2011

Irregular tessellated surface model map algebras to define flow directions and delineate catchments using LiDAR bare earth sample points

Gerald B. Gabrisch
Western Washington University

Irregular Tessellated Surface Model Map Algebras to Define Flow Directions and Delineate Catchments Using LiDAR Bare Earth Sample Points

By

Gerald B. Gabrisch

Accepted in Partial Completion

of the Requirements for the Degree

Master of Science

Moheb A. Ghali, Dean of the Graduate School

ADVISORY COMMITTEE

Chair, Dr. Scott Miles

Dr. Michael Medler

Dr. Robert Mitchell

MASTER'S THESIS

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Western Washington University, I grant Western Washington University the non-exclusive royalty-free right to archive, reproduce, distribute, and display the thesis in any and all forms, including electronic format, via any digital library mechanisms maintained by Western Washington University.

I represent and warrant this is my original work, and does not infringe or violate any right of others. I acknowledge that I retain ownership rights to the copyright of this work, including but not limited to the right to use all or part of this work in future works, such as articles or books.

Library users are granted permission for individual, research and non-commercial reproduction of this work for educational purposes only. Any further digital posting of this document requires specific permission from the author.

Any copying or publication of this thesis for commercial purposes, or for financial gain, is not allowed without my written permission.

Signature _____

Date _____

Irregular Tessellated Surface Model Map Algebras to Define Flow Directions and Delineate Catchments Using LiDAR Bare Earth Sample Points

A Thesis

Presented to

The Faculty of

Western Washington University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

By

Gerald B. Gabrisch

November 2011

Abstract

Flow directions and catchment algorithms have historically utilized raster-based data models. A significant body of literature focuses on raster-based interpolation errors, and the subsequent surface reconditioning to compensate for those errors, that together degrade the accuracy of the derived flow directions and catchments. This research seeks to improve upon the raster-based approach by developing and evaluating a vector-based approach to generating flow directions and delineating catchments that preserves the accuracy of the input point data through the use of irregular tessellated surface models. Specifically, the Python computer programming language was used in conjunction with a geographic information system (GIS) to develop ITSMHydro, a custom toolset that creates a Delaunay triangulated irregular network (TIN) from LiDAR bare-earth sample point data, and subsequently generates flow directions, delineates basins, and processes spurious sink catchments. Surface model accuracy, and area, shape, and overlap of the resulting catchments were compared with catchments delineated using industry-standard raster-based digital terrain models. The vector-based approach implemented through ITSMHydro was limited to file sizes less than approximately 120,000 LiDAR strikes that processed in approximately 30 hours, whereas the industry-standard raster-based approach transformed 111,000,000 LiDAR strikes across the study area into a 3-feet pixel surface model and generated catchment boundaries in approximately 36 hours. A root mean square analysis of surface models indicates that surface model quality is more heavily degraded when LiDAR sample points are interpolated to raster grids as opposed to surface

models relying on Delaunay TIN interpolation, suggesting that the vector-based approach maintains the quality and precision of the LiDAR input data. For the four test areas in which the two approaches were compared, ITSMHydro generated catchments that were generally smaller (percent difference in areas ranged from -83.97% to 9.39%) and with more complex boundaries (i.e. lower isoperimetric quotient in 3 out of 4 test areas) than the associated raster-based catchments. Coefficient of areal correspondence (CAC), a measure of overlap between catchments generated by the two methods where a value of 1 indicates perfect overlap, ranged from 0.28 to 0.80 in the four test areas. Given the lower relative accuracy of raster-based surface models evident in the study area, these differences suggest use of the raster-based approach may compromise accuracy in area, shape, and location of the resulting catchments. A vector-based approach that preserves the accuracy of the input data is preferred, especially in areas of low topographic relief. The file size constraints limit application of the approach developed herein, however, at least until technological advances and/or code revisions improve computer processing speed and file size capacity.

Acknowledgements

I would like to extend my deepest thanks to: Scott Miles, Michael Medler and Robert Mitchell for serving on my committee and helping me to develop and expand my body of knowledge; the citizens of the Lummi Nations for graciously sharing their GIS data; Jeremy Freimund for providing direction, encouragement, criticism, and allowing me to incorporate my research into my workload; and my friends and family for supporting my efforts and long hours away from home. Additionally, I would like to thank Treva Coe for her firm and extensive editing skill, countless hours of childcare, and her warmth, love, and support of this project over the years.

Contents

List of Figures

List of Tables

List of Equations

Section 1: Introduction

The legal, cultural, and economic implications of surface water and ground water quality and availability necessitate high-quality boundary delineations and flow direction models for watersheds. A number of commercially available Geographic Information Systems (GIS) provide toolsets that allow the delineation of basin boundaries using raster-based surface models (Garbrecht and Martz 2000; Maidment 2002). These watershed delineation tools, in combination with the nationwide coverage of United States Geologic Survey (USGS) raster-based digital terrain models (DTM), provide a convenient and popular means for delineating watershed boundaries.

A significant body of literature raises questions about the quality of raster-based surface models for delineating catchment boundaries. Criticism stems from the over or underestimation of pixel values resulting from the interpolation algorithms used to generate raster-based surface models, the flow direction constraints of the raster-based surface models, and limitations of raster cell size (Mark 1984; O'Callaghan and Mark 1984; Mark 1988; Fairfield and Leymarie 1991). Advances in computer processing times, data storage capacities, and surface elevation data collection are rapidly improving the science and practice of watershed delineation. DTMs generated from higher quality and higher precision airborne-remote-sensing Light Distance and Ranging (LiDAR) datasets can surpass the quality of the photogrammetric techniques used to generate 10-meter pixel USGS digital terrain models (Campbell 2002). LiDAR can produce a high density of randomly scattered sample points that can be interpolated into regular tessellated surface models, raster surface models, or used to build irregular tessellated surface models such as Triangulated Irregular Networks (TIN), and Voronoi Diagrams.

The toolsets necessary to delineate catchment boundaries using vector-based-irregular-tessellated surface models are not readily available in a GIS. This research details the generation and application of algorithms that produce catchment delineations using a combination of TIN surface models and Voronoi diagrams generated from LiDAR bare-earth sample points, and compares and contrasts the catchment delineation results against raster-based surface models generated from the same LiDAR sample points. The irregular tessellated catchment delineations are evaluated for differences in shape, area, and processing speed against industry-standard raster-based catchment delineation algorithms. This research addresses the question of whether tessellated surface models can produce a higher quality, higher precision, catchment delineation than the basin delineation generated from a raster-based surface model.

This research is important to the field of geographic information sciences because the algorithms presented in this research do not require additional levels of data transformation or abstraction that can degrade the data and compromise the quality of catchment boundary. Hence, my irregular tessellated surface models generate higher quality catchment boundaries. Such catchments represent a more legally defensible delineation, and therefore may affect jurisdictional responsibilities with respect to water rights and other water resources-related issues.

This thesis is divided into the following sections: a literature review, data sources, methods, results, discussion and conclusion. The literature review focuses on the LiDAR data collection process and research into field of catchment delineations using raster-based surface models. The data section details the study area and data inputs. The methods section explaines the processing steps of ITSM Hydro, a Python-based series of algorithms that delineate flow direction lines and

create catchment boundaries using LiDAR bare-earth sample points. The methods section also details the raster-based catchment delineation methods that I used to compare and contrast the results of the Python algorithms and several different metrics used to assess raster-surface-model quality and to quantify differences in area and shape between the tessellated surface-model catchment delineations and the raster-based catchment delineations. The results section contains maps of catchment delineations for four test areas, and the comparative metric results detailing the comparisons of catchment delineations for both raster and ITSMHydro methods. The discussion section highlights differences between the tessellated and the raster-based delineations, and the limitations and conditions of the ITSMHydro tool set. Finally, the conclusion and future work section is a discussion of suggested future enhancements and improvements to the tools developed for this research.

Section 2 Literature Review

This literature review examines trends in GIS and geomorphometry to correct data model errors and extract hydrography and basin boundaries from digital surface models in a GIS. This literature review will also address recent approaches to overcoming sampling error for the purpose of constructing higher quality surface models for the extraction of hydrographic features, and addressing the error introduced by the DTM. Finally, this literature review highlights the use of tessellated surface data models to predict overland flow directions.

This section is organized thus: Section 2.1 describes the LiDAR data collection process. Section 2.2 reviews the use of regular gridded surface models for defining flow direction surfaces, flow accumulation surfaces, and defining catchment boundaries. Section 2.3 reviews the process of interpolating a regular gridded surface model from a random distribution of sample points and methods to compensate for error introduced in the interpolation process. Section 2.4 introduces the concept of irregular tessellated surface models and summarizes the research on their use in hydrologic modeling.

2.1 LiDAR

LiDAR data is widely recognized as a means to improve the spatial accuracy and precision of surface elevation data over the USGS DTM. LiDAR is a remote sensing technique wherein a pulse laser is attached to the bottom of aircraft containing a high-accuracy global positioning system (GPS), a sensor to capture the reflecting laser pulse, and an on-board computer to correlate the plane's altitude and position with the individual LiDAR pulse returns (Campbell 2002).

Pulses from the laser are reflected from ground surfaces (bare earth, vegetation, buildings and other structures on the ground) and captured by the sensor. The onboard computer calculates the coordinate of the pulse strike and records the x, y, and z values based on the time difference between the pulse emission and the pulse return, and the known current position of the aircraft (Wehr 1999). In ideal conditions, LIDAR can produce sub-meter elevation accuracy for each square meter of surface, which is an improvement over the 7 to15 meter error of the traditional 10 to 30 meter DEMs interpolated using stereo-photogrammetry techniques implemented by the USGS (Garbrecht and Martz 2000). LiDAR data collection results in a point cloud, or a series of different LiDAR strike returns, including a first return dataset that records the upper elevation values of surface vegetation (a false panchromatic aerial photograph based on the strength of the returning LiDAR strike) and a last return, or bare-earth surface for all LiDAR strikes that penetrate the vegetation canopy (Campbell 2002).

2.2 Overview of Raster DTMs in Surface Water Analysis

Raster-based surface models are the dominant data structure for predicting overland flow directions and defining watershed boundaries within a GIS (Garbrecht and Martz 2000). Raster DTMs are the standard input into a number of GIS software packages such as ArcGIS, WMS, HEC-RAS, and GRASS (Maidment 2000). Raster-based GIS flow direction algorithms iterate through the surface model matrix and computationally define flow direction based on the steepest slope to the surrounding coincident cells, otherwise known as the deterministic-eight-direction (D8) algorithm (Figure 1;Maidment 2000). The flow direction calculations are then used to

computationally define watershed boundaries by defining cells that share flow connectivity (Figure 2).

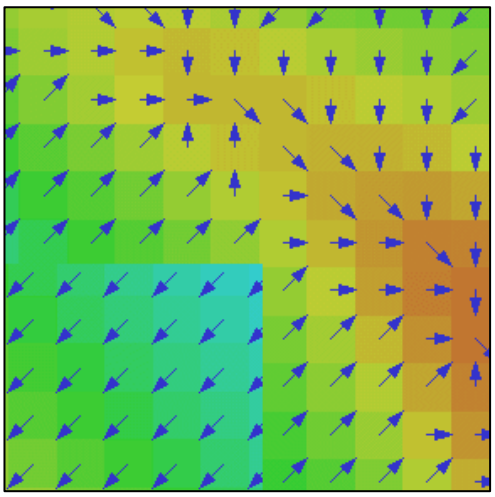

Figure 1 A typical raster DTM showing the resulting D8 flow direction calculations.

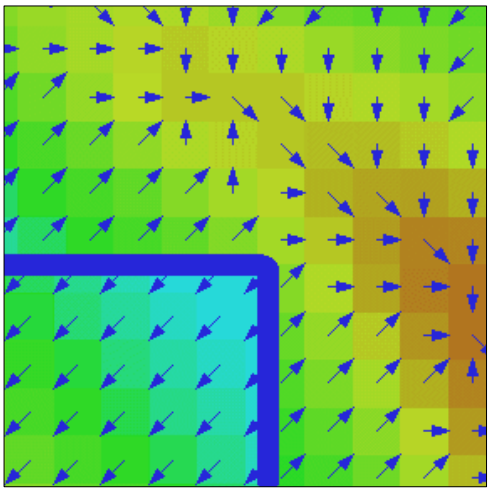

Figure 2 Flow direction calculations and the resulting basin delineation line in blue.

The popularity of raster-based surface models for hydrologic modeling is due to the availability of nation-wide DEM data sets, the rapid computer processing times of the raster data model within a GIS, and the limitations imposed by computer storage capacities (Garbrecht et al. 2001). Limitations of data availability, computer processing speeds, and data storage capabilities have historically outweighed the facts that cell resolution, overestimation and underestimation of cell values due to interpolation, and flow direction constraint (to eight cardinal directions) of the raster data model each degrade the quality of the flow direction and watershed boundary calculations. In areas of high topographic relief, the errors in raster-based flow directions due to cell resolution, interpolation, and flow direction constraints are less pronounced since flow is typically unidirectional and flow paths converge to a single discharge point (Jones 2002). In areas of low topographic relief, however, subtle differences in elevation values compound with

interpolation uncertainty and increase the number of artificial sink cells (cells with no outflow), limiting the quality of flow direction certainty across the surface model (Jones 2002).

2.3 Raster DTM Creation, Interpolation, Resolution, and Correction

Garbrecht and Martz (2000) provided a succinct overview of the issues affecting the definition of stream channels and the delineation of watershed boundaries based on raster DTMs. The authors excluded from their discussion areas influenced by urban development, which might alter surface flow patterns.

A number of issues limiting accuracy were highlighted in the paper including: (1) three quality levels of USGS DEMs and the techniques used to generate DTMs of those qualities; (2) how USGS DEMs store integer values for elevation, which in turn limits slope calculations and subsequently flow directions in areas of low spatial relief; and (3) cell resolution, or pixel size. Cell resolution is a critical issue with DTMs because cell resolution affects the number of sinks, or pits, i.e., cells that do not have flow out paths. Pits are a problem exacerbated by low cell resolution and low topographic relief that leads to incomplete drainage patterns. Cell resolution also affects the resulting stream length, with low-resolution DTMs producing shorter stream lengths than the actual channel.

Garbrecht and Martz (2000) also highlighted some issues with the D8-direction-flow-path algorithm common in most GIS applications. Because flow is restricted to only one of eight cardinal directions, divergent flow is not captured in areas of low spatial relief over convex slopes, resulting in biased flow directions. The authors acknowledged that if the intended outcome is a

watershed delineation, the D8 is an adequate choice over multiple flow path algorithms. Sink cells, whether actual or artificially generated by DTM interpolation, are problematic for the D8 regardless of whether they are a result of over- or underestimations of cell value. A number of techniques, including an artificial computational leveling of sinks to enforce hydraulic connectivity (filling) or a computational lowering of obstructions (breaching) were mentioned, but specific methods were not discussed in detail. Finally, the authors stated that methods to define flow across flat areas, whether actual or artificially created, are elusive and the user will have to 'contend with approximations.'

Barber and Shortridge (2005) addressed the issue of raster cell resolution in a comparative analysis of the hydrography and watershed boundary results from a 6-meter cell LIDAR-generated raster surface model compared with standard 30-meter USGS National Elevation Dataset (NED) DTMs. Barber and Shortridge compared the results in an area of high spatial relief and an area of low spatial relief in North Carolina using the standard hydrography toolset found in a popular GIS system, ESRI ArcGIS 8. For both the LIDAR and NED data, stream networks were calculated, random sample points field-validated, and the resulting values compared statistically between the different ArcGIS hydrologic model outputs produced by the authors. Barber and Shortridge found that the LIDAR surface model did not produce significantly better results for stream networks, but it did show a modest improvement in watershed boundaries, especially for the area of low relief. Barber and Shortridge mentioned the error introduced by surface feature artifacts like bridges, but the investigators do not describe the implementation of fill procedures or stream burning. However, the metadata for the 20 meter LIDAR DTM produced by the state of North Carolina (from which the author's 6-m DTM was interpolated) indicates it was corrected for

known bridges. Furthermore, the interpolation method used is a potential source of error (Wechsler 2007) and likely skews the stated results of this paper.

Haile and Rientjes (2005) focused mainly on the issues of DTM interpolation and cell resolution to address issues in modeling the effects of flooding. Beginning with a 1.5-meter resolution LIDAR-derived surface model, the researchers employed a number of re-sampling methods to derive lower resolution surface models. The lower resolution surface models were then compared for computer processing time and the accuracy of the model hydraulic outputs. For example, a resampled 15-meter DTM took approximately 1 day to process in their flood inundation model, which included the generation of surface flow patterns, while a 2.5-meter DTM took 13 days to process (1.5GHz Pentium IV). The resulting inundation area was significantly affected by the raster cell size, with the coarse 15-meter DTM showing a 3-fold increase in inundated depth compared to a 5-meter resolution DTM. The authors employed nearest neighbor, bilinear, and bicubic resampling techniques and generated a range of different output cell sizes, and the elevation differences of the new surfaces were compared against the original 1.5-meter surface. For all three resampling methods, the 4.5-meter raster resulted in a mean ~0.54 meter overestimation, and the 10-meter DTM was associated with an underestimation of 0.14-0.45 meters. While this article is a good treatise on the different resampling methods and processing times, and althoughthe authors made a strong case that model accuracy is related to DTM cell size, the authors never stated the total relief of the study area to give the reader an indication of the significance of the differences in the interpolation comparisons (Haile 2005).

Wechsler (2007) provided a succinct cautionary outline of the fundamental problems associated with the DTM data structure. Wechsler systematically addressed DTM sampling error,

differences in the algorithms used to derive surface features, DTM grid resolution, interpolation from the raw sample data to a raster DTM, and the use of surface modification to enforce flow connectivity as potential constraints on any watershed analysis. The most pertinent information he provides is the application of a Monte Carlo simulation to evaluate the bias introduced through the *filling* of the DTM to remove pits. When filling a DTM, an algorithm passes over each DTM cell and identifies which cells are lower than the eight neighboring cells; if a cell is identified as lower than its neighbors, the cell is marked as a sink, or depression. The fill process computationally corrects the *z*-value of the sink to ensure that there will be flow connectivity across the DTM; that is, the *z*-value of the cell is raised to equal the elevation of its neighbors, thereby enforcing flow connectivity. Wechsler's work showed that filling the DTM influences slope and alters the flow regime of the original surface, and he demonstrated that the problem of surface abstraction is aggravated in areas of relative flatness, like agricultural fields.

Similarly, Lindsay and Creed (2005) addressed error introduced through the process of filling DTMs, but they also presented a method to reduce error associated with removing depressions from the surface model. When breaching, the connectivity of the sink cell with surrounding cells is enforced by lowering the z values for any cells that form an obstruction between the sink and cells at a predefined distance from the sink. Neither filling (the raising of sink cells) nor breaching (the lowing of obstruction cells) provides a useful way to enforce flow connectivity alone since the utility of each fill or breach depends on the cause of the sink, which is an unknown. For example, if the sink is a result of an underestimation of the cell value, then filling is the preferred method, and if the sink is a result of an overestimation of neighboring cells, then breaching is the preferred method. Lindsay and Creed acknowledged the shortfalls of filling

and breaching and then presented a method called the impact reduction approach (IRA) to overcome this shortfall. From the initial surface model, the IRA generates two additional copies of the surface model and one *result* raster at the same spatial extent of the original surface. All sinks are filled in the first surface model copy and breached in the second copy. The two new surfaces are analyzed computationally on a cell-by-cell basis, comparing the number of fills/breaches and the mean absolute difference from the original surface. The resulting number is written to the result raster. The result raster is iterated cell-by-cell and the resulting values determine a fill or breach of the coincident cells of the original surface model. After a substantial and detailed statistical analysis of the differences in the fill/breach methods, the authors concluded that the filling method typically employed in commercial GIS software packages greatly impact the derived terrain attributes, particularly in areas of flat bottomlands. The authors showed that their IRA method is a substantial improvement in the construction of hydrologically enforced surface models.

2.4 Overview of Irregular Tessellated DTMs

Two tessellated data structures for representing a continuous elevation surface are the Triangulated Irregular Network (TIN) (Peucker et al. 2002), and Voronoi diagram (VD) (Gold 1989). With a TIN, a given a set of scattered elevation sample points become nodes in the TIN mesh, and the nodes are connected by lines which form triangles, all of which share a topological relationships with their adjacent neighbors (Figure 4;Peucker 1977). TIN surface models in Environmental Systems Research Institute (ESRI) GIS software package ArcGIS utilize a Delaunay

algorithm which result in short triangle facets that satisfy the criterion that, for any triangle, a circumscribed circle that intersects the triangle nodes will not contain nodes from any coincident triangles (Figure 3).

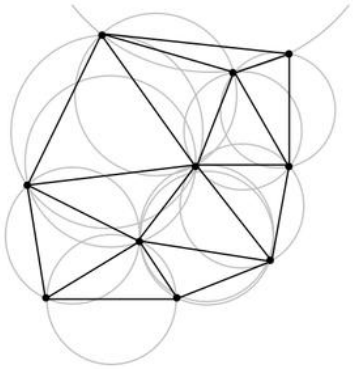

Figure 3 For each triangle in a Delaunay TIN, a circle that intersects the triangles nodes will contain no other nodes (de Berg 2008).

Voronoi diagrams (VDs), or Thiessen Polygons, can also be used to represent surface elevation. VDs define an asymmetrical polygon region around each sample point such that any area bound by the Voronoi polygon is closer to the sample point than any other sample point (Gold 1989). Because polygon coverage is continuous over the surface, the inherent topological relationship between adjacent polygons in the data structure lends itself to spatial modeling (Gold 1997).

A number of triangulation algorithms exist which will result in different triangulations for the same points, for example, to minimize or maximize triangle angles, and quadtree(Sack 2000). The edges in a Delaunay triangulation, or triangle sides, connect nodes that share the same spatial

relationship as the adjacent polygons of a Voronoi diagram created from the same nodes where other triangulation methods do not maintain this relationship (Figure 4). TIN surface models are gaining popularity in the GIS community as researchers gain access to the popular LiDAR elevation sample data, but the tools to use TINs in hydrologic modeling are not readily available to GIS users.

