Data Elements and Algorithms for the Tinfour Library

A Triangulated Irregular Network (TIN) package written in Java

G. W. Lucas
November 2017

Author's Note

This document is informal in nature. It is intended to provide assistance to developers. It is not a scholarly work. That being said, I'd at least like to get it right. So, if you if you have corrections or suggestions for improvements that would make this document more useful, feel free to let me know.

Acknowledgements

I began the Tinfour project while registered in the Masters of Geographic Information Systems program at Penn State University. Inspiration for the project is due, in large part, to Justine Blanford whose course in Geospatial Analysis demonstrated just how interesting and useful that topic can be. I am also grateful to my instructors Mike Renslow and Karen Schuckman who introduced me to airborne lidar, the technology which set the bar so high for performance considerations in the Tinfour software.

I'd also like to acknowledge my friend, the late Ron Patton, who first got me interested in the study of bathymetry and the modeling of underwater environments. Those applications started me down the road to the development of Delaunay triangulation software. And I'd like to thank my co-worker Scott Martin, one the most talented software developers that I have ever met. Scott provided many of the key insights that made Tinfour possible.

Finally, I'd like to thank my wife, Kat, who put up with my many late nights and early mornings spent in the creation of this software. I could not have accomplished the implementation without her support.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

Tinfour is a Java-based software library that provides tools for organizing and modeling unstructured data using the Triangulated Irregular Network (TIN) structure.

Tinfour is intended to be a free, open-source project. Therefore, it seems appropriate to provide background information to assist those who wish to use the code for their own applications. Nothing in Tinfour can be properly described as novel. The techniques it uses follow directly from well-known results from computational geometry. Its algorithms are not more complicated than those you would encounter in a college-level programming course. Even so, some of the underlying ideas that form the basis of the software may be obscured by the coding and implementation details. With that in mind, I have written these notes to help clarify the logic and design choices that are embedded in its source code. The algorithms Tinfour uses are taken from published works, most of them dating from the 1970's and 80's. Citations are provided for the convenience of the reader. The theory of the Delaunay triangulation was presented by Boris Delaunay in 1934. Many of the concepts used in Tinfour were inspired by the work of Jonathan Shewchuk, particularly his excellent Triangle library (Shewchuk, 1996).

1.1 Delaunay Triangulations

The primary structure created by Tinfour is a triangular mesh that is optimal in the sense of the Delaunay criterion. Because the Delaunay triangulation is a well-known topic in graph theory and widely documented on the Internet, a brief discussion will suffice. For any reasonably large set of vertices, there are many ways to connect them in a triangular mesh. But not all of them provide a favorable representation of the spatial relationships between points. The Delaunay criterion provides a way of selecting a mesh that is optimal in many regards. To illustrate this idea, Figure 1 shows an example of what happens when a set of points is connected in an arbitrary manner versus one that observes the Delaunay criterion.

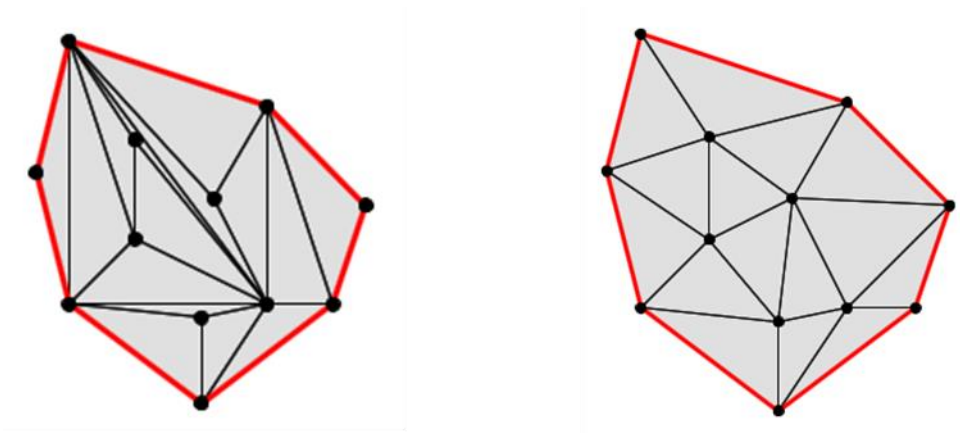

Figure 1 – A non-optimal triangulation (left) and the Delaunay (right)

The Delaunay triangulation is advantageous when modeling sets of unstructured sample points for a number of reasons. First, the triangles it produces tend to be, on the whole, more nearly equiangular than in other triangulations. By minimizing the frequency of “skinny” triangles, the Delaunay mesh improves the numeric stability of the floating-point calculations that are commonly used for modeling surfaces. Secondly, the Delaunay ensures that any point on the surface will be contained in a triangle formed from the closest sample points in the set. Since the closest points are generally the most relevant when performing interpolations, this property of the Delaunay leads to more accurate interpolation results. Finally, the Delaunay triangulation provides an efficient tool for identifying features and adjacency relationships within a set of sample points.

Figure 2 below shows an example of how data from a set of unstructured sample points can be organized into a triangulated mesh. The data for this figure was taken from a lidar-based elevation survey conducted in the area of Bear Mountain in Northwest Connecticut (NOAA, 2011a). Lidar systems use laser distance measuring equipment aboard low-flying aircraft to obtain surface elevation data. These systems provide a highly accurate and detailed view of the surface, typically collecting millions of samples per square kilometer. The data in Figure 2 covers a 480 by 240 meter area. In the original lidar data set, the area included over 133 thousand samples. For depiction purposes, the collection was subsampled to 675 vertices. Color coding was added with blue tones being assigned to the lowest elevations and red tones to the highest.

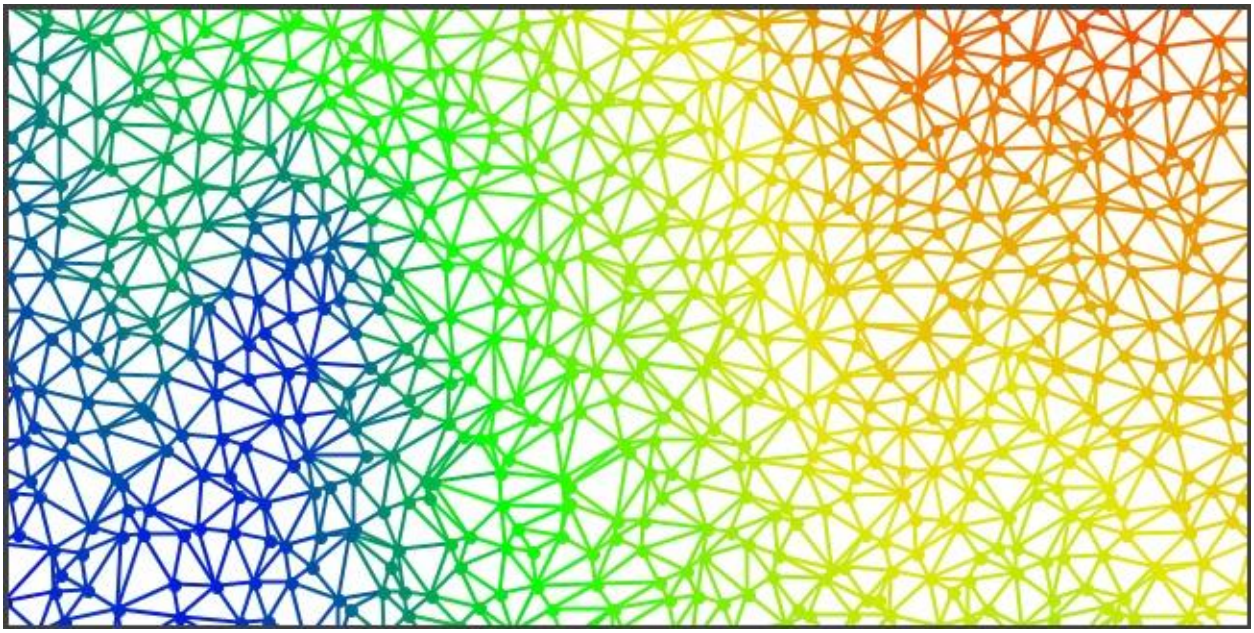

Figure 2 – A Delaunay triangulation created using unstructured sample points.

Figure 3 shows a surface that was derived from the reduced set of sample points by using a technique called Natural Neighbor Interpolation. The technique, which was described by Sibson (1981), takes advantage of the properties of the Delaunay triangulation to produce a pleasingly smooth surface with only moderate processing.

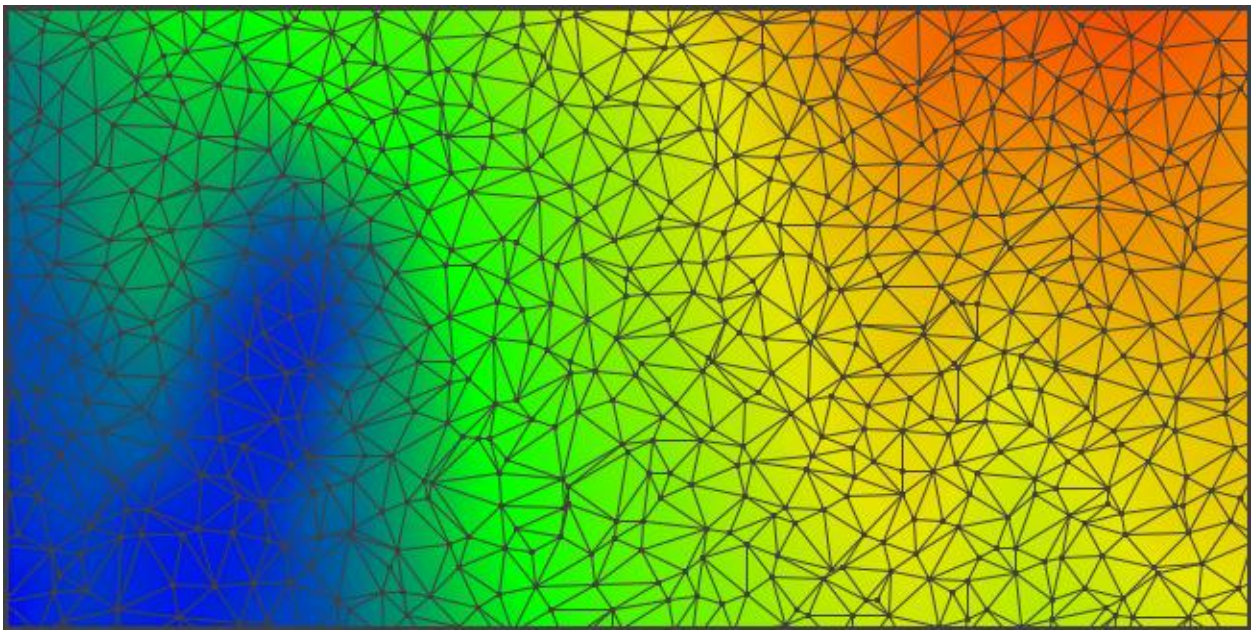

Figure 3 – A surface interpolated from unstructured data using a triangular mesh.

Finally, the image in Figure 4 shows effect of using the full resolution of the Bear Mountain lidar data set that was used for the figures above. Tinfour was used to build a triangle mesh from the complete sample set, providing an efficient method for computing the surface normal at each position in the output image. A simple illumination model used the resulting surface normal to produce lightning and shading effects. Color coding was assigned according to elevation using the same palette as in the above two images. The enhanced level of detail in Figure 4 demonstrates the quality of the data in the original lidar survey.

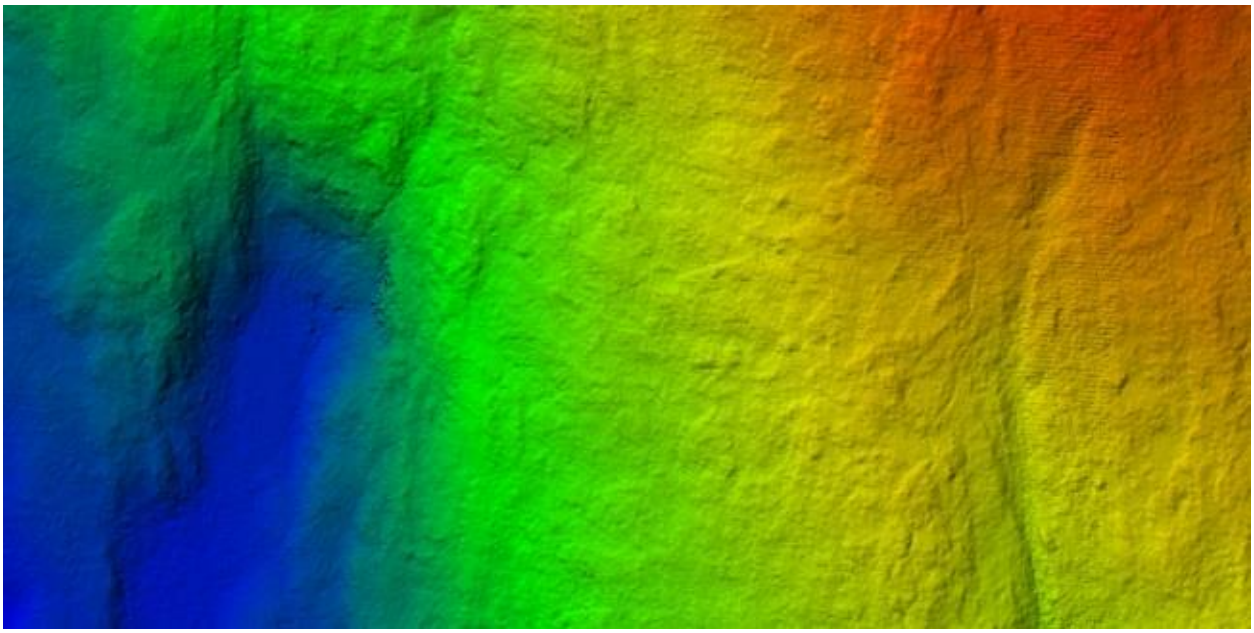

Figure 4 – Hillshade and elevation coded lidar sample at full resolution.

1.2 Voronoi Diagrams

In addition to being an important result in Computational Geometry, the Delaunay triangulation is also closely related to another prominent structure, the Voronoi diagram. The figure below shows a Voronoi diagram that was constructed from the same sample points as used for the figures above. In the diagram, the vertices from the triangular mesh are treated as "seed" points, each of which defines a sub region of the plane. Each region contains the set of all points that are closer to their respective seed point than any other sample in the data set.

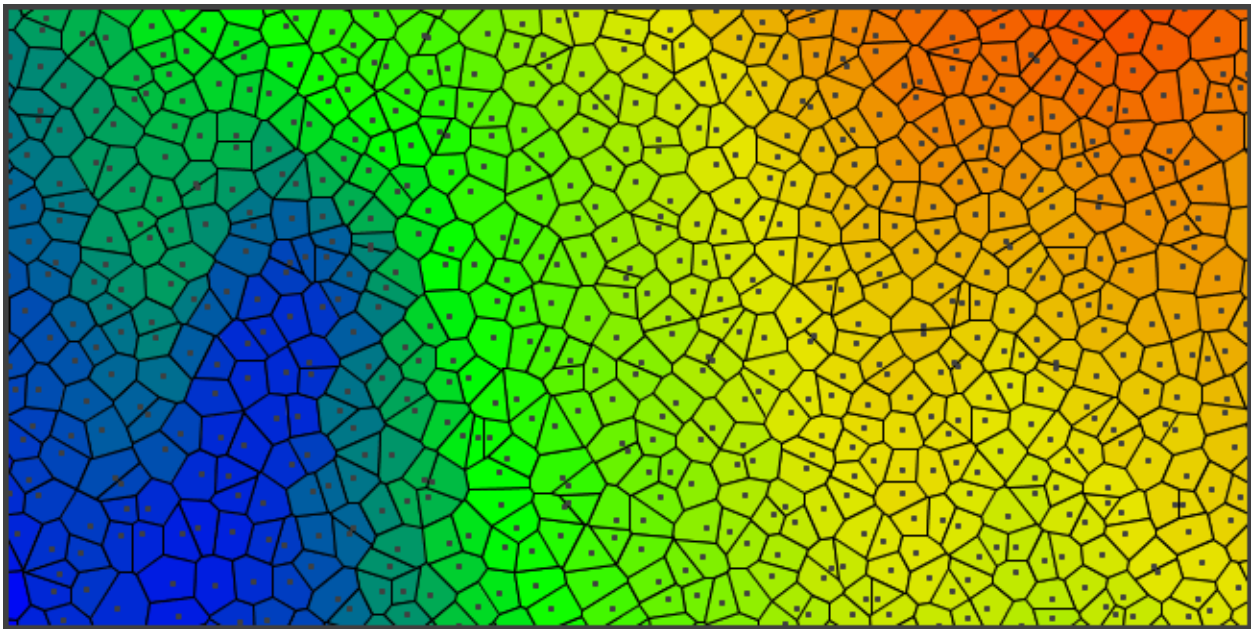

Figure 5 – A Voronoi diagram derived from unstructured samples.

Like the Delaunay triangulation, the Voronoi diagram is widely used to analyze data represented by unstructured samples over a surface. For example, consider the case where a number of mold spores are distributed at random in a culture dish. As the spores grow and reproduce, they form colonies that expand outward at a uniform rate until they encounter other colonies. The boundaries between colonies constitute a Voronoi diagram. Alternately, imagine a case where we wish to construct a map of areas served by airports so that a pilot can determine the nearest airport in the event of an emergency. Again, the boundaries of these areas would constitute a Voronoi diagram.

The Voronoi diagram is sometimes referred to as the "dual structure" of the Delaunay triangulation. In other words, for each Voronoi diagram, there is a unique Delaunay triangulation, and vice versa. This property makes it possible to construct one structure from the other with minimal processing (though there are many algorithms that construct the structures directly).

2 Building a Triangulated Irregular Network (TIN)

2.1 Performance and Memory

2.1.1 Time to Build Mesh as a Function of Sample Size

Currently, there are two versions of the Tinfour mesh-building implementation: the standard version and the semi-virtual version. The standard version represents all edges as objects in memory. The semi-virtual version maintains edges as a set of in-memory arrays of data primitives (integers, vertex references, etc.), and creates objects only as needed on a short-term basis. The standard version is simpler than the semi-virtual version. It also tends to process data faster than its counterpart. Unfortunately, the standard version requires substantially more memory than the semi-virtual-edge version: 246 bytes per sample in the standard version versus 120 for the semi-virtual version. So while it is not a true virtual (out-of-core) implementation, the semi-virtual version avoids the considerable memory overhead due to representing edges as persistent objects. Details of the two variations are discussed below.

