17 shows the original tessellation of the boundary data plus the addition of 5000 LE points distributed as a Gaussian cluster to the NE of the single island.

Step 3 – boundary seed

With the tessellation completed in step 2 we need to provide z-field data that represents the real values associated with the LE's. This assigns to each point the mass of the LE divided by the Voronoi area for that point. The boundary points on the other hand are zero boundaries so we should reset the values defined in the bSeg array to z-values of zero. Contouring the resulting z-value field will allow a complete representation of the floating (inside) LEs with arbitrarily high values near the shore, but no mass crossing the shoreline, or under islands. The lower panel in Figure 17 shows the contour lines and Map boundary of the floating LEs and conserves mass as a continuum tin representation.

References

Edelsbrunner, Herbert (2001) Geometry and Topology for Mesh Generation. Cambridge Monographs on Applied and Computational Mathematics, Cambridge University Press, Cambridge, UK

Edeosbrunner,Herbert and Ernst Peter Mucke (1988) “Simulation of Simplicity: A technique to cope with degenerate cases in geometric algorithms”, Fourth Annual ACM Symposium on Computational Geometry. 118-133.

Knuth, D. E (1992) Axioms and Hulls. Lecture Notes in Computer Science 606: Springer – Verlag, New York, NY

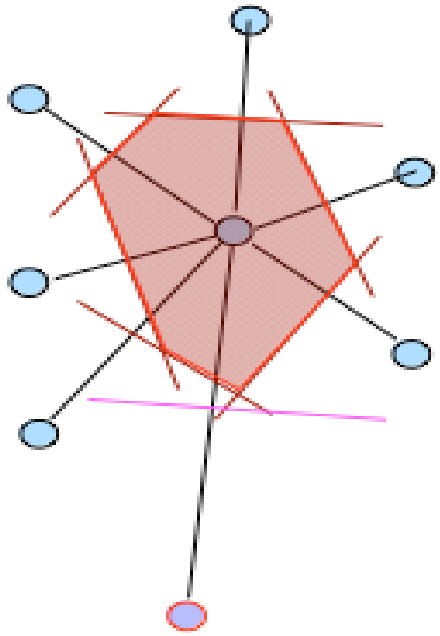
LeVeque, Randal J. (1992) Numerical Methods for Conservation Laws. Lectures in Mathematics, ETH, Birkhauser Verlag, Basel, Switzerland

Sedgewick, Robert (1992) Algorithms in C++. Addison-Wesley Publishing Co., Inc. , Reading, MA USA.

Zienkiewicz, O. C. (1971) The Finite Element Method in Engineering Science (2nd Ed). McGraw-Hill, London UK

Figures

Figure 1 – Thessian Polygon or the area associated with a single vertex. The union of all Thessian Polygons makes up a Voronoi Diagram.



$$V_p = (x \in \mathbb{R}^2 \mid \|x-p\| \leq \|x-q\|, \forall q \in S)$$

Figure 2 – Delaunay triangles represent the topological dual of a Thessian polygon. The three way junction of Thessian polygons is the center of a circle with a radius just reaching three vertices which make up the Delaunay triangle. The circle does not contain any other vertices from the set.

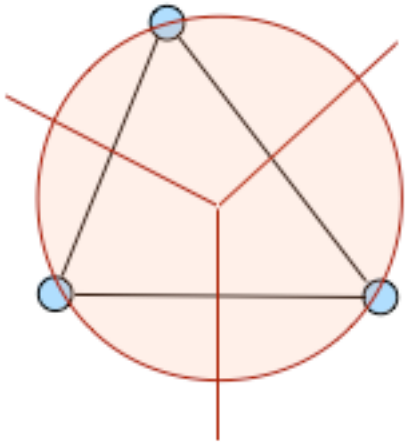


Figure 3 - Area of Delaunay triangles surrounding vertex V_i that define the spatial span of the local influence function.