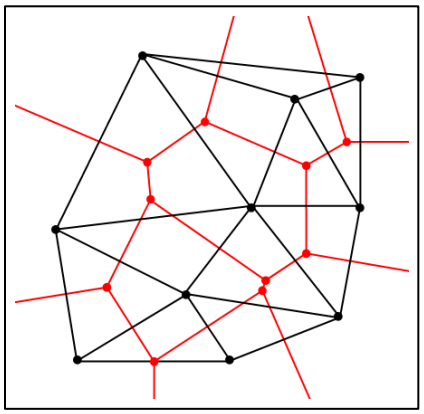

Figure 4: shows the relations ship between the Voronoi diagram (red), and the TIN edges (black). The TIN edges connect the nodes of adjacent Voronoi polygons (de Berg 2008).

Gehegan and Lee (2000) provided a clear overview of the different types of tessellated surfaces, focusing their research on the Voronoi diagram. The authors made the argument that the raster-based surface and the interpolation needed to derive the DTM lose important information contained in the collected discrete point data. Most notable are the loss of spatial relationships between the adjacent sample points and the relative difficulty in updating or changing the surface model. While the authors recognized the utility of the DTM, they raised the concern that the DTM might not be the best surface model for all applications. They suggest that ordinary Voronoi diagrams (all areas closest to the sample point), farthest Voronoi diagrams (all areas farthest from the sample point), higher order Voronoi diagrams (two or more points are bound by the polygon), and Delaunay triangulations provide the file architecture necessary to avoid DTM abstraction. When describing the creation process of the tessellations, the article provides the fundamental methodology necessary to construct flow convergence networks from the

tessellations through an iterative process of sample point selection, coincident Voronoi polygon selection, attribute retrieval, and attribute recalculations.

Dakowicz and Gold (2007) employed the use of Voronoi diagrams to model surface runoff using tessellations and stated that the tessellation approach bypasses a number of the abstractions imposed by the DTM, namely the error introduced by interpolation, the loss of the original sample points, and the flow direction constraints imposed by the D8 algorithm. Dakowicz and Gold used a series of sample points to create a TIN surface and a Voronoi diagram. They calculated the area of each Voronoi cell, determined the rate of precipitation, calculated the rate of flow from each cell to its downslope neighbor from the TIN edge, and passed the volume over time to the downslope-Voronoi polygon. While Dakowicz and Gold provided a simple argument for using tessellations to determine flow directions, what they fail to state in the article are specific methods for how the tessellated data structure permitted the surface runoff simulation to pass information to the downslope cell. The Dakowicz and Gold Voronoi model does bypass the sink problem since precipitation volume per Voronoi cell over time is passed to its downhill neighbor, and when the height in the Voronoi exceeds the pit height, flow passes down slope. Although not reviewed here, Li and Piltner (2004) suggested that the file architecture of the tessellated surface incorporates a related database record for each individual tessellation, and that this database allows the storing, altering, and adding of attribute information, which can be returned for this type of iterative analysis. Dakowicz and Gold may have used attribute passing to overcome computer memory limitations in their analysis, but they failed to mention the computation time needed to accomplish the attribute passing, or the areal extent of their research area, which might influence model execution times. It is widely acknowledged that DTM-based processes are executed faster

than vector-based processing, but at a sacrifice of spatial accuracy. Providing some statistical information of the processing time per area would have indicated whether the tessellation approach is computationally feasible with a commercially available desktop computer for hydrologic modeling.

Finally, Tucker (Tucker et al. 2001) described a set of tools used to develop a distributed rainfall-runoff model using TIN data structures generated with a Delaunay triangulation and the associated Voronoi diagram. This article gives a detailed overview of the topological relationship among TIN objects (nodes, edges, and triangles). Each node stores a pointer to the incoming edge and the outgoing edge, each edge has a pointer to both nodes and both triangles, and each triangle has pointers to its nodes, edges, and coincident triangles. This set of pointers is exploited in several examples of pseudo-computer code to define flow path based on the relationship between the TIN edges of the underlying Voronoi diagrams. The topological relationship between TIN nodes is used to define flow directions, while at the same time the TIN node's corresponding VD area is returned and passed to the downstream TIN node. The VD areas are summed and used to define the total contributing watershed area.

Section 3: Data

This section describes the study area and the data inputs including data sources. LiDAR bare-earth sample data for a northwestern section of the USA are arbitrarily selected as sample data to test algorithms that utilize irregular tessellated for the purpose of generating catchment boundaries. These data are intended to address and bypass issues of error by raster interpolation, constraints of raster-based flow routing routines, and pixel resolution as introduced by the raster-based surface model. Section 3.1 details specific information about the study area, section 3.2 describes the data including the LiDAR source and quality and other contributing datasets used to produce the highest quality raster surfaces and validate the quality of the raster surfaces used to compare and contrast results.

3.1 Study Area

The study areas for this thesis are those lands that contribute to overland flow onto the Lummi Indian Reservation, located near Bellingham, Washington (Figure 5), at approximately 48.79N degrees latitude and -122.62W degrees longitude. The Lummi Reservation is best described as Puget Sound lowlands, including forested uplands, agricultural fields, cleared or partially forested floodplains, river deltas, and rural residential density with some concentrations of housing developments. The Lummi Reservation, and the adjacent non-Reservation lands, typically has low topographic relief, with a maximum elevation of 600 feet and a mean elevation of 80 feet over the 36.69 square mile study area.

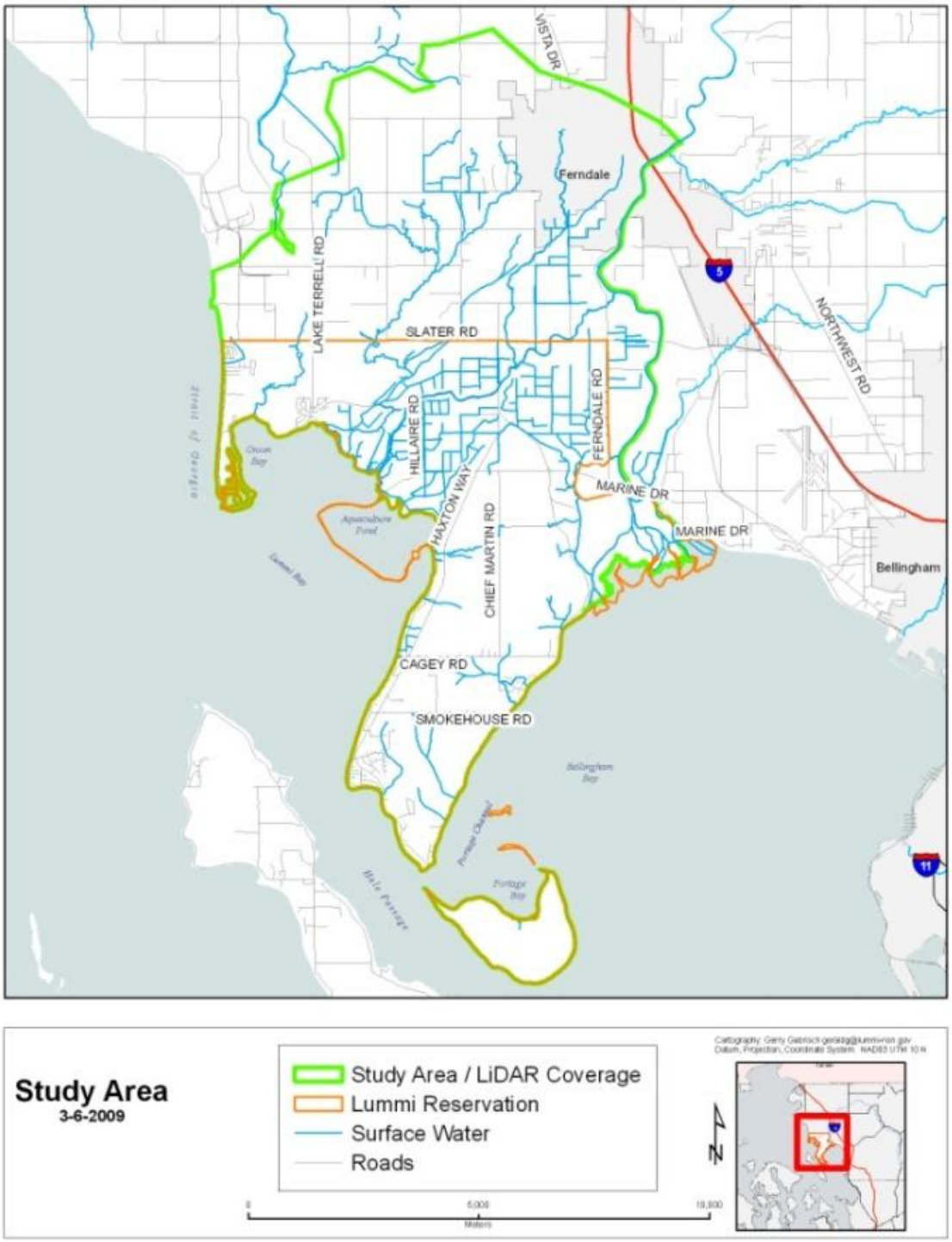

Figure 5: The Lummi Reservation shown in orange and the extent of LiDAR coverage (shown in green). The vendor-provided LiDAR data was edited to exclude any sample point lower than the mean higher high water line (NAVD88 vertical datum), and east of the Nooksack River.

3.2 DATA

Between March 6th 2005, and May 4th, 2005, LiDAR data were collected and processed by Terrapoint USA Inc. of The Woodlands, Texas (Terrapoint). Terrapoint used a 40 kHz ALTMS (Airborne Laser Terrain Mapping System), a Trimble 4700 GPS receiver, a Honeywell H764 IMU, and two Sokkia GSR2600 dual frequency GPS receivers mounted to a fixed wing aircraft averaging an elevation 3500 feet to collect LiDAR data. The Terrapoint Project Report reports the following accuracies (Terrapoint 2005):

> *Accuracy is as follows, quoted at the 95% confidence level (2 sigma),*
>
> *Absolute Vertical Accuracy:*
>
> *+/- 15-20 centimeters on Hard Surfaces (roads and buildings)*
>
> *+/- 15-25 centimeters on Soft/Vegetated Surfaces (flat to rolling terrain)*
>
> *+/- 25-40 centimeters on Soft/Vegetated Surfaces (hilly terrain)*
>
> *Absolute Horizontal Accuracy:*
>
> *+/- 20 – 60 centimeters on all but extremely hilly terrain.*
>
> *Contour Accuracy:*
>
> *2ft Contour National Map Accuracy Standard (NMAS)*

All horizontal coordinate data were collected and referenced to North American Vertical Datum of 1988 (NAVD88) and delivered in US State Plane Zone Washington North with coordinate and elevation values listed in US Survey feet. Space-delimited text files of bare-earth sample points provided by Terrapoint were transformed into ESRI shapefiles. The resulting shapefiles were manually edited to exclude marine waters below the NAVD88 mean higher high water (MHHW) tidal line. Figure 5 shows the extent of LiDAR data coverage (show in green) after those marine waters lower

than the MHHW line were removed from the dataset to reduce overall file size and computer processing time.

In 2006, Lummi Indian Business Council (LIBC) GIS staff used 6 ft. bare-earth raster DTM surface models provided by Terrapoint, coupled with 2006 Pictometry imagery, to edit an existing surface-water-hydrography data set, including stream and river networks and agricultural drainage ditches, to conform to channels apparent from the DTM and the aerial imagery.

In 1998, an LIBC staff hydrologists and LIBC Water Resource Division staff conducted a field inspection of all areas of the Lummi Reservation (Reservation) to identify the location of storm water facilities (culverts, tide gates). The positions of storm water facilities were captured using a mapping grade Trimble GeoXT Global Positioning System (GPS). Similarly, a survey of storm water facilities was conducted by the Whatcom County Public Works (WC) department to capture the point locations of storm water facilities off Reservation. An ESRI geodatabase-point-feature-class was provided by Whatcom County, but no further detail are known about these WC data.

Table 1: GIS data summary. This table details the spatial datasets utilized in this research. LiDAR point locations were either used directly in the ITSMHydro analysis, or used to create raster surface models. Hydrography and storm water facility data were used to hydrologically correct raster surface models, and survey point data were used to assess the quality of the raster-based surface models.

Data Provider	Description	Data Type	Data Model	Use
Terrapoint USA Inc.	Post processed LiDAR bare-earth sample points.	Text	Tab delimited text files	Surface model creation.
Lummi Indian Business Council	Hydrography, rivers, streams and irrigation ditches.	Vector	ESRI line Shapefiles	Surface model reconditioning.
Lummi Indian Business Council	On-Reservation storm water facilities (culverts)	Vector	ESRI point Shapefile	Surface model reconditioning.
Whatcom County	Off-Reservation storm water facilities (culverts)	Vector	ESRI Geodatabase point feature class	Surface model reconditioning.
Pacific Surveying and Engineering	Surveyed elevation control points.	Vector	ESRI point Shapefile	Surface model evaluation.
Aspect Engineering	Surveyed elevation control points.	Vector	ESRI point Shapefile	Surface model evaluation.

Section 4: Methods

This section details a sequence of geoprocessing tools developed in the Python programming language that interface with ERSI ArcGIS v9.3.1- v10 to delineate flow direction lines and catchments from a random distribution of sample points. The toolset, collectively called Irregular Tessellated Surface Model Hydrography (ITSMHydro), requires the generation of a geodatabase workspace and a TIN surface model to facilitate the spatial relationship and geoprocessing of neighboring sample points (ESRI 2010). This section also details the methods used to create and assess the quality of several raster-based surface models using different pixel resolutions and interpolation. The raster surface models provide an industry-standard benchmark against which to compare the ITSMHydro catchment delineations. Known watercourses and storm water facilities serve to hydrologically correct the raster surface models prior to catchment delineation using established ArcGIS ArcHydro methodologies to maximize the quality of the raster delineations.

Subsections 4.1 and 4.2 detail the preprocessing steps required for execution of the ITSMHydro tools, including the creation of a geodatabase workspace to hold model output files and the creation of a bounding polygon to define the analysis extent. Subsection 4.3 describes the methods used to create a hydrologically corrected TIN that 'burns' stormwater facilities and known stream networks into the TIN surface model. Subsections 4.4 – 4.7 summarize the ITSMHydro Python algorithms which, respectively: (1) export TIN nodes (LiDAR sample points), edges (lines connecting LiDAR points) and polygons (triangles formed by the TIN generation process) into the ESRI file-geodatabase data workspace; (2) generate a new feature class of flow direction lines that

describes the steepest path of descent from each LiDAR point; (3) utilize the flow direction lines to create catchment boundaries for each set of connected flow direction lines; and (4) fill sinks. Subsection 4.8 outlines the methods used to create a series of raster surface models using different pixel resolutions and interpolation methods, assess the quality of those raster surface models, and generate catchment boundaries from the highest quality surface.

4.1File Geodatabase Creation

An ESRI feature dataset within a geodatabase is required to store all geoprocessing outputs from ITSMHydro. No specific naming conventions are required for the feature dataset or the geodatabase. ITSMHydro was tested using ESRI file geodatabases due to the improved performance and file storage capacity of the file geodatabase over the Microsoft Access personal geodatabase (ESRI 2010).

4.2 Bounding Polygon Creation

A user-defined bounding polygon must be stored within the feature dataset prior to the ITSMHydro code execution. This bounding polygon defines and limits the extent of the analysis area.

4.3 TIN Creation

For consistency with the methodologies that generate the raster surface models used to evaluate the ITSMHydro basin delineations, hydrologically-corrected Delaunay TINs were generated to establish hydrologic connectivity. Delaunay TINs were choosen over other triangulations methods because the Delaunay TIN polygons maintain the same spatial adjacent

relationship between LiDAR points as the ordinary Vorinoi diagram, thus providing a way to identify those points most spatially related.

LiDAR technology cannot capture the flow path of storm water facilities underneath roads because those flow paths are blocked from the aerial view of the LiDAR beam. To enforce hydrologic connectivity in those areas traversed by raised road beds, 'culvert burning' was used to establish flow paths through storm water facilities (Duke 2003). A 50-feet buffer polygon around each storm water facility point was created to sufficiently span the width of raised roadbeds. A new attribute was added to the resulting 50-feet buffer polygons and assigned a value equal to the lowest elevation value recorded in the TIN. The 50-feet buffer polygon feature class was used for TIN creation as shown in

Table 2 to replace any LiDAR nodes bound within the 50-feet polygons. Hydrography data provided by the Lummi Nation were also used to enforce hydrologic connectivity by stream burning the stream course into the TIN. The ArcGIS editing tool *divide* was used to insert vertices in the stream courses at 3-foot intervals. The stream line's vertices were converted to a point feature class. A new attribute column was added to the point feature class as a numerical data type. This new numeric attribute was populated with descending values beginning with -1, and descending -1 foot for each point along the line course. The points served as an input to create a TIN file using the artificially generated numeric values as the z values as input parameter for the new TIN as shown in

Table 2.

Table 2: Input data to create a hydrologically corrected TIN surface. This table shows which datasets served as input parameters for the construction of raster surface models. Column one lists the data set name, column two shows which value from the attribute table contributed elevation values for raster creation, and column three lists the ArcGIS parameter name, where mass point contribute are evaluated as TIN nodes, hard lines contribute z values with no TIN interpolations occurring across the line, and hard replace replaces any TIN values bound withing the polygon.

Feature Class	Z values Used	Input Type
LiDAR points	Attribute z	Mass Point
Stream Division Points	Attribute z	Mass Point
Stream	None	Hard Line
Storm Water Facilities Buffer	Attribute z	Hard Replace
Bounding Polygon	None	Hard Clip

4.4 LOAD TIN COMPONENTS

ITSMHydro requires the extraction of the TIN geometry as points, triangles, and triangle edges into the feature dataset. The Python/geoprocessing tool *0_LoadDataFromTINToGeoDataBase.py* (Appendix 1) automates the TIN component extraction process and ensures the correct naming conventions for nodes, triangles, and edges required by all

ITSMHydro tools. Input parameters for the *0_LoadDataFromTINToGeoDataBase.py* tool include the path to the TIN and the path to the feature data set. *0_LoadDataFromTINToGeoDataBase.py* also requires the user to set the output feature classes coordinate geometry resolution equal to the precision of the original LiDAR data, thus guaranteeing all node geometries are coincident (ESRI 2010).

0_ LoadDataFromTINToGeoDatabase.py extracts feature classes called nodes, edges, and triangles from the TIN and stores them in the feature dataset using ArcGIS methods available in Python using the ArcGIS Application Programming Interface. After importing the required ArcGIS libraries, *0_LoadDataFromTINToGeoDatabase.py* loads the required ESRI 3D analyst toolset into Python. Prior to execution, the user must define the script variables for the working directory, the path to the feature dataset, the x, y, z resolution, and the path to the TIN. The resolution settings shown in Appendix A were set to a precision that matched the LiDAR point text files recorded in hundredths of feet. Finally, the TIN edges (TIN triangles as lines) , TIN nodes (the LiDAR bare-earth sample points), and the TIN triangles are extracted from the TIN and stored in the feature dataset as ESRI point, line, and polygon vector data models.

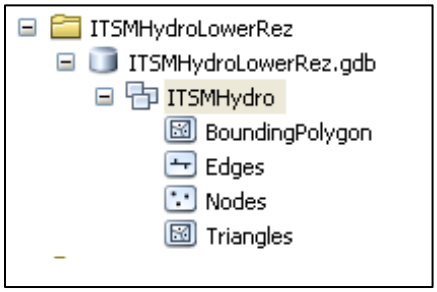

Figure 6: The working directory after 0_ LoadDataFromTINToGeoDatabase.py execution.

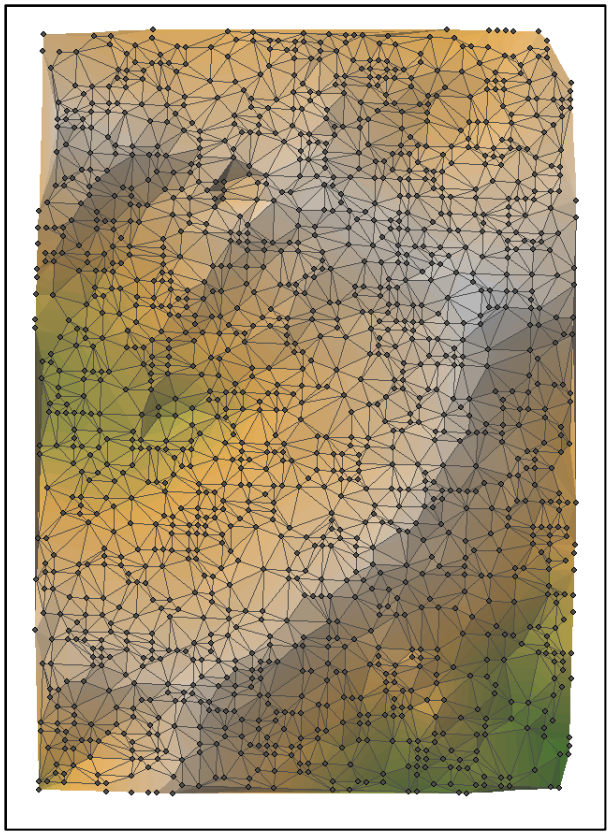

Figure 7: The resulting edge and node feature classes generated by *0_LoadDataFromTINToGeoDatabase.py.*

4.5 TIN Flow Directions

The second algorithm *1_FlowDirectionsFromTIN.py* (Appendix B) utilizes the node and edge components from a TIN surface model to generate a new feature class called *FlowDirections*. Figure 8 shows an abridged version of the processing steps of *1_FlowDirectionsFromTIN.py*. The resulting *FlowDirections* feature class represents the steepest path of descent from each node to its adjacent neighbors as defined by the triangulation.