The plots in Figure 6 show timing values collected across several runs of Tinfour using a lidar data set collected near Pole Creek in Oregon that contains just over 12 million samples including ground points and other features (USGS, 2014). The data was processed using an application called TimeDueToSampleSize that is one of the test applications included in the Tinfour software distribution. To measure the effect of sample size on runtime, the data was randomly subsampled into smaller data sets of various sizes and processed by the library. The statistics reflect only the time spent for data processing. The time to read the input data files and select the sample input data was excluded from the measurements. The trend lines and equations for the processing times were obtained using linear regression.

The results for this test were obtained using Oracle's HotSpot Java Virtual Machine (JVM) running on a Windows 7 host with 8 gigabytes of installed memory, a 2.9 GHz CPU with 512 kilobytes of L2 fast cache memory and 4096 kilobytes of L3 memory. The Tinfour build process is a non-concurrent (single thread) process, so the number of processors was largely irrelevant. No other major processes were running at the time. So the results for this test depend on favorable conditions that do not apply in all cases, but are not unrealistic when running Tinfour in a production environment with ample system resources. In the case of the standard configuration, the library processed an average of 1.86 million samples per second, though it exceeded that for smaller sample sizes. The slower, but less memory intensive semi-virtual configuration averaged 1.34 million samples per second.

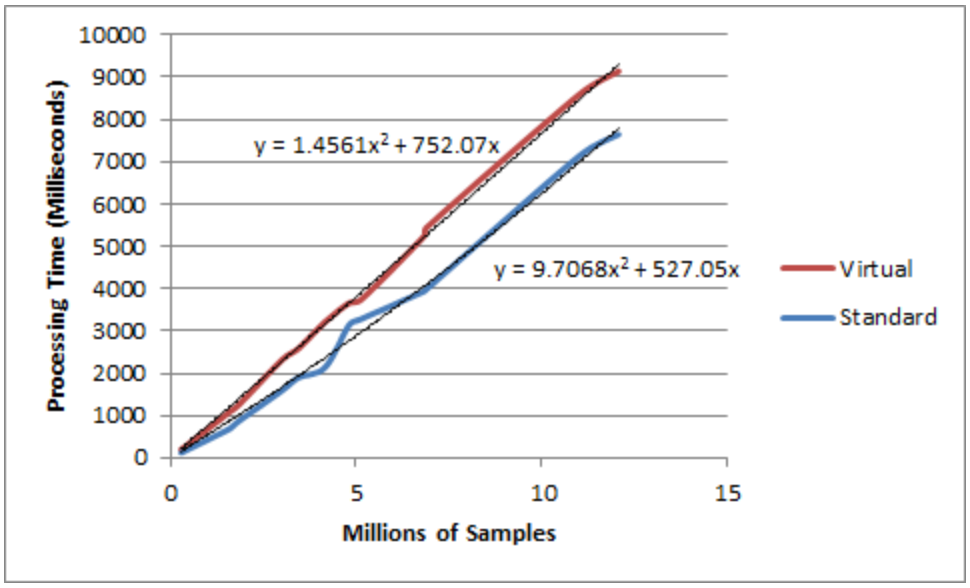

Figure 6 – Measured build times for mesh construction

Like most contemporary software, the Tinfour library is based on an object-oriented approach. In the standard implementation, the elements of the triangular mesh – vertices and edges – are all represented by individual objects. Each sample in the source data requires, on average, 7 object instances in the standard implementation: one object for each vertex and 6 objects to represent edges. On the other hand, the semi-virtual implementation does not use explicit objects to represent edges, and so requires an average of only 1.019 objects per sample: one for the vertex and a small amount of overhead for internal bookkeeping. But even with the reduction due to the semi-virtual representation, a set of 12 million samples would still require a very large number of objects.

In most Java applications, the overhead associated with the construction of an object is negligible. However, conventional Java applications seldom create objects a million at a time. When processing sample sets containing millions of points, the creation of objects (and the inevitable garbage collection that accompanies it) is a significant contributor to the cost of processing. Figure 7 shows the result of timing tests conducted using the same set up as described above, except that the cost of object allocation was excluded. The equations in the figure do not show quadratic terms because they were of very small magnitude and not statistically significant. Note that the performance change for the semi-virtual configuration, which allocates fewer objects per sample, is much smaller than that of the standard configuration.

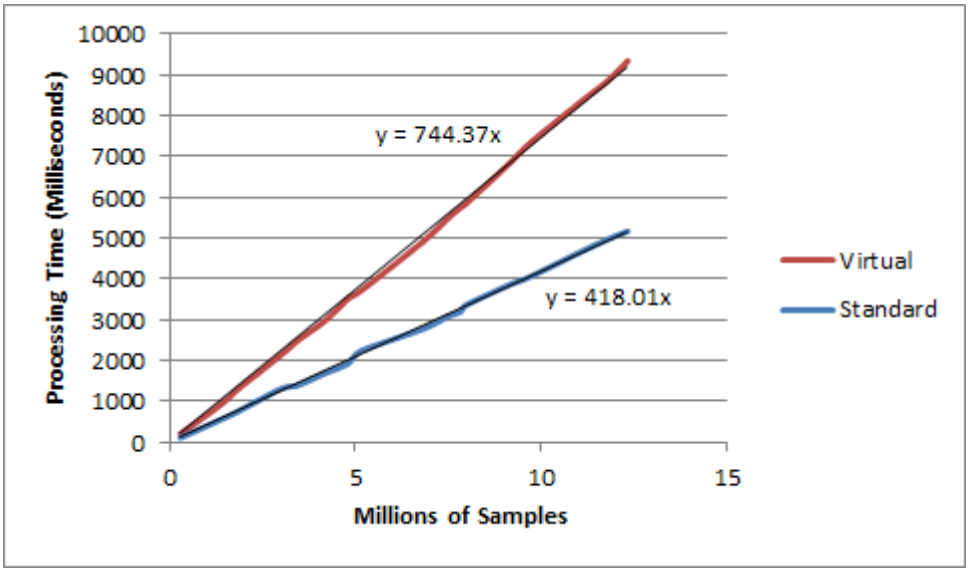

Figure 7 – Measured build times for mesh construction, object allocation excluded.

Tinfour does permit an application to reduce the cost of object allocation and garbage collection in cases where multiple data sets are being processed individually. The main classes for the library – IncrementalTin and SemiVirtualIncrementalTin – are essentially collections of vertices. When processing multiple data sets, it is possible to reuse a single instance of these classes for each data set by clearing its content between tasks. When the content is cleared, the internal objects that were constructed to represent the edges in the TIN are not removed. Instead, they are simply marked as available for re-use. Doing so avoids both the cost of allocating new objects and the overhead due to garbage collection.

2.2 Algorithms and Structures for Building a Triangular Mesh

2.2.1 Mesh Building through Incremental Insertion

The problem of constructing an optimal triangulated mesh is an important topic in computational geometry and has been extensively studied. Su and Drysdale (1996) identified three broad classes of algorithms for building triangulated mesh: divide-and-conquer methods, sweep-line methods, and incremental insertion. The Tinfour library uses an incremental insertion algorithm. In this process, an initial mesh of three vertices is created using a "bootstrap" process. Once the initial mesh is constructed, vertices are added to the one-at-a-time. The process is illustrated in Figure 8 below. Vertices 3 and 4 are inserted to the interior of the existing mesh. Vertex 5 extends the mesh. Note that each addition changes the structure of the triangles and has the potential of destroying previously existing edges (segments). For example, the insertion of vertex 4 has the effect of destroying edge 2-3 and replacing it with new edges 3-4 and 2-4.

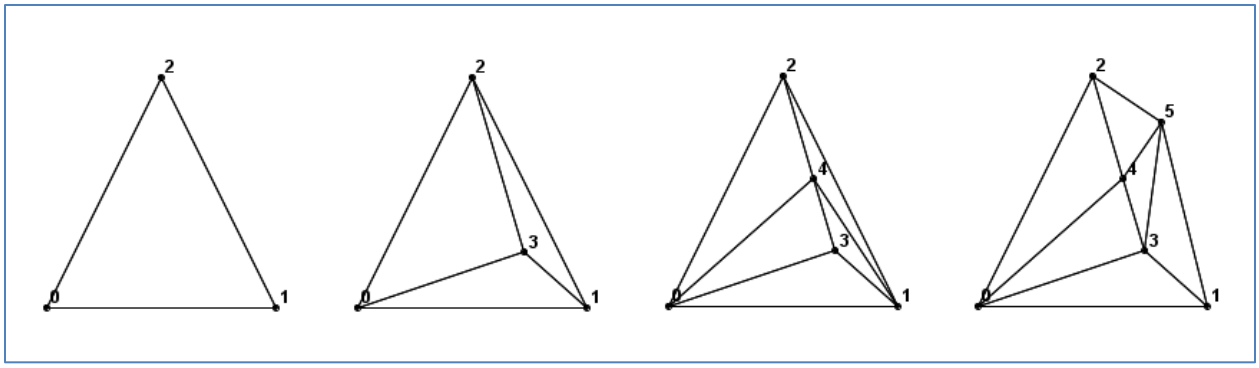

Figure 8 – Mesh building using incremental insertion

In the course of building a TIN that includes a substantial number of vertices, edges may be constructed and then replaced many times. Tinfour tracks the number of replacement operations as part of its regular processing. In testing with lidar data from the Bear Mountain sample, the average number of replacements ran about 6.5 (for a set of over 3 million edges). This statistic suggests that an efficient way of handling edge replacements is a requirement for the design of a good TIN implementation. Tinfour accomplishes this efficiency by using a reusable-object pool known as the EdgePool collection. Slightly different versions of the EdgePool are used for the standard and semi-virtual implementations.

The figure above also illustrates a notable characteristic of a TIN. The perimeter of a TIN is always a convex polygon.

2.2.2 The Delaunay Triangulation

As mentioned above, the fundamental product of the Tinfour implementation is a Delaunay triangulation. The Delaunay criterion requires that the triangular mesh be constructed so that no point lies within the circumcircle of a triangle to which it is not a member. On the left side of Figure 9 below, point D does lie not within the circumcircle of triangle ΔABC, so the Delaunay criterion is met by the pair of triangles ΔABC and ΔCBD . If point D were inside the circumcircle as shown on the right, the triangulation would need to be reorganized by flipping the edge BC so that it connected edge AD to form two alternate triangles ΔABD and ΔDCA. Note that in both cases, the points are always given so that they specify the edges of the triangle in counterclockwise order.

Each time a new vertex is inserted into the triangular mesh, Tinfour adjusts the local edges as necessary to ensure that the criterion is observed. Thus at all stages of construction, the software maintains a triangulation that is properly Delaunay.

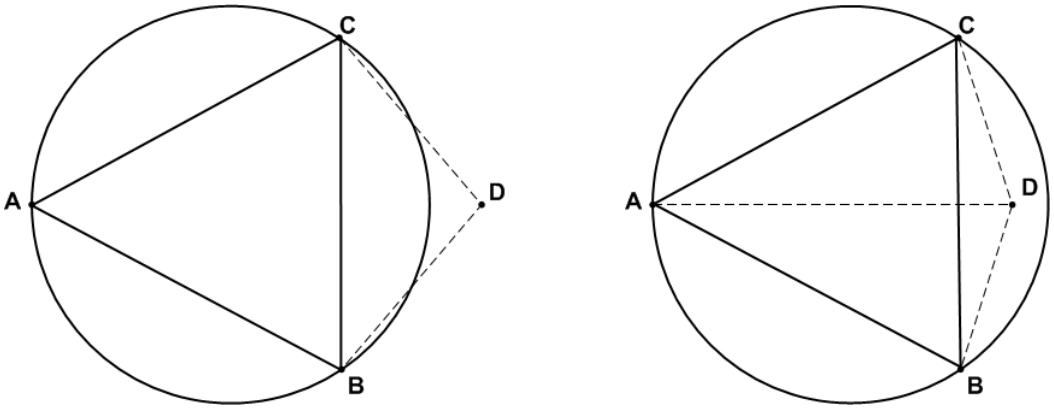

Figure 9 – Circumcircle criteria used for Delaunay triangulations

Cheng (2013, p. 57) provided a computation for determining whether a point D given by coordinates (d_x, d_y) is inside the circumcircle of a triangle ΔABC with coordinates (a_x, a_y), (b_x, b_y), and (c_x, c_y) by evaluating the following determinant :

$$\text{InCircle(a,b,c,d)} = \begin{vmatrix} a_x - d_x & a_y - d_y & (a_x - d_x)^2 + (a_y - d_y)^2 \\ b_x - d_x & b_y - d_y & (b_x - d_x)^2 + (b_y - d_y)^2 \\ c_x - d_x & c_y - d_y & (c_x - d_x)^2 + (c_y - d_y)^2 \end{vmatrix}$$

If the value for InCircle(a,b,c,d) is greater than zero, D lies inside the circumcircle of ΔABC and the Delaunay criterion is violated. To restore the Delaunay property, we must perform an edge-swap operation as described above. If the value is less than zero, then D is outside the circumcircle and the criterion is met. If the value is exactly zero, the point is on the circumcircle and either arrangement of points is acceptable based on the Delaunay criterion. In this ambiguous case, some other criterion must be applied to select the preferred construction.

One question that arises from inspecting the drawings in Figure 9 is whether the fact that point D is outside the circumcircle of ΔABC tells us that we can be sure that point A is outside that of ΔCBD. A little thought reveals that the computation for InCircle(c,b,d,a) is equivalent to swapping the rows in the determinant for InCircle(a,b,c,d) an even number of times which, by the row property of determinants, will produce the same value as the original order. Indeed, any permutation of rows that preserves the counterclockwise ordering of triangle vertices always requires an even number of swaps. Thus the evaluation of one determinant is all that is required to decide whether an edge flip operation is necessary.

2.2.3 Data Primitives and Structures for Representing Graphs

A triangulated mesh can be viewed as consisting of three geometric primitives:

1. Vertices
2. Edges
3. Triangles

Delaunay showed that, as the number of vertices in a Delaunay triangulation grows large, the number of each kind of feature approaches the following values:

For N vertices:

- N×3 Edges
- N×2 Triangles
- An average of 6 edges connect to each Vertex

These relationships are maintained at all sufficiently large sub-regions of the overall TIN except near the outer boundary.

For each sample in a data set, we construct one vertex. In data sets such as lidar surveys, where the number of samples usually runs in the millions, the number of edges and vertices would be similarly large.

2.2.4 The Quad-Edge Data Structure

The triangular mesh created by Tinfour is built from a collection of linked edges represented using the quad-edge data structure which was popularized by Guibas and Stolfi in the mid 1980's (Guibas, 1985, p. 74). It is suited to the construction of many different classes of polygon-based graphs including Delaunay triangulations and Voronoi diagrams.

A single instance of a quad-edge structure is used to represent a single edge consisting of a pair of vertices and links to 4 adjacent edges. As shown in Figure 10 below , the vertices A and B define a segment AB and its "dual" BA. Edges in Tinfour are always treated as having direction, and every edge has a dual in the opposite direction. The links for the edges depend on their direction. The forward link from AB would be represented by a second quad-edge for vertices BR. The reverse edge from AB would be quad-edge PA. Taken together, these quad-edges can be used to indicate the existence of a polygon. In a TIN, all such polygons are triangles and all links are populated, though this restriction does not necessarily apply to other kinds of graphs. Links for the edge are given in Table 1.

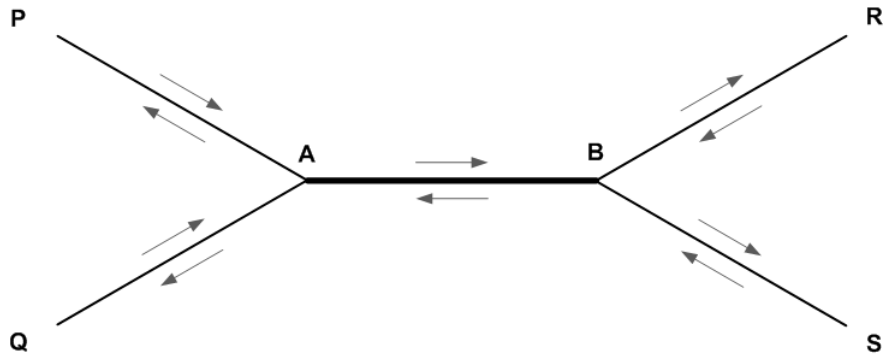

Figure 10 – The quad-edge data structure for edge AB/BA and neighbors

Table 1 – Links for quad-edge AB/BA and neighbors

Edge	Forward	Reverse	Dual
AB	BR	PA	BA
BA	AQ	SB	AB
PA	AB	Not shown	AP
AP	Not shown	QA	PA
QA	AP	Not shown	AQ
AQ	Not shown	BA	QA
BR	Not shown	AB	RB
RB	BS	Not shown	BR
BS	Not shown	RB	SB
SB	BA	Not shown	BS

Two adjacent triangles would be represented as shown in Figure 11 below.

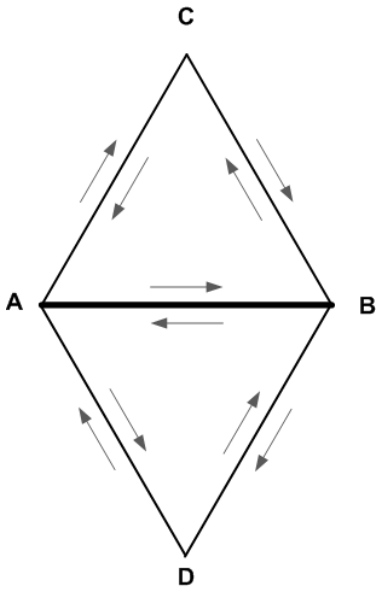

Figure 11 – Quad-edge links adjacent triangles ΔABC and ΔDBA

Table 2 – Links depicted for triangles ΔABC and ΔDBA

Edge	Forward	Reverse	Dual
AB	BC	CA	BA
BC	CA	AB	CB
CA	AB	BC	AC
BA	AD	DB	AB
AD	DB	BA	DA
DB	AB	AD	BD

The mesh representation in Tinfour does not specify triangles as an explicit object. Triangles are implied by the links associated with the set of edges in the mesh collection. The data objects that represent vertices do not carry any information that explicitly ties them to edge. The edges know about vertices, the vertices do not know about edges. Thus, software that uses Tinfour can define vertices as immutable objects or simply pass them to the library without fear that they will be altered.