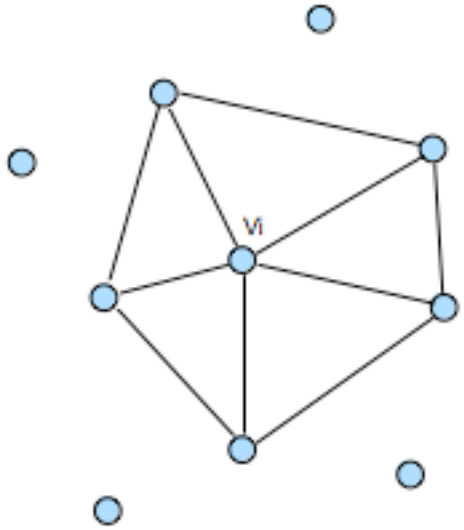


Figure 4 - The CCW predicate takes three vertices as arguments and based on the first two determines which of the semi-infinite half planes the third point is in. It returns either right (!CCW) of left (CCW) for the text point relative to the directed line segment (V_1=>V_2)

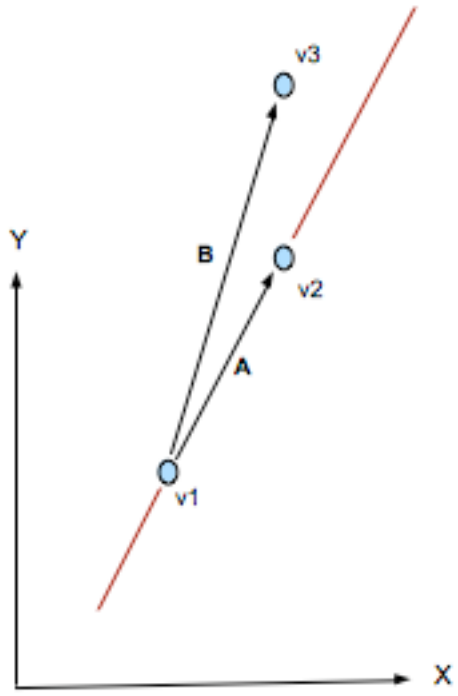


Figure 5 – Example of test output running CCW predicate. The test program picked two random points which defined a directed line segment. It then looped through 5000 points and plotted them after setting the pen color depending on the CCW predicate. As can be seen the two semi-infinite half planes are distinguished by color. A number of test runs using micro degree data indicate that about 1,000,000 CCW predicates can be evaluated per second on a small lap-top.

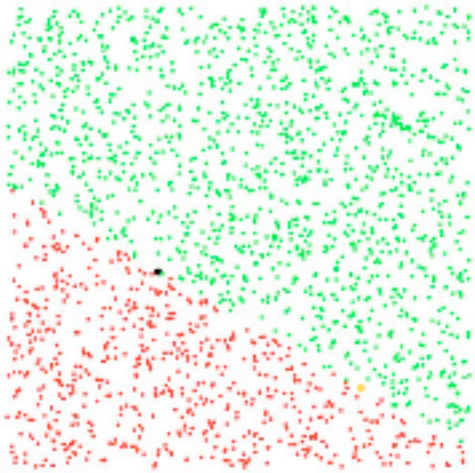


Figure 6 – Any two adjacent triangles will be defined by four vertices. Taking the three vertices of one triangle in CCW order and using the opposite vertex of the second triangle as a test point the InCircle predicate will return true if the triangle pair are non-Delaunay, otherwise false. Non-Delaunay pairs will become Delaunay by flipping the orientations of the common side.

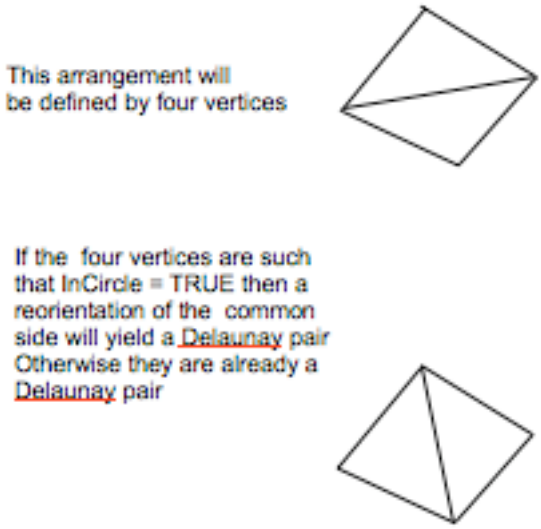


Figure 7 - Example of test output running Incircle predicate. The test program picked three random points with their CCW order defining a circle. It then looped through 5000 points and plotted them after setting the pen color depending on the Incircle predicate. As can be seen the plane is distinguished as inside or outside the circle by color. A number of test runs using micro degree data indicate that about 250,000 Incircle predicates can be evaluated per second on a small lap-top.