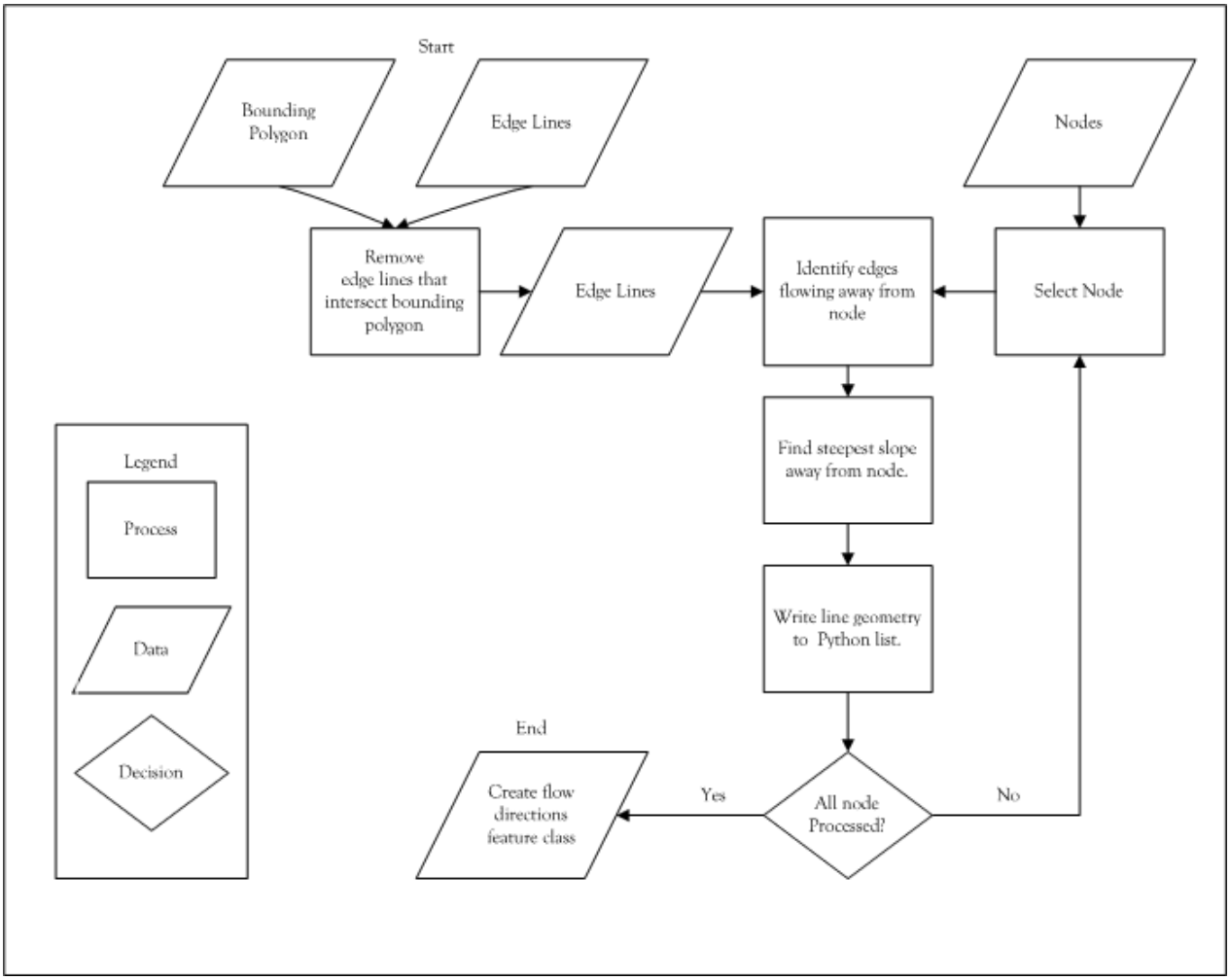

Figure 8: An overview of the processing steps of 1_FlowDirectionsFromTIN.py.

For each LiDAR point, the feature geometry and the x, y, and z values are written to a Python list data type. Similarly, the feature geometry of all edges are written to another Python list (the line's start node and end node coordinate values). For each LiDAR point, every line that shares a coordinate value equal to the LiDAR point is connected to that LiDAR point. Lines with a distal-end z value higher than the LiDAR point's z value cannot represent flow away from that LiDAR point and are ignored. If the line's distal-end z value is lower than the z value of the

LiDAR point, that line represents a potential flow path away from the LiDAR point. The distance formula (equation 1) is used to calculate the edge length based on the line's start and end node x and y values. The line's z values are used to calculate a rise by subtracting the higher z value from the lower z value, which is then converted into the percent slope (equation 2). The percent slope value of each line is appended to the Python list of edge coordinate values.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

d = distance

(x_1, y_1) = coordinate geometry of node.

(x_2, y_2) = coordinate geometry of opposing line end.

Equation 1 The distance formula for calculation the distance between two coordinates.

$$100 * \frac{rise}{run}$$

Equation 2 Formula for calculating the percent slope of a line where run is the value of d from equation 1.

The list of lines that represent potential flow paths away from the LiDAR point are sorted in ascending order based on the percent slope value. The last item in the list is that line with the steepest path of descent away from the LiDAR node. The coordinate geometry associated with these steepest path lines is converted to a new feature class in the geodatabase called *FlowDirections*. The resulting *FlowDirections* feature class represents lines of the steepest path of descent from each node to its adjacent neighbors defined by the Delaunay triangulation. After code execution, the

working directory will contain the feature classes detailed in Figure 9, including the newly created *FlowDirections* feature class (Figure 10).

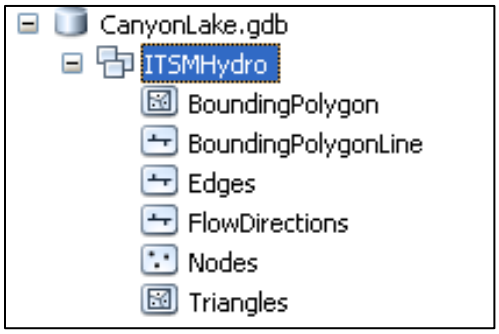

Figure 9: Shows the contents of the working directory after execution of *1_FlowDirectionsFromTIN.py.*

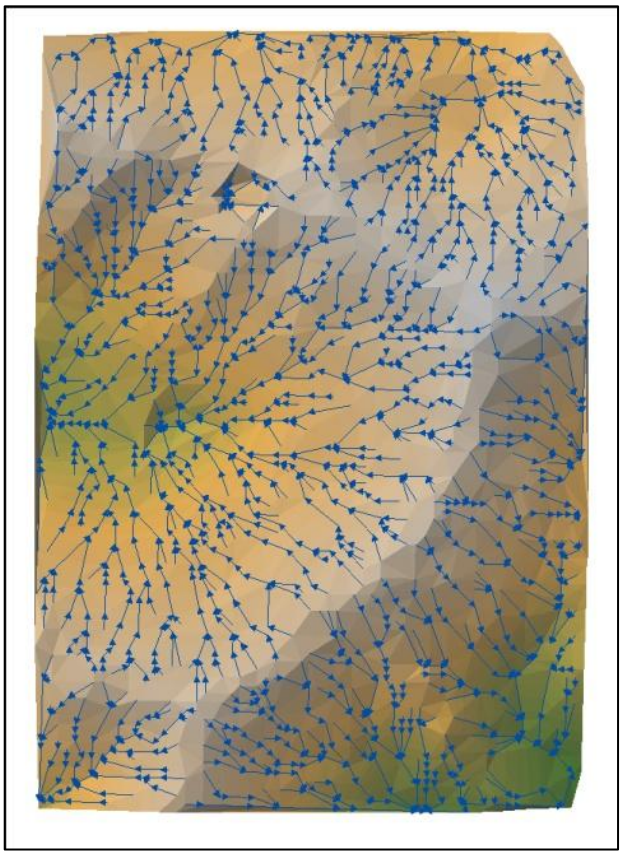

Figure 10: The flow direction lines generated by *1_FlowDirectionsFromTIN.py.*

4.6 TIN DTM Basin Delineation

Given the flow direction outputs of the 1_FlowdirectionFromTin.py tool, an algorithm is used to generate a polygon boundary that encapsulates those flow direction lines that share connectivity, and therefore represent a catchment boundary for those connected flow direction

lines. The algorithm *2_CreateCatchmentPolygons.py* (Appendix C), utilizes the node, edge, and triangle components of the TIN, and the newly generated *FlowDirections* feature class, to generate a new polygon feature class called *Catchments*. Those areas bound by the resulting *Catchments* feature class represent those areas that contribute to overland flow to a single pour point. Figure 11 shows an abridged version of the processing steps executed by *2_CreateCatchmentPolygons.py*.

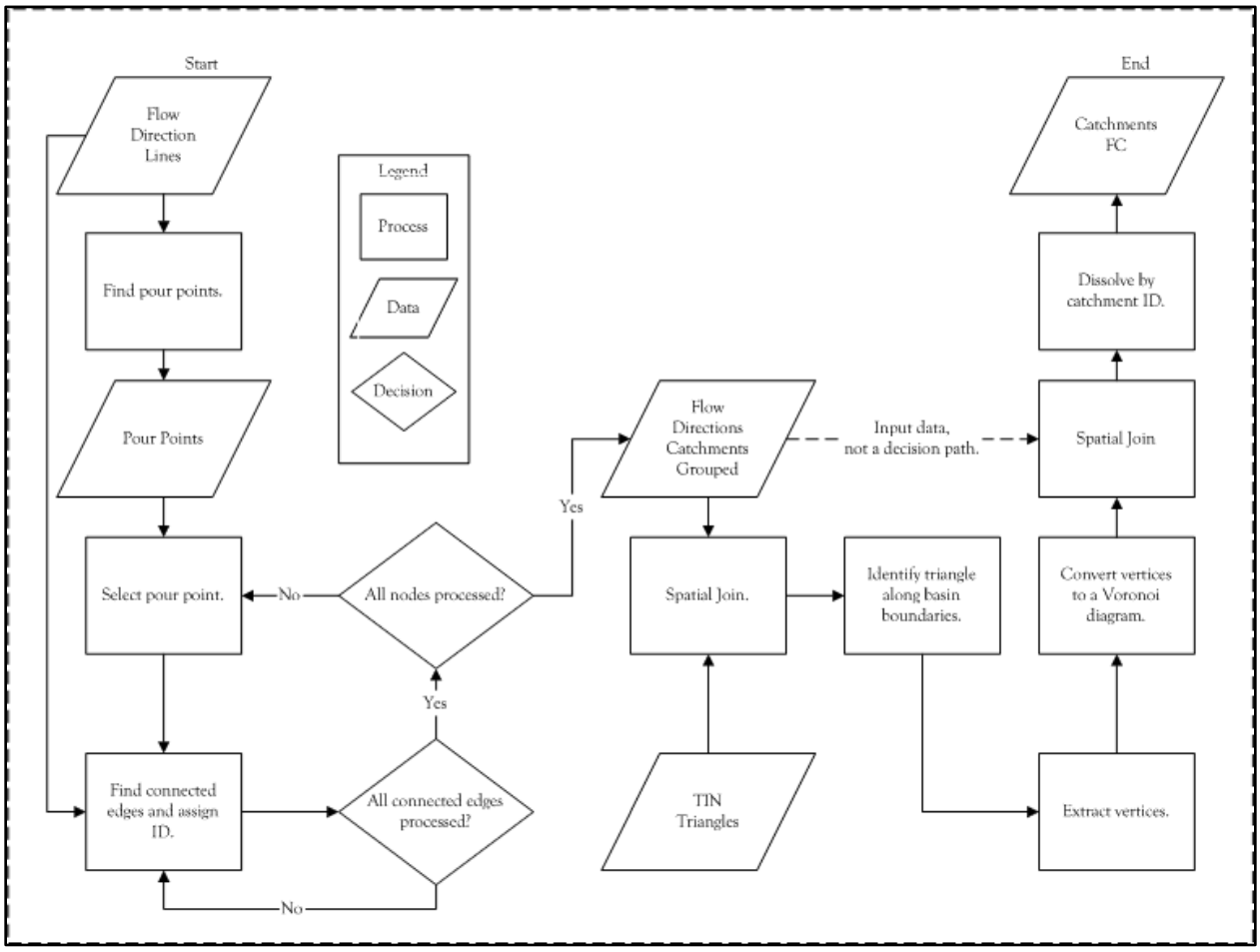

Figure 11: An overview of the geoprocessing steps of 2_CreateCatchmentPolygons.py.

A Python list stores the coordinate geometry of each flow direction line. Using the *Extract Line Vertices to Points* method in ArcGIS, all line end nodes and start nodes are exported to two

new point feature classes. Any end point that is not coincident with a start point represents the location of a pour point for that catchment. These points are selected from the feature class of vertices and their feature geometry is written to a new Python list. The list of points are sorted in ascending order based on z values. For each point item, all lines that intersect that pour point are assigned a catchment ID integer value starting with one. The coordinate values of these lines are passed into a Python function that identifies any connected upstream lines. This process is repeated recursively, thereby 'walking up' each branch of the *Flow Direction* geometry, assigning the same catchment ID to each branch of the flow direction lines. After all connected flow direction lines are assigned the same catchment ID value for that pour point, the next pour point is selected from the pour point list, one is added to the catchment ID, and the process repeats until all lines have been assigned a catchment ID. Based on the catchment ID, a new feature class is created called *FlowDirections_CatchmentGrouped* (Figure 12).

Figure 12: The grouped flow direction lines generated by *2_CreateCatchmentPolygons.py*.

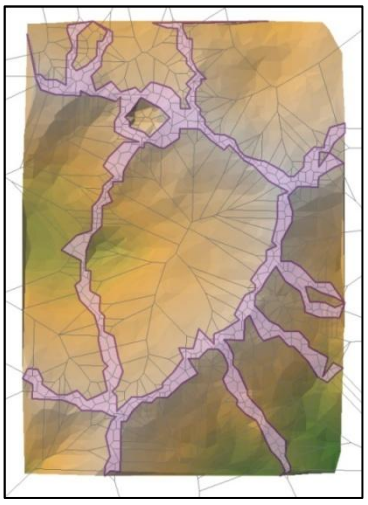

Figure 13: The areas in purple show those areas that exist on the periphery of any group of flow direction lines and represent those areas that contain catchment boundaries. The lines shown in gray are the Voronoi polygons generated from the vertices of the purple area.

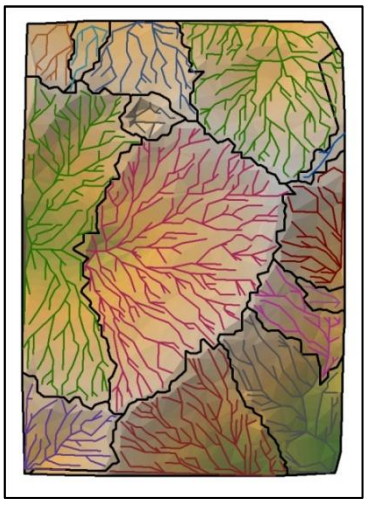

Figure 14: Final catchment delineations and grouped flow directions lines.

The *FlowDirections_CatchmentGrouped* feature class is spatially joined with the TIN triangle polygons using the ArcGIS spatial join method. A resulting attribute of the spatial join is a join count, which holds the number of *FlowDirections_CatchmentGrouped* features each triangle touches. Any triangle that intersects only one *FlowDirections_CatchmentGrouped* feature has a join count of one, any triangle that is adjacent to more than one *FlowDirections_CatchmentGrouped* feature has a join count of two or more. Join counts greater than one indicate triangles that exist on the periphery of two or more basins. The vertices of triangles with join count greater than one are converted to Voronoi polygons (Figure 13). The Voronoi polygons are then spatially joined to FlowDirections_CatchmentGrouped feature class and dissolved based on the basin ID values. The

resulting feature class called *Catchments* defines catchment boundaries around each group of connected flow lines (Figure 14). After code execution, the file geodatabase will show the feature classes detailed in Figure 15; users can define a setting in 2_CreateCatchmentPolygons.py to delete temporary files no longer required by ITSMHydro (Figure 16).

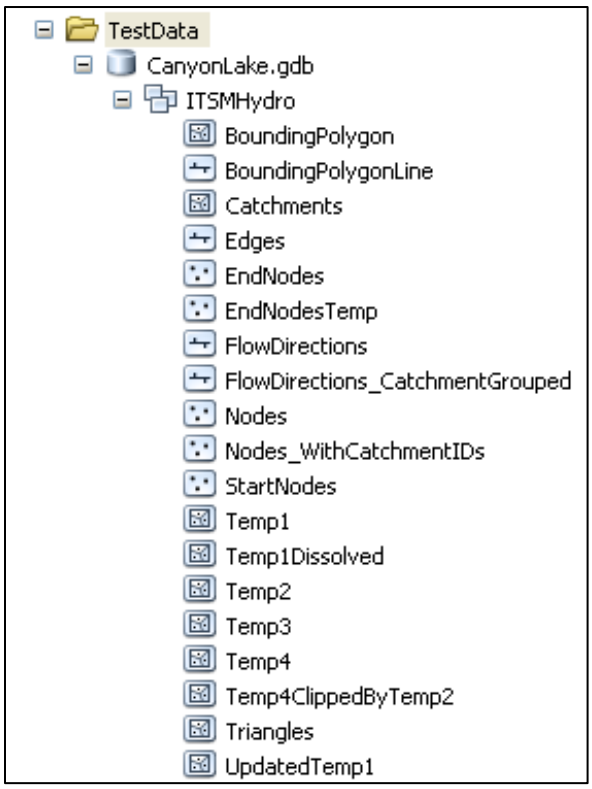

Figure 15: Working directory showing all files generated by 2_CreateCatchmentPolygons.

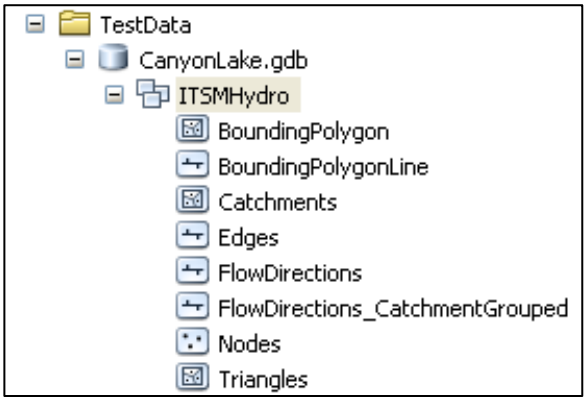

Figure 16: A user defined setting in *2_CreateCatchmentPolygons.py* will delete temporary files from the working directory no longer required by ITSMHydro.

4.7 SINK PROCESSING

LiDAR data may contain spurious pits or sinks, i.e., nodes that have a z value lower than the surrounding LiDAR points and, therefore, have no connected outflow path. The algorithm *3_AggregateSinkCatchments.py* (Appendix 4) merges those sink polygons with adjacent basin

polygons by assuming that all sink catchments will puddle, fill to capacity, then pour into one of the adjacent polygons based on the path of least resistance defined by the triangle edges. A sink catchment is defined as any catchment delineation that does not intersect the bounding polygon. Sink catchments are merged with one-and-only-one adjacent catchment. Figure 17 shows an abridged version of the processing steps executed by *3_AggregateSinkCatchments.py*.

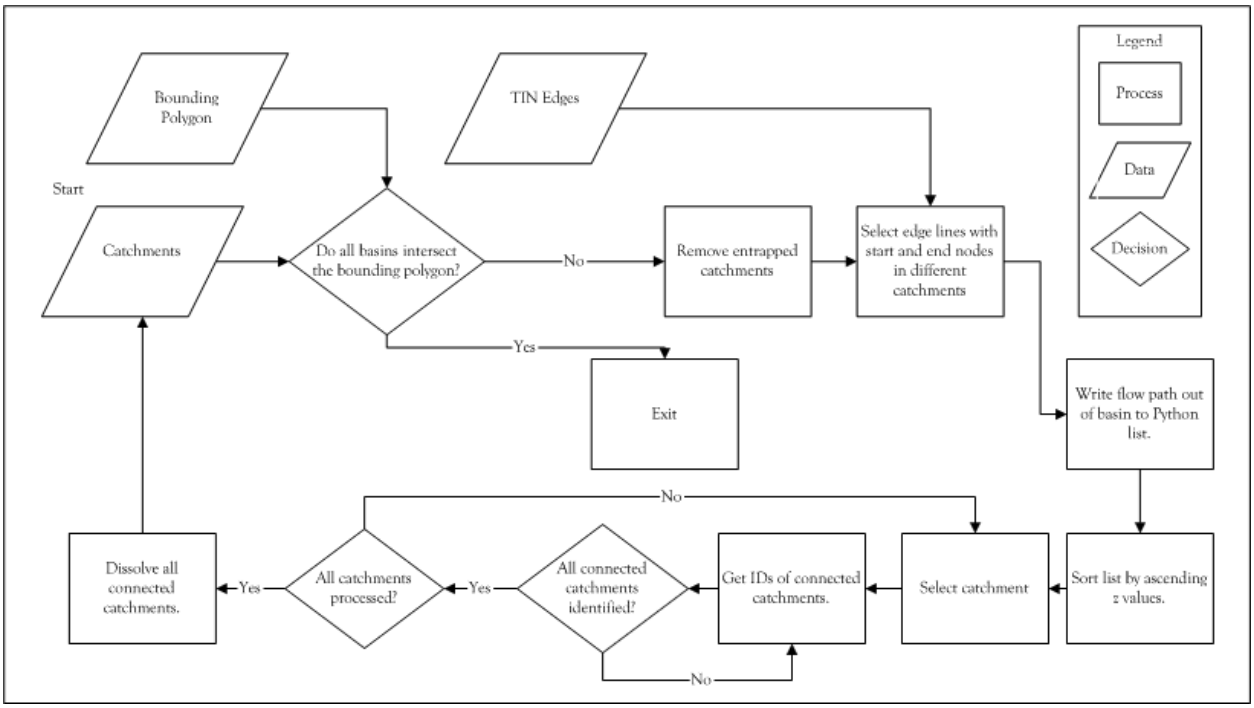

Figure 17: An overview of the geoprocessing steps executed by *3_AggregateSinkCatchments.py*.

3_AggregateSinkCatchments.py first identifies any polygon that forms an annulus and deletes the feature geometry of the interior portion of the annulus and the catchment delineation that fills the inner portion of the annulus. All lines that cross the catchment boundaries are selected from the data set of edge lines, and the feature geometry of those edge lines are written to a Python list. The lines that cross catchment boundaries are spatially joined with the basin Object ID, resulting

in two values: the catchment ID from which the edge line originates and the catchment ID to which the edge line flows. Any line that originates and terminates in the same catchment is removed from further computations. From the remaining lines, that line which has the lowest z value out of the catchment is selected as the flow path away from the sink polygon (Figure 18).