2.2.5 The Ghost Vertex and Bootstrap Layout

Because both the previous examples were only fragments of a mesh, some of the links were not recorded. One key factor in the construction of a triangular mesh using the quad-edge structure is the stipulation that each link in the structure be populated. Doing so simplifies many coding problems, but does require special logic to handle the edges that lie on the perimeter of the mesh.

There are different strategies for avoiding or otherwise managing null links in different triangular mesh implementations, Tinfour depends on a concept known as the "ghost vertex" (Cheng, 2013, p. 61). Imagine a simple triangular mesh containing a single triangle with three perimeter edges. To populate the null links for these edges, Tinfour specifies the existence of an imaginary point, the ghost vertex, which connects to each vertex on the perimeter of the TIN. By doing so, it ensures that the forward and reverse links for the perimeter edges are all populated. Some implementations give the ghost point an actual geometric specification by imagining that it exists in a higher-order dimension from all the other points in the mesh. For example, in a 2D triangular mesh for a set of coordinates organized over a plane, the ghost point could be treated as existing in a third dimension, being raised some distance above the plane. Tinfour does something a little different, implementing the ghost vertex as a null object reference.

Figure 12 below illustrates the links for a mesh containing three points as it would be configured after the initial bootstrap operation. In the figure, solid lines are actual edges, while the dotted lines indicate connections and arrows indicate link direction. The mesh consists of three actual vertices – A, B, and C – and a single ghost vertex. Even though the ghost vertex in the figure is shown in three positions, it is a single entity and, so, is always labeled as g.

In addition to ensuring that no edges have null links, the bootstrap operation also establishes geometric relationships that will be maintained in all subsequent point insertions. In particular, the forward links for the interior edges of triangle ΔABC establish a counterclockwise ordering for the triangle. Tinfour maintains all triangles in the interior of the TIN in counterclockwise order. While the exterior links have no true geometry (because the ghost point is null), an order is imposed on each loop based on the direction of the perimeter edge it includes.

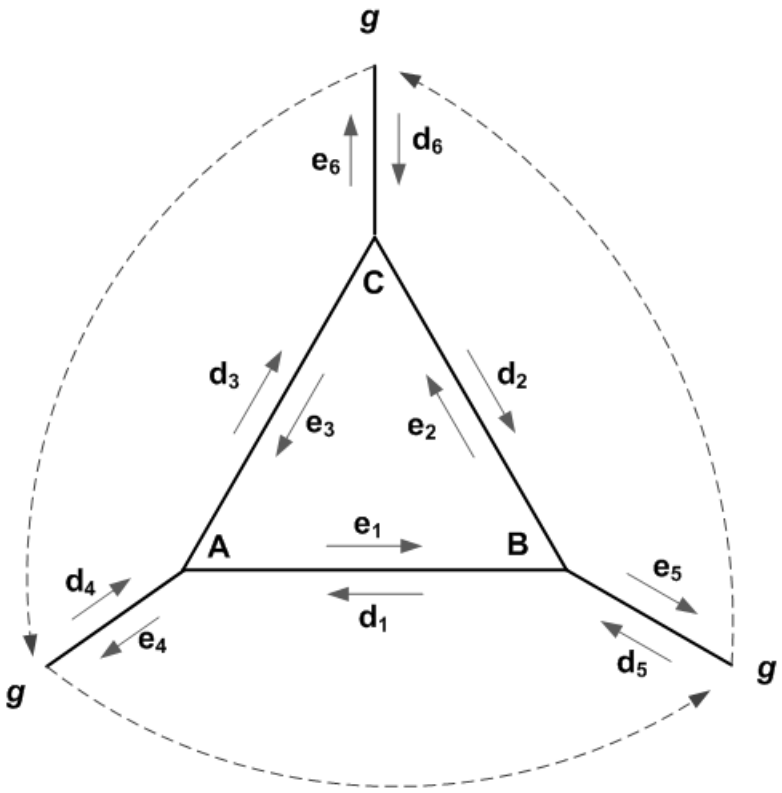

Figure 12 – Initial geometry and links after completion of bootstrap operation.

Table 3 – Links after completion of bootstrap operation

Edge	Start Vertex	end Vertex	Forward	Reverse
e_1	A	B	e_2	e_3
e_2	B	C	e_3	e_1
e_3	C	A	e_1	e_2
e_4	A	g	d_5	d_1
e_5	B	g	d_6	d_2
e_6	C	g	d_4	d_3
d_1	B	A	e_4	d_5
d_2	C	B	e_5	d_6
d_3	A	C	e_6	d_4
d_4	g	A	d_3	e_6
d_5	g	B	d_1	e_4
d_6	g	C	d_2	e_5

2.2.6 Edge Traversal and Navigating through the Mesh

Many operations in the Tinfour library involve some kind of traversal from one edge to a neighbor. For example, given a starting edge it is possible to construct a triangle by moving across the forward links until the traversal returns to the original edge. Figure 13 below shows the traversal from a starting edge, e, to edges in its vicinity.

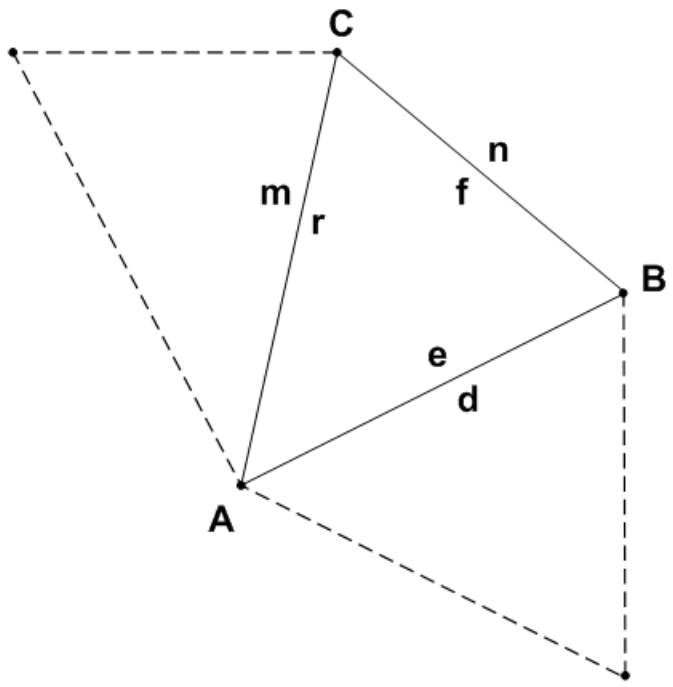

Figure 13 – Traversal for adjacent edges

Table 4 – Psuedocode for edge traversal

Edge	Vertices	Links
e	AB	Starting edge
d	BA	e.getDual()
f	BC	e.getForward()
r	CA	e.getReverse(), also e.getForward().getForward().
n	BC	e.getDualFromForward()
m	AC	e.getDualFromReverse()

As mentioned above, Tinfour maintains links so that all triangles forming the mesh are oriented in a counterclockwise order under forward traversal. So the result of three subsequent getForward() operations results in a complete loop of a triangle.

As a last illustration of edge traversal, the following fragment of Java code shows an operation nicknamed “the pinwheel” . The code collects a list of all the vertices that are joined to a central “anchor” vertex by a set of connecting edges. At the start of the operation, we are given an edge that begins with the anchor vertex A. The getA() method of that edge would obtain the anchor vertex. The getB() method obtains the vertex at the other end of the edge. In the loop that follows, the getDualFromReverse() method is used to traverse across the edges that connect to the anchor so that the adjacent vertices can be extracted and added to the result list. The collection effort terminates when the traversal makes a complete loop around the anchor vertex and reaches the initial edge.

```
IQuadEdge e;  // given e starts with vertex A
ArrayList<Vertex> result = new ArrayList<>(); // a vertex collection

IQuadEdge cursor = e;
Do{
   Vertex b = cursor.getB();
   result.add(b);
   cursor = cursor.getDualFromReverse();
}while(!cursor.equals(e));
```

2.2.7 The Realization of the Quad-Edge Structure in Code

When we consider the problem of edge-traversal, it becomes apparent that any code that navigates a triangular mesh needs to be aware of the direction in which the edge is traversed. If we wish to represent edges using Java objects, then information about direction should be part of the class design. Tinfour addresses that requirement by implementing each edge as a pair of linked objects, one for each direction of traversal. In effect, it splits the quad-edge structure into two pieces. Each piece is an individual Java object. Each piece has a reference to its dual. Both pieces are instantiated at the same time and tied together by setting their dual references to their counterparts.

The main class for edge representation is named QuadEdge. Each instance of QuadEdge is accompanied by a companion object from the class QuadEdgePartner, which is derived from QuadEdge. Thus, there are two objects associated with each edge in the TIN. Because there are approximately 3 edge pairs constructed for each vertex in a Delaunay triangulation, and the number of vertices in a data sample can run in the millions, the number of object instances in a fully populated TIN can grow quite large. Therefore a compact class design is essential for conserving memory. For example, two vertices define a line segment, so it follows that each edge would requires a reference to two vertex objects. But the QuadEdge implementation only implements one. Since a QuadEdge object is always associated with a QuadEdgePartner object, each object only needs to carry one reference. The second vertex reference for either side of the pair can always be obtained from its counterpart.

Each instance of a QuadEdge object requires 32 bytes when running under the HotSpot virtual machine with the compressed references option. QuadEdgePartner requires the same. Table 5 shows the layout of the elements in the class. Because each edge requires a pair of objects, each edge requires 2×32=64 bytes of memory. Since there are 3 edge pairs per vertex, the total per-vertex memory use for the QuadEdge representation is 3×64=196 bytes. Instances of the Vertex class itself require 40 bytes. So the average memory use per data sample, including both edges and vertices, is 196+40=226 bytes. Additional overhead due to the JVM memory management raises this value to the 246 bytes cited in paragraph *2.1 Performance and Memory.*

Table 5 – Memory use for QuadEdge instance

Element	Size
Java management overhead	8 bytes
Reference to Java class definition	4 bytes
Reference to dual	4 bytes
Reference to vertex A	4 bytes
Reference to forward edge (link)	4 bytes
Reference to reverse edge (link)	4 bytes
Integer edge index (Tinfour bookkeeping)	4 bytes
Total	**32 bytes**

Some of the fields in the QuadEdge class are used to represent actual data, but 12 bytes of each instance is due to the unavoidable cost of allocating objects in Java. That amounts to 24 bytes of overhead per edge pair, or an average of 72 bytes per vertex. On top of that, the creation of so many objects adds considerable processor overhead in terms of construction and garbage collection.

The SemiVirtualEdge implementation does not represent the edges in the triangular mesh as actual objects, but stores metadata for the edges in conventional data primitives. When edge-related objects are required, they are constructed as needed and quickly discarded. Since the edges do not exist as persistent objects, the number of objects constructed by the semi-virtual implementation is reduced by a factor of more than 1000. Thus the SemiVirtualEdge implementation reduces the total cost of all edges to an average of 76.07 per vertex, using a total of approximately 120 bytes per sample including memory for vertices and JVM overhead.

Because the SemiVirtualEdge implementation reduces memory use, it permits an application to process substantially more vertices without increasing the maximum memory allocation of a JVM. Furthermore, by reducing the number of objects that are constructed, it substantially reduces the burden of garbage collection. Unfortunately, the memory reduction has a cost of its own. The overhead required for accessing Java arrays and interpreting the integer indices for forward and reverse links substantially increases the run time. As demonstrated by the measured build times for the two variations shown above in *Figure 6 – Measured build times for mesh construction*, the SemiVirtualEdge variation requires 40 percent more time than the standard QuadEdge implementation to process one million points.

2.2.8 Vertex Insertion Process

Tinfour inserts vertices into the mesh using an algorithm based on two famous papers by Bowyer (1981) and Watson (1981). The thing that makes these papers famous is that both were submitted to the same journal at about the same time and both presented important and closely related results. When the editors of *Computer Journal* received the two papers, they elected to run them side-by-side in the same issue.

By way of introduction, I will preface the discussion of the Bowyer-Watson algorithm with an earlier and simpler technique that illustrates some of its underlying principles. The edge-flipping algorithm, which was described by Lawson (1977), was actually the first algorithm implemented by Tinfour. It had the appeal of being compact to code and easy to understand. However, when it was replaced with the Bowyer-Watson method, the time required to build a TIN was reduced by a factor of 50 percent.

2.2.8.1 Simple Insertion with Edge Flipping

Lawson's original algorithm creates a Delaunay mesh using a simple insertion procedure. Starting with an initial mesh of three points (which Tinfour calls the "bootstrapped mesh"), the algorithm inserts each vertex using the following steps:

1. Locate the containing triangle.
2. Insert the vertex into the triangle by linking it to each of the vertices in the existing triangle.
3. Recursively "flip" edges as necessary to restore the Delaunay property.

The key to Lawson's approach is the third step. When a vertex is inserted into the containing triangle, any or all of the resulting triangles may be non-Delaunay. Without some kind of correction, the results would gradually come to have the same sub-optimal appearance as the example of a non-Delaunay mesh versus a Delaunay triangulation that was given in Figure 1 above. Lawson restored the Delaunay property by testing each new edge to see if the triangles on its opposite sides met the Delaunay criteria. If they did not, the edge was "flipped" resulting in an alternate set of triangles as shown in the figure below.

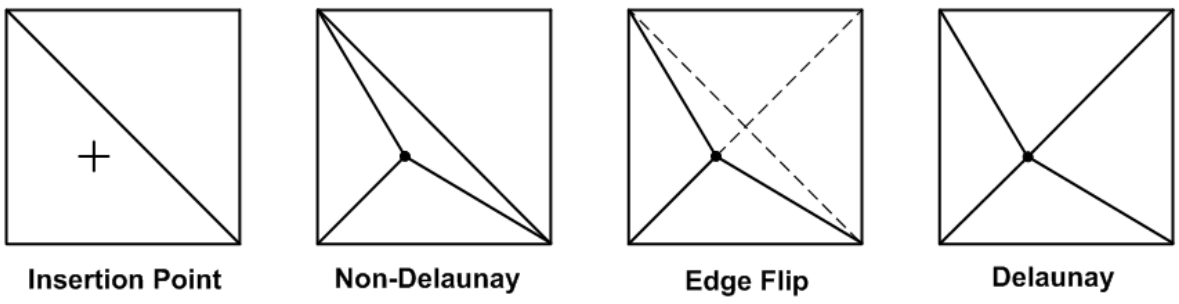

Figure 14 – Restoring the Delaunay property by flipping edges

Unfortunately, when a “non-Delaunay” edge is flipped, the job of restoring the Delaunay optimality is not necessarily finished. When the insertion point lies within the circumcircle of the immediately adjacent triangle, it may also lie within the circumcircles of one or more of the triangles adjacent to the neighbor. So when the insertion logic detects a non-Delaunay triangle, it must recursively search the "neighbor's neighbors" looking for additional edges that need to be flipped to restore Delaunay optimality. Fortunately, when the search encounters a "Delaunay edge" (one that does not need to be flipped), it does not need to continue beyond that point. Also, if the search encounters a perimeter edge, there is no need to continue further. Thus the recursive search will always terminate

Even though it is guaranteed to terminate, the recursive search may radiate outward and affect several layers of neighboring triangles. How many layers? In theory an insertion can affect the entire TIN. When processing the Bear Mountain sample, the early implementation encountered a case where the flipping operations radiated outward to 43 layers of surrounding triangles. In practice, the restoration of the Delaunay property usually involves no more than two layers (or six edges). Even so, the overhead related to testing and modifying edge links was sufficient to warrant an alternate approach.

2.2.8.2 Improved Performance using the Bowyer-Watson Algorithm

The insertion of a vertex using the Bowyer-Watson algorithm proceeds in 4 phases as illustrated in Figure 15 below. Once the containing triangle is located, the process creates a cavity in the TIN by removing non-Delaunay edges. It then connects the insertion vertex to the interior edges of the cavity, restoring the triangle mesh. Bowyer and Watson’s papers show that the resulting mesh is Delaunay optimal.

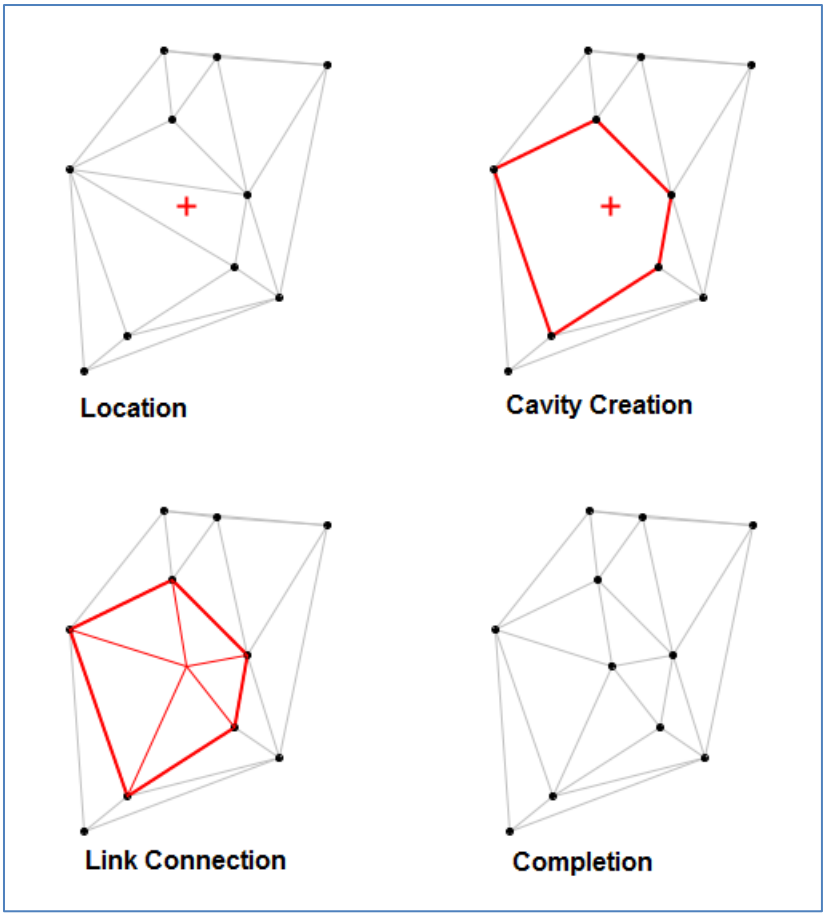

Figure 15 – Four phases of Bowyer-Watson insertion

As a further refinement, Tinfour combines the Cavity Creation and Link Connection steps into a single operation. Doing so improves the performance of the insert routine by reducing the number of times edge links must be reassigned. It does, however, complicate the code. For clarity, these notes will describe the insertion algorithm as separate steps. Readers interested in the details of the actual implementation may review the source code for the addWithInsertOrAppend() method in the IncrementalTin class.