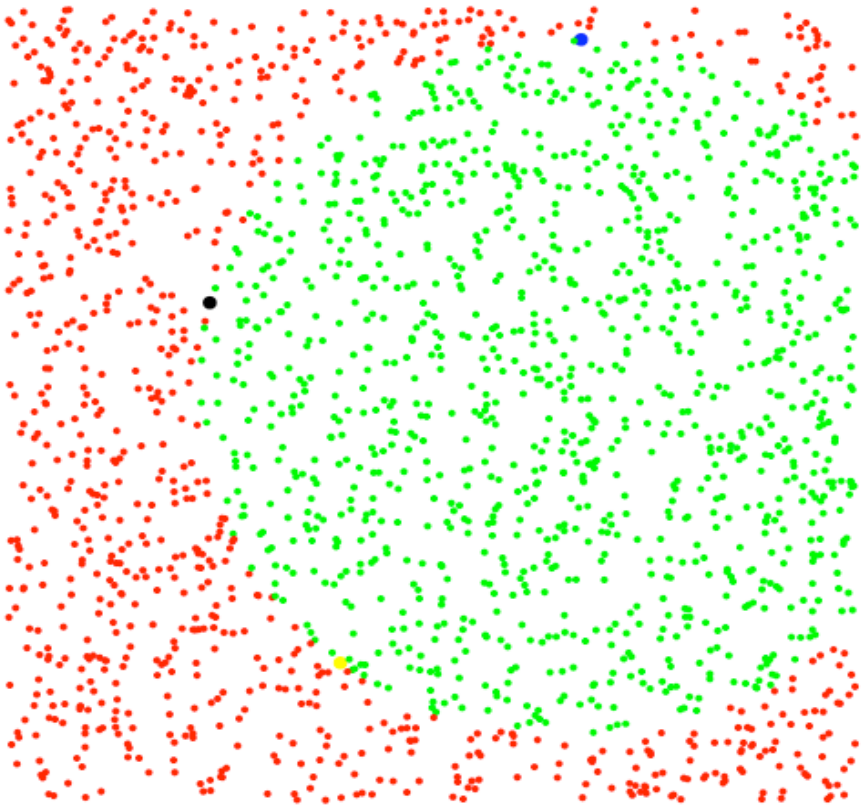


Figure 8 - Basic arc structure that define a triangle. Each triangle is made up of three Arcs that have associated data fields as shown