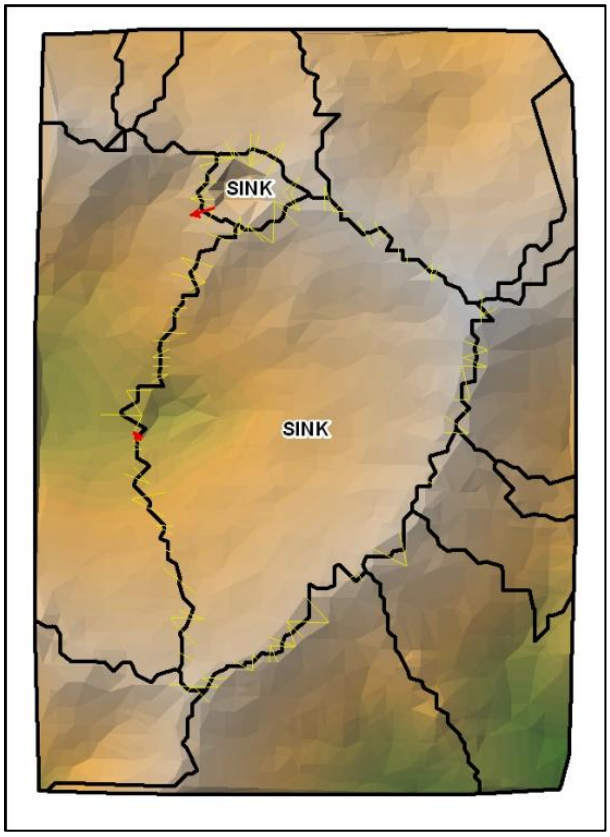

Figure 18: This surface has two sinks. The yellow lines are those edge lines with a direction of flow away from the sink. The red lines are the first path water would take out of the sink if the sink were filled with water.

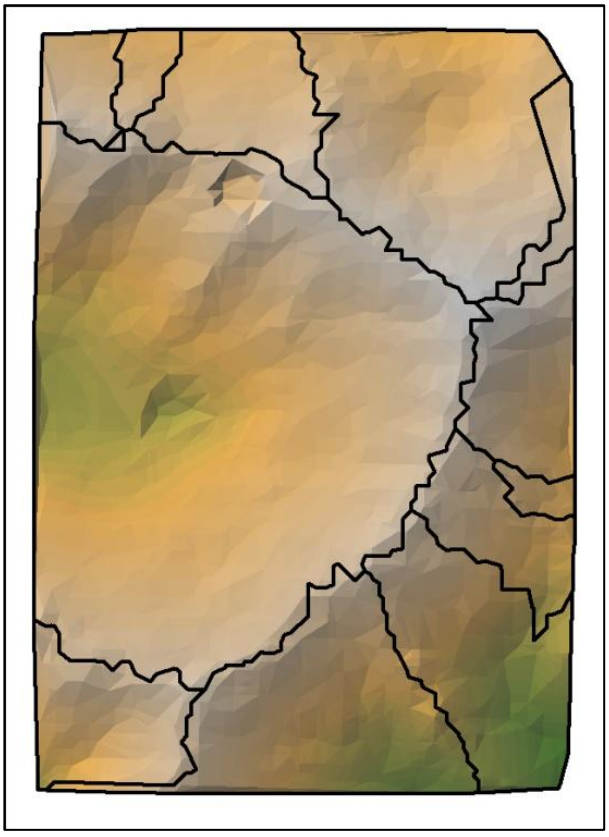

Figure 19: The final delineation is called *catchments* in the working directory. The catchment delineation shown in figure 18 was renamed to *catchments1*. All numerically numbered catchment feature classes are retained in the working directory.

If there is more than one line that has the same lowest z value, that line with the steepest path as defined by the line's slope is selected as the steepest flow path. A new Python list is generated to store the ID of the sink and the ID of the catchment that receives its flow. The sink with the lowest z value flow path is assigned a nominal identifier; the neighboring recipient basin is assigned the same nominal identifier. This process is repeated recursively until all connected polygons have been assigned the same nominal identifier. This recusion process loops until allpolygons have been assigned a nominal value. All polygons that have the same nominal value are dissolved together. The feature class called Catchments is renamed Catchments *n*, and this process loops until all catchment polygons touch the user-defined bounding polygon. The final feature class defining catchment boundaries is called Catchments (Figure 19). For each iteration, a feature class called Catchments *n* is written to the geodatabase (i.e., Catchments 1, Catchments 2, Catchments 3...). After code execution, the file geodatabase will show the feature classes detailed in Figure 20 (see Figure 21 for file structure with the "delete temporary files" setting invoked).

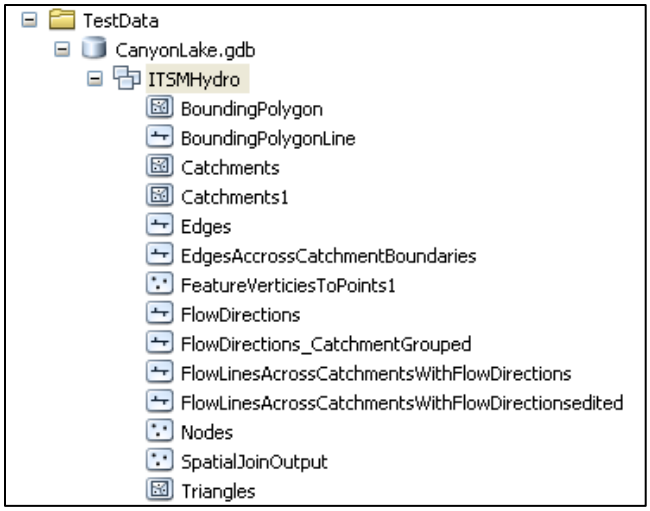

Figure 20: Working directory showing all files generated by *3_AggregateSinkCatchments.py*.

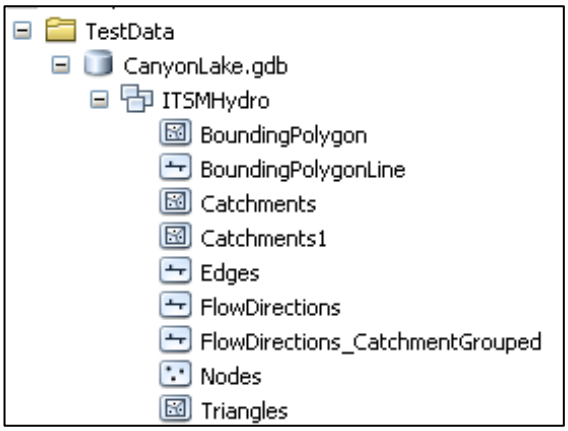

Figure 21: Working Directory showing all files generated by *3_AggregateSinkCatchments.py* using the "delete temporary files" setting.

4.8 Raster DTM Creation and Basin Delineation

This sub-section details the methods used to create raster DTMs and delineate catchment boundaries from LiDAR data using the raster-based geoprocessing tools in the ArcGIS 9.3 ArcHydro extension. Different pixel cell size and interpolation method combinations are used to identify which raster surface model produced the highest-quality raster DTM for the purpose of evaluating the ITSMHydro catchment delineation tools detailed in sections 2.2 – 2.5. The LiDAR points were used to create an ESRI Terrain data model. From that terrain data model, eight ESRI Grid surface models were created using five different pixel sizes and two different interpolation methods available in the ArcGIS v 9.3 software package (Table 3). While there are many different types of interpolation algorithms, the natural neighbors and the linear interpolation methods were selected because these two interpolation methods are default parameters for the ESRI Terrain to Raster conversion tool.

Table 3 Surface model cell resolutions and interpolation methods. Column one lists the pixel resolution, column two details the interpolation method used to transform LiDAR points into regular tessellations, and column three shows which surfaces were used a comparisons against the ITSMHydro toolset.

Raster Grid Resolution/Cell Size	Interpolation Method	Used For Catchment Delineation
30-feet	Linear	Yes
30-feet	Natural Neighbor	Yes
6-feet	Linear	No
6-feet	Natural Neighbor	No
3-feet	Linear	Yes
3-feet	Natural Neighbor	Yes
1-foot	Linear	No
0.5-feet	Linear	No

A root mean square error (RMSE) calculation was performed on each dataset to quantify elevation precision and, therefore, determine the highest quality watershed delineation. The RMSE value determines the average difference between the interpolated pixel values of the surface model and the elevation values of surveyed locations (Wu 2008). The surveyed sample points used for the RMSE calculations included 63-point locations with surface elevation values determined by professional survey.

Equation 3 Root Means Square Error equation to determine the average difference between the interpolated cell values and surveyed point elevation values where X_1 represents the interpolated pixel value at the surveyed elevation location of X_2, and n represents the total count of surveyed locations.

.

$$\mathrm{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(x_{1,i} - x_{2,i})^2}{n}}.$$

Storm water facility buffers were converted to a 3-foot raster Grid surface model and assigned an elevation value equal to the minimum value of the entire LiDAR dataset. The pixel values of the storm water facility Grid were used to computationally replace the coincident pixels in the surface models, thereby establishing a connective flow path across the "obstruction" created by the raised roadbeds.

The hydrography vector lines were manually edited to ensure that, for each individual line segment, the line direction of flow matched the direction of flow detailed in the Lummi Nation Storm Water Inventory. The ESRI ArcHydro geoprocessor cannot calculate flow directions in a network of looping flow paths, for example braided streams or interconnected drainage ditches (Maidment 2002). For this reason, some hydrography lines had their uphill node disconnected from the network of flow paths to ensure that no flow line formed closed loops.

The resulting 'culvert burn' surface model and the non-looping hydrography data set were imported into an ArcHydro geodatabase. The ArcHydro database allowed the stream network (hydrography) to be 'burned' into the surface models, enforcing flow connectivity based on the configuration of the stream network (Maidment 2002). The resulting hydrologically-corrected

surfaces were filled using the ArcGIS *fill* function to remove sinks and obstructions from the surface models that might impede the analysis.

The filled surface models were used to generate flow direction surfaces detailing the flow direction from each cell to one of its eight adjacent neighbors (Figure 22). The flow direction surfaces were then used to generate a flow accumulation surface where the numeric value of each pixel represents the total count of individual cells that flow into that cell (Figure 23). The flow accumulation surfaces were used to generate watershed boundaries where all cells that share a pour point are assigned unique nominal numeric value (Figure 24). The basin output was transformed from its Grid format into a polygon data structure.

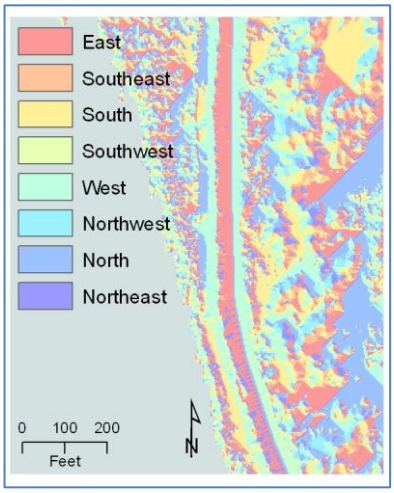

Figure 22: A typical flow direction surface detail. Each cell stores a numeric value detailing the flow direction in one of eight cardinal directions.

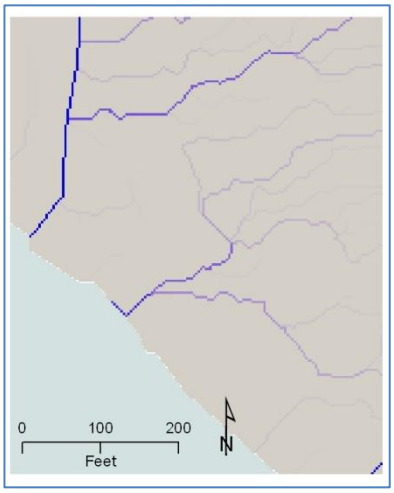

Figure 23: A typical flow accumulation surface detail; each cell stores the count of cells that pour into that cell. Higher cell counts are displayed as a darker blue.

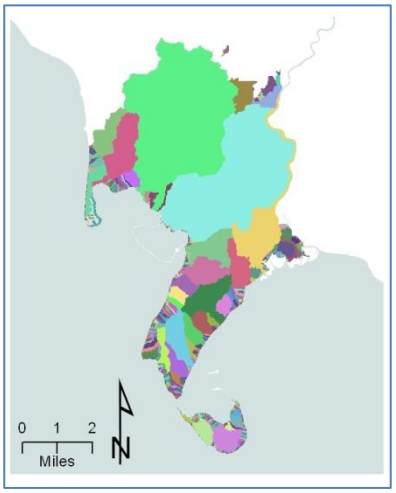

Figure 24: Resulting catchment boundaries generated from a 3 ft. natural neighbor DTM.

4.9 Comparative Metrics for Catchment Polygons

Three different comparative metrics were used to quantify the differences in catchment delineations between the raster-based catchments and the ITSMHydro delineations: (1) percent difference, which highlights differences in area (Equation 4); (2) coefficient of area correlation, which expresses difference in footprint surface areas (Equation 5); and (3) the isoperimetric quotient, a measure of the length of the line for polygons with normalized areas (Equation 6).

Equation 4: Formula to calculate percent difference between catchment areas.

$$PD = \frac{(\text{ITSMHydro Area} - \text{Raster Area})}{\left(\frac{(\text{ITSMHydro Area} + \text{Raster Area})}{2}\right)} * 100$$

The coefficient of correspondence (C_A) is a measure of areal association between two polygons as the ratio of the area of the intersection divided by the area of the union (Taylor 1977). If two polygons are identical in shape and coincident, the C_A will be one, if two polygons do not intersect, the C_A will be zero.

Equation 5: The formula for calculating the coefficient of area correspondence as expressed as the ratio of the area of the intersections of two polygons over the area of the union of two polygons.

$$C_A = \frac{X \cap Y}{X \cup Y}$$

C_A= The coefficient of areal correspondence where $X \cap Y$ equals the area of the intersect of two polygons and
$X \cup Y$ = the area of the union of two polygons

With the exception of areas and perimeters, there are a limited number of empirical metrics available to quantify the differences in the shapes of two asymmetrical polygons. The isoperimetric quotient (IQ) of a polygon provides one method to quantify a polygon's perimeter with respect to the polygon's area and is defined as the ratio of the polygon area to the area of a circle with same perimeter as the polygon where a perfect circle has an IQ of 1 (Equation 6;Osserman 1978). The isoperimetric quotient was used as an inverse indicator of two catchments relative 'sinuosity', or the amount of curves and bends formed by the line defining each polygon's perimeter. In other words, if the ITSMHydro catchments, and the raster-based catchments have a similar shape, and the isoperimetric quotient values were different, the catchment with the lower isoperimetric quotient value had more perimeter for the area it encloses.

Equation 6: The formula for calculating the isoperimetric quotient as an indicator of the sinuosity of the catchment polygon.

$$IQ = \frac{(4*\pi*\text{area})}{(\text{perimeter}^2)}$$

Section 5: Results

This section presents maps showing the resulting catchment delineations for four test areas and the results of the comparative metrics used to identify differences between the raster-based catchment delineation method and the ITSMHydro vector-based catchment delineation. The test areas were selected represent areas which required the least amount of surface reconditioning and small enough in area to be processed by the ITSMHydro toolset.

Subsection 5.1 details the results of RSME analysis for eight different raster surfaces generated from LiDAR bare-earth sample points. Subsection 5.2: Maps showing four different catchment delineations from natural -neighbors-interpolated raster DTMs overlaid with four ITSMHydro interpolations. Subsection 5.2.1: A number of comparative metrics are presented, including differences in catchment areas, differences in catchment perimeters, percent difference in areas, the coefficient of correspondence, and the isoperimetric quotient for each test area.

5.1 Raster Catchment Delineation and RMSE Analysis Results

Table 4 lists the results of the RMSE analysis for eight raster surfaces generated from LiDAR bare earth sample points using different pixel resolutions and interpolation methods and for a TIN derived from the LiDAR bare-earth sample points.

Table 4 also lists the RMSE value for a USGS 7.5-minute ten-meter DEM to highlight the accuracy gain of the LiDAR collection process over traditional stereophotogrammetric methods for surface model generation. The RMSE value represents the average difference between the surface models elevation values at surveyed elevation points. As shown in Table 4, the RMSE for all surface models generated using natural-neighbors interpolation were slightly lower than the RMSE values for surface models generated using linear interpolation, indicating that natural-neighbors interpolation produces higher quality surface models for these LiDAR data. The highest RMSE value is associated with the USGS ten-meter surface model, indicating that this is the lowest quality surface model.

The LiDAR data has a sample density of approximately 1126 points/1000 ft^2 over the entire study area. The higher RMSE values for coarser pixel surface models are expected since pixel values are subject to LiDAR point averaging during the interpolation process. The LiDAR sample density most closely matches the surface area of the three-feet-pixel raster surface model that returned the lowest RMSE values, indicating that the three-feet-pixel surface model is less subject to error introduce by interpolation. For pixels smaller than three feet, the pixel area is less than the LiDAR sample density, and therefore, each pixel value relies more heavily on the interpolation. The 3-foot pixel (nearest neighbor interpolation) raster surface model had the lowest RMSE and was used to generate catchments to compare with the LiDAR TIN-generated catchments.

Table 4. Surface model Root Mean Square Error values based on pixel sizes and interpolation methods where the RMSE value represents the average distance between known surveyed elevations and the interpolated pixel values. Each row in this table represents a raster surface model generated from bare earth LiDAR sample points. Each raster has either a different pixel size or using a different interpolation method. The RMSE for the USGS 10-meter is based on photogrammetric techniques resulting in a less precise surface model. Notice the RMSE column: for all LiDAR-based surface models the RMSE decreases as the pixel size decreases, up to 1-foot. Additionally, the RMSE for all natural neighbor interpolations are slightly lower than the linearly interpolated surfaces. Also shown ("TIN") is the RMSE of the LiDAR sample points as reported by the LiDAR vendor, TerraPoint.

Surface Model Resolution	Interpolation Method	RMSE (feet)
USGS 10-meter	Unknown	6.583
30-feet	Linear	1.478
30-feet	Natural Neighbor	1.473
6-feet	Linear	1.393
6-feet	Natural Neighbor	1.388
3-feet	Linear	1.393
3-feet	Natural Neighbor	1.387
1-foot	Linear	1.390
0.5-feet	Linear	1.469
TIN	Delaunay Triangulation	0.102

5.2 ITSMHydro Catchment and Raster Catchment Delineation Comparison and Analysis

Figure 25 through 29 show the resulting catchment delineations generated by ITSMHydro compared to a 3 ft. natural neighbor raster DTM catchment delineation. For test area 4, the Onion Creek watershed, the number of LiDAR sample points contained too many points to

process by ITSMHydro (see discussion for more details on processing times and limitations). For test area 4, to highlight issues related to less dense LiDAR point spacing, a number of LiDAR points were randomly selected using a random number generator to approximate the pixel count as a 30-meter pixel raster for the same area; those points were processed through ITSMHydro. For each test area, a number of comparative metrics were calculated to highlight catchment differences resulting from differences in data type (Table 5). For test areas 1, 3, and 4, the ITSMHydro catchments were smaller that the raster catchments, with PD ranging from -83.97% to -3.09%; only test area 2 returned a slightly larger relative percent difference in area (9.39%). The CAC, a measure of overlap between the raster and ITSMHydro catchments (with value of 1.0 indicating complete overlap), ranged from 0.28 to 0.80. Finally, the IQ, a measure of a catchment's area relative to its perimeter (an IQ of a perfect circle is one and a lower IQ value indicates greater boundary complexity), was lower for ITSMHydro catchments in 3 of 4 test areas, with the 4th being virtually identical (Test Area 1); IQ ranged from 0.13 to 0.26 among ITSMHydro catchments and from 0.21 to 0.37 among raster catchments.

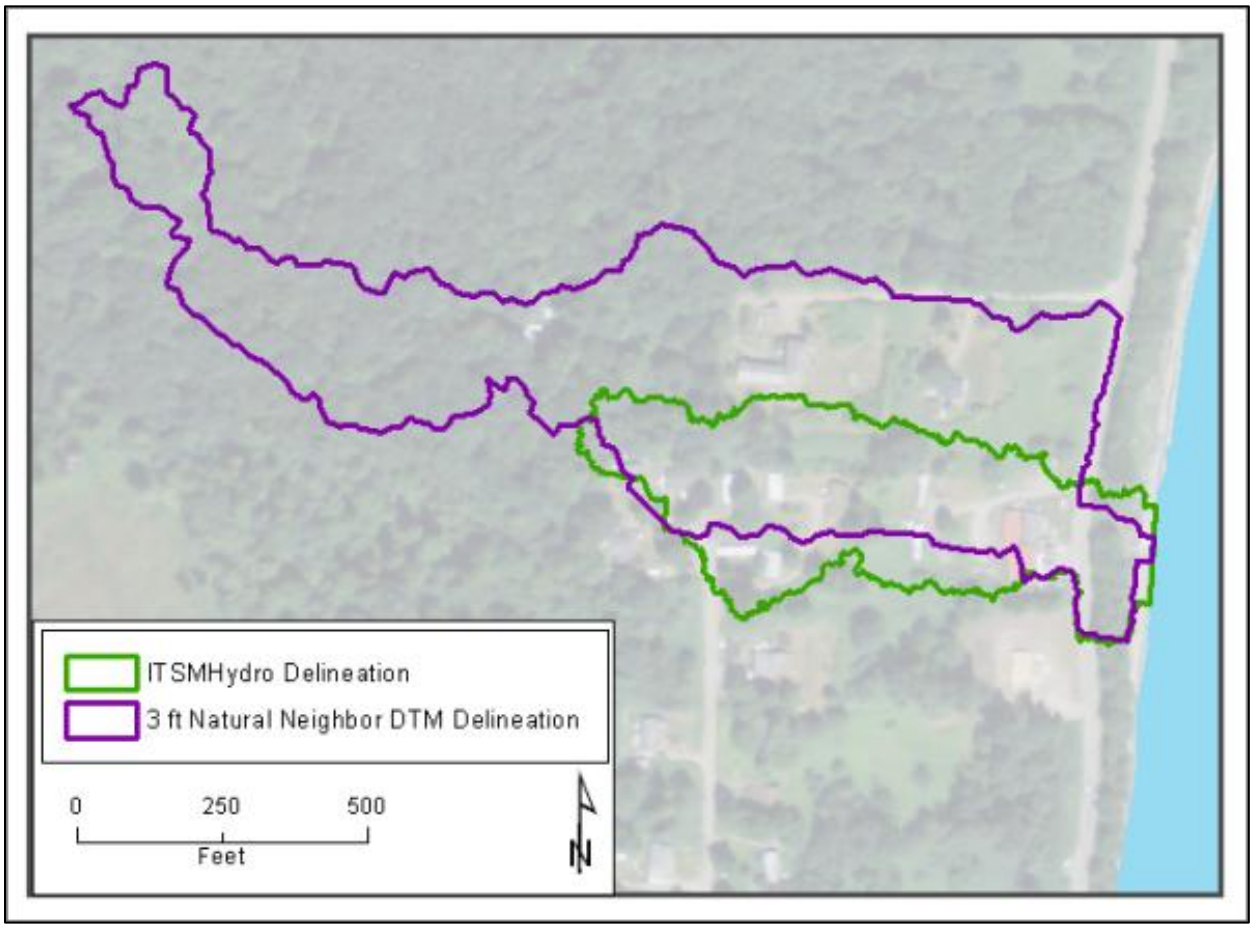

Figure 25: Test area 1 catchment comparison between the ITSMHydro delineation and a 3 ft. NN raster DTM delineation using all available LiDAR points.