Incidentally, the term "vertex insertion" is also used to describe the case where the vertex to be added lies outside the TIN. Cheng (2013) provides details on how a "ghost triangle" (one that includes a perimeter edge and the ghost vertex) can be processed with minor alterations to the overall logic described below (p. 59).

2.2.8.3 Overview of Bowyer-Watson insertion

Once the bootstrap operation is complete and an initial triangulated mesh is available, the Bowyer-Watson algorithm inserts vertices into the mesh using the following steps:

1. Location: For each vertex to be inserted, identify the enclosing triangle. If the vertex is outside the TIN, locate the ghost triangle such that the perimeter edge of the triangle is closest to the insertion point.
2. Uniqueness: By definition, every vertex in a TIN must have unique horizontal coordinates. When a vertex is added to Tinfour, it tests the insertion vertex against the three vertices of the enclosing triangle to determine if is distinct. If an insertion vertex is not unique, it is not added to the TIN. Instead, it is combined with the pre-existing vertex in a "vertex group". The structure of the TIN is not changed.
3. Insertion: If an insertion vertex is unique, identify mesh vertices that must be connected to the insertion vertex, removing edges as necessary to ensure that the mesh remains Delaunay optimal (this step also includes extending the mesh when the added vertex lies outside the perimeter of the TIN).

2.2.8.4 Vertex Location

The most direct method for Tinfour to locate the triangle that contains an insertion vertex is a sequential search through all existing triangles until it found a match. Unfortunately, such a process is slow, having a time complexity of $O(n^2)$ depending on the number of vertices in the input set. A faster approach using a "walk" algorithm was proposed by Lawson (1977). Figure 16 illustrates the concept of a traversal between two triangles in a Delaunay triangulation. As long as an algorithm can identify a reasonably direct path, the number of steps in the traversal is substantially less than the number of vertices in the mesh. Because such a path is readily obtained from a Delaunay triangulation, the insertion algorithm can use it to expedite the point location process. A comprehensive discussion of walk algorithms was given by Soukal (2012).

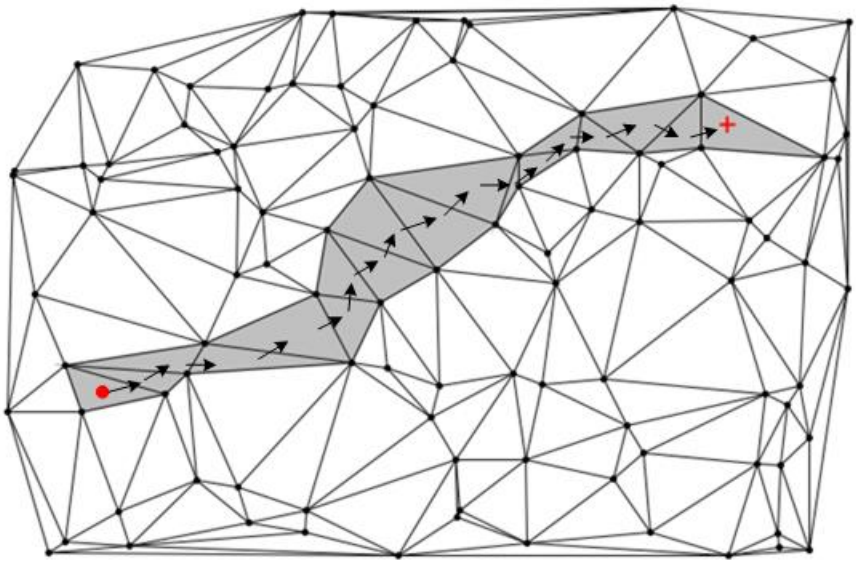

Figure 16 – A "walk" across a triangular mesh

Tinfour performs a vertex location operation using the following steps:

1. Recall that all triangles in the mesh are oriented in counterclockwise order. Therefore, if a vertex is contained by a triangle, it will be lie within the half plane to the left side of each interior edge.
2. For the first insertion after bootstrapping, select a “starting edge” using one of the interior sides of the initial triangle. For all subsequent searches, pick a starting edge from the most recently constructed triangle.
3. Test to see if the insertion vertex lies on or to the left side of the starting edge. If it does, proceed to step 4. If it does not, then it will lie to the left side of the dual of the starting edge, so transfer to the dual of the starting edge.
4. Repeat the following steps until the containing triangle is located or the traversal transfers to the exterior of the TIN:
    a. Obtain the forward edge. If the vertex is to the right of the forward edge, transfer to its dual and continue to step 5.
    b. Obtain the reverse edge. If the vertex is to the right of the reverse edge, transfer to its dual and continue to step 5.
    c. If the vertex is to the left of both the forward and reverse edges, then it must be in the interior (or on the edge) of the current triangle. The traversal terminates.
5. The search has transferred to the dual of an edge such that the vertex is to the left side of that edge. If the edge is an interior edge, continue the search from step 4.
6. If the edge is an exterior edge, identify the edge that subtends the vertex by moving to the left or right perimeter edges until the subtending edge is located. Terminate the search.

The steps above work properly for a unique, optimum Delaunay triangulation. Unfortunately, a non-optimum mesh may include regions in which the walk algorithm falls into a cyclic path and never

reaches a containing triangle. Lawson showed that an infinite loop may be avoided through by randomly alternating the order in which the forward or reverse edges are considered in steps 4.a and 4.b. Even if the walk lands in a potentially cyclic sequence of triangle hops, it will eventually transfer out of it from it by switching the order in which the neighboring edges are considered. Because of the randomization element in the walk algorithm, this approach is often called the “Stochastic Lawson’s Walk”.

Between each operation, Tinfour keeps track of the so-called "starting edge" so that each subsequent walk starts where the previous walk ended up. If two subsequent vertices within the overall sample set are spaced closely together (compared to the distance between other vertices), the number of steps required for the walk operation is reduced. On the other hand, if the set of samples were randomly positioned, the walk operation would tend to jump back and forth across the sample domain so that the overall length of the walk operations would be increased. So Tinfour's walk operations tend to be more efficient when subsequent vertices tend to be closer together than non-subsequent vertices. Such data sets, which have a "high degree of sequential spatial autocorrelation", can be processed more efficiently than those that do not have this characteristic. Fortunately, that is just the case in a typical lidar data set. Because the points in a lidar data set are collected using a scanning laser, and most lidar samples are given in the order collected, vertices derived from lidar will usually feature a high degree of sequential spatial autocorrelation. For the Bear Mountain data set, it took an average of 3.38 steps to locate a the triangle containing a vertex by using Lawson’s walk algorithm (this value was obtained using the SingleBuildTest described below).

2.2.8.5 Reducing Walk Lengths using a Sort Based on the Hilbert Curve

There is an obvious case where the assumption of sequential spatial autocorrelation does not apply: random samples. When samples are given at random positions across the input domain, it is unlikely that one sample will be positioned near its predecessor. For randomly positioned samples, the length of the average walk tends to be proportional to the square root of the number of points in the mesh (e.g. it is proportional to the length of the diagonal across the collection of points). The time-complexity of the point location, integrated over a large number of vertex additions, would approach $O(n^{3/2})$.

To reduce the number of steps required to insert a set of points with poor sequential spatial correlation, the Tinfour library implements a class for sorting the samples using an ordering scheme based on the Hilbert space-filling curve (Hilbert, 1891). Each point in the sample is projected onto the nearest segment of the Hilbert curve and assigned a sorting key based on its distance along the curve as shown in Figure 17 below. Because the Hilbert curve folds back on itself, points that are close together tend to have similar distance values. Thus the sort ensures that closely positioned points will be near each other in the resulting sequence of samples. This operation greatly improves the sequential spatial autocorrelation of the sample set. Thus, the time complexity of the vertex location process is reduced to that of the Java sort itself, which is typically better than $O(n \cdot \log n)$.

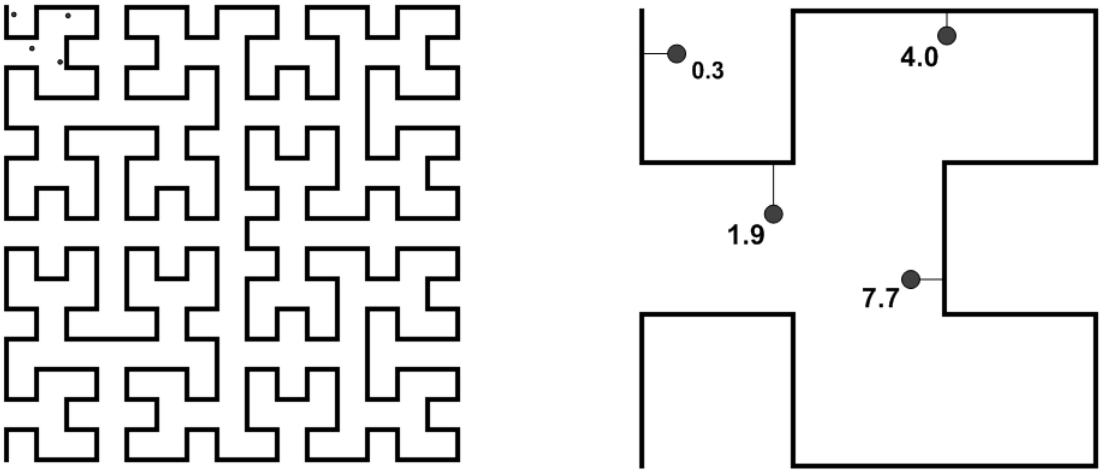

Figure 17 – Points Projected onto the Hilbert Space-filling Curve

Although the Hilbert sort can be beneficial when processing samples with poor autocorrelation, it is not appropriate for all data sets. For example, the Hilbert sort is seldom required for lidar samples because they usually have a high degree of sequential spatial autocorrelation. In fact, the sort can increase the overall processing time for lidar samples by adding an extra step which carries an upfront cost of its own and offers only modest reduction in the vertex location time. For example, performing a Hilbert sort on the Bear Mountain sample reduces the average traversal length from 3.38 to 3.12 steps. When that sample was tested with the Hilbert sort option, the time spent constructing the TIN was reduced by 106 milliseconds compared to the unsorted input. Unfortunately, the sort itself cost 236 milliseconds. So sorting the vertices before building the TIN led to a net increase of 130 milliseconds for the overall processing time. Clearly, the Bear Mountain sample was not a good candidate for the Hilbert sort. But in cases where an application has knowledge, *a priori,* that a sample has weak sequential spatial correlation, it can provide improved efficiency for processing. The sort can also be useful in application such as the Tinfour Viewer (described below) in which the same data set is processed multiple times (so that the cost of a single sort is amortized over many subsequent operations).

The logic for computing the Hilbert "rank" is based on the Lam & Shapiro method as described in Warren (2013, p. 358).

2.2.8.6 Vertex Uniqueness

Tinfour tests each insertion vertex to ensure that it is unique based on a minimum distance criterion. If the horizontal coordinates of a vertex is identical, or nearly identical, to those of an existing vertex, it is not inserted into the mesh. Instead, Tinfour creates a "vertex group" treating the non-distinct vertices as a single entity.

The VertexMergerGroup class extends Vertex by adding a list of vertices as one of its member elements. The first time Tinfour encounters a case where the insertion vertex is non-unique, it replaces the pre-existing vertex object with an instance of VertexMergerGroup constructed with its horizontal

coordinates. Both the insertion vertex and the pre-existing vertex are added to the group. When an application requires a vertical (z) coordinate for a vertex group, Tinfour extracts either the minimum, maximum, or mean value of the vertices depending on which access options have been set for the TIN. If an application uses the incremental TIN class's accessor methods to request a list of all vertices currently in the mesh, the resulting vertex collection (a Java List) contains the vertex group as an element. The insertion vertices that were bundled into the group are not included in the result, but can be obtained by accessing the group object that contains them.

2.2.8.7 Cavity Creation

In the procedure that follows, we describe an edge as being "Delaunay" if and only if the insertion vertex is outside the circumcircle of the triangle that lies to the opposite side of the edge. The algorithm creates the cavity by removing all edges that are non-Delaunay. As the edges are removed, the forward and reverse links of the adjacent edges are adjusted so that the cavity is bounded by a properly linked set of edges.

The cavity creation proceeds as follows:

1. Arbitrarily choose one edge of the enclosing triangle to be the "starting edge".
2. Designate the initial vertex of the starting edge as the "starting vertex".
3. Define an element as the "cursor" edge and set it to the starting edge.
4. If the cursor edge is Delaunay with respect to the insertion vertex, it will not be removed. If the opposite vertex is the ghost vertex, the edge will not be removed and the edge will be treated as "effectively" Delaunay. Use the InCircle calculation to determine if the cursor edge is Delaunay. If the InCircle calculation is ambiguous, treat the edge as being Delaunay. Is the edge Delaunay?
    a. Yes: Do not remove the edge. Transfer the cursor to its own forward edge.
    b. No: Remove the edge from the mesh, adjusting the links of the adjacent edges to maintain the cavity polygon links. Transfer the cursor to its dual's forward edge.
5. If the initial vertex of the cursor edge is the starting vertex, the cavity creation procedure is complete. Otherwise, repeat from step 4.

It is possible that all edges of the enclosing triangle will be properly Delaunay and that the "cavity" polygon will simply be the original enclosing triangle.

2.2.8.8 Link Connection

The resulting polygon may be convex or non-convex, but the work of Bowyer and Watson shows that it will be ordered strictly in counterclockwise order. Furthermore, all edges constructed between the insertion vertex and the polygon vertices will be Delaunay optimal. The link connection procedure is straightforward and the resulting triangles will be specified in counterclockwise order. Furthermore, as long as all the InCircle calculations were unambiguous, the resulting mesh will be Delaunay optimum and unique. Otherwise, it will be "nearly Delaunay" and non-unique. Although Tinfour could implement additional rules for "disambiguating" situations where the InCircle calculation gives an ambiguous (zero) result, none are in place at this time. Thus, it is possible that the same set of sample points may give rise to different TINs depending on the order in which they are added to the mesh.

2.2.9 Coordinates and Numerical Issues

Computational geometry applications are notorious for issues with numerical precision. Computations based on expressions with exact algebraic solutions often fail due to round off or approximation errors due to the limits of floating-point arithmetic. Specific issues are discussed as they arise in the discussion that follows, but two general considerations are worth noting:

1. Nearly identical vertices: The triangulation algorithms used in Tinfour depend on each vertex in the mesh being unique. If vertices are too close, numeric computations that combine their values could cause errors in the construction of the TIN. In order to avoid issues with two vertices being spaced so closely together that computations fail, Tinfour must define a threshold distance for treating "nearly identical" vertices as the same point.
2. Situations requiring extended precision arithmetic: In some cases, Tinfour will use extended precision arithmetic to determine geometric relationships between features (for example, on which side of a line a vertex lays). Because extended arithmetic requires more processing than standard floating point calculations, Tinfour implements threshold values so that when some standard calculations produce a value "close to zero", the alternate extended precision value calculations can be employed.

The assignment of threshold values depends on the magnitude of the data being modeled. Coordinate values for an application for modeling the distribution of nutrients in a call culture would be of a much different scale than those for one based on weather observations taken hundreds of kilometers apart.

When computing threshold values, the constructors for the Incremental TIN classes allow an application to specify a value related to the average spacing of the vertices to be built into the TIN. The default constructor assumes a value of 1 unit (meters, feet, parsecs, etc.). Other constructors allow an application to specify values as appropriate.

The threshold for considering two vertices identical is $1/10000^{th}$ of the average point spacing. Applications may vary this by using the Thresholds class defined in the Tinfour project.

2.2.9.1 Large-magnitude coordinates for vertices

Although Tinfour is by no means limited geographic applications, geophysical modeling is a major use of Triangulated Irregular Networks and requires that software be able to manage large-magnitude coordinates. The Earth is a big place. Our planet is 40081299 meters around its equator. But geographic applications often deal with situations in which one meter is a significant distance. So, in some cases, resolving a coordinate to the nearest meter requires at least eight digits of precision. Doing so requires the use of Java's 8-byte double floating-point data type which provides about 16 decimal digits of precision. The 4-byte float type, which provides only 7 digits, is usually sufficient for *vertical coordinates* (elevations), but is inadequate for *horizontal coordinates* (x/y, latitude/longitude, etc.).

For example, one of the datasets used to test Tinfour was a lidar survey conducted near Pole Creek in Oregon. The area contained about 5 million ground points with an average spacing of about 0.5 meters. Horizontal coordinates were given in a projected coordinate system (x/y values in meters rather than

latitude/longitude). Coordinates from the first six points are given below. Although the horizontal coordinate values are quite large, the differences between subsequent values are small.

X	Y	Z
605075.31	4891924.60	1610.51
605074.19	4891924.54	1610.56
605072.07	4891924.84	1610.90
605071.97	4891925.62	1611.99
605067.98	4891925.29	1611.64
605055.42	4891925.26	1611.71

The x and y coordinates listed above require 8 and 9 digits of precision respectively, and all within the range supported by an 8-byte floating point value. But consider what happens when they are used in arithmetic that depends on the product of these coordinates such as that used in the well-known formula for the area of a polygon. Given a set of points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ that describe a polygon, with subscript n+1 corresponding to point 1, the area *A* of the polygon is:

$$A = \frac{1}{2} \sum_{i=1}^{n} x_i y_{i+1} - x_{i+1} y_i$$

The 17 digits of precision required for the product of the x and y can exceed the precision of an ordinary 8 byte floating point value, so that the lower-order digits wash out of the expression. But because the points in a lidar sample are close together, the low-order digits are just the ones that interest us. In taking the product and difference of these terms, the information they carry will be lost.