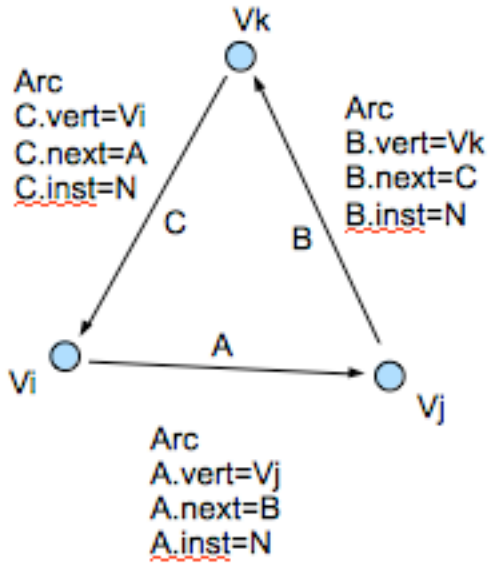


Figure 9 - Topological relationship between Arc's and Node's

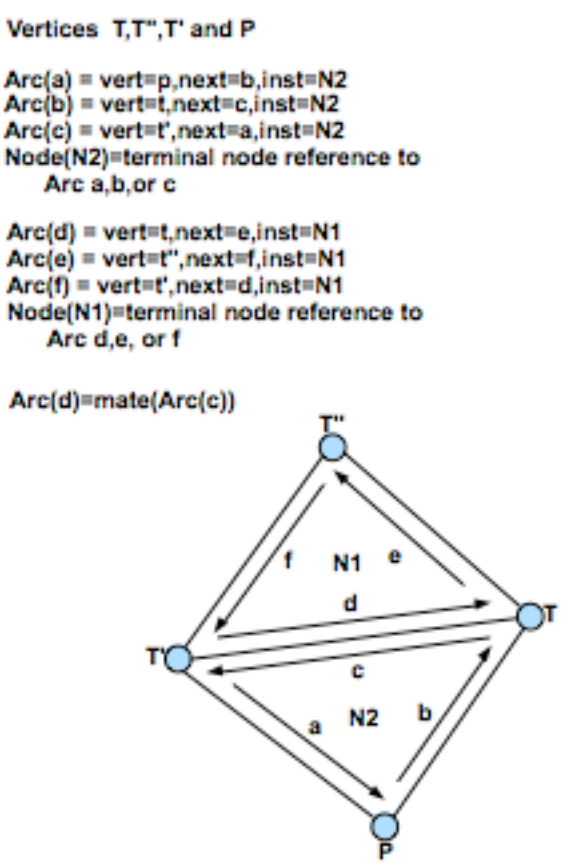


Figure 10 – Example of contours drawn using a Delaunay triangular mesh representing a Gaussian plume at a geophysical scale. This figure used the internal "helper" Contour object that is part of the DagData definition.

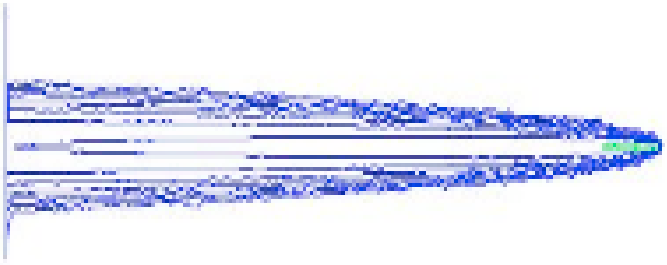


Figure 11 - Node and Arc structure after the initialization of DagData. The first two vertices V0 and V1 have been added to the structure. There are three Nodes and six Arcs which are related as indicated.

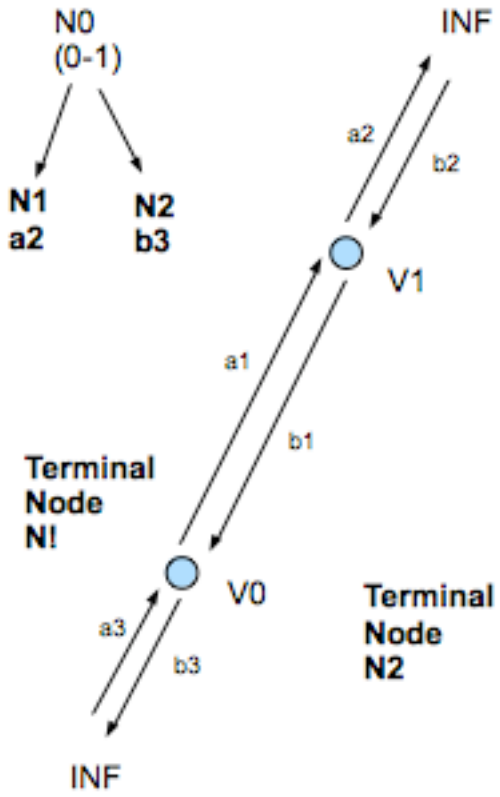


Figure 12 - The Node and Arc structure after the 4th vertex has been added. It references two real triangles, four "wedge" triangles with a vertex at INF and a total of eighteen arcs. Note that the DagTree also includes "fossil" node N1 that no longer references an existing edge but non the less is fundamental in partitioning the 2-D vertex space.