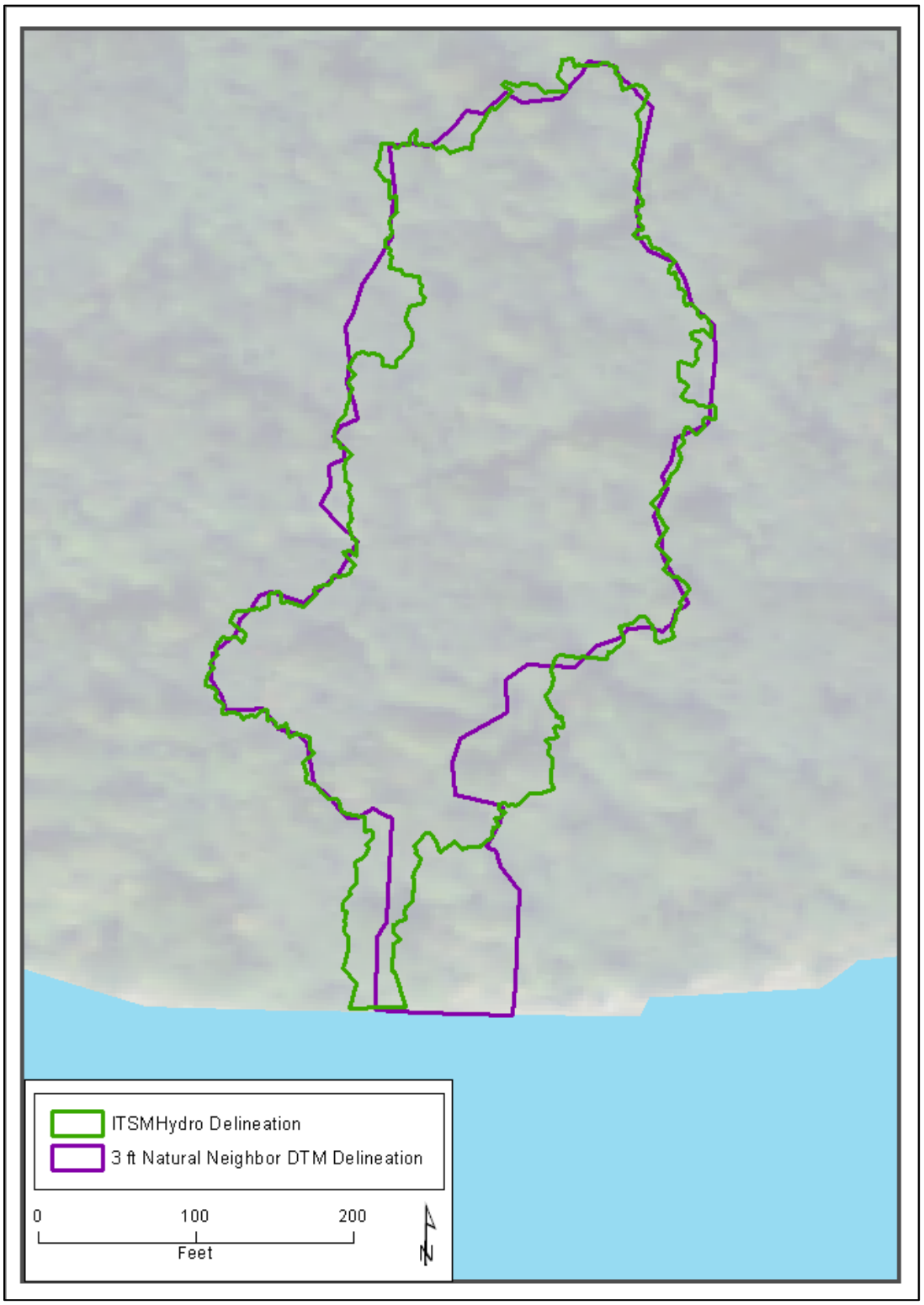

Figure 26: Test area 2 catchment comparison between the ITSMHydro delineation and a 3 ft NN raster DTM delineation using all available LiDAR points.

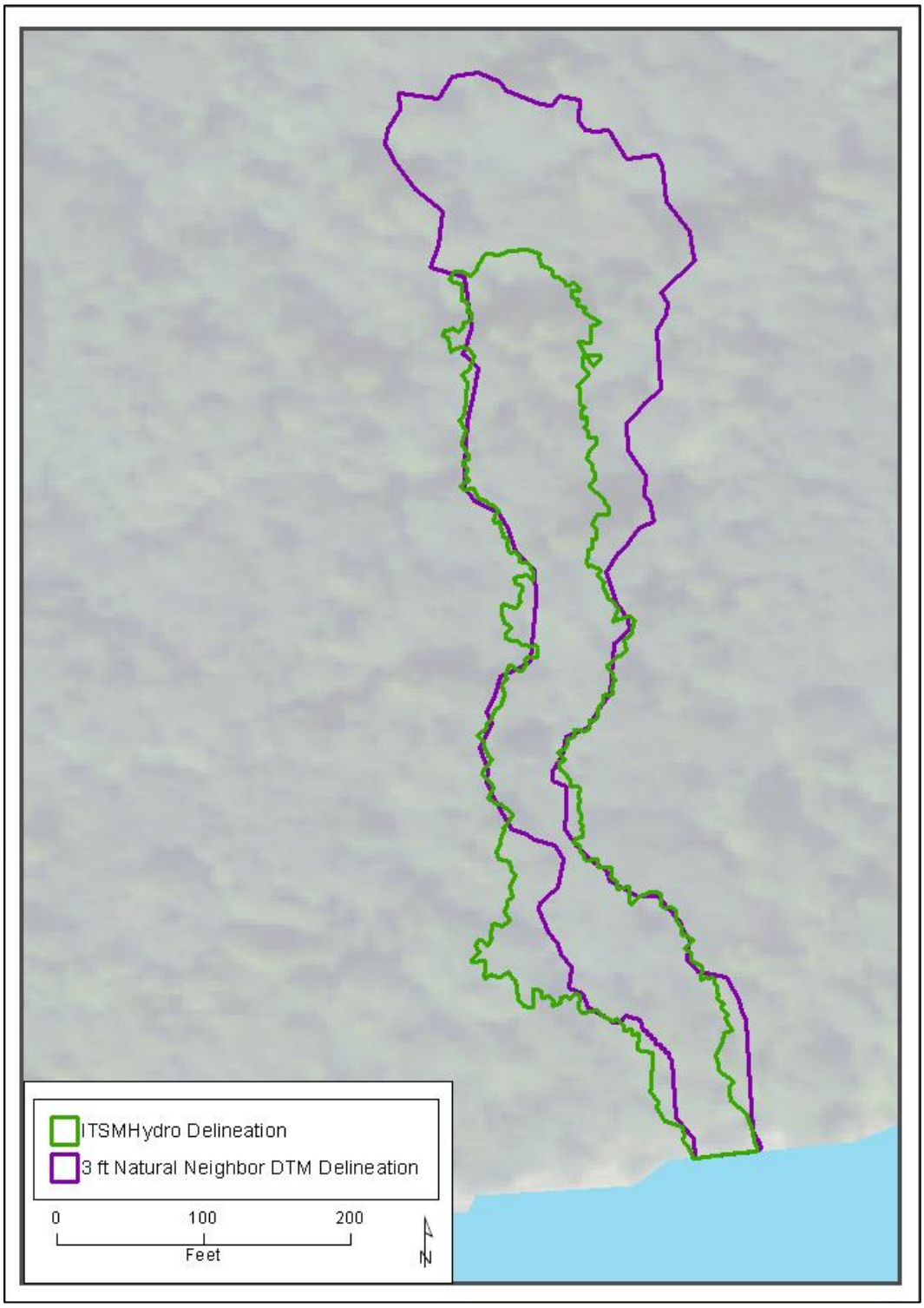

Figure 27: Test area 3 catchment comparison between the ITSMHydro delineation and a 3 ft. NN raster DTM delineation using all available LiDAR.

Figure 28: Test area 4 catchment comparison between the ITSMHydro delineation and a 30 ft NN raster DTM delineation using a LiDAR point appoximatly equal to the pixel density of a 30 ft pixel DTM.

Table 5: Comparative metrics for ITSMHydro and raster catchment delineations. Four different three-feet natural neighbors interpolated raster catchment delineations are compared against four ITSMHydro delineations. For each delineation, the area and perimeter measures of each catchment polygon are shown below with the area values used as inputs to calculate the relative percent difference in area. For test areas 1, 3, and 4, the ITSMHydro catchments were smaller that the raster catchments, with test area 2 returning a slightly larger relative percent difference in area. The CAC values show the ratio of the area of the intersection over the area of the union of both the raster and ITSMHydro catchments. A CAC value of 1.0 would indicate that both catchment delineation methods returned exactly the same catchment polygons, a value of 0.5 would indicate that 50 percent of both polygons could be described as occupying the same space. Therefore, a lower CAC value indicates a greater difference in shared area and a higher CAC value indicates more similarity in shared area. The IQ values for each catchment measure the amount of area relative to the length of its perimeter. A perfect circle would return an IQ value of one and a lower IQ value, when compared to a polygon of similar shape, indicates that the polygon with the lower value has more perimeter for the area it occupies. Only test area 1 returned nearly identical IQ values, with all other ITSMHydro catchments returning lower IQ values than the raster catchments, suggesting that the ITSMHydro catchments formed boundaries that are more complex.

Test Area	Delineation Method	Area (sq. ft.)	Perimeter (feet)	Percent Difference in Area (PD)	Coefficient of Areal Correspondence (CAC)	Isoperimetric Quotient (IQ)
Test Area 1	ITSMHydro	239,381.9	3,716.4	-83.97	0.28	0.22
	3 ft. NN Raster	585,854.5	5,914.2			0.21
Test Area 2	ITSMHydro	104,312.5	2,249.7	9.39	0.80	0.26
	3 ft. NN Raster	94,954.7	1,789.4			0.37
Test Area 3	ITSMHydro	45,923.0	2,122.2	-38.15	0.51	0.13
	3 ft. NN Raster	67,570.5	2,010.9			0.21
Test Area 4	ITSMHydro	34,085,641.5	44,813.2	-3.09	0.79	0.21
	30 ft. NN Raster	35,156,334.1	36,898.7			0.32

Section 6: Discussion

This sections details limitations and sources of error for the ITSMHydro toolset and the implications of those tool limitations on catchment delineations. Subsection 6.1 discusses the implications of the RMSE analysis for raster surface creation for different interpolation methods and pixel size. Subsection 6.2 discusses the percent difference in area, coefficient of aerial correspondence, and the isoperimetric quotient values calculations. Subsection 6.3 discusses the processing time differences between the raster-based catchment delineations and the ITSMHydro catchment delineations. Subsection 6.4 explains the necessity of the bounding polygon prior to ITSMHydro code execution and the errors that can result from a poorly defined bonding polygon. Subsection 6.5 discusses how inconsistencies between irregular tessellations (TIN and Voronoi diagrams) can result in flow direction lines that cross or intersect catchment boundaries. Subsection 6.6 discusses the problems associated with model verification.

6.1 RMSE Analysis of Interpolated Surfaces

Table 4 shows the RMSE values for a number of raster surface models generated from the LiDAR data and is consistent with the literature in that raster surface models are affected by a number of different factors including pixel size and interpolation methods (Garbrecht and Martz 2000); (Garbrecht; et al. 2001); (Haile and Rientjes 2005); (Jones 2002) . The USGS 10-meter surface model is the product of aerial photogrammetry and human interpretation, and the RMSE for this surface model showed the greatest difference between the pixel value and the surveyed elevation data; it is, therefore, the lowest quality surface model when compared against other raster surface models generated from LiDAR. Raster surfaces interpolated from LiDAR showed a

substantial drop in RMSE relative to the 10-m surface model, indicating that the pixel values at all survey elevation locations were more similar, supporting Garbrecht and Martz (2000) in that the LiDAR data collection process results in higher quality surface data. For those surfaces generated from LiDAR, as the pixel size decreased and approached the point density of the LiDAR data, the RMSE decreased. This decrease in the RMSE as the pixel size decreased also supports the literature by demonstrating that more course pixel surfaces are subject to rounding errors (Garbrecht and Martz 2000). Furthermore, differences in RMSE between similar pixel sizes, but different interpolation methods, indicates that the natural neighbors interpolation method produces a higher quality surface model for these data.

The three feet pixel surface models more closely matched the sample density of the LiDAR data. Two surfaces were interpolated with pixel sizes smaller that the sample density of the LiDAR data: a 1-foot pixel surface and a 0.5-foot pixel surface. For both the 1-foot and the 0.5-foot pixel surfaces, the RMSE increased higher than the three feet natural neighbors' surface, suggesting that the interpolation introduced error and degraded surface quality.

6.2 Raster – Vector Delineation Comparisons for Shape and Area

ITSMHydro returned catchments that expressed variations in catchment areas, the extent of those areas as measured by the CAC, and differences in IQ when measured against raster delineation for the same area. Because those vector-based differences in delineation area and delineation extent were neither consistently over nor under the areas and extent of those of the raster-based approach suggests that the process of interpolation has an unpredictable effect on the

quality of catchment delineation. Three test areas returned ITSMHydro isoperimetric quotient values lower than the raster-based approach, suggesting that the raster-based approach has the effect of smoothing the catchment boundary. This smoothing of the catchment boundary is expected in a raster surface model since the pixel value is a generalize elevation value over the pixel area as defined by the regular tessellation of the raster, and not defined by the discrete sample point values. Test area one returned an ITSMHydro isoperimetric quotient slightly higher than the corresponding raster catchment, but this may be due to the considerable difference in catchment areas or because LiDAR sample points below the mean higher high water line were removed from the analysis to reduce file sizes and speed processing times resulting in poorly defined boundaries along the shoreline.

6.3 ITSMHydro Processing Times and File Size Limitations

Processing times for ITSMHydro were significantly longer that the processing times of the raster-based approach. The three-feet pixel raster surface model contained approximately 111 million pixels, and it was possible to fill sinks, create flow direction lines, and generate catchment delineations for the entire surface area in approximately 36 hours of computer processing time. ITSMHydro took approximately 30 hours to process a 120,000 node TIN. As the number of LiDAR points increases, the demands on the computer's processor and available memory increases, thereby increasing the processing times as detailed in Table 6. Graphs showing the processing time for each tool.

Table 6: Number and type of feature geometry iterations required by each ITSMHydro tool. The ITSMHydro tools rely on iterations of the point X, Y, Z data stored in arrays, where an iteration is defined as a computational 'visit' to each member of the array, and a comparison of those values to line geometries stored in different arrays. Increasing the number of LiDAR points in the data set increases the demand on the computer's processor and available memory, thereby increasing processing times. For each LiDAR point, the create Flow Direction Lines tool requires a visit to the LiDAR point, and a visit to every edge line to evaluate whether that line intersects that point. The Create Basin Boundary tool, and the Aggregate Sinks tool also utilize iterations of the data, but also use recursive functions resulting in two or more iterations occurring simultaneously, thus compounding the demands on the computer's processor and memory. Furthermore, Aggregate Sinks needs to execute x number of times where x is the total number of sinks bound within sinks. Column 2 in the table below show the number of iterations required for each tool and column 3 summarizes the iteration type. Since Create Flow Direction Lines only require simple iterations of the data, this tools executes faster than both the Create Basin Boundary, which is executing multiple iterations simultaneously, or Aggregate Sinks, which is executing multiple iteration simultaneously and repeatedly until all sinks have been assigned to a catchment.

ITSMHydro Tool	Iteration Count	Iteration Type
Create Flow Direction Lines	node count * edge count	Iterator
Create Basin Boundaries	(pour point count * flow direction line count) + (branch count flow direction line count)	Iterator and Recursion
Aggregate Sinks	((sink count * edge count) + branch count * edge count))* maximum number of nested sinks	Looped Iterator and Recursion.

The ITSMHydro tools utilize iterators to evaluate connections between LiDAR points (nodes) and edge line or flow direction lines. Point and line X, Y, and Z values are stored in matrices within the computer's memory and each point or line is 'visited and evaluated' for connectivity using iteration of the features geometry stored in arrays. Each LiDAR point (node) evaluated by the Create Flow Directions tool requires an iteration of every edge, to determine if that edge intersects that node.

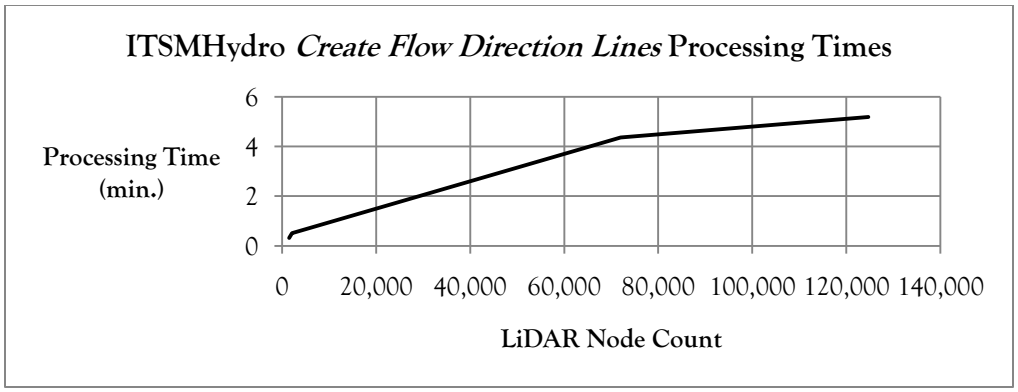

Figure 29: Processing time for the Create Flow Direction tool. This graph shows the processing time required to generate flow direction lines from a TIN with x number of nodes on a 2.93 Ghz Duo Core processor with 4 GB of RAM. Because this tool relies on the iteration of two arrays, the processing time is largely linear with respect to node count. The initial curve from 0 – 4.5 minutes on the y-axis likely results from the time required to initialize the Python interpreter, load required code libraries, and establish the connection to the ArcGIS geoprocessor.

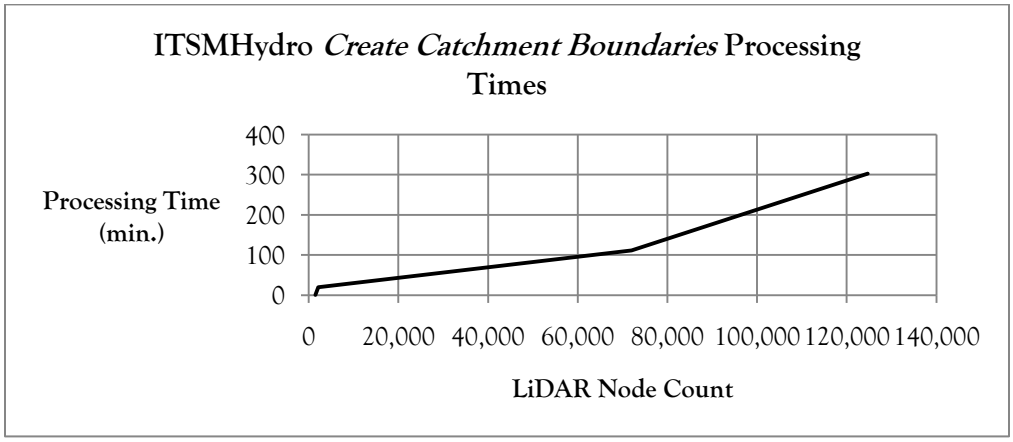

Figure 30: Processing time for Create Basin Boundaries. This graph shows the processing time required to generate catchment boundaries for a TIN with x number of nodes on a 2.93 Ghz Duo Core processor with 4 GB of RAM. The time required to process a 125,000 node TIN is about 5 hours, demonstrating the computational burden of this type of recursive iteration. Processing TINs greater than 125,000 nodes resulted in out-of memory errors; it is likely the line of this graph would more clearly define an exponential function due to the added processor and memory burdens of additional points.

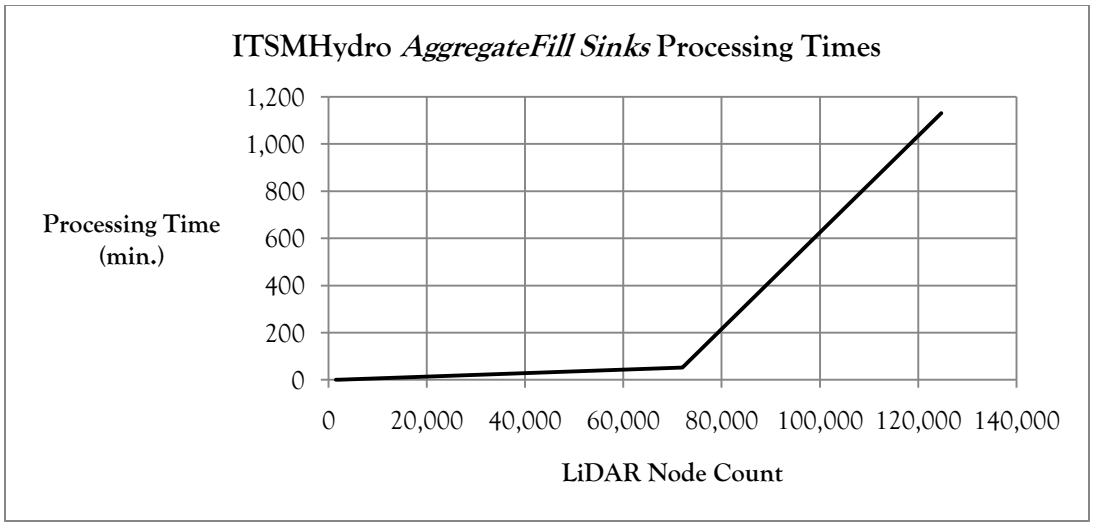

Figure 31. Processing times for Aggregate Sinks. This tool requires the most processing time, requiring almost 20 hours to aggregate sink catchments generated from a 125,000 node TINs on a 2.93 Ghz Duo Core processor with 4 gb of RAM. Processing times for this tool are substantially longer than the Create Catchments tool, requiring almost 20 hours, demonstrating the computational burden of this type of looped recursive iteration. Processing TINs greater than 125,000 nodes resulted in out-of-memory errors. It is likely the line of this graph would more clearly define an exponential function due to the added processor and memory burdens of additional points. Presumably, processing time is related to topographic relief where a LiDAR data set from a mountainous area would produce catchments with less sink areas, requiring fewer loops and thus speeding processing times.