Some TIN implementations attempt to address this problem by normalizing all input coordinates (scaling them to the range 0 to 1 while building the TIN). Because one of the goals for Tinfour was to maintain data elements in their original form, it does not modify input coordinates. Instead, it avoids numeric issues by using alternate algebraic forms of numerically vulnerable expressions. For example, the area of a polygon does not change when it is moved rigidly to another position. So if we reduce the magnitude of the coordinates in the expression above by subtracting a fixed offset, we can avoid numeric issues without affecting the resulting area value. The following expression shows one way of translating the polygon coordinates to a position close to the origin.

$$A = \frac{1}{2} \sum_{i=2}^{n-1} (x_i - x_1)(y_{i+1} - y_1) - (x_{i+1} - x_1)(y_i - y_1)$$

Although the area example is rather contrived, the, the InCircle function described above is a good example of where an alternate algebraic form is used to avoid loss of significant digits. Different authors

have used different forms of the InCircle determinant, but the form given by Cheng avoids precision issues.

For more detail, see the Wikipedia article on Loss of significance.

2.2.9.2 Special Considerations for Geographic and Projected Coordinates

Data collected for geophysical data samples present a special issue when using a TIN for processing: geographic coordinates are not isotropic. Coordinates given in the X (longitude) and Y (latitude) directions do not have a consistent unit of measure. This phenomenon is easily visualized by picturing the way meridians (lines of longitude) converge at the poles. The distance between parallels (lines of latitude) is uniform everywhere on the globe, but the distance between meridians decreases at latitudes away from the equator.

Users with experience in Geographic Information Systems (GIS) are used to the issue of converting data from geographic coordinates to some local projected coordinate system (map-based coordinate system) so that features can be processed using a 2D Cartesian coordinate system with a consistent unit of measure in all directions. Many geographic calculations depend on this quality, including the InCircle and surface gradient calculations performed by Tinfour.

Because geographic coordinate are not isotropic, most lidar products are initially produced in a projected coordinate system. However, some of the best sources for data on the Internet present samples in geographic coordinates. For example, all data available at the NOAA Digital Coast site (NOAA, 2015) is converted to geographic coordinates before being posted. In 2011, the U.S. Department of Agriculture collected a sample of lidar data over a Litchfield County in Northwest Connecticut (NOAA, 2011a). The sample consisted of 1742 one-kilometer squares. The coordinates for the original data were given in the Universal Transverse Mercator (UTM) projection, zone 18N. However, before NOAA posted the data to their FTP site, they converted it to a geographic coordinate system. In the region covered by the survey, one degree of latitude represents a distance of about 35.4 kilometers, but one degree of longitude represents a distance of about 26.4 kilometers. So the representation of the data for that area is distinctly non-isotropic.

If at all possible, when processing data specified in geographic coordinates, it is useful to transform it to a projected coordinate system. If that is not possible, at least be sure to specify a reasonable value for the average point spacing when initializing the Tinfour incremental TIN classes. In the case of the Connecticut data above, the average point spacing is about 1 meter. So, in geographic coordinates, the average separation between data points would be about one 35.4 thousandth of a degree (if aligned vertically).

2.2.10 The Constrained Delaunay Triangulation

The Delaunay techniques described above are based on the assumption that the triangulation process is free to associate neighboring vertices based on the Delaunay criterion. In some cases, however, doing so is not necessarily the best treatment of the data. Turning again to the example of elevation data, consider the case when the terrestrial surface of interest include a cliff, road cut, escarpment, or even a

body of water. Connecting vertices on opposite sides of such a boundary may not necessarily be the best treatment of the data.

The Constrained Delaunay Triangulation allows the insertion a collection of edges into the triangular mesh that supersede the Delaunay criterion and constrain the way vertices are connected in the mesh. The figure below illustrates the concept. The data shown occurs in two separate regions. The ordinary Delaunay creates connections between the separate vertices at will. The constrained Delaunay adds more information to the system in the form of edges that define the limits of the data regions. In the figure, the constraint is shown as the vertical edge at the center of the triangulation.

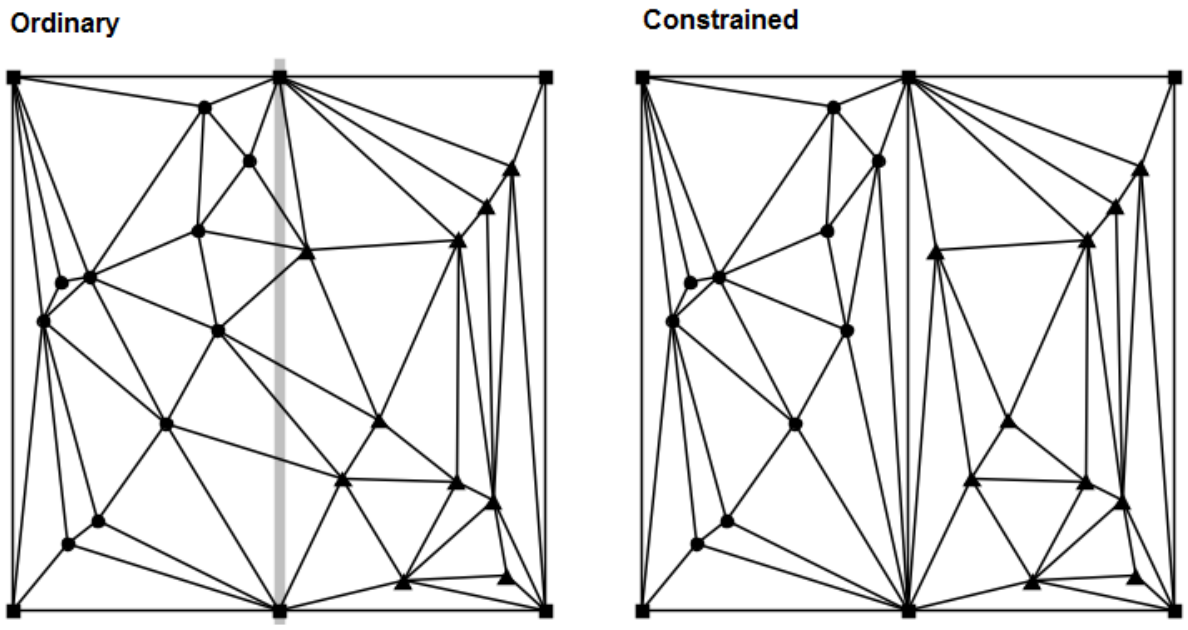

Figure 18 – Ordinary and constrained Delaunay triangulations

One of the drawbacks of adding constraints to the triangulation is that not all the triangles in the mesh will necessarily conform to the Delaunay criterion. In particular, the constraint may give rise to "skinny" triangles such as those that appear near the constrained edge in the figure. Such artifacts are often undesirable when using the triangulation to interpolate values or model a surface. Also, many applications take advantage of the fact that the Delaunay Triangulation is easily mapped to another important graphical structure, the Voronoi Diagram. If the addition of constraints renders the triangulation non-Delaunay, it no longer has an associated Voronoi Diagram.

One way to restore Delaunay optimality was described by Rognant, et al. (1999), who also offered a brief mathematical proof of the technique. The technique adds synthetic points along the constraint edges as illustrated in the figure below. The constraint edges are subdivided into smaller edges and the resulting triangles all conform to the Delaunay criterion. Optimality is restored.

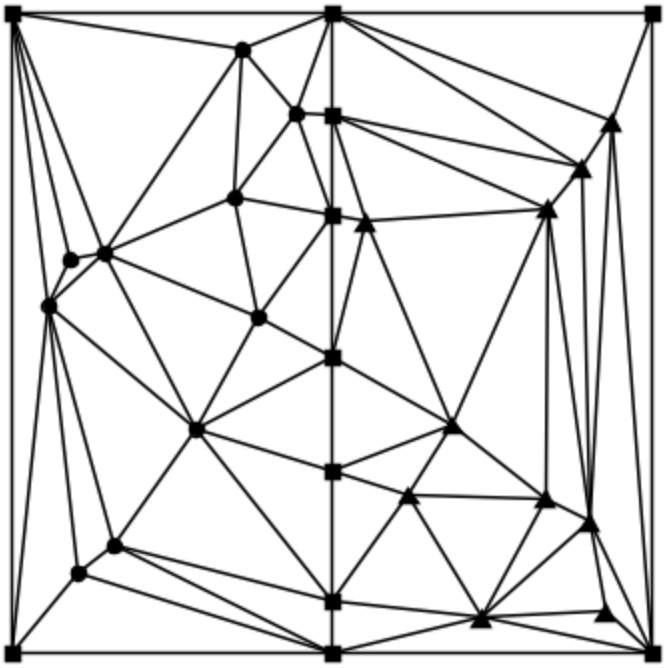

Figure 19 – Optimality restored

The applications of the CDT extend beyond terrain modeling into many areas of data modeling. An example of the CDT that has nothing to do with surface elevation is provided by the LogoCDT application which is included in the Tinfour software distribution. An image from the application is shown below.

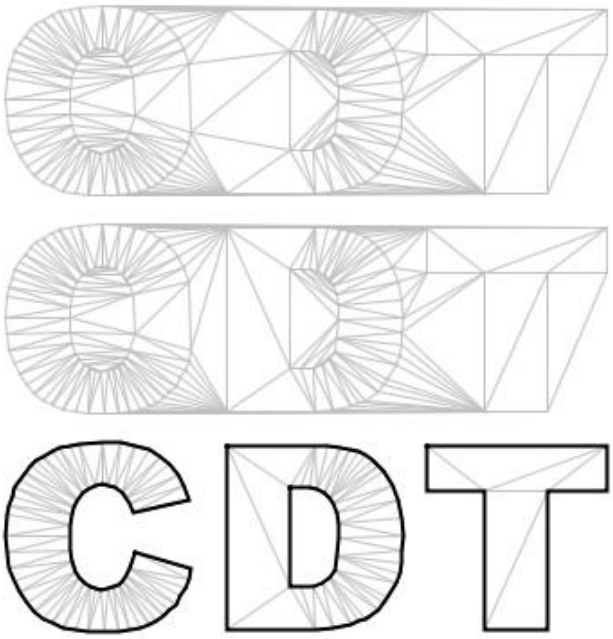

Figure 20 – The ordinary Delaunay, the constrained, and the constrained with exterior removed.

Tinfour implements a concept called the “constrained region” that permits polygon-based constraints to define regions with application-defined metadata. This metadata is usually specified in the form of Java objects added to the constraint when it is constructed. For example, the picture below was composed using a global-scale product from the public-domain Natural Earth map project. Each of the country polygons was populated with a Java Color object. A test application was written to render the interior edges of from the resulting TIN using the colors registered with the associated polygons.

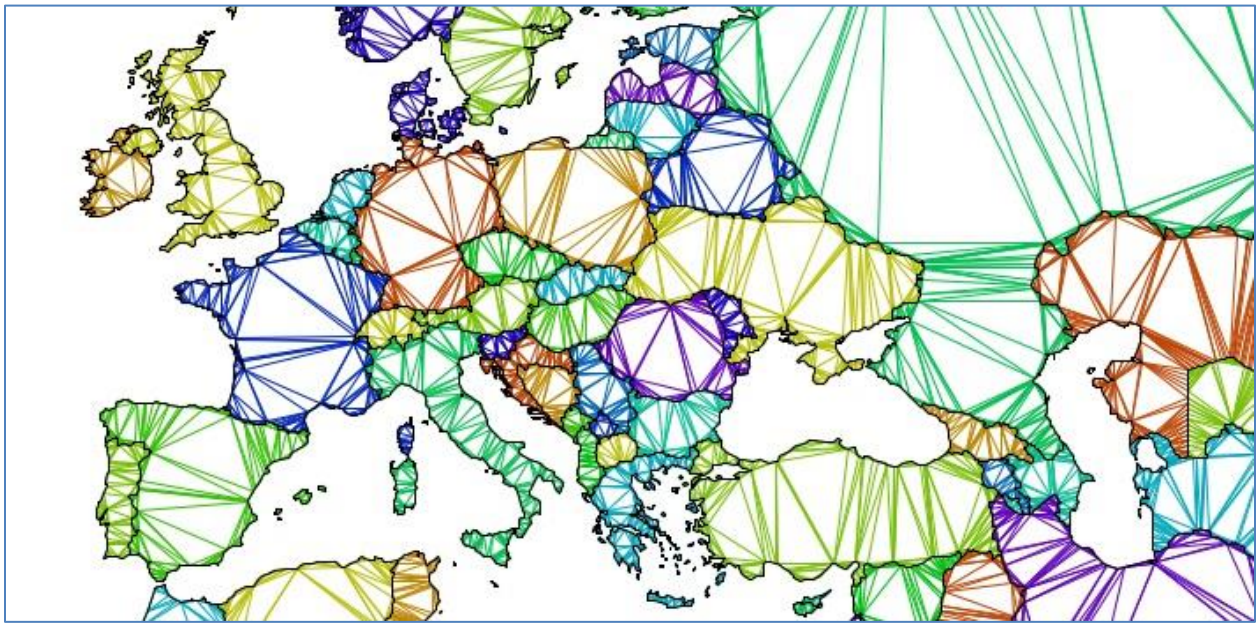

Figure 21 – Image rendered using the constrained-region feature

2.3 Interpolation

Interpolation is probably the most common application of a triangular mesh. Tinfour implements three different interpolation techniques: Triangular Facets, Natural Neighbors, and Geographically Weighted Regression Polynomials. Because all three techniques are implemented so that they access the TIN on a read-only basis, it is possible operate multiple instances of any of the Tinfour interpolation classes in parallel using a multi-threaded approach.

2.3.1 Techniques Implemented by Tinfour

The interpolation methods implemented by Tinfour are described below.

2.3.1.1 Triangular Facets (TriangularFacetInterpolator.java)

The triangular facets technique determines a value for the interpolation point (x,y) by using the three vertices of the triangle that contains the point to derive a the equation for a plane z=f(x,y) and then solving for the z value at the specified coordinates. If the point lies outside the bounds of the TIN, it is projected onto its subtending boundary edge and assigned a value using a two-point interpolation. While this method is fast and robust, it is not suitable for most applications because the first-derivative of the resulting surface is not continuous across the edges of adjacent triangles.

2.3.1.2 Natural Neighbors (NaturalNeighborInterpolator.java)

Sibson's Natural Neighbor interpolation technique uses a weighted sum of vertices in the neighborhood of (x,y) to compute a z value. The technique is fast, accurate, and provides first-derivative continuity across the TIN except at its vertices (Sibson proposed an additional method that provided first-derivative continuity everywhere, but it is not yet implemented in Tinfour). The Natural Neighbor interpolation provides good results and is an excellent choice for many applications. However, it is not defined on the edges or the exterior of the TIN. Also the Tinfour logic for computing surface normal tends to be non-responsive to local changes in slope, so that visual results will appear to have filtered away finer details.

Unlike many Natural Neighbor implementations, the Tinfour algorithm does not alter the TIN when it is used to perform an interpolation. In effect, it accesses the TIN on a read-only basis. Thus it is possible to use the interpolator in concurrent, multi-threaded applications that perform analysis or other operations over the TIN.

2.3.1.3 Geographically Weighted Regression (GwrTinInterpolator.java)

One of the fundamental ideas of geographic information analysis is the idea that data collected near a point of interest is more relevant than that from farther away. So if we were to interpolate a point on a surface by using a regression technique, it seems a natural to assign a greater weight to those sample points nearby than to those further away. That, in effect, is what the Geographically Weighted Regression (GWR) technique attempts to do.

The GwrTinInterpolator class selects a limited set of input samples in the vicinity of (x,y) and applies a weighting factor to each sample based on the inverse of their distance from the interpolation point.

Although the regression technique is the most processor-intensive of the methods supported by Tinfour, it is the only technique which provides a statistics describing the quality of the interpolated result including both the standard deviation and prediction interval for its interpolated value. Also, because the result of this interpolation is a polynomial (of degree 1, 2, or 3), it can be used to find derivatives and second derivatives for the surface in the neighborhood of (x,y). This feature makes the regression technique useful in applications requiring surface normal, slope, or curvature values. The GWR is the interpolation of choice for Tinfour's example hillshade implementation.

The GWR technique also has the feature that it is not intrinsically tied to the triangulated mesh as the other techniques. It requires only a collection of samples in the vicinity of the interpolation coordinates. So in addition to the TIN-based implementation GwrTinInterpolator, Tinfour also includes a class called GwrInterpolator which accepts an array of sample points for interpolation purposes. Your application can collect these sample points using whatever means is appropriate, and then use GwrInterpolator to perform its interpolation. No TIN would be required. The Tinfour library also exposes an internal class, SurfaceGWR, which performs most of the internal operations related to GWR modeling.

More detail on the GWR technique is given below.

2.3.2 Cross Validation

One of the precepts of geospatial analysis is the idea that features near to each other tend to have similar properties. We expect that two elevation samples taken close together will be more likely to have similar values than samples taken far apart. In geographic information analysis, this idea is often referred to as spatial autocorrelation.

With this precept in mind, the question arises as to how well the value of a particular sample point can be predicted by its neighbors. The cross validation process explores that question using the following process:

1. Construct a TIN from a set S of sample points.
2. One at a time, temporarily remove a sample point $s_i = (x_i, y_i, z_i)$ from the TIN and then use the TIN to interpolate a value for z at (x_i, y_i).
3. Compute the error $e_i = |z-z_i|$.
4. Restore sample point s_i back to the TIN and continue to process the rest of the samples in the TIN.

The mean value and variance of the error terms provides a metric for evaluating both the degree of spatial autocorrelation of the sample set and the relative success of a particular interpolation strategy. The following output was obtained using one of the example applications included in the Tinfour source distribution, ExampleCrossValidation, to process part of the Bear Mountain sample.

```
Tested 663592 of 1036879 vertices
Method                        mean |err|  std dev |err|    range of err
Triangular Facet               0.049917      0.047752     -2.524    2.473
Natural Neighbor               0.048697      0.046385     -2.516    2.381
GWR, Fixed Bandwith 0.86       0.048607      0.046385     -2.395    3.586
GWR, Proportionate  0.45       0.048021      0.045532     -2.447    2.318
```

The text shows results from all three interpolation classes with two variations of the Geographically Weighted Regression method which are discussed below. In general, all produce values close to 5 centimeters (about 1.96 inches). Examining the source data shows, not surprisingly, that the most severe errors occur in areas with very steep or discontinuous surfaces such as cliffs or escarpments.