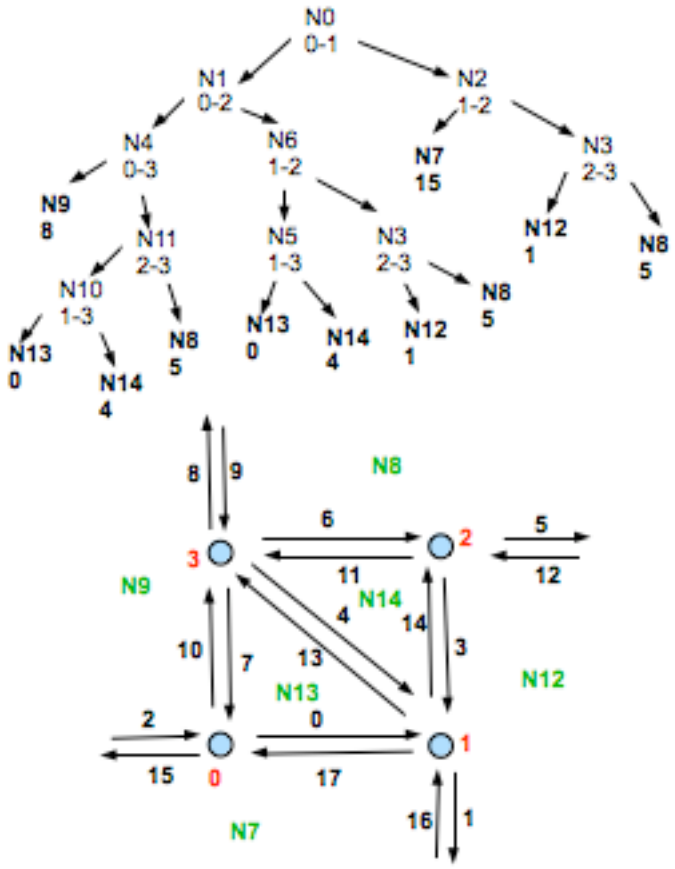


Figure 13 – Three possible cases for the addition of a fifth vertex to the DagTri structure. Case 1 when the vertex falls within an existing real triangle. Case 2 when the vertex falls in a "wedge" triangle, but does not break the existing "convex hull". And, Case 3 when the vertex falls within a "wedge" triangle and the existing "convex hull" is broken and will require that it be repaired by the "flip" of one or more "wedge triangles".

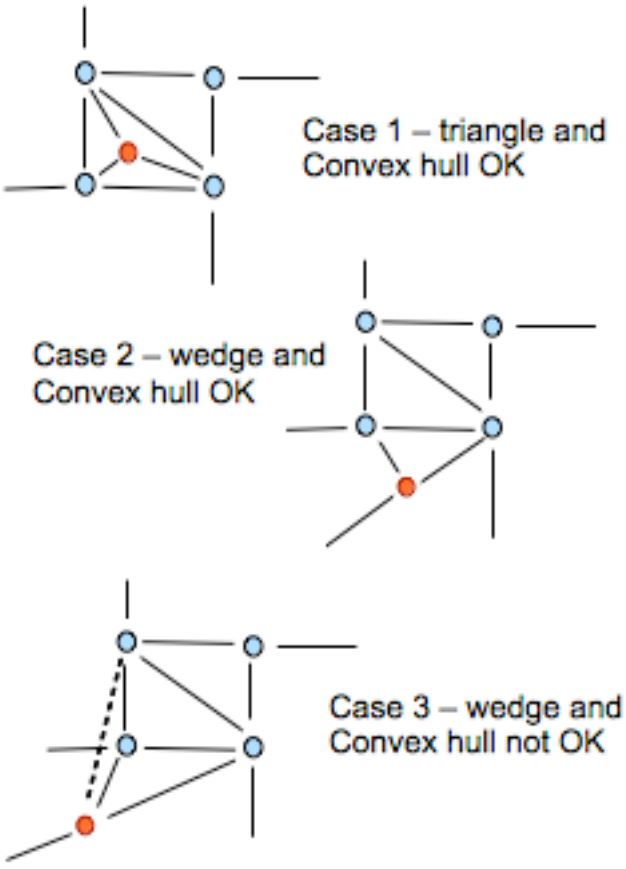


Figure 14 – Details of the changes in the DagData structure as the result of a flip operation.