For the Create Basin Boundary tool, each pour point requires an iteration of each flow direction line to determine if that flow direction line connects to that pour point. If a line is found to connect to the pour point, that line is evaluated and any flow direction line that connects to that line is evaluated until every line connected to the pour point is identified. Since this type of tree-spanning algorithm requires recursion, functions that call themselves, and since these recursive iterations of the arrays occur simultaneously in RAM, the Create Basin Boundary tool demands a significant amount of processing resources.

The Aggregate Sinks tool functions similarly to the Create Basin Boundary tool in that it also relies on recursive iterations to define those sinks that share hydrologic connectivity. Create Basin Boundary is a further burden on the computer processor since this tools needs to execute multiple times (loop) until all sinks, or sinks within sinks, are assigned and aggregated into single catchments. Because of these computational demands, ITSMHydro is limited in the number of LiDAR nodes it is able to process. A 150,000 node TIN surface failed to process on the Create Basin Boundaries tools and returned an error message that the computer was out of memory. Presumably, improvement in computer processing speeds and RAM will improve the performance of the ITSMHydro tools.

6.4 Bounding Polygon Considerations

The TIN generation process will create triangles on the periphery of the TIN that satisfy the definitions of a Delaunay triangulation but may form erroneous connections between two pour points for two discrete catchments. ITSMHydro exports all edge lines for all tessellations, and edge lines on the periphery of the TIN can extend a significant distance between sample points. **Error! Reference source not found.** shows the triangle edges and nodes outputs from a TIN using ArcGIS. The gray area to the left of the red line shows examples of some triangles and edges that could potentially skew the resulting flow direction calculations. If one of those lines erroneously formed a connecting edge line between two catchment pour points, ITSMHydro would evaluate that line as flow direction line, thereby merging two adjacent catchments and resulting in an over-estimation of catchment area. For example, if the nodes labeled *A* and *B* in Figure 34 are pour points for two distinct catchments, and point *A* was slightly higher in elevation

then *B*, and the line *A-B* is not excluded by the bounding polygon, then ITSMHydro would recognize the line *A-B* as a valid flow direction line away from *A* resulting in a merging of the two catchments formed upslope of pour point A and B.

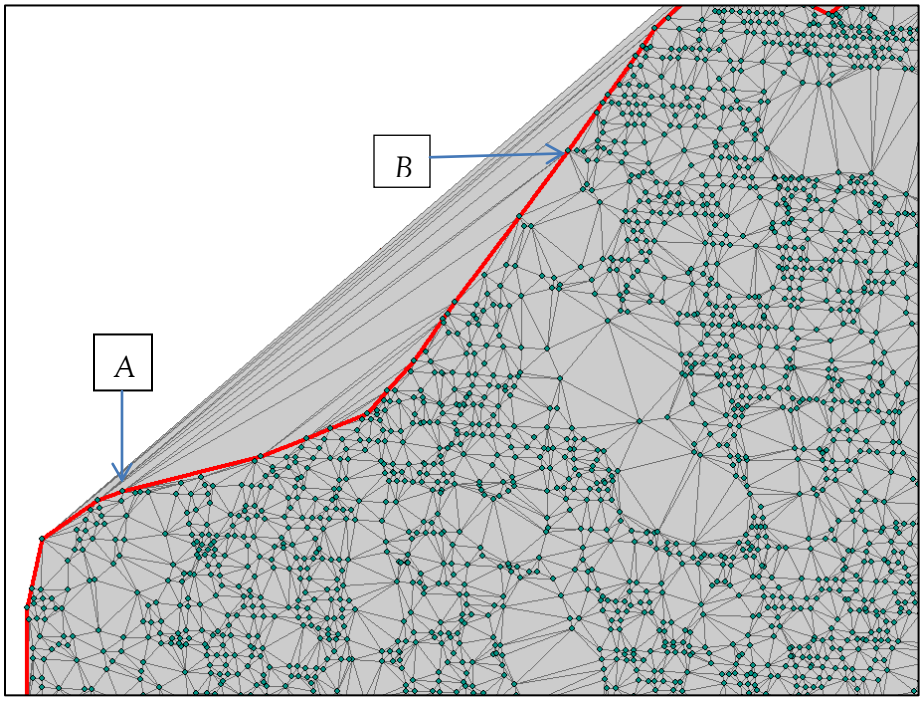

Figure 32: A hypothetical TIN where the edge lines on the periphery of the TIN connect node a substantial distance apart. The bounding polygon (shown in red) must only capture those areas considered for flow direction lines. A line shown above as A – B, if not excluded by a properly defined bounding polygon could define a flow direction line that joined two pour points A and B resulting in the joining of two different catchments.

6.5 Flow Directions Lines and Catchment Delineations

All flow directions lines generated by ITSMHydro are coincident with the triangulation lines. The generation of the catchment boundary relies on the creation of a Voronoi diagram from those nodes that exist on the leaves of the outermost branches of the flow direction network.

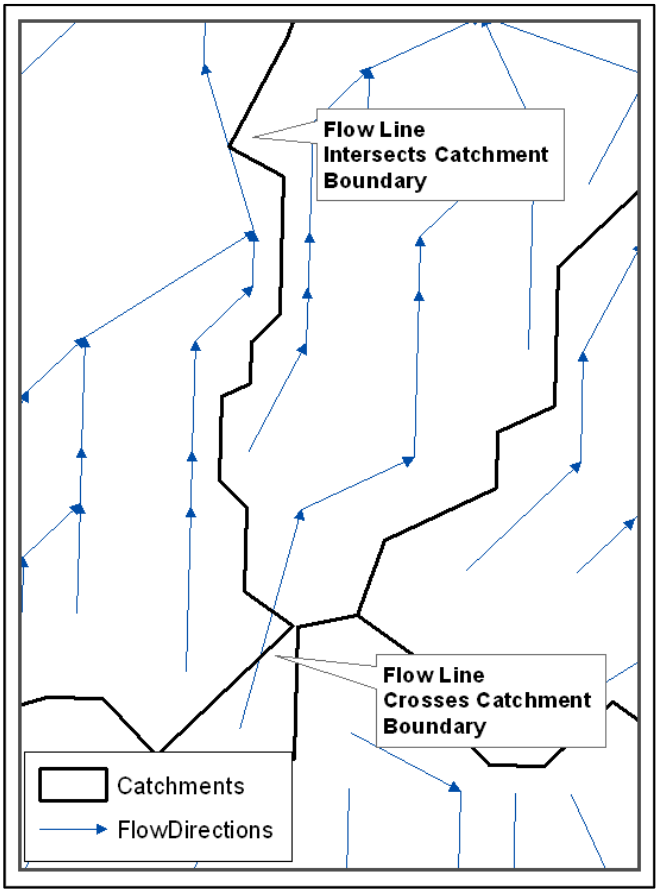

Figure 33: Showing an example of flow direction lines that cross or intersect the catchment boundary.

While the Delaunay triangulation and the Voronoi diagram share a geometric relationship in that the nodes bound by adjacent Voronoi polygons are connected by the TIN lines that connect those nodes, the Voronoi diagram and the TIN are different geometric objects. This reliance on difference geometries will result in flow direction lines across catchment boundary lines, or that are coincident with catchment boundary line segments (Figure 33). As such, either there is a slight error in estimations in catchment areas, or the flow direction lines do not represent the true flow path. Whether the catchment areas are erroneous, or the flow lines are erroneous is based on which premise is accepted, in other words whether the Delaunay triangulation or the TIN is the primary geometry. Accepting one geometry over the other, and

altering the algorithms to adjust the line work, or redeveloping the algorithms to utilize only one geometry type would solve the problem of inconsistent flow lines and catchments.

6.6 Discussion on model validation

ITSMHydro flow directions and catchment outputs were verified using an artificial dataset with a large range of topographical relief and well-defined catchment boundaries prior to execution using actual LiDAR data. ITSMHydro has not been validated beyond a comparison between the catchment delineations generated using industry standard raster processing tools.

True catchment boundaries involve complex interactions among topography, soils, underlying geology, and vegetation cover. Extensive amounts of fieldwork surveying the elevation, soils, geology and vegetation to determine a catchment line for model validation would likely involve subjectivity, sample error, and enough uncertainty to invalidate field-delineation of a catchment boundary for model validation.

Finding some impervious surface, for example a paved parking lot covered by LiDAR data, that resulted in different raster vs. ITSMHydro catchment delineations, then measuring the discharge from that area during a rain event would result in a discharge volume total. That discharge volume may correlate with the discharge volume expected from one of the catchment delineations, but any number of catchment boundaries may return that discharge volume and would not provide conclusive proof that any calculated delineation was a sound delineation. It may be possible to manufacture an artificial surface in a laboratory, find those areas where there are differences in the catchment boundaries and measure which direction water flows to better

determin the catchment in those areas. Repeated iterations of this experiment may provide statistical evidence that one delineation method is superior.

Section 7 Conclusion and Future Work

This work involves a number of algorithms generated in the Python programming language that interface with ESRI's ArcGIS to delineate catchment boundaries from LiDAR bare-earth sample points that are not constrained by the limitations imposed by the raster surface model. LiDAR sample points are transformed into a TIN surface model and the TIN geometry are extracted from the TIN as point, triangle polygon, and polygon line ESRI feature-dataset feature-classes. The node, triangle and line feature classes serve as inputs into three different tools which define flow directions from the lines based on the slopes between LiDAR nodes, define catchment boundaries based on Voronoi polygons around connected flow direction lines, and aggregate areas defined as sinks into final catchment delineations.

Historically, GIS users have transformed surface elevation data into raster-based surface models for hydrologic modeling, including the delineation of flow direction lines, defining catchment boundaries, and managing sink areas within the dataset. The raster-based surface model, and the algorithms used to generate flow directions and catchments, are influenced by a number of factors inherited by the pixel data structure. Pixel values are subject to error by raster-interpolation, filling algorithms result in areas of data loss, and flow directions are constrained by the eight-cardinal directions dictated by the raster cell structure. These data structure constraints can each influence flow directions and catchment delineations resulting in error.

The research presented in this thesis indicates that the LiDAR point data provided by Terrapoint Inc. for the area on and near the Lummi Indian Reservation are of a sufficient sample density and precision to define catchment boundaries using irregular tessellations to define the

spatial adjacency of those points. This research indicates that ESRI TIN surface models represent a higher quality dataset than traditionally-generated raster surface models, consistent with the research of Garbrecht and Martz (2000). Delaunay-interpolated TIN surface models generated from the LiDAR data returned much lower RMSE, proving theTIN data were of a higher quality to any of the raster-interpolated surface models, supporting the literature that raster interpolation introduces more error into the raster dataset (Mark (1984); O'Callaghan and Mark (1984); Mark (1988); Fairfield and Leymarie (1991); and Gehegan and Lee (2000). Furthermore, this research shows that the processing algorithms presented are sufficient for hydrologic modeling when those data are transformed into irregular tessellations as demonstrated by the Voronoi flood modeling work of Dakowicz and Gold (2007).

The algorithms presented produced catchment boundaries that varied in size and shape to a raster-based catchment delineation. The ITSMHydro delineations ranged from substantially smaller to slightly larger than raster delineations in the four different test areas, indicating the surface-model-data-structure can have a dramatic impact on catchment delineations. These algorithms also produced catchment boundaries with generally higher boundary complexity, suggesting that a move away from the D8 flow direction algorithm (Maidment 2000) avoids any generalization of the flow direction surface, thereby producing a more accurate catchment boundary. While the ITSMHydro algorithms executed as planned, a number of issues could be addressed to improve the overall utility of the ITSMHydro vector-based catchment delineation tools, including an automated method of producing a bounding polygon, model validation, and code optimization. The automated generation of the bounding polygon would remove the human error associated with the construction of the bounding polygon and reduce the data preprocessing

time need to prepare the data for ITSMHydro execution, resulting in high-quality delineations. Validating the catchment delineations for both the vector-based and raster-based approach would definitively determine the utility of the ITSMHydro tools. Additionally, there are number of code alterations that, if implemented, could speed ITSMHydro execution times to make ITSMHydo a more viable tool for hydrologic modeling.

ITSMHydro, like the ArcHydro toolset, only considers topography when delineating catchment boundaries. Neither ITSMHydro nor raster-based approaches produce the true catchment, since neither approach considers all the data necessary to capture the true catchment boundary. However, it is relevant to evaluate which method most closely approximates the catchment area. While there is no known method to definitively conclude that the vector-based approach presented herein is superior to a raster-based approach, RMSE values indicate that the TIN model generated from LiDAR points resulted in a remarkably improved surface model over the raster-based surface model in terms of elevation accuracy. The ITSMHydro algorithms were able to produce catchment boundaries for several test areas that demonstrate differences in areas, location, and shape suggesting that the surface model has a significant influence on catchment boundary delineation.

This work contributes to the field of geographic information sciences in a number of different ways. Several original subroutines were developed to perform a number of tasks that were previously unavailable to GIS professionals, including subroutines to collapse the inner rings of donut polygons, reassign line from-to directions based on node z values, write line geometries to feature class attribute tables, identify lines sloping away from points, dissolving single-part line

features that share node connectivity, and recursive spatial selections. Given the lower relative accuracy of raster-based surface models evident in the study area, the observed differences in size, shape, and location between raster- and vector-based catchments suggest that use of the raster-based approach may compromise accuracy in area, shape, and location of the resulting catchments. A vector-based approach that maintains the integrity of the sample data is preferred, especially in areas of low topographic relief with high sample point density . Since the TIN surface model more closely matched surveyed elevation values, ITSMHydro catchments are better suited for producing a more legally defensible catchment boundary and therefore, may affect jurisdictional responsibilities with respect to water rights and other water resource-related issues. The file size constraints limit application of the approach developed herein, however, at least until technological advances and/or code revisions improve computer processing speed and file size capacity. Most importantly, this research advances the growing field of geographic information science by exposing the assumptions of historically accepted practices.

Works Cited

Campbell, J. (2002). Introduction to Remote Sensing. New York, The Guilford Press.

de Berg, M., Cheong, O, van Kreveld, M, Oveermars, M (2008). Computational Geometry: Algorithms and Applications. Santa Clara, CA.

Duke, G. D., Kienzel, S. W., Johnson, D. L., Byrne, J.M. (2003). "Improving overland flow routing by incorporating ancillary road data into Digital Elevation Models." Journal of Spatial Hydrology **3**(2): 26.

ESRI. (2010). "About TIN Surfaces." from http://webhelp.esri.com/arcgiSDEsktop/9.3/index.cfm?TopicName=About_TIN_surfaces.

ESRI. (2010). "File Geodatabase Resolution." from http://webhelp.esri.com/arcgisserver/9.3/java/index.htm#geodatabases/feature_class_basics.htm.

ESRI. (2010). "Geodatabases." from http://webhelp.esri.com/arcgisserver/9.3/java/index.htm#geodatabases/types_of_geodatabases.htm.

Fairfield, J. and P. Leymarie (1991). "Drainage networks from grid digital elevation models." Water Resour. Res. **27**(5): 709-717.

Garbrecht, J. and L. W. Martz (2000). Digital Elevation Model Issues In Water Resources Modeling.

Garbrecht, J., F. L. Ogden, et al. (2001). "GIS and Distributed Watershed Models. I: Data Coverages and Sources." Journal of Hydrologic Engineering **6**(6): 506-514.

Gold, C. M., Remmele, P. R., Roos, T. (1989). Spatial adjacency - a general approach. Auto-Carto 9, Baltimore, Md, USA.

Gold, C. M., Remmele, P. R., Roos, T. (1997). Voronoi Methods in GIS. Algorithmic Foundations of Geographic Information Systems. Berlin/Heidelberg, Springer. **1340/1997:** 21 - 35.

Haile, A. T., Rientjes, T. H. M. (2005). Effects of LIDAR DEM Resolution in Flood Modeling: A Model Sensitivity Study for the City of Tegucigalpa, Honduras. ISPRS WG III. Enschede, The Netherlands.

Jones, R. (2002). "Algorithms for using a DEM for mapping catchment areas of stream sediment samples." Computers & Geosciences **28**(9): 1051-1060.

Maidment, D. (2002). ArcHydro, GIS for Water Resources. Redlands, Ca, ESRI Press.

Maidment, D. R. (2002). Arc Hydro, GIS for Water Resources. Redlands, ESRI Press.

Maidment, D. R., Djokic, D. (2000). Hydrologic and Hydraulic Modeling Support with Geographic Information Systems. Redlands, ESRI Press.

Mark, D. M. (1984). "Automatic detection of drainage networks from digital elevation models." Cartographica **21**(2/3): 168-178.

Mark, D. M. (1988). Modeling Geomorphologic Systems. Chichester, John Wiley & Sons.

O'Callaghan, J. F. and D. M. Mark (1984). "The extraction of drainage networks from digital elevation data." Computer Vision, Graphics, and Image Processing **28**(3): 323-344.

Osserman, R. (1978). "The Isoperimetric Inequality." Bulliten of the Mathematical Society **84**: 1182 - 1238.

Peucker, T. K., R. J. Fowler, et al. (2002). Digital Representation of Three-Dimensional Surfaces by Triangulated Irregular Networks (TIN). REVISED.

Peucker, T. K., Fowler, Robert J., Little, James J., Mark, David M. (1977). The Triangulated Irregular Network. D. o. Geography. Burnaby, Canada, Simon Fraser University.

Sack, R., Urruita, J (2000). Handbook of Computational Geometry. Amsterdam, Elsevier.

Taylor, p. (1977). Quantitative Methods in Geography. Tyne, United Kingdom, Waveland Press, Inc.

Tobler, W. (1970). "A computer move simulating urban growth in the Detroit region." Economic Geography **46**(2): 234 - 240.

Tucker, G. E., S. T. Lancaster, et al. (2001). "An object-oriented framework for distributed hydrologic and geomorphic modeling using triangulated irregular networks." Computers & Geosciences **27**(8): 959-973.

Wehr, A. L. U. (1999). "Airborne laser scanning-an introduction and overview." ISPRS Journal of Photogammetry and Remote Sensing **54**: 68-82.

Wu, S., Li, J., Huang, G. (2008). "Characterization and Evaluation of Elevation Data Uncertainty in Water Resource Modeling." Water Resources Management: 959-972.

Appendix A- 0_LoadDataFromTINToGeoDataBase.py

```
try:
    print "Load data from TIN To ESRI File Geodatabase for ITSMHydro"
    import sys, arcgisscripting
    gp = arcgisscripting.create()
    gp.CheckOutExtension("3D")
    gp.workspace = TheWorkingDirectory
    gp.RefreshCatalog(TheWorkingDirectory)

    ######    User Defined Variables    ##########
    TheWorkingDirectory = r"E:\Thesis\ITSMHydro2011CorrectNames\TestData\CanyonLake.gdb\ITSMHydro"
    TIN = r"E:\Thesis\ITSMHydro2011CorrectNames\TestData\testtin"
    gp.overwriteoutput = 1
    gp.ZResolution = "0.01"
    gp.XYResolution = ".01"
    ################################################

    print "Export TIN edges."
    Edges = "Edges"
    gp.TinEdge_3d(TIN, Edges, "DATA")
    print "Export TIN nodes."
    Nodes = "Nodes"
    gp.TinNode_3d(TIN, Nodes, "", "Tag_Value")
    print "Export TIN triangles."
    Triangles = "Triangles"
    gp.TinTriangle_3d(TIN, Triangles, "PERCENT", "1", "", "")
    print "\nFinished loading TIN Data."
except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
except Exception, ErrorDesc:
    print ErrorDesc.message
    print "Error with python"
    sys.exit()
```