2.3.3 The Geographically Weighted Regression (GWR) Technique

The Geographically Weighted Regression technique was introduced in paragraph 2.3.1.3 above.

Geographically Weighted Regression (GWR) is a statistics technique that is used in situations where the nature of the data may vary over a geographic area, a characteristic which is referred to as "non-stationarity". In GWR, a weighting factor is applied to each sample based on its distance from a point of interest. Thus, the more distant a sample, the less information it contributes to the regression estimate. While GWR is a very general technique, the Tinfour implementation focuses on the special case where the data is a real-valued surface (a "field") with isotropic coordinates. This treatment is well suited to terrain analysis and related problems. More general implementations are available on the web through the GWR4 application[1] and through modules for the R statistics application. More information about the technique is available on the web from Wheeler and Páez (2010).

Tinfour's GwrTinInterpolator class provides optional interpolation arguments that are not used by the other interpolators: surface model and bandwidth selection. The surface model option allows an application to specify the degree of the polynomial that the interpolator uses to model the surface. An application may specify a planar model, a quadratic model, or a cubic model. The bandwidth selection option permits an application to specify different strategies for assigning weights to the samples.

2.3.3.1 Surface Models

The polynomials that are derived in response to the surface model selection are given in the following forms which correspond directly to the states in the Java enumeration SurfaceModel which is part of the Tinfour interpolation package. The interpolating coefficients $\beta_0, \beta_1, \ldots, \beta_n$ are derived from the input samples using the regression operation.

planar	$\hat{z}_1 = \beta_0 + \beta_1 x + \beta_2 y$
planar with cross terms	$\hat{z}_2 = \beta_0 + \beta_1 x + \beta_2 y + \beta_3 xy$
quadratic	$\hat{z}_3 = \beta_0 + \beta_1 x + \beta_2 y + \beta_3 x^2 + \beta_4 y^2$
quadratic with cross terms	$\hat{z}_4 = \beta_0 + \beta_1 x + \beta_2 y + \beta_3 x^2 + \beta_4 y^2 + \beta_5 xy$

[1] Unfortunately, at the time of this writing I was unable to find a copy of GWR4 on the web except at a site that wanted to push out adware. Testing was conducted using an older version which is no longer available.

cubic $\hat{z}_5 = \beta_0 + \beta_1 x + \beta_2 y + \beta_3 x^2 + \beta_4 y^2 + \beta_5 x^3 + \beta_6 y^3$

cubic with cross terms $\hat{z}_6 = \beta_0 + \beta_1 x + \beta_2 y + \beta_3 x^2 + \beta_4 y^2 + \beta_5 xy + \beta_6 x^2 y + \beta_7 xy^2 + \beta_8 x^3 + \beta_9 y^3$

When an interpolation is computed for coordinates (x, y), the Tinfour methods uses the TIN to identify a set of samples in the neighborhood of the interpolation coordinates, selecting a number of input samples appropriate to the degree of the surface model. Next, weighting factors are assigned to each sample based on their distance from the interpolation point. Finally, the horizontal coordinates of the input samples are adjusted by the offset (-x, -y) before the regression is performed. With that adjustment, the interpolation point is treated as the origin and the estimating function is evaluated for coordinates (0, 0). Thus the value of the interpolation is just the resulting estimator β_0 from the interpolation. The main purpose of this approach is to reduce the impact of numerical issues and the loss of precision when performing the linear regression. But it also has the side effect of greatly simplifying the calculations at the interpolation coordinates, particularly when obtaining slope, surface normal, or second derivatives.

For all six models evaluated at coordinates (0,0), the following results apply.

$$z_{Interp} = \beta_0$$

$$z_x = \frac{\partial z}{\partial x}(0,0) = \beta_1$$

$$z_y = \frac{\partial z}{\partial y}(0,0) = \beta_2$$

With the unit normal to the surface at the interpolation point being given by

$$\vec{n} = \frac{(-z_x, -z_y, 1)}{\sqrt{{z_x}^2 + {z_y}^2 + 1}} = \frac{(-\beta_1, -\beta_2, 1)}{\sqrt{{\beta_1}^2 + {\beta_2}^2 + 1}}$$

The *aspect* of the surface at the slope at the interpolation point is defined as direction of steepest ascent taken counterclockwise from the x axis. It is computed as $\theta = \tan^{-1}(z_y/z_x)$ with the appropriate adjustment of angle for quadrant based on the signs of the two input arguments. Many programming languages, including Java, include a function called atan2() which automatically makes this adjustment when computing tangent. Using the simplifications due to treating the interpolation point as the origin, the aspect is given by

$$\theta = atan2(\beta_2, \beta_1)$$

Applications dealing with hydrography are often more interested in the direction of *steepest descent* which, of course, gives the expected direction for the flow of surface water:

$$\theta = atan2(-\beta_2, -\beta_1)$$

The *slope* at the interpolation point is derived from the magnitude of the gradient vector $|\nabla z|$ as

$$s = \sqrt{{z_x}^2 + {z_y}^2} = \sqrt{{\beta_1}^2 + {\beta_2}^2}$$

Second derivatives are easily computed for the quadratic and cubic models:

$$z_{xx} = \frac{\partial^2 z}{\partial x^2}(0,0) = 2\beta_3$$

$$z_{xx} = \frac{\partial^2 z}{\partial y^2}(0,0) = 2\beta_4$$

And for the quadratic and cubic models *with cross terms*:

$$z_{xy} = \frac{\partial^2 z}{\partial x \partial y}(0,0) = \beta_5$$

Wilson and Gallant (2000, p. 53) give the following expression for *profile curvature* which is the rate of change of slope measured along a unit vector pointing in the direction of descent.

$$K_p = -\frac{z_{xx}{z_x}^2 + 2z_{xy}z_x z_y + z_{yy}{z_y}^2}{\left({z_x}^2 + {z_y}^2\right)\left({z_x}^2 + {z_y}^2 + 1\right)^{3/2}}$$

The value for profile curvature, K_p, and the other curvatures described below is given in radians per unit distance.

According to Peckham (2011, p 27), the extra factor $({z_x}^2 + {z_y}^2 + 1)^{3/2}$ in the denominator defines curvature "based on differential movements along the 3D streamline curve (that lies on the surface)" rather than on the horizontal plane. Wilson and Gallant also provide the following expression for *contour curvature* (sometimes called *plan curvature*) which describes the curvature of a contour line at a point of interest on the surface. Contour curvature essentially describes the rate at which a flow path on the surface would be converging (negative) or diverging (positive):

$$K_c = -\frac{z_{xx}{z_y}^2 - 2z_{xy}z_x z_y + z_{yy}{z_x}^2}{\left({z_x}^2 + {z_y}^2\right)}$$

By again adding the extra factor to the denominator, Peckham obtains *tangential curvature* which is adjusted to match the curvature along the streamline curve rather than the horizontal plane.

$$K_t = -\frac{z_{xx}{z_y}^2 - 2z_{xy}z_x z_y + z_{yy}{z_x}^2}{\left({z_x}^2 + {z_y}^2\right)\left({z_x}^2 + {z_y}^2 + 1\right)^{3/2}}$$

Peckham gives the streamline curvature which describes the rate of change in the direction of the flow (as measured on the horizontal plane):

$$K_s = -\frac{z_x z_y\left(z_{xx} - z_{yy}\right) + \left(z_y{}^2 - z_x{}^2\right) z_{xy}}{\left(z_x{}^2 + z_y{}^2\right)^{3/2}}$$

Peckham gives a clear and concise derivation for all of the expressions given above and offers useful insights on applying them to the modeling of surface flow and related phenomena.

Finally, it is worth noting that the expressions given above can be used for points other than the interpolation point provided that the appropriate offset is added to the coordinates of interest. While the simplifications described above cannot be used when computing z_x , z_y , etc., the algebra is straightforward.

2.3.3.2 Bandwidth Strategies for Sample Weighting

The minimum number of samples required to perform a linear regression depends on the number of coefficients in the target polynomial. Tinfour automatically selects a set of sample points from the neighborhood of the interpolation coordinates based on the specified model. Thus when performing an interpolation, the larger order polynomials have two characteristics: they will require more sample points, and the average distance of the sample points from the interpolation coordinates will increase resulting in an increase of the overall area from which the samples are collected. Since one of the ideas of geospatial analysis is that the nearby points are more relevant than the more distant ones, some method is required to account for the expanded data set characteristics.

In Geographically Weighted Regression, the relative contribution of neighboring sample points is adjusted by an inverse-distance weighting function. Wheeler (2010, p. 463) and other authors have identified several functions for distance weighting. Tinfour uses the Gaussian kernel

$$W_i = e^{-\frac{1}{2}\left(\frac{d_i}{\gamma}\right)^2}$$

Where d_i gives the distance for the *ith* sample from the interpolation coordinates and W_i gives the weighting factor to be used for that sample in the regression computation. The parameter γ in the equation above is referred to as the "bandwidth" of the weighting function. It should be expressed in the same unit of measurement as the distance. The idea of bandwidth is roughly analogous to the use of the term in communications theory in that it controls how much information is accepted from distant samples when performing a regression. If γ is large, it reduces the size of the distance term and increases the overall weight of the sample. Thus with a larger bandwidth value, more distant samples receive larger weights and contribute more to the overall interpolation. For a sufficiently large bandwidth, the distance term would be effectively zero and the weight of all samples approach unity. In that case, the GWR would be equivalent to an ordinary least squares estimation. If the bandwidth parameter is small, the distance term becomes more significant and the contribution of more remote samples is decreased.

Tinfour provides four methods for bandwidth selection as described below:

1. Fixed bandwidth supplied by the application.
2. Proportional bandwidth selected as a multiple of average distance of the locally selected samples from the interpolation coordinates based on a parameter supplied by the application.
3. Automaticallly selected bandwidth in which an optimal bandwidth is selected based on the Akaike Information Criterion with correction for small sample sizes (AICc method).
4. Ordinary Least Squares (uniform weighting of all samples).

The fixed bandwidth method is preferred in cases where an application has information or requirements specific to the data and needs to enforce a particular criterion. It has the disadvantage that an application must implement its own logic for establishing the bandwidth parameter. There are no fixed rules for deriving this parameter. For the Bear Mountain sample shown in the discussion on cross validation above, the ExampleCrossValidation application simply used the average point spacing for the overall sample set.

The proportional bandwidth is preferred in cases where an application has no *a priori* information about the data set and wishes to apply a bandwidth that is proportionate to the average distance of the locally selected samples from the interpolation coordinates. This method has the advantage of adjusting the bandwidth to meet local density of the samples.

Both the fixed and proportional bandwidth methods have the disadvantage of being arbitrary settings that are not driven by a statistical model of the data. In contrast, the automatic bandwidth method is based on recognized statistical techniques for model selection (bandwidth being considered an element of the model). The AICc criterion provides a way of judging the relative performance of different combinations of surface model and bandwidth value for a particular set of samples. In conventional regression applications, the criterion is often used to determine which combination or subset of interpolation variables lead to the best quality estimator for a response surface. In the automatic selection process, Tinfour use the criteria to select a surface model selection and bandwidth combination that produce a good representation of the surface in the vicinity of an interpolation point.

Unfortunately, the calculation of the AICs score is computationally expensive. It requires far more processing than the regression itself. Consequently, the overall process of automatic model and bandwidth selection is quite slow. Processing rates of 100 to 200 interpolations per second are typical. The problem is exacerbated by the fact that the treatment of AICc score as a function of bandwidth over a given set of samples often yields a highly complicated curve with many local extrema and inflection points. Thus simple numerical techniques like Newton's method cannot be used to find optimal values. Instead, processor-intensive search techniques must be employed.

2.3.3.3 Interpreting the Results

The results below were performed by the cross-validation application with the automatic model and bandwidth options enabled. Because the automatic method is so slow, only a subarea of the lidar sample was processed. For this area, the overall error tallies are somewhat larger than they were in the

example above. The details below also provide the mean and standard deviation of the bandwidth values that were selected as well as counts of how many times particular surface models were chosen.

```
Tested 9453 of 1036879 vertices (153.10/sec)
Method                        mean |err|  std dev |err|     range of err     sum err
Triangular Facet               0.052792      0.047142     -0.334    0.533     2.242
Natural Neighbor               0.051293      0.045680     -0.342    0.414    -3.003
GWR, Fixed Bandwith 0.86       0.051572      0.045812     -0.340    0.403     0.748
GWR, Proportionate  0.45       0.051091      0.045509     -0.336    0.405    -0.345
GWR, Automatic BW AICc         0.052512      0.047208     -0.346    0.377    -1.037

Values for automatically selected bandwidth
   Mean:               0.728962   (0.400464 of mean dist)
   Std Dev             0.133318   (0.071733)
   Min,Max:            0.370358,     1.217216  (0.300, 0.650 of mean dist)

Number of Ordinary Least Squares: 0
Planar                             10
PlanarWithCrossTerms               81
Quadratic                        1490
QuadraticWithCrossTerms           814
Cubic                            1360
CubicWithCrossTerms              5698
```

Considering how much processing the automatic bandwidth selection method requires, the fact that it yields the largest mean absolute error of all the regression methods is disappointing. One could reasonably ask whether the implementation is correct. While that remains an open question (see *The State of the GWR Implementation* below), there is an alternate explanation. One of the reasons that the error is somewhat higher for the GWR is that the AICc criterion is designed to select the configuration that produces the best *overall* interpolating polynomial for the results. The elevation at the interpolation point is based on the use of just one coefficient from the result: β_0. The AICc process attempts to find an optimal set of interpolating coefficients: $\beta_0, \beta_1, \ldots, \beta_n$. Thus a bandwidth and model selection that yields a strong β_0, but weak higher-order coefficients, would have a correspondingly poor AICc score and would not be preferred to a selection that produces good coefficients overall. So in cases where an application was as interested in estimates of characteristics derived from surface derivatives – including slope, curvature, and surface normal – the weaker cross validation results would be of less concern.

Also, it is worth noting that success of GWR in a cross-validation technique is a measure of how well the values of a sample point can be predicted from that of its neighbors. That, in turn, is a direct reflection of the kind of terrain in a prediction area. The Bear Mountain sample represents rough terrain with numerous boulders, escarpments, and steep slopes. A more moderate terrain from a different tile in the Connecticut survey area (20111218_18TXM2835.las, which is mostly farmland) produced quite different results. These are shown in the table below. While the automatic bandwidth selection still features the largest magnitude errors, that deviation is only 2.3 centimeters… a value less than the width of two fingers. Results like that give one an appreciation of just how good Lidar technology has become.

```
Tested 12675 of 1193868 vertices (148.57/sec)
Method                        mean |err|  std dev |err|     range of err     sum err
Triangular Facet               0.022933      0.025198     -0.281    0.265    -0.083
Natural Neighbor               0.022473      0.024584     -0.320    0.249    -2.920
GWR, Fixed Bandwith 0.80       0.022708      0.024311     -0.303    0.270    -1.236
GWR, Proportionate  0.45       0.022473      0.024254     -0.303    0.266    -0.208
```

```
GWR, Automatic BW AICc          0.023215      0.025326     -0.343     0.312     1.819
```

2.3.3.4 Application Access to GWR Results

The following snippet of code shows how the results for an interpolation at coordinates (x,y) could be obtained and output to a Java PrintStream (ps).

```
// Perform the interpolation for coordinates (x,y) given a valid TIN
// using a specified surface model and the Automatic Bandwidth selection
GwrTinInterpolator gwr = new GwrTinInterpolator(tin);
double z = gwr.interpolate(
  SurfaceModel.Cubic,
  BandwidthSelectionMethod.AutomaticBandwidth, 0,
  x, y, null);

// Obtain the associated results for the most recent interpolation,
double[] beta = gwr.getCoefficients();
double []predictionInterval = gwr.getPredictionInterval(0.05);
double zX = beta[1];
double zY = beta[2];
double zXX = 2*beta[3];
double zYY = 2*beta[4];
double zXY = beta[4];
double azimuth = Math.atan2(zY, zX);
double compass = Math.toDegrees(Math.atan2(zX, zY));
if(compass<0){
  compass+=360;
}
double grade = Math.sqrt(zX*zX+zY*zY);
double slope = Math.toDegrees(Math.atan(grade));
double kP = (zXX*zX*zX+2*zXY*zX*zY + zYY*zY*zY) /
  ((zX*zX+zY*zY)*Math.pow(zX*zX+zY*zY+1.0, 1.5));

ps.format("Estimated z:                          %12.5f\n", z);
ps.format("Prediction interval (95%% confidence): %12.5f to %6.5f (%f)\n",
   predictionInterval[0], predictionInterval[1],
   predictionInterval[1]-predictionInterval[0]);
ps.format("Zx:                                   %12.5f\n", beta[1]);
ps.format("Zy:                                   %12.5f\n", beta[2]);
ps.format("Azimuth steepest ascent               %12.5f\n", azimuth);
ps.format("Compass bearing steepest ascent          %05.1f\u00b0\n", compass);
ps.format("Grade                                 %8.1f%%\n", grade*100);
ps.format("Slope:                                %8.1f\u00b0\n", slope);
ps.format("Profile curvature:                    %12.5f\n", kP);
```

The following text gives an example output from the code above at the coordinates (627520, 4655800) from the Bear Mountain sample (ground points only):

```
Estimated z:                          612.54953
Prediction interval (95% confidence):  612.37857 to 612.72049 (0.341922)
Zx:                                     -0.01685
Zy:                                      0.03897
Azimuth steepest ascent                  1.97891
Compass bearing steepest ascent        336.6°
Grade                                      4.2%
Slope:                                     2.4°
Profile curvature:                       0.00833
```

2.3.3.5 The State of the GWR Implementation

The GWR implementation is the area of the Tinfour package that could most benefit from expert attention. Clearly the automatic bandwidth and model-selection option is too slow for bulk production of interpolated values, as when producing an elevation or hillshade grid. During development, the results from the Tinfour application were compared to those of GWR4 to verify *correctness of implementation*. When Tinfour is used with application-specified bandwidths, the results from the interpolation and evaluation statistics (variance, confidence intervals, AICc scores) match.