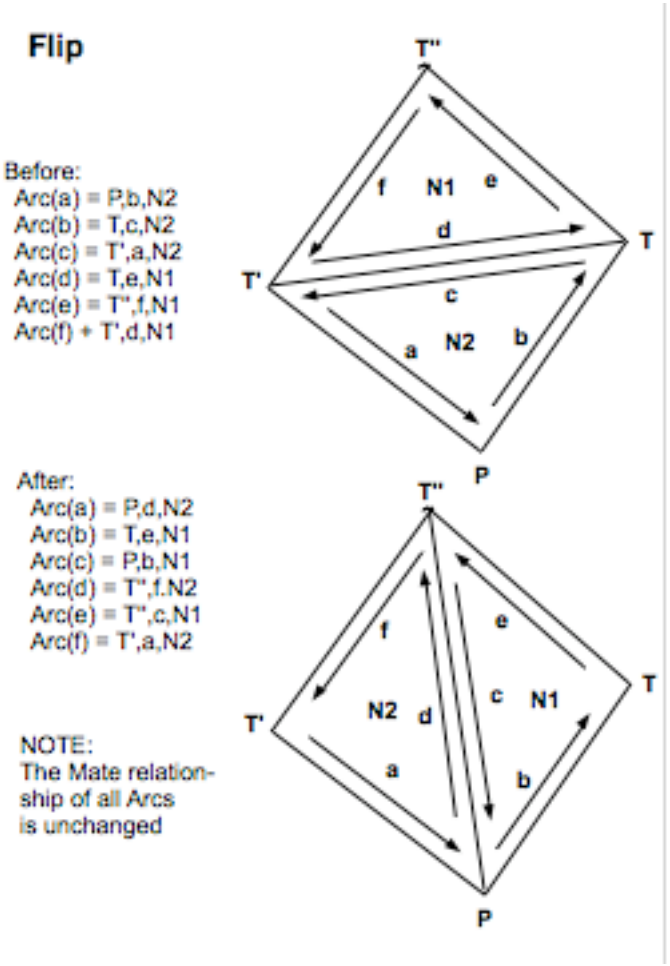


Figure 15 - Ten random vertex points with DagTri tessellation first unconstrained and then with edge V0 – V1 as a “sacred side”

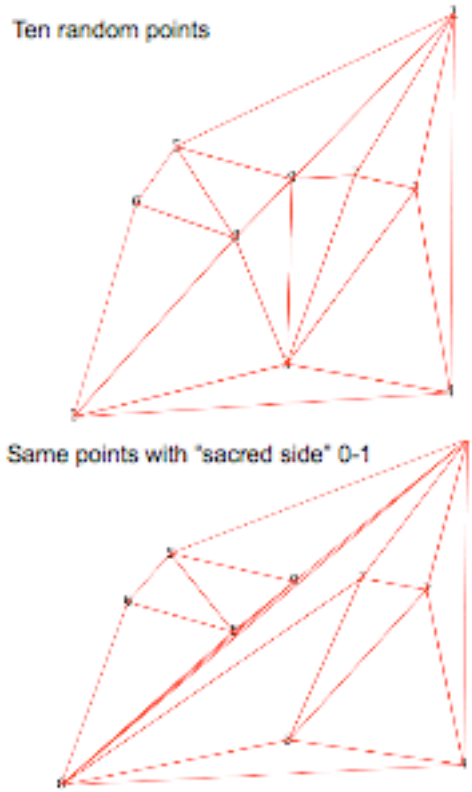


Figure 16 - Realistic geophysical domain with a single island. Upper panel shows shoreline. Central panel shows the constrained DagTri tessellation with the "sacred sides" preserved. The lower panel shows the interior triangles ignoring the remaining triangles of the convex hull that are outside of the boundary.

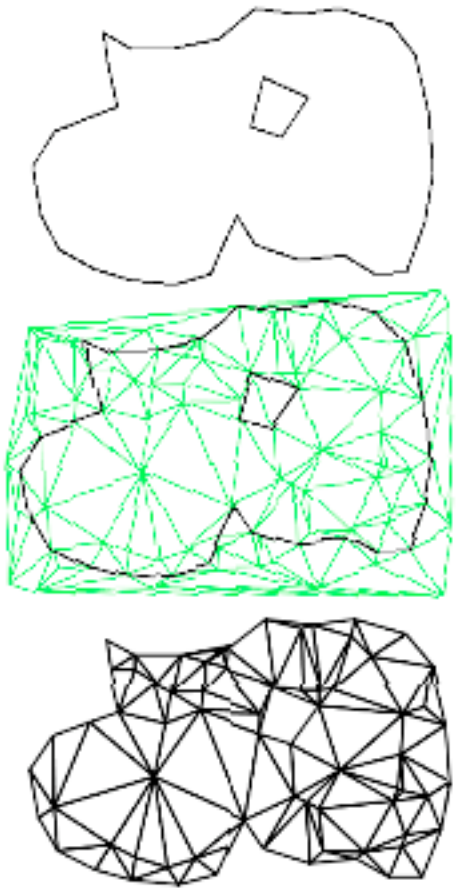


Figure 17 - Left panel initial tessellation of boundary segments. Center panel the same as the left panel with the addition of 5000 Lagrangian Particles distributed as a Gaussian cluster NE of the island. Right panel shows the contours generated from the z-value field associated with the LE density distribution which allows arbitrarily high values along shore lines but do not allow contours to cross the boundary or pass under islands.