Appendix B- *1_FlowDirectionsFromTIN.py*

```
#This script will take a scattered distribution of point and identify those
#lines that represent surface water flow paths based on a Delaunay Triangulation
#generated from those points.
#For each record in the feature class called node, those lines that intersect that node are identified.
#The line distance is calculated using the distance formula and a slope value
#is generated.  For each node, the intersecting lines are sorted based on slope, the line with the steepest
#slope is writen to a new feature class.  If more that one line have the same steepest slope, that line
#in the last sort position is returned.

#User Defined Variables    ###################################################
#The path to the feature dataset in a geodatabase
TheWorkingDirectory = r"E:\Thesis\ITSMHydro2011CorrectNames\TestData\CanyonLake.gdb\ITSMHydro"
#A path and name.txt of a text file, user must have write permission to this directory.
textfile = r"C:\Temp\ITSMHydroflowdirectionfromTIN.txt"
#1 to overwrite all geoprocessing outputs, 0 to not overwrite.
OverWriteOutput = 1
#The resolution of the origial point data
XYResolution = r".01"
# ###############################################################################

print "Create Flow Direction lines from a TIN"
print"Created by Gerry Gabrisch. \nAugust 2009  gerry@gabrisch.us\n"
import sys, string, os, arcgisscripting, math, operator, exceptions, shutil, time
def GetTime(t):
    #Convert time to a readable format.
    theyear = t[0]
    themonth = t[1]
    theday = t[2]
    thehour = t[3]
    theminutes = t[4]
    theseconds = t[5]
    thedate = str(themonth) + r"/" + str(theday) + r"/" + str(theyear)
    starttime = str(thehour) + ":" + str(theminutes) + ":"+str(theseconds)
    return thedate +", "+ starttime
```

```
def EndTime(starttime):
    #Calculate the processing time and
    #print the processing time to the screen.
    endtime = time.time()
    t = time.localtime(endtime)
    print "Finished at " + GetTime(t)
    totalseconds = endtime - starttime
    hours1 = totalseconds/3600
    hours = int(hours1)
    minutes2 = hours1 - hours
    minutes1 = minutes2*60
    minutes = int(minutes1)
    seconds = int((minutes1-minutes)*60)
    print "Time to Process; "+str(hours) +": "+str(minutes) +": "+str(seconds)
    print GetTime(t)
try:
    starttime = time.time()
    t = time.localtime(starttime)
    print "Start Time = " + GetTime(t)
    #Create Geoprocessing object, set workspace, set extensions, and purge schema locks.
    gp = arcgisscripting.create()
    gp.workspace = TheWorkingDirectory
    gp.RefreshCatalog(TheWorkingDirectory)
    gp.CheckOutExtension("3D")
    gp.OverWriteOutput = OverWriteOutput
    gp.XYResolution = XYResolution
    #Required Variables for this script.
    Edges = "Edges"
    BoundingPolygon = "BoundingPolygon"
    BoundingPolygonLine = "BoundingPolygonLine"
    FlowDirectionLines = "FlowDirections"
    Edges_Layer = "Edges_Layer"
    BoundingPolygonLine_Layer = "BoundingPolygonLine_Layer"
    #The geoprocessor will not overwrite a text file. If it exists, delete the existing copy.
    if os.path.exists(textfile):
        os.remove(textfile)
```

```
    print "Purge lines that intersect the bounding polygon of the TIN."
    #Convert the bounding polygon to a line feature class to facilitate the
    #selections that remove edge lines on the periphery of the TIN.  Select any
    #lines that intersect the bounding polygon and delete them.
    gp.FeatureToLine_management(BoundingPolygon, BoundingPolygonLine, "", "ATTRIBUTES")
    gp.MakeFeatureLayer_management(Edges, Edges_Layer, "", "", "Index Index VISIBLE NONE;EdgeType EdgeType VISIBLE NONE;Shape_Length Shape_Length
VISIBLE NONE")
    gp.MakeFeatureLayer_management(BoundingPolygonLine, BoundingPolygonLine_Layer, "", "", "")
    gp.SelectLayerByLocation_management(Edges_Layer, "INTERSECT", BoundingPolygonLine_Layer, "", "NEW_SELECTION")
    gp.DeleteFeatures_management(Edges_Layer)
    #Create a text file that will store the feature geometry of the
    #flow direction lines.
    print "Create text file."
    f = open(textfile,'a')
    thestring = "polyline\n"
    f.writelines(thestring)
    f.close()
    #Start a search cursor to get the feature geometry of the
    #edge lines that intersect the nodes.
    print "Start Search Cursor."
    TheObjectID = 1
    desc = gp.Describe(Edges_Layer)
    shapefieldname = desc.ShapeFieldName
    rows2 = gp.SearchCursor(Edges_Layer)
    row2 = rows2.Next()
    superlist = []
    print "Reading feature geometry."
    while row2:
        feat = row2.GetValue(shapefieldname)
        thefeature = row2.getvalue(desc.OIDFieldName)
        partnum = 0
        partcount = feat.PartCount
        #Templist store the line geometry for each node.
        templist = []
        while partnum < partcount:
            part = feat.GetPart(partnum)
            pnt = part.Next()
```

```
pntcount = 0
#Enter while loop for each edge, get the first x, y, z returned for each feature.
while pnt:
    z = round(pnt.z,2)
    x = round(pnt.x,2)
    y = round(pnt.y,2)
    #If the list is empty, write xyz to the templist.
    if templist == []:
        templist=[x, y, z]
    #If the list is not empty, then this is the second node in the
    #line.  Evaluate the z values of the two nodes
    #and write them to a list with the higher elevation first.
    #Also, create a unique ID consisting of a string of the from
    #coordinate values of each lines start and end nodes.
    else:
        #For cases where the first node returned is the lower end of the line do this.
        if z < templist[2]:
            x1 = str(templist[0])
            y1 = str(templist[1])
            IDKey = x1 + y1
            tempsuperlist = [IDKey, templist[0], templist[1], templist[2], x, y, z]
            #Calculate the line length using the distance formula.
            therun = math.pow(((math.pow((tempsuperlist[1]-tempsuperlist[4]),2)) + (math.pow((tempsuperlist[2] - tempsuperlist[5]),2))),.5)
            #Calculate the rise by subtracting the z values of the two line end nodes.
            therise = tempsuperlist[3]- tempsuperlist[6]
            #Convert the rise to the percent slope.
            percentslope = abs(therise/therun *100)
            tempsuperlist.append(percentslope)
            #Add the results of this line to the 'super list of all line IDs, coordinates, and slopes.
            superlist.append(tempsuperlist)
            #Reset templist to an empty list.
            templist = []
        #For cases where the first node returned is the higher end of the line do this.
        else:
            x1 = str(x)
            y1 = str(y)
            IDKey = x1+y1
```

```
                tempsuperlist = [IDKey, x, y, z, templist[0], templist[1], templist[2]]
                #Calculate the line length using the distance formula.'"
                therun = math.pow(((math.pow((tempsuperlist[1]-tempsuperlist[4]),2)) + (math.pow((tempsuperlist[2] - tempsuperlist[5]),2))),.5)
                #Calculate the rise by subtracting the z values of the two line end nodes.
                therise = tempsuperlist[3]- tempsuperlist[6]
                #Convert the rise to the percent slope.
                percentslope = abs(therise/therun *100)
                tempsuperlist.append(percentslope)
                #Add the results of this line to the 'super list of all line IDs, coordinates, and slopes.
                superlist.append(tempsuperlist)
                #Reset templist to an empty list.
                templist = []
            pnt = part.Next()
            pntcount += 1
            if not pnt:
                pnt = part.Next()
        partnum += 1
    #Add one to the TheObjectID.
    TheObjectID += 1
    row2 = rows2.Next()
print "Total Lines Processed = "+ str(TheObjectID)
#Sort (ascending)the list items based on the values stored in [0](the from id)
superlist = sorted(superlist, key=operator.itemgetter(0))
linesfromanode =[]
TheObjectID = 0
print "Finding steepest path out from each TIN node."
#Superlist[] now contains all lines grouped by the 'from' coordinates. Get all records with the same ObjectID,
#write them to a new list, and sort them by slope so that the greatest slope as the last record in the list.
for item in superlist:
    #If the list is empty, grab the current item and add it to a new list.
    if linesfromanode == []:
        linesfromanode.append(item)
    #If the list is not empty, and the index of this item is equal to the index of the item currently
    #in the list, add it to the list.
    else:
        if linesfromanode[-1][0] == item[0]:
            linesfromanode.append(item)
```

```
            #There are no more lines with the same index, so process the data and identify the
            #line with the steepest path from the node.
            else:
                #Sort the values by the slope, the last item in the list.
                linesfromanode = sorted(linesfromanode, key=operator.itemgetter(7))
                #Now the last item in the list hase the greatest slope and is therefore, the line out.
                lineout = linesfromanode.pop()
                #Write the coordinates and elevations (from-to) to a string formated for ArcGIS.
                thestring = str(TheObjectID) + " 0\n" + "0 "+ str(lineout[1])+" "+str(lineout[2]) +" " + str(lineout[3]) + "\n" + "1 " + str(lineout[4]) + " " + str(lineout[5]) + " " +
str(lineout[6])+"\n"
                f = open(textfile,'a')
                f.writelines(thestring)
                f.close()
                TheObjectID += 1
                #All finished with this node, now clear out the list of lines from this node.
                linesfromanode = []
                #The item returned is not part of the same node, use it to evaluate the lines out of this node.
                linesfromanode.append(item)
    #All lines have been processed and writen to a text file. Finish formatting the text file.
    f = open(textfile,'a')
    thestring = "END"
    f.writelines(thestring)
    f.close()
    #Read the text file of geometry and construct a new feature class
    #that represents the flow direction lines from each node.
    print "Building geometry."
    inSep = "."
   #Convert the text file to a shapefile.
    gp.CreateFeaturesFromTextFile_samples(textfile, inSep, FlowDirectionLines, "#")
    gp.RefreshCatalog(TheWorkingDirectory)
    #Delete variables.
    try:
        #Delete the geoprocessor.
        del gp
    except:
        pass
    EndTime(starttime)
```

```
    x = raw_input("Finished, press enter to quit")
    sys.exit(0)
except arcgisscripting.ExecuteError:
    print "ArcGIS error in FlowDirectionsFromTIN.py."
    print gp.GetMessages(2)
    sys.exit()
except Exception, ErrorDesc:
    print ErrorDesc.message
    print "Python error in FlowDirectionsFromTIN.py."
    sys.exit()
```