However, when using automatic bandwidth selection, the version of GWR4 that was used for testing (4.0.80) failed to execute successfully and did not report any useful diagnostic information. So the results of the automatic bandwidth testing method have not been compared to other applications. Instead, testing was conducted through a self-test process as follows:

1. For selected sample point sets, a regression was performed using the "fixed bandwidth" option for a large number of bandwidths within the range supported by Tinfour (from 0.3 to 1.0 times the mean distance of the samples from the interpolation point). The AICc score was recorded for each. The test with the best AICc score was considered the "optimal" bandwidth.
2. The optimal bandwidth obtained from the above test was compared to the bandwidth estimated using the automatic selection method in the Tinfour implementation.
3. The test was considered successful if the bandwidth selections were within 5 percent of each other (computed as the difference of the two bandwidths divided by the range of evaluation).

Since the Tinfour AICc calculation was verified using a separate software package (GWR4), the self-test procedure has some merit. However, because the computation of AICc for GWR is so time-consuming, only a few hundred samples were considered. Therefore these results should be viewed with caution.

Beyond the details of implementation, the basic assumption of GWR deserves attention. GWR addresses the issue of heteroscedasticity in observed data by treating the significance of samples distant from a point-of-interest as less than that of those nearby. While this assumption is reasonable as far as it goes, it does not consider the degree to which samples correlate with each other. Consider the case where two relatively distant elevation samples are spaced close to each other and so capture a feature (such as a large boulder) with an increased elevation compared to the surrounding terrain. While the distance-weighting logic will reduce the contribution of each sample, there are still two of them in the data set. Simple GWR does not account for the fact that, taken together, those two samples over-represent a heteroscedastic feature. As this example shows, addressing collinearity and self-correlation among samples is an area that presents potential improvement to the Tinfour implementation.

2.3.3.6 Background on the GWR Technique

General discussions of the GWR technique are readily available on the Internet (see Wheeler 2010, etc.).

The calculations used to derive regression coefficients are adapted from Walpole and Myers (1989, p. 401-442) Chapter 10, "Multiple Linear Regression". Walpole and Myers provide an excellent introduction to the problem of multiple linear regression, its evaluation statistics (particularly the prediction interval), and their use. The calculations for a *weighted* regression are not covered in their work, but were derived from the information they provided. Because these calculations are not taken from published literature, they have not been vetted by expert statisticians.

Details of the evaluation statistics specific to a weighted regression are taken from Leung, et al. (2000, p. 9-32). The authors were particularly interested in presenting mathematically sound formulations for unbiased statistics and reliable ways of detecting “nonstationarity” (heteroskedasticity) in a sample set.

Information related to the AICc criteria as applied to a GWR was found in Charlton, Martin and Fotheringham, A (2009), a white paper downloaded from the web. A number of other papers by Brunsdon, Fotheringham and Charlton which provide slightly different perspectives on the same material can be found on the web.

3 Tests and Demonstrations

The Tinfour software distribution includes a Java package (folder) named "test" which contains high-level test programs and example applications that demonstrate how to use the package. Most of these applications implement the ability to build Delaunay triangulations from an input "vertex file". The vertex files may be given either as a text file (a tab, space, or comma separated file giving x, y, and z coordinates) or from an industry standard lidar LAS file. Some of the applications measure software performance or test for correctness of implementation. Others demonstrate the use of Tinfour for performing analysis of a set of unstructured sample points.

The test applications described below are mostly small, non-interactive implementations that exercise or demonstrate a single key feature of the Tinfour package. But the final application in this section is an entirely different matter. The Tinfour Viewer is a user-interface based application that provides a graphical display and interactive functions that permit a user to explore surfaces built from unstructured data. In a sense, it is the culmination of the Tinfour development project (at least so far). As such it is the last item discussed in this section.

3.1 The Test Environment

Both the tests and example applications are intended for use by software developers. All the current test applications can be run from either an Integrated Development Environment (IDE) such as Netbeans and Eclipse, or from a command-line environment. Specifications for running the test are accepted from arguments passed in to the Java main.

3.1.1 Command-Line Arguments

The general input options for the Tinfour test and demonstration applications are described below. In cases where an application has unique command-line options, these are given in the usage text or documentation for the specific application. Options are not case sensitive. Boolean settings do not take an argument, but are given in the form "-option" or "-noOption" as appropriate.

Table 6 – Input and output options

Option	Arguments	Description
-in	File path	File to be used as input source for samples. May be a .LAS (lidar), .CSV (comma-separated-value), or character-delimited text file (by default, space or tab). When whitespace is used as a delimiter, Tinfour treats multiple spaces as a single element.
-delimiter	Character	The delimiter character to be used for a text-based input file if something other than whitespace is used.
-out	File path	Output path specification for those applications that produce file output. If the application produces multiple outputs (such as those that produce both raster elevation grid files and image files), the file extension will be replaced as appropriate.

Table 7 – Processing options

Option	Arguments	Description
-cellSpace	Floating point distance	For applications that produce an output grid, the spacing between grid cells. Spacing should be a distance specified in the same system of units as the source data. Applications that produce grids have the ability to select cell spacing automatically, though an explicit setting is generally preferred by most users.
-frame	xMin xMax yMin Ymax	For applications that can focus on a sub-region of the overall sample set, the area-of-interest for processing. By default, the coordinates of the "frame" are the overall bounds of the input data set. So, unless a frame specified, the entire data set will be considered. The frame option usually does not affect which samples are read in from a source file. Instead, it provides options related to the way in which the samples are processes.
-interpolator	String	For applications that perform interpolation, a string indicating the interpolation method to be used: • NaturalNeighbor (default) • TriangularFacet • Regression
-lidarClass	Integer : 0 to 255 for a specific classification or -1 to process all.	For files that process lidar data, the classification of the lidar points to be included for processing. Hillshade and elevation-related applications will default to a value of 2 (for ground points).
-lidarThinning	Floating point value greater than zero and less than or equal to one.	For input files, indicates that a subset of points are to be randomly selected. Typically used when an input file is too large for the available memory on a system. The name of this option is a misnomer in that it applies to all input formats.
-nTests	Integer: 1 to maximum integer value.	For applications that perform repetitive tests, the number of iterations to be performed. Values greater than 3 are recommended.
-palette	Named palette	Palette for applications that provide color rendering of TIN. Palettes are built in to the test applications and use the following names: • BlueToYellow • PurpleTones • RedToYellowToWHite • Rainbow • BlackToGray
-preAllocate,	None (Boolean)	Indicates if storage for edges is to be pre-allocated before

-noPreAllocate		building a TIN. Used to isolate the runtime cost of object allocation version algorithm-related processing.
-preSort, -noPreSort	Non (Boolean)	Indicates whether the data should be processed using the Hilbert sort method before building a TIN.
-seed	Integer	For all applications that use randomization methods to select or generate sets of input sample point, a random seed for the process. The seed follows the general contract of the java.util.Random class.
-tinClass	String giving fully qualified Java class name.	The class to be used for building the TIN. At present, two specifications are supported: tinfour.standard.IncrementalTin tinfour.semivirtual.SemiVirtualIncrementalTin Specifications follow the Java conventions and are case-sensitive.
-classA -classB	String giving fully qualified Java class name.	Specifies the class to be test for the TwinBuildTest

Some test utilities may create a random set of unstructured input sample points. A few use grids in which the horizontal coordinates are perturbed by small random adjustments to test for weaknesses in the TIN-building implementation. In addition to the –seed option described above, such applications may use the options listed below:

Table 8 – Random sample generation options

Option	Argument	Description
-nColumns	Integer:2 to the maximum integer size	Number of columns in a grid
-nRows	Integer: 2 to the maximum integer size	Number of rows in a grid
-nVertices	Integer: 3 to maximum integer size	Number of vertices to be included in TIN.

3.2 Examples and Demonstrations

3.2.1 Example Elevation and Hillshade Grid from Vertex Files

The Java application ExampleGridAndHillshade can accept vertices from either a text file or an LAS lidar file and produce both an elevation grid and a hillshade image such as the one shown in Figure 22. The figure shows a hillshade image derived from the lidar samples covering a slightly larger region in the

vicinity of Bear Mountain area mentioned above. The area is almost entirely wooded and free of paved roads. The linear feature that cuts across the image is an unimproved fire road. This road is used as part of a loop trail to the Bear Mountain summit and as an access path to the Appalachian Trail which is slightly to the southeast of the image. A footpath which leads north to the top of the mountain is faintly visible in the lower-right corner of the image. A large cairn at the top of the mountain is clearly visible as a "bump" in the upper right corner of the figure.

This hillshade images in the figure below and in Figure 4 above were generated using an example application called ExampleGridFromFile. The demonstration program performs the following operations:

1. Accept a command-line argument specifying an input lidar LAS file or a text file.
2. Accept a command-line argument specifying an output file path for an Esri ASCII raster file with the extension .asc.
3. Read a set of vertices (samples) from the LAS file.
4. Determine the triangulation class based on estimated memory use and select either the standard or semi-virtual variations for construction of a Delaunay triangulation.
5. Interpolate a grid of elevation values and store the information in the Esri ASCII raster format as follows:
    a. A grid is computed based on the extent of the input data and a cell size specified as a command-line argument (if omitted, the overall estimated points spacing for the sample is used). In selecting the cell-size, pick a value that is consistent with the data specified.
    b. Elevations are interpolated using one of three interpolation methods that can be specified on the command line:
        i. Linear (simple triangular facets)
        ii. Natural Neighbor Interpolation (the default)
        iii. Linear Regression
6. Generate a hillshade image and write it to a PNG file using the following:
    a. Create a Java BufferedImage instance sized according to the number of rows and columns in the output grid computed above.
    b. If the command-line arguments include a palette specification, use it to color code elevations. If not, a flat white backdrop is used. Figure 22 was generated without the use of a palette. Figure 4 used the "Rainbow" palette.
    c. Use Tinfour's Linear Regression interpolator to create a cubic equation describing the surface in the vicinity of each grid point.
    d. Taking the partial derivatives of the cubic equation, compute the slope in the direction of the X and Y axes. From these slopes, obtain the vector normal to the surface at each grid point.
    e. Using a simple diffuse-lightning model and assuming a point-illumination source, compute relative shading.
    f. Adjust the brightness of the output pixels according to the illumination values and output the image in the form of a PNG file.

Figure 22 – Hillshade image derived from the Bear Mountain lidar sample

3.2.2 Point Thinning using the Hilbert Sort

Figure 2 and Figure 3 both showed triangular meshes created using a subsample of the Bear Mountain data set. The most direct method to obtain a subset from a list of samples is to loop through the list selecting samples at fixed intervals. To reduce a set of 1000 samples to 100, simply select every tenth sample, etc.

One shortcoming of the simple approach is that it doesn't guarantee complete coverage of the region defined by the sample points. This situation is exacerbated when the samples are given in random or semi-random order with non-uniform density.

Tinfour includes an experimental application called ExampleWireframeWithThinning that was used to produce the thinned meshes in Figure 2 and Figure 3. The application attempts to improve the coverage of a subsample by performing a Hilbert sort on the data before selecting samples. In testing, this approach has always yielded good results. However, the idea that it represents a significant improvement over simple selection remains conjectural.

3.2.3 Multiple Concurrent Processes for Surface Interpolation

Tinfour constructs a TIN using a sequential process that does not benefit from concurrent processing. But, because the interpolation and grid analysis routines operate over the TIN on a read-only basis, they are suitable for parallel processing using Java's multi-threading capability. The application ExampleMultiThreadTest demonstrates how the time required for the relatively expensive process of surface interpolation can be reduced by employing multiple concurrent threads to process data.

3.3 Test Applications

3.3.1 The Single Build Test for Correctness of Implementation

The test application SingleBuildTest builds a TIN using the input from a source file and performs post-construction testing for correctness of implementation. It also prints out diagnostic information about memory use and statistics about the TIN construction process.

At the end of the construction, the SingleBuildTest performs an "integrity test" to verify that the resulting TIN conforms to the Delaunay criterion and that all data elements are properly populated. In effect, this is a correctness-of-implementation test.

A sample output is given below.

```
TIN class:    tinfour.semivirtual.SemiVirtualIncrementalTin
Input File:   C:\CT_NW_Lidar\A1_Bear\bear_all.las
Number of vertices in file (all classes): 4874727
Number of vertices to process:  1036887
Range x values:       627000.000,   628000.010, (1000.010000)
Range y values:      4654999.980,  4655999.990, (1000.010000)
Range z values:          538.434,      709.032, (170.597961)
Est. sample spacing:       0.862
Time for pre-sort:            460.14
```

```
Pre-alloc is not used
Begin insertion
Time build TIN:              2961.20
Total time for TIN:          2961.20
Checking memory

Memory use (bytes/vertex)
   All objects:              120.48
   Vertices only:             44.08
   Edges and other elements:  76.40

Total for application (mb): 316.00


Descriptive data
Number Vertices Inserted:      1036887
Coincident Vertex Spacing:    0.000010
   Sets:                             8
   Total Count:                     16
Number Edges On Perimeter:          37
Number Ordinary Edges:         3110597
Number Ghost Edges:                 37
Number Edge Replacements:      5096883    (avg: 4.9)
Max Edge Replaced by add op:        70
Average Point Spacing:               1.12
Application's Nominal Spacing:       1.00
Number Triangles:              2073719
Average area of triangles:           0.482
Samp. std dev for area:              0.310
Minimum area:                        0.000300
Maximum area:                       14.024
Total area:                     999993.4

Construction statistics
Number of SLW walks:           1036887
   exterior phase:                6947
   tests:                      6364282
   extended:                      1124
   avg steps to completion:          3.12
InCircle calculations:        13273748
   extended:                         14
   conflicts:                         0


Edge pool diagnostics
   Edges allocated:              3110634
   Edges free:                       278
   Pages:                           3038
   Partially used pages:               1
   Total allocation operations:  3173435
   Total free operations           62801

Performing integrity check
Test complete
```

3.3.2 The Repeated Build Test for Performance Evaluation

The repeated build provides a way of assessing the performance of the Tinfour implementation. To do so, it runs the sample data repeatedly, measuring the time required for builds. In the example output

text below, note that the processing time for the first two tests is slightly longer than those of subsequent tests. This effect is due to the cost of the Java class loader and JIT compiler in the initial stages of execution. Experience shows that some systems take as many as three iterations for the run times to settle down to a steady state. Therefore, the first three tests runs are not included in the overall averages.

```
Date of test:       24 Feb 2016 02:09 UTC
Input file:         C:\CT_NW_Lidar\A1_Bear\bear_partial.las
TIN class:          tinfour.tin.IncrementalTin
Time for pre-sort   0.0
Number of vertices  287889
run,        build,    avg_build,     total_mem,   alloc_time
  0,      600.949,        0.000,       68.819,     1314.329
  1,      475.069,        0.000,       68.825,      205.250
  2,      415.067,        0.000,       68.825,      130.842
  3,      428.102,      428.102,       68.825,      122.653
  4,      424.691,      426.397,       68.825,      129.509
  5,      406.460,      419.751,       68.825,       77.295
  6,      403.428,      415.670,       68.825,       61.259
  7,      413.498,      415.236,       68.825,      119.233
Avg max removed:        412.019
```

3.3.3 The Twin Build Test for Tuning Performance and Optimization

Optimization and performance tuning can be tricky subjects. Sometimes, the most promising concepts turned out to be major disappointments. Ideas that seemed like “no-brainers” revealed themselves to be gross misconceptions. Logic that succeeded in unit tests actually slowed down processing when integrated into the code base.

The software development effort for Tinfour included extensive experiments with different ideas and approaches for improving processing speed using an application called TwinBuildTest. The TwinBuildTest is designed to overcome the fact that a modern operating system is a noisy test environment. It is difficult to control or even predict when background processes, security software, or other system processes will compete with a test program for resources. To provide an objective measurement of how changes to the code affected performance, the development process depended on a “twin build test”.

1. Implement two versions of the same mesh-building class (or classes) introducing the software approach to be tested in one of them.
2. Repeatedly run both versions in the same Java process, alternating approaches. Record the run time for each.
3. Inspect the run times for and observe trends. Compare the relative time required for processing by each. If the statistics includes obviously anomalies, throw away the results and run the test again.

The following text shows the output from a test that compared the time to build a TIN from a lidar sample containing about 290 thousand points running on a computer with limited resources and CPU power. Internally, Tinfour uses a buffer that permits it to reuse edges when building a TIN. The test evaluated whether the buffer was actually contributing to performance. The TwinBuildTest was used to

compare a standard implementation (Class A) to a version in which the buffer was disabled (Class B). Run times are given in milliseconds. As in the repeated build test, the first three run times are not included in the tabulated results.

```
TwinBuildTest
Date:    27 Dec 2015 16:57
Class A: tinfour.standard.IncrementalTin
Class B: tinfour.standard.IncrementalTin1
Preallocation enabled: true
Sample Data: C:\CT_NW_Lidar\A1_Bear\bear_partial.las

Number of vertices  287889

27 Dec 2015 16:57 UTC
run,          build1,   avg_build1,          build2,    avg_build2
  0,        3160.465,        0.000,         2308.458,        0.000
  1,        1607.773,        0.000,         1599.363,        0.000
  2,         747.325,        0.000,          805.106,        0.000
  3,         749.661,      749.661,          804.241,      804.241
  4,         764.466,      757.064,          812.882,      808.562
  5,         744.030,      752.719,          811.557,      809.560
  6,         749.223,      751.845,          815.042,      810.931
  7,         776.388,      756.754,          805.807,      809.906
avg with max removed       751.845,                        808.622

comparative time method a/b: 0.9297858109283054
comparative time method b/a: 1.0755165203065342
```

The average run time for the standard implementation is about 750 milliseconds while that for the modified implementation is about 800 milliseconds. So the version with the buffer runs about 7 percent faster than the version without. The results give evidence that the buffer is an effective performance enhancement and should remain in the implementation.

Looking down the columns of values, note that the run times for the two methods were fairly consistent, indicating a fairly stable system environment during the test. Sometimes, the entries in a column will bounce around in an apparently random manner, indicating that something may have been happening on the system that competed with the test application for resources. In testing, it is common for the data may include just one run that is substantially longer than all others. This happens often enough that the twin-build test application automatically disregards the worst run of the main series when tabulating statistics. However, in cases where the run times are obviously inconsistent, it is best to simply discard the results and run the test again.

The Twin Build Test application accepts two special command line arguments, -classA and -classB which given the fully qualified names for the classes to be tested. By default it uses the standard IncrementalTin and semi-virtual SemiVirtualIncrementalTin.

3.3.4 Time to Process TIN Due to Sample Size

The TimeDueToSampleSize test preforms repeated build operations in which it randomly selects a subset of input data from a sample set and measures the time to build it. Because the size of each random selection is varied, this test application provides a realistic method for measuring the actual cost of processing a TIN as a function of the number of input vertices.

3.4 The Tinfour Viewer

When first considering a software package like Tinfour, it is reasonable to ask questions like "what is it good for?" and "what can I do with it?" These notes were intended, at least in part, to address those questions. And so it seems natural that they conclude with a discussion application that integrates the major features of Tinfour into a single application which demonstrates its capabilities. The Tinfour Viewer is uses the Delaunay Triangulation as a tool for viewing and analyzing surfaces built from unstructured data. The figure below shows a screenshot from Tinfour Viewer taken while inspecting a lidar sample collected near a highway overpass where I-95 crosses Church Street in Guilford, CT.

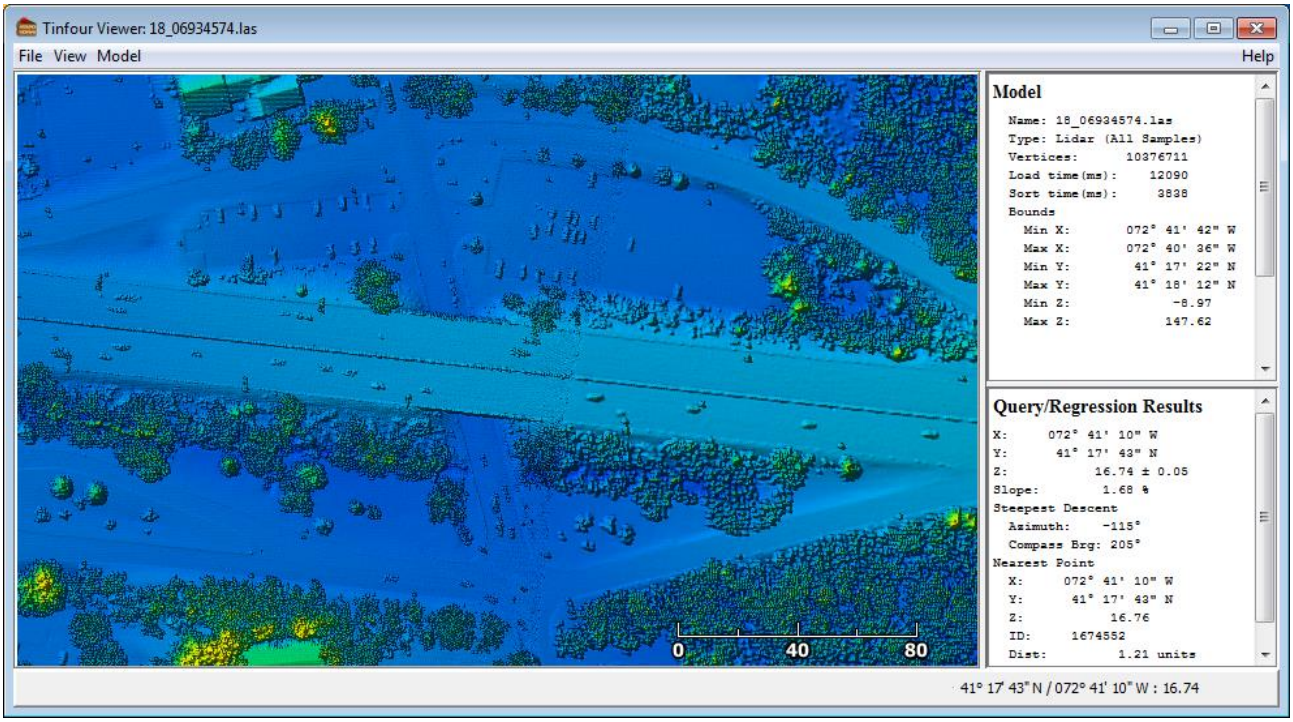

Figure 23 – Tinfour Viewer image constructed from lidar data over Church Street and I-95 in Guilford, CT (data source NOAA, 2011b).

The Viewer application combines many of the major functions from Tinfour that were discussed in the example applications above including:

1. Elevation modeling and surface interpolation use for raster rendering.
2. Hillshade functions.
3. Point-thinning to produce reduced-complexity meshes for inspection in the “wireframe” option.
4. Concurrent processing to speed production of raster images.
5. Interactive data queries for inspecting samples.

The wireframe option is used to depict the structure of the TIN. By using the point-thinning techniques described above, it manages the density of the points according to the scale of view. The figure below was produced from data collected in a coastal region near Byram Connecticut and processed to thin the number of points by a factor 256 to 1. One interesting feature of the display is that the density of points is not uniform across the collection area. Conventional lidar systems use infrared laser sources that are generally scattered rather than reflected by water surfaces. So it is common for the sample density to drop off sharply over water. In the figure below, only a few data samples were obtained over the water area. Thus the distribution of samples in the picture reflects the different surface characteristics in the sample.

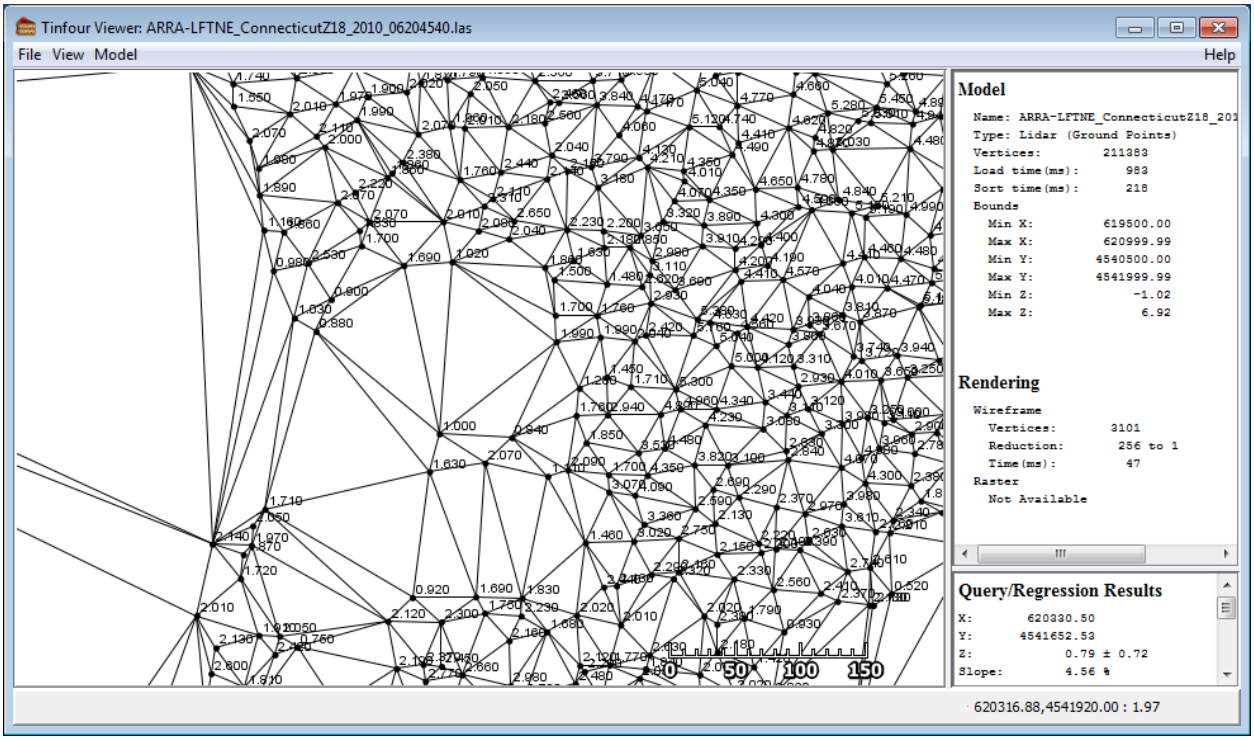

Figure 24 – Wireframe rendering of lidar sample in coastal region (data source NOAA, 2011c)

4 Lidar Data Samples

Many examples provided in these notes use Lidar products as a data source. Laser-measured elevation data sets offer a good test case for Tinfour implementations because they are large and because the real-world data they provide describes features with which all of us are familiar. If you wish to download and use Lidar products for your own use, there are a few issues with which you should be familiar.

4.1 LAS and LAZ Format

LAS is an industry standard format for the exchange of Lidar data. It is a non-proprietary standard that was developed by the American Society for Photogrammetry & Remove Sensing (ASPRS) to promote the sharing of Lidar information. Because LAS is a straightforward, binary format that follows modern computer numeric data representation standards, it promotes efficient processing and reasonably compact data storage. LAS files also support random-access read operations, a feature which permits applications to better manage the amount of data that is read into memory when processing Lidar data. Finally, the LAS format has a clean, simple design that made it easy to write code for the Tinfour library that would read LAS files directly.

The LAS format is described at http://www.asprs.org/wp-content/uploads/2010/12/LAS_1_4_r13.pdf

Of course, even though LAS using an efficient binary representation of data, the sheer volume of Lidar data samples lead to very large file sizes. Data size is particularly an issue when downloading data via the Internet because LAS files do not compress well using conventional tools such as ZIP.

Fortunately, an independent software developer named Martin Isenburg developed a variation of LAS named LAZ which embeds data compression into the format. Because the compression is customized specifically for the representation of Lidar data, it achieves excellent size reduction ratios. Also, the implementation makes smart choices that preserve the random-access capabilities of the original. Best of all, Mr. Isenburg released his compression utility as an open-source project which has led to the widespread adoption of LAZ as an industry standard.

The data compression algorithms used by the LAZ format are fairly sophisticated and were only recently implemented in Java by Reutegger (2016) in the laszip2j project. The laszip2j library is currently used by the Tinfour test and data viewer applications.

4.2 Ground Points and Lidar Data Classification

When Lidar data is collected from an aircraft, the elevation-measuring sensors don't know if the features that are reflecting their laser sources are flat ground, trees, bushes, or man-made structure. In fact, it is not uncommon for Lidar to detect birds or swarms of insects. So when Lidar data is prepared for distribution, it usually goes through a post-collection "classification" process in which each measurement is categorized according to what kind of object produced the sample. As you can imagine, this process is computationally intensive and often involves the use of supplemental data sources such as multi-spectral imagery (aerial photographs) and ground-based survey information. By convention all samples from a survey are included in a LAS data set. Nothing is thrown away. Even those that are

rejected as anomalous are simply marked as "unclassified" or "withheld" (one man's anomalous data is another's "research find").

The LAS standard defines several categories for data and assigns each a numeric code. The most important of these is the "ground point" classification, which is assigned the numeric index 2. Other classifications – including surface water, roads, vegetation, etc. – are described in the LAS specification (ASPRS, 2013). The example applications provided by Tinfour primarily focus on ground-point data, though there is nothing in the software package that would limit the use of other data classifications.

4.3 Geographic Coordinates

Native Lidar data is rarely collected using geographic coordinates. As discussed in paragraph *2.2.9 Coordinates and Numerical Issues* geographic coordinates are not isotropic and so provide a poor representation of 3D features over the small area of the Earth's surface that comprises a Lidar survey. So in order to facilitate geospatial analysis on 3D data samples, Lidar collections will usually base the horizontal coordinate system on coordinates projected to a plane using well established mapping techniques. For example, the above cited Lidar samples taken in Connecticut were originally collected using horizontal coordinates based on the Universal Transverse Mercator Zone 18N map projection.

Distribution of Lidar data is another matter. One of the largest collections of Lidar data available on the web is the NOAA Coastal Lidar repository. For various reasons, NOAA has elected to convert all the data in its collection to geographic coordinates. Converting the original Lidar data to geographic coordinates presents two problems:

1. The data is no longer in its original format because the horizontal coordinates (and perhaps the vertical coordinates) have undergone a mathematical transformation.
2. Because the data is non-isotropic, the horizontal coordinates needs to be projected to a flat coordinate plane for processing.

In NOAA's defense, they've elected to convert Lidar data from hundreds of different surveys based on dozens of incompatible horizontal coordinate systems to a single unified geographic coordinate system, one which non-GIS practitioners are familiar with. However, it means that before the data can be analyzed by Tinfour, it needs to be converted.

The LAS access classes provided by Tinfour do this conversion automatically, though the implementation is still a bit rough and lacks important features (it does not use metadata from the LAS files, but deduces that coordinates are geographic and assumes that the vertical units are in meters). If you require more control over the representation of data, the lastools library includes tool that permit you to convert the horizontal coordinate system in LAS files from geographic to projected values.

5 References

Bowyer, Adrian (1981). “Computing Dirichlet tessellations”, *The Compute Journal 24(2),* p. 162-166.

Cheng, Siu-Wing; Dey, Tamal Krishna; Shewchuk, Jonathan R. (2013). *Delaunay Mesh Generation*. Boca Raton, FL: CRC Press.

Delaunay, Boris (1934). "Sur la sphère vide", *Otdelenie Matematicheskikh i Estestvennykh Nauk* **7**, p. 793–800

Guibas, L., Stolfi, J (1985). “Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams”, *ACM Transactions on Graphics, Vol 4, No. 2*, April 1985, p 74-123.

Hilbert, D. (1891). “Ueber die stetige Abbildung einer Linie auf ein Flächenstück”, Mathematische Annalen 38 (1891), 459-460. Accessed from http://www.digizeitschriften.de/dms/img/?PID=GDZPPN002253135

Lawson, C.L. (1977). “Software for C^1 surface interpolation”, *Mathematical Software III*, Rice, J.R. ed., pages 161-194, Academic Press, NY.

Leung, Yee , Chang-Lin Mei, Wen-Xiu Zhang (2000).”Statistical Tests for Nonstationarity Based on the Geographically Weighted Regression Model”, *Environment and Planning A*, vol.32, 2000, p.9-32.

Natural Earth (2017) “Free vector and raster map data at 1:10m, 1:50m, and 1:110m scales”. Accessed October 2017 from http://www.naturalearthdata.com/.

Pennsylvania, Commonwealth of, Department of Conservation and Natural Resources, Bureau of Topographic and Geologic Survey (2006). *PAMAP Program LAS Files (LiDAR Data of Pennsylvania).* Website. Accessed December 2015 from http://www.pasda.psu.edu

Peckham, S. (2011). “Profile, Plan, and Streamline Curvature: A Simple Derivation and Applications”. Electronic document downloaded March 2016 from http://geomorphometry.org/Peckham2011a

Reutegger, Marcel (2016). “laszip4j – The LASzip library ported to Java”. Accessed February 2017 from https://github.com/mreutegg/laszip4j

Rognant, et al (1999). The Delaunay constrained triangulation: The Delaunay stable algorithms. "IEEE International Conference on Information Visualization 1999", pg. 147-152.

Shewchuk, Jonathan R. (1996). “Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator”, *Applied Computational Geometry: Towards Geometric Engineering*, ed. Lin, Min C, & Manocha, Dinesh, *Lecture Notes in Computer Science*, vol. 1148, pages 203-222, Springer-Verlag, Berlin, May 1996. Downloaded December 2015 from http://www.cs.cmu.edu/~quake/tripaper/triangle0.html

Sibson, Robin (1981). “A Brief Description of Natural Neighbor Interpolation”. *Interpreting Multivariate Data*, pages 21-36. Ed. Barnett, V., John Wiley & Sons, Inc. Chichester (1981).

Soukal, R., Málková, M., Kolingerová, I. (2012). "Walking algorithms for point location in TIN models", *Computational Geoscience* vol. 16, pages 853-869.

Su, P., Drysdale, R. (1996). "A comparison of sequential Delaunay triangulation algorithms", *Computational Geometry 7* (1997) p. 361-385

U.S. Geological Survey [USGS] (2014). "USGS Lidar Point Cloud (LPC) OR_PoleCreek_2013_000100 2014-09-26 LAS". Downloaded December 2015 from https://www.sciencebase.gov/catalog/item/542fe8d4e4b092f17df623c7

U.S. National Oceanic and Atmospheric Administration [NOAA] Coastal Services Center (2011a). "2011 U.S. Department of Agriculture- Natural Resources Conservation Service (USDA-NRCS) Topographic Lidar: North West Connecticut" , file 20111217_18TXM2755.laz. FTP site. Downloaded December 2015 from ftp://coast.noaa.gov/pub/DigitalCoast/lidar1_z/geoid12a/data/2597/

U.S. National Oceanic and Atmospheric Administration [NOAA] Coastal Services Center (2011b). "2011 FEMA Lidar: Quinnipiac River Watershed (CT) Point Cloud files with Orthometric Vertical Datum NAVD88 using GEOID12A", file 18_06934574.laz. FTP site. Downloaded December 2015 from https://coast.noaa.gov/htdata/lidar1_z/geoid12a/data/1472/

U.S. National Oceanic and Atmospheric Administration [NOAA] Coastal Services Center (2015). "Lidar Datasets at NOAA Digital Coast" Website. Accessed December 2015 from https://coast.noaa.gov/htdata/lidar1_z/

Walpole, R., and Myers, R. (1989). *Probability and Statistics for Engineers and Scientists (4th ed.).* Macmillan Publishing Company, New York City (1989).

Warren, Henry S., Jr. (2013) *Hackers Delight, 2nd Edition*. Pearson Education, Inc. Upper Saddle River, New Jersey (2013).

Watson, David F. (1981). "Computing the n-dimensional Delaunay t tessellation with application to Voronoi polytropes", *The Compute Journal 24(2),* p. 167-173.

Wheeler, David C., Páez, Antonio (2010). "Geographically Weighted Regression", *Handbook of Applied Spatial Analysis: Software Tools, Methods and Applications,* pages 461-486, Springer-Verlag, Berlin 2010.

Wilson, John P., Gallant, John C. (2000). *Terrain Analysis: Principles and Applications.* John Wiley & Sons, Inc. New York (2000).