Appendix C- *2_CreateCatchmentPolygons.py*

```
print "Assign sink polygons to catchments "
########################################################################
#UserDefinedInputData
#The path to the geodatabase feature dataset
TheWorkingDirectory = r"C:\Temp\ITSMHydro2011CorrectNames\TestData\CanyonLake.gdb\ITSMHydro"
#A textfile path and name (must be a read\write space.)
textfile =  r"C:\Temp\GBGTempTextFileForReassignLineDirections.txt"
#Set to 1 to delete the temp feature classes.
overwriteoutput = 1
deletetempdata = 0
########################################################################
import sys, string, os, arcgisscripting, math, operator, exceptions, shutil, time

def GetTime(t):
    #Convert time to a readable format and to time code execution.
    theyear = t[0]
    themonth = t[1]
    theday = t[2]
    thehour = t[3]
    theminutes = t[4]
    theseconds = t[5]
    thedate = str(themonth) + r"/" + str(theday) + r"/" + str(theyear)
    starttime = str(thehour) + ":" + str(theminutes) + ":"+str(theseconds)
    return thedate +", "+ starttime

def EndProgram(starttime, deletetempdata):
    #Quit the program, delete temporary data file if requested, and print code execution time.
    if deletetempdata == 1:
        print "Delete temp data"
        EdgesAccrossCatchmentBoundaries = "EdgesAccrossCatchmentBoundaries"
        FlowLinesAcrossCatchmentsWithFlowDirections ="FlowLinesAcrossCatchmentsWithFlowDirections"
        SpatialJoinOutput = "SpatialJoinOutput"
        FeatureVerticiesToPoints1 = "FeatureVerticiesToPoints1"
        FlowLinesAcrossCatchmentsWithFlowDirectionsedited = "FlowLinesAcrossCatchmentsWithFlowDirectionsedited"
        gp.Delete_management(EdgesAccrossCatchmentBoundaries)
        gp.Delete_management(FlowLinesAcrossCatchmentsWithFlowDirections)
```

```
        gp.Delete_management(SpatialJoinOutput)
        gp.Delete_management(FeatureVerticiesToPoints1)
        gp.Delete_management(FlowLinesAcrossCatchmentsWithFlowDirectionsedited)
    endtime = time.time()
    t = time.localtime(endtime)
    print "Finished at " + GetTime(t)
    totalseconds = endtime - starttime
    hours1 = totalseconds/3600
    hours = int(hours1)
    minutes2 = hours1 - hours
    minutes1 = minutes2*60
    minutes = int(minutes1)
    seconds = int((minutes1-minutes)*60)
    print "\n\nTime to Process; "+str(hours) +": "+str(minutes) +": "+str(seconds)
    x = raw_input("Finished! Press enter to quit")
    sys.exit(0)

def FillDonut(inputfeatureclass):
    #Catchment delineations can have other polygons bound within them. Delete any
    #verticies of a polygon the represent the interior portion of the feature
    #by identifing interior nodes and delete them using a feature geometry array.
    #The bounding polygon is no longer an annulus but a different record exists
    #that represents the area covered by the inner hole.  Find those remaining
    #features and delete them. The resulting feature class represents an irregular
    #tesselation of polygons with no polygons bound within any other polygon.

    try:
        print "\nCall FillDonut()"
        desc = gp.Describe(inputfeatureclass)
        shapefield = desc.ShapeFieldName
        rows = gp.UpdateCursor(inputfeatureclass)
        row = rows.next()
        arrayObj = gp.CreateObject("Array")
        arrayOuter = gp.CreateObject("Array")
        ListOfIDs = []
        while row:
            feat = row.getValue(shapefield)
```

```
            qInterior = False
            for partNum in range(feat.partCount) :
                part = feat.getPart(partNum)
                qInterior = False
                for ptNum in range(part.count):
                    pt = part.next()
                    if pt != None:
                        arrayOuter.add(pt)
                    else :
                        qInterior = True
                        break
                arrayObj.add(arrayOuter)
                arrayOuter.RemoveAll()
            if qInterior :
                row.setValue(shapefield,arrayObj)
                rows.updateRow(row)
                ListOfIDs.append(row.OBJECTID)
            arrayObj.RemoveAll()
            row = rows.next()
        del rows,row
        Catchments_Layer = "Catchments_Layer"
        gp.MakeFeatureLayer_management(inputfeatureclass, Catchments_Layer, "", "", "")
        if ListOfIDs != []:
            print "Filling features bound within other catchments."
            for item in ListOfIDs:
                gp.SelectLayerByAttribute_management(Catchments_Layer, "NEW_SELECTION", "\"OBJECTID\" = " + str(item))
                gp.SelectLayerByLocation_management(Catchments_Layer, "COMPLETELY_WITHIN", Catchments_Layer, "", "NEW_SELECTION")
                gp.DeleteFeatures_management(Catchments_Layer)
        else:
            print "No features bound within other features."
        print "Finished with FillDonut()."
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error in FillDonut()."
```

```
        sys.exit()

def SelectEdgeLinesThatCrossCatchmentBoundaries(Edges, Catchments, BoundingPolygonLine,starttime, deletetempdata):
    #Identifies those edge lines that cross catchment boundaries. Any polygon touching the
    #convex hull are excluded meaning all edge polygons can recieve flow from
    #non-edge polygons, but that cannot flow back into interior polygons.  All
    #edge polygons must flow off the surface model.
    #This function also calculates the number of polygons that touch the convex hull
    #and the count of those that don't touch the convex hull.  If those counts are the same, then all
    #aggregations are complete and the function calls EndProgram()
    try:
        print "\nCall SelectEdgeLinesThatCrossCatchmentBoundaries()."
        Output_Layer = "Output_Layer"
        Catchments_Output_Layer ="Catchments_Output_Layer"
        EdgesAccrossCatchmentBoundaries = "EdgesAccrossCatchmentBoundaries"
        BoundingPolygon_Output_Layer= "Convex_Hull_Output_Layer"
        #Add an attribute to indentify if this catchment touches the edge of the convex hull
        gp.AddField_management(Catchments, "IS_POUR_PT", "DOUBLE", "", "", "", "", "", "NON_REQUIRED", "")
        print "Make Feature Layer."
        #Make Feature Layers...
        gp.MakeFeatureLayer_management(Edges, Output_Layer, "", "", "")
        gp.MakeFeatureLayer_management(Catchments, Catchments_Output_Layer, "", "", "")
        gp.MakeFeatureLayer_management(BoundingPolygonLine, BoundingPolygon_Output_Layer, "", "", "")
        #Select only those sinks that do not touch the edge of the surface model, This tool assumes that all
        #polygons touching the convex hull can receive flow but do not flow into the surface model.
        CatchmentsCount = gp.GetCount_management(Catchments)
        print "CatchmentsCount = ", CatchmentsCount
        gp.SelectLayerByLocation_management(Catchments_Output_Layer, "BOUNDARY_TOUCHES", BoundingPolygon_Output_Layer, "",
"NEW_SELECTION")
        CatchmentsTouchingBoundingPolygon = gp.GetCount_management(Catchments_Output_Layer)
        print "CatchmentsTouchingBoundingPolygon = ", CatchmentsTouchingBoundingPolygon
        if CatchmentsCount == CatchmentsTouchingBoundingPolygon:
            print "\n\nAll Catchments intersect the convex hull."
            EndProgram(starttime, deletetempdata)
        gp.CalculateField_management(Catchments_Output_Layer, "IS_POUR_PT", "1", "VB")
        #Get a list of all the catchments touching the convex hull.
        ListOfPourPointCatchments = []
```

```
        rows2 = gp.SearchCursor(Catchments_Output_Layer)
        row2 = rows2.Next()
        while row2:
            if row2.IS_POUR_PT == 1:
                ListOfPourPointCatchments.append(row2.Catchments)
            row2 = rows2.next()
        del rows2, row2
        #Select only those catchment polygons that are interior polygons.
        gp.SelectLayerByLocation_management(Catchments_Output_Layer, "", "", "", "SWITCH_SELECTION")
        print "Select all edges that cross the boundary of the selected polygons."
        gp.SelectLayerByLocation_management(Output_Layer, "CROSSED_BY_THE_OUTLINE_OF", Catchments_Output_Layer, "", "NEW_SELECTION")
        print "Create feature class EdgesAccrossCatchmentBoundaries."
        gp.CopyFeatures_management(Output_Layer, EdgesAccrossCatchmentBoundaries, "", "0", "0", "0")
        del Output_Layer, Catchments_Output_Layer, EdgesAccrossCatchmentBoundaries, BoundingPolygon_Output_Layer
        print "Finished SelectEdgeLinesThatCrossCatchmentBoundaries()"
        return ListOfPourPointCatchments
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error with SelectEdgeLinesThatCrossCatchmentBoundaries()"
        sys.exit()

def AlterLineGeometryFlowsFrom2FlowsTo(EdgesAccrossCatchmentBoundaries, textfile):
    #ESRI feature class geometry holds a start node and an end node.  These nodes are independent of
    #poly z objects.  Because lines can have a start node with a lower elevation than a end node,
    #reassign the line direction so that the highest z value is that start node.
    #Feature geometry is read using cursors identifying the start and end node z values and written to a list.
    #The line-node coordinates are flipped if necessary so that the line start has the highest z value.
    #The resulting geometry is written to a text file and that text file is used to create a new feature class so
    #that line direction is the same as the flow direction.
    try:
        print "\nCall AlterLineGeometryFlowsFrom2FlowsTo()."
        FlowLinesAcrossCatchmentsWithFlowDirections = "FlowLinesAcrossCatchmentsWithFlowDirections"
        if os.path.exists(textfile):
            os.remove(textfile)
```

```
print "Create Text File."
f = open(textfile,'a')
thestring = "polyline\n"
f.writelines(thestring)
f.close()
print "Read Feature Geometry. Create Correct Flow Directions."
desc = gp.Describe(EdgesAccrossCatchmentBoundaries)
shapefieldname = desc.ShapeFieldName
rows = gp.SearchCursor(EdgesAccrossCatchmentBoundaries)
row = rows.Next()
while row:
    feat = row.GetValue(shapefieldname)
    FeatureID = str(row.getvalue(desc.OIDFieldName))
    partnum = 0
    partcount = feat.PartCount
    while partnum < partcount:
        ThePart = str(partnum)
        part = feat.GetPart(partnum)
        pnt = part.Next()
        pntcount = 0
        Thecurrentpart = []
        while pnt:
            Thecurrentpart.append(pnt.x)
            Thecurrentpart.append(pnt.y)
            Thecurrentpart.append(pnt.z)
            pnt = part.Next()
            pntcount += 1
            if not pnt:
                pnt = part.Next()
                if pnt:
                    print "Interior Ring:"
        partnum += 1
    #If the from z is lower than the to z, flip them, write the results to a text file...
    if Thecurrentpart[2]<Thecurrentpart[5]:
        thestring = FeatureID +" 0" + "\n" + "0 " + str(Thecurrentpart[3])+ " "+ str(Thecurrentpart[4])+ " "+ str(Thecurrentpart[5])+ "\n" +"1 "+
str(Thecurrentpart[0])+ " "+ str(Thecurrentpart[1])+ " "+ str(Thecurrentpart[2])+"\n"
    else:
```

```
            thestring = FeatureID +" 0" + "\n" + "0 " + str(Thecurrentpart[0])+ " "+ str(Thecurrentpart[1])+ " "+ str(Thecurrentpart[2])+ "\n" +"1 "+
str(Thecurrentpart[3])+ " "+ str(Thecurrentpart[4])+ " "+ str(Thecurrentpart[5])+"\n"
            f = open(textfile,'a')
            f.writelines(thestring)
            f.close()
            row = rows.Next()
        f = open(textfile,'a')
        thestring = "END"
        f.writelines(thestring)
        f.close()
        del row, rows
        #Create a new feature class that has the correct flow directions.
        print "Create Features from Text File."
        #Process: Create Features From Text File...
        gp.CreateFeaturesFromTextFile_samples(textfile, ".", FlowLinesAcrossCatchmentsWithFlowDirections, "")
        print "Finished AlterLineGeometryFlowsFrom2FlowsTo()."
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error with AlterLineGeometryFlowsFrom2FlowsTo()."
        sys.exit()

def AddCatchmentIDsToFlowLines(FlowLinesAcrossCatchmentsWithFlowDirections, Catchments, ListOfPourPointCatchments):
    # This function appends to the attribute table of FlowLinesAcrossCatchmentsWithFlowDirections the
    #catchment ID for the catchment the line originates in and the catchment it ends in by exporting line nodes
    #to a new feature class, using a spatial join to append the catchment ID to the nodes.  The resuting nodes
    #catchment values are read with a cursor and stored in a Python list.  Finally, this list is iterated and the
    #catchment IDs are written to the attribute table of the lines.  The FlowLinesAcrossCatchmentsWithFlowDirections
    #feature class will store the catchment IDs in  attributes called From_ID, and To_ID.
    #Any line that originates in catchment x and flow back into it that same catchment is removed from the feature class
    #because this will cause closed loops which are not- reconsilable with the spanning tree funtion.
    try:
        print "\nCall AddCatchmentIDsToFlowLines()"
        SpatialJoinOutput = "SpatialJoinOutput"
        FeatureVerticiesToPoints1 = "FeatureVerticiesToPoints1"
```

```
print "Adding Fields"
gp.AddField_management(FlowLinesAcrossCatchmentsWithFlowDirections, "From_ID", "SHORT", "", "", "", "", "", "NON_REQUIRED", "")
gp.AddField_management(FlowLinesAcrossCatchmentsWithFlowDirections, "To_ID", "SHORT", "", "", "", "", "", "NON_REQUIRED", "")
gp.AddField_management(Catchments, "Catchment1", "SHORT", "", "", "", "", "", "NON_REQUIRED", "")
#Get the catchment IDs for the lines that cross the catchment boundaries.
print "Export Vertices to Points."
gp.FeatureVerticesToPoints_management(FlowLinesAcrossCatchmentsWithFlowDirections, FeatureVerticiesToPoints1, "BOTH_ENDS")
print "Join Vertices to Catchment IDs"
fieldmappings = gp.CreateObject("FieldMappings")
fieldmappings.AddTable(Catchments)
fieldmap = fieldmappings.GetFieldMap(fieldmappings.FindFieldMapIndex("Catchments"))
field = fieldmap.OutputField
field.Name = "Catchments"
fieldmap.OutputField = field
fieldmappings.ReplaceFieldMap(fieldmappings.FindFieldMapIndex("Catchments"), fieldmap)
gp.SpatialJoin_analysis(FeatureVerticiesToPoints1, Catchments, SpatialJoinOutput, "JOIN_ONE_TO_ONE", "KEEP_ALL",fieldmappings )
#Enummerate the points and get the values of the catchments
#Write the catchments to a list....
#Because the FeatureVerticiesToPoints1 points are in order (from to) and
#by original line FID, you can read the points and append to the lines.
print "Searching Feature Attributes."
rows2 = gp.SearchCursor(SpatialJoinOutput)
row2 = rows2.Next()
CatchmentNumbers = []
while row2:
    catchment = row2.Catchments
    CatchmentNumbers.append(catchment)
    row2 = rows2.next()
del rows2, row2
#Each line flow from one catchment to another.
#Add the from-catchment-id and the to-catchment-id
#to the attribute table of the lines across catchments.
#Remove any lines that flow back into themselved causing loop
print "Writing Feature Attributes."
counter = 0
rows = gp.UpdateCursor(FlowLinesAcrossCatchmentsWithFlowDirections)
row = rows.Next()
```

```
        while row:
            try:
                From_ID = CatchmentNumbers[counter]
                To_ID = CatchmentNumbers[counter + 1]
            except:
                pass
            counter += 2
            #Edges can cross catchment boundaries but from-to the same catchment
            #causes a closed loop that chokes the spanning tree.
            #If this happens, delete that row to avoid sinks in sinks.
            if From_ID == To_ID or From_ID in ListOfPourPointCatchments:
                    rows.DeleteRow(row)
            else:
                row.From_ID = From_ID
                row.To_ID = To_ID
                rows.UpdateRow(row)
            row = rows.Next()
        del row, rows
        print "Finished with AddCatchmentIDsToFlowLines()."
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error in AddCatchmentIDsToFlowLines()."
        sys.exit()
def FeatureZGeometryFromAPolylineZToList(InputFeatureClass):
    #Gets the feature geometry from a polyline z and
    #write the xyz values of the from nodes and to nodes
    #to a Python list.
    try:
        print "\nCall FeatureZGeometryFromAPolylineZToList()"
        desc = gp.Describe(InputFeatureClass)
        shapefieldname = desc.ShapeFieldName
        print "Start search cursor."
        rows = gp.SearchCursor(InputFeatureClass)
        row = rows.Next()
```

```
        Alltheparts = []
        while row:
            feat = row.GetValue(shapefieldname)
            partnum = 0
            partcount = feat.PartCount
            while partnum < partcount:
                part = feat.GetPart(partnum)
                pnt = part.Next()
                pntcount = 0
                Thecurrentpart = []
                while pnt:
                    Thecurrentpart.append(pnt.z)
                    pnt = part.Next()
                    pntcount += 1
                    if not pnt:
                        pnt = part.Next()
                partnum += 1
            Alltheparts.append(Thecurrentpart)
            row = rows.Next()
        return Alltheparts
        del row, rows, Thecurrentpart
        print "Finished FeatureZGeometryFromAPolylineZToList()"
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error writing feature z values to a list."
        sys.exit()

def WriteFeatureGeometryToTheAttributeTableLines(InputFeatureClass, ListOfZValues):
    #Read the feature geometry from the Python list generated by
    #FeatureZGeometryFromAPolylineZToList and writes that feature geometry
    #to the line file's attribute table.
    try:
        print "\nCall WriteFeatureGeometryToTheAttributeTableLines()."
        try:
```

```
            print "Add  fields"
            gp.AddField_management(InputFeatureClass, "FROM_Z", "DOUBLE", "", "", "", "", "", "NON_REQUIRED", "")
            gp.AddField_management(InputFeatureClass, "TO_Z", "DOUBLE", "", "", "", "", "", "NON_REQUIRED", "")
            #gp.AddField_management(InputFeatureClass, "FLOW_LINE", "DOUBLE", "", "", "", "", "", "NON_REQUIRED", "")
            gp.AddField_management(InputFeatureClass, "Per_Slope", "Float", "", "", "", "", "", "NON_REQUIRED", "")
        except:
            print "Fields already exist, passing."
            pass
        counter = 0
        print "Start UpdateCursor."
        rows = gp.UpdateCursor(InputFeatureClass)
        row = rows.Next()
        while row:
            row.FROM_Z = ListOfZValues[counter][0]
            row.TO_Z = ListOfZValues[counter][1]
            therise = ListOfZValues[counter][0]- ListOfZValues[counter][1]
            percentslope = abs(therise/row.Shape_Length *100)
            row.Per_Slope = percentslope
            counter += 1
            rows.UpdateRow(row)
            row = rows.Next()
        del row, rows, ListOfZValues, counter
        print "Finished with WriteFeatureGeometryToTheAttributeTableLines()."
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error with WriteFeatureGeometryToTheAttributeTableLines()."
        sys.exit()

def IdentifyFlowLineOutOfSinkPolygons(InputFeatureClass1):
    #The feature class called FlowLinesAcrossCatchmentsWithFlowDirections
    #represents all flow lines extending out of each polygon.
    #This functions analyses each line for each polygon and identifies that line which that
    #has the lowest 'flows from' z value.  Because this line represent the most likely
    #path water would take if the polygon was filled, this line identifies the connective
```

```
#route between two sink polygons.  If there is more than one line that share the same
#lowest z value, then the line with the steepest slope is selected.  If there are
#more that one line with the same lowest z out value, and the same slope, the last
#item returned by the Python sort method is selected.
#A new feature class called FlowLinesAcrossCatchmentsWithFlowDirectionsedited is created.
#FlowLinesAcrossCatchmentsWithFlowDirectionsedited are those lines that define the path
#water would take if filled and flowed into its neighbor.

try:
    print "\nCall IdentifyFlowLineOutOfSinkPolygons()"
    InputFeatureClass1 = "FlowLinesAcrossCatchmentsWithFlowDirections"
    InputFeatureClass = "FlowLinesAcrossCatchmentsWithFlowDirectionsedited"
    print "copy features"
    gp.Copy_management(InputFeatureClass1, InputFeatureClass)
    print "Write all objectid, from catchment ids, fromz, and slopes to a list"
    rows = gp.SearchCursor(InputFeatureClass)
    row = rows.Next()
    SuperList = []
    while row:
        templist = []
        templist.append(row.OBJECTID)
        templist.append(row.From_ID)
        templist.append(row.FROM_Z)
        templist.append(-1 * row.Per_Slope)
        SuperList.append(templist)
        row = rows.next()
    del rows, row
    #Sort ascending order fromid, fromz, and descending perslope.
    print "Sort list by ascending catchment id, ascending from z, and descending slope values."
    SuperList = sorted(SuperList, key=operator.itemgetter(1,2,3))
    CatchmentList = []
    ObjectIDList = []
    #The first item in superlist is the line in that catchment with the lowest fromz and
    #the steepest slope if more than one, save this line and purge the rest.
    for item in SuperList:
        if item[1]not in  CatchmentList:
            #The first from catchment returned is that line with the lowest z value out, and the steepest slope.
```

```
            #Save that from catchment id to a list and save that object id, this identifies the flow out lines.
            CatchmentList.append(item[1])
            ObjectIDList.append(item[0])
        #Now delete any lines not the line out.
        rows = gp.UpdateCursor(InputFeatureClass)
        row = rows.Next()
        while row:
            #If the objectid is in the object id list, this is a flow out line, keep it.
            #Otherwise, remove it from the feature class.
            if row.OBJECTID in ObjectIDList:
                pass
            else:
                rows.DeleteRow(row)
            row = rows.next()
        del rows, row#, SuperList, templist, CatchmentList, ObjectIDList
        print "Finished IdentifyFlowLineOutOfSinkPolygons()"
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error with IdentifyFlowLineOutOfSinkPolygons()."
        sys.exit()

def CreateListOfFromAndToCatchmentValues(InputFeatureClass):
    #The lines in FlowLinesAcrossCatchmentsWithFlowDirectionsedited store all the catchment id that they
    #flow from, flow into, and the z value of the flows to end.  This function reads that feature
    #class and writes these values to a Python list including a new 'aggregated catchment ID value.  The
    #resulting list is formatted for use the SpanTheTree().
    try:
        InputFeatureClass = "FlowLinesAcrossCatchmentsWithFlowDirectionsedited"
        print "\nCall CreateListOfFromAndToCatchmentValues()"
        ToFromList = []
        rows = gp.SearchCursor(InputFeatureClass)
        row = rows.Next()
        counter = 0
        while row:
```

```
            templist = []
            templist.append(row.From_ID)
            templist.append(row.To_ID)
            templist.append(row.To_Z)
            templist.append(0)
            ToFromList.append(templist)
            row = rows.next()
            counter +=1
        print "Finished with CreateListOfFromAndToCatchmentValues()"
        del rows, row
        return ToFromList
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error in CreateListOfFromAndToCatchmentValues()."
        sys.exit()
def SpanTheTree(ToFromList, From_ID, counter):
    #The ToFromList is sort by increasing flows to z values. This
    #funtions iterates the list and identifies any connected catchments by walking up
    #the connected graph and checking flows from -flows to values.
    #The variable counter is used to store a nominal value used to
    #identify which sinks are connected.
    try:
        for item in ToFromList:
            currentCatchment = From_ID
            for item2 in ToFromList:
                if item2[3] == 0 and item2[1] == From_ID:
                    item2[3] = counter
                    From_ID = item2[0]
                    ToFromList = SpanTheTree(ToFromList, From_ID, counter)
        return ToFromList
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
```

```
        print ErrorDesc.message
        print "Error in SpanTheTree()."
        sys.exit()

def AggregateCatchmentSinksToNewCatchments(Catchments, ToFromList):
    #Dissolves catchments together that share connected flow by iterating the ToFromList.
    #If the catchment value exists in the ToFromList, it is assigned that new catchment ID value
    #from an item in the ToFromList which is writen to a new attribute called Catchments1.
    #If the catchment is not found in the ToFromList that polygon is asigned an arbitrary unique
    #nominal value.
    try:
        CatchmentsCount = gp.GetCount_management(Catchments)
        print "\nStart AggregateCatchmentSinksToNewCatchments()"
        rows = gp.UpdateCursor(Catchments)
        row = rows.Next()
        while row:
            #Identify any catchment polygon without flow in or out (edge polygons)
            noflowpathcatchment = 0
            for item in ToFromList:
                if item[0] == row.Catchments or item[1] == row.Catchments:
                    row.Catchment1 = item[3]
                    #This row has flow in or out, so assign noflowpathcatchment a value of 1 and break the iteration.
                    noflowpathcatchment = 1
                    break
            #The ToFromList was iterated and no connective flow found, give the catchment
            #a unique catchment1 id.  The CatchmentCount is used to assign a values that will not conflict
            #with the catchment1 IDs defined earlier in the code.
            if noflowpathcatchment == 0:
                row.Catchment1 = int(CatchmentsCount)
                CatchmentsCount -=1
            rows.UpdateRow(row)
            row = rows.Next()
        print  "Create Catchments featureclass."
        CatchmentsFirstFill = "CatchmentsFirstFill"
        gp.Dissolve_management(Catchments, CatchmentsFirstFill, "Catchment1", "", "MULTI_PART", "DISSOLVE_LINES")
        gp.AddField_management(CatchmentsFirstFill, "Catchments", "SHORT", "", "", "", "", "", "NON_REQUIRED", "")
        gp.CalculateField_management(CatchmentsFirstFill, "Catchments", "[OBJECTID]", "VB", "")
```

```
        gp.CalculateField_management(CatchmentsFirstFill, "Catchment1", "[OBJECTID]", "VB", "")
        print "RenameFiles and Proceed."
        dummy = 1
        counter = 1
        while dummy == 1:
            newcatchment = "Catchments" + str(counter)
            print "Check for " + newcatchment
            if gp.Exists(newcatchment):
                counter +=1
            else:
                print "rename Catchments to ", newcatchment
                gp.Rename_management(Catchments, newcatchment, "FeatureClass")
                print "Rename CatchmentsFirstFill"
                gp.Rename_management(CatchmentsFirstFill, Catchments, "FeatureClass")
                print "reset dummy"
                dummy = 0
        print "Finished with AggregateCatchmentSinksToNewCatchments()."
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error with AggregateCatchmentSinksToNewCatchments()."
        sys.exit()
###############
try:
    try:
        print "Create geoprocessor."
        gp = arcgisscripting.create()
        print "Set product type to ArcInfo."
        gp.SetProduct("ArcInfo")
        gp.overwriteoutput = overwriteoutput
        print "Check out 3D and SA extentions."
        gp.CheckOutExtension("3D")
        gp.CheckOutExtension("sa")
        print "Set workspace directory."
        gp.workspace = TheWorkingDirectory
```

```
    gp.RefreshCatalog(TheWorkingDirectory)
    starttime = time.time()
    t = time.localtime(starttime)
    print "Start Time = " + GetTime(t)
    #Script defined (required) variables. Do not alter these variable names...
    print "Define variables."
    Edges = "Edges"
    Catchments = "Catchments"
    FlowLinesAcrossCatchments = "FlowLinesAcrossCatchments"
    Nodes = "Nodes"
    BoundingPolygon = "BoundingPolygon"
    BoundingPolygonLine = "BoundingPolygonLine"
    EdgesAccrossCatchmentBoundaries = "EdgesAccrossCatchmentBoundaries"
    FlowLinesAcrossCatchmentsWithFlowDirections = "FlowLinesAcrossCatchmentsWithFlowDirections"
    FlowLinesAcrossCatchmentsWithFlowDirectionsLayer = "FlowLinesAcrossCatchmentsWithFlowDirectionsLayer"
    CatchmentsFirstFill = "CatchmentsFirstFill"
    NodesFeatureLayer = "NodesFeatureLayer"
    FlowLinesAcrossCatchmentsFeatureLayer = "FlowLinesAcrossCatchmentsFeatureLayer"
    startnodes = "startnodes"
    startnodes_Output_Layer = "startnodes_Output_Layer"
    FlowLinesAcrossCatchmentsWithFlowDirectionsedited = "FlowLinesAcrossCatchmentsWithFlowDirectionsedited"
    EndNodesWithCatchmentIDs = "EndNodesWithCatchmentIDs"
    FeatureVerticiesToPoints1 = "FeatureVerticiesToPoints1"
    EndNodes = "EndNodes"
    SpatialJoinOutput = "SpatialJoinOutput"
    EndNodesTemp = "EndNodesTemp"
    EndNodes_Dissolve = "EndNodes_Dissolve"
    CatchmentsFirstFillTemp = "CatchmentsFirstFillTemp"
    ListOFZValues = []
except:
    print "Error in creating gp or declaring variables."
    sys.exit()
#Keep doing this until the EndProgram() is called.
while True:
    FillDonut(Catchments)
    ListOfPourPointCatchments = SelectEdgeLinesThatCrossCatchmentBoundaries(Edges, Catchments, BoundingPolygonLine,starttime, deletetempdata)
    AlterLineGeometryFlowsFrom2FlowsTo(EdgesAccrossCatchmentBoundaries, textfile)
```

```
        AddCatchmentIDsToFlowLines(FlowLinesAcrossCatchmentsWithFlowDirections, Catchments, ListOfPourPointCatchments)
        ListOFZValues = FeatureZGeometryFromAPolylineZToList(FlowLinesAcrossCatchmentsWithFlowDirections)
        WriteFeatureGeometryToTheAttributeTableLines(FlowLinesAcrossCatchmentsWithFlowDirections, ListOFZValues)
        IdentifyFlowLineOutOfSinkPolygons(FlowLinesAcrossCatchmentsWithFlowDirections)
        ToFromList = CreateListOfFromAndToCatchmentValues(FlowLinesAcrossCatchmentsWithFlowDirectionsedited)
        ToFromList = sorted(ToFromList,key=operator.itemgetter(2))
        counter = 1
        for item in ToFromList:
            currentCatchment = item[1]
            for item2 in ToFromList:
                if item2[1] == currentCatchment and item2[3] == 0:
                    item2[3] = counter
                    From_ID = item2[0]
                    itemcounter = 0
                    FromToCatchmentIDs = SpanTheTree(ToFromList, From_ID, counter)
            counter +=1
        AggregateCatchmentSinksToNewCatchments(Catchments, ToFromList)
except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
except Exception, ErrorDesc:
    print ErrorDesc.message
    print "General Python Error."
```

Appendix D- *3_AggregateSinkCatchments.py*

```
print "Assign sink polygons to catchments "

######User Defined Variables#####################################
#The path to the geodatabase feature dataset
TheWorkingDirectory = r"C:\Temp\ITSMHydro2011CorrectNames\TestData\CanyonLake.gdb\ITSMHydro"
#A textfile path and name (must be a read\write space.)
textfile =  r"C:\Temp\GBGTempTextFileForReassignLineDirections.txt"
#Set to 1 to delete the temp feature classes.
overwriteoutput = 1
deletetempdata = 0
#################################################################

import sys, string, os, arcgisscripting, math, operator, exceptions, shutil, time

def GetTime(t):
    #Convert time to a readable format and to time code execution.
    theyear = t[0]
    themonth = t[1]
    theday = t[2]
    thehour = t[3]
    theminutes = t[4]
    theseconds = t[5]
    thedate = str(themonth) + r"/" + str(theday) + r"/" + str(theyear)
    starttime = str(thehour) + ":" + str(theminutes) + ":"+str(theseconds)
    return thedate +", "+ starttime

def EndProgram(starttime, deletetempdata):
    #Quit the program,delete temporary data file if requested, and
    #print code execution time.
    if deletetempdata == 1:
        print "Delete temp data"
        EdgesAccrossCatchmentBoundaries = "EdgesAccrossCatchmentBoundaries"
        FlowLinesAcrossCatchmentsWithFlowDirections ="FlowLinesAcrossCatchmentsWithFlowDirections"
        SpatialJoinOutput = "SpatialJoinOutput"
        FeatureVerticiesToPoints1 = "FeatureVerticiesToPoints1"
        FlowLinesAcrossCatchmentsWithFlowDirectionsedited = "FlowLinesAcrossCatchmentsWithFlowDirectionsedited"
        gp.Delete_management(EdgesAccrossCatchmentBoundaries)
        gp.Delete_management(FlowLinesAcrossCatchmentsWithFlowDirections)
```

```
        gp.Delete_management(SpatialJoinOutput)
        gp.Delete_management(FeatureVerticiesToPoints1)
        gp.Delete_management(FlowLinesAcrossCatchmentsWithFlowDirectionsedited)
    endtime = time.time()
    t = time.localtime(endtime)
    print "Finished at " + GetTime(t)
    totalseconds = endtime - starttime
    hours1 = totalseconds/3600
    hours = int(hours1)
    minutes2 = hours1 - hours
    minutes1 = minutes2*60
    minutes = int(minutes1)
    seconds = int((minutes1-minutes)*60)
    print "\n\nTime to Process; "+str(hours) +": "+str(minutes) +": "+str(seconds)
    x = raw_input("Finished! Press enter to quit")
    sys.exit(0)

def FillDonut(inputfeatureclass):
    #Catchment delineations can have other polygons bound within them. Delete any
    #verticies of a polygon the represent the interior portion of the feature
    #by identifing interior nodes and delete them using a feature geometry array.
    #The bounding polygon is no longer an annulus but a different record exists
    #that represents the area covered by the inner hole.  Find those remaining
    #features and delete them. The resulting feature class represents an irregular
    #tesselation of polygons with no polygons bound within any other polygon.
    try:
        print "\nCall FillDonut()"
        desc = gp.Describe(inputfeatureclass)
        shapefield = desc.ShapeFieldName
        rows = gp.UpdateCursor(inputfeatureclass)
        row = rows.next()
        arrayObj = gp.CreateObject("Array")
        arrayOuter = gp.CreateObject("Array")
        ListOfIDs = []
        while row:
            feat = row.getValue(shapefield)
            qInterior = False
            for partNum in range(feat.partCount) :
                part = feat.getPart(partNum)
                qInterior = False
                for ptNum in range(part.count):
```

```
                pt = part.next()
                if pt != None:
                    arrayOuter.add(pt)
                else :
                    qInterior = True
                    break
            arrayObj.add(arrayOuter)
            arrayOuter.RemoveAll()
        if qInterior :
            row.setValue(shapefield,arrayObj)
            rows.updateRow(row)
            ListOfIDs.append(row.OBJECTID)
        arrayObj.RemoveAll()
        row = rows.next()
    del rows,row
    Catchments_Layer = "Catchments_Layer"
    gp.MakeFeatureLayer_management(inputfeatureclass, Catchments_Layer, "", "", "")
    if ListOfIDs != []:
        print "Filling features bound within other catchments."
        for item in ListOfIDs:
            gp.SelectLayerByAttribute_management(Catchments_Layer, "NEW_SELECTION", "\"OBJECTID\" = " + str(item))
            gp.SelectLayerByLocation_management(Catchments_Layer, "COMPLETELY_WITHIN", Catchments_Layer, "", "NEW_SELECTION")
            gp.DeleteFeatures_management(Catchments_Layer)
    else:
        print "No features bound within other features."
    print "Finished with FillDonut()."
except arcgisscripting.ExecuteError:
    print gp.GetMessages(2)
    sys.exit()
except Exception, ErrorDesc:
    print ErrorDesc.message
    print "Error in FillDonut()."
    sys.exit()

def SelectEdgeLinesThatCrossCatchmentBoundaries(Edges, Catchments, BoundingPolygonLine,starttime, deletetempdata):
    #Identifies those edge lines that cross catchment boundaries. Any polygon touching the
    #convex hull are excluded meaning all edge polygons can recieve flow from
    #non-edge polygons, but that cannot flow back into interior polygons.  All
    #edge polygons must flow off the surface model.
    #This function also calculates the number of polygons that touch the convex hull
    #and the count of those that don't touch the convex hull.  If those counts are the same, then all
```

```
    #aggregations are complete and the function calls EndProgram()
    try:
        print "\nCall SelectEdgeLinesThatCrossCatchmentBoundaries()."
        Output_Layer = "Output_Layer"
        Catchments_Output_Layer ="Catchments_Output_Layer"
        EdgesAccrossCatchmentBoundaries = "EdgesAccrossCatchmentBoundaries"
        BoundingPolygon_Output_Layer= "Convex_Hull_Output_Layer"
        #Add an attribute to indentify if this catchment touches the edge of the convex hull
        gp.AddField_management(Catchments, "IS_POUR_PT", "DOUBLE", "", "", "", "", "", "NON_REQUIRED", "")
        print "Make Feature Layer."
        #Make Feature Layers...
        gp.MakeFeatureLayer_management(Edges, Output_Layer, "", "", "")
        gp.MakeFeatureLayer_management(Catchments, Catchments_Output_Layer, "", "", "")
        gp.MakeFeatureLayer_management(BoundingPolygonLine, BoundingPolygon_Output_Layer, "", "", "")
        #Select only those sinks that do not touch the edge of the surface model, This tool assumes that all
        #polygons touching the convex hull can receive flow but do not flow into the surface model.
        CatchmentsCount = gp.GetCount_management(Catchments)
        print "CatchmentsCount = ", CatchmentsCount
        gp.SelectLayerByLoc

ation_management(Catchments_Output_Layer, "BOUNDARY_TOUCHES", BoundingPolygon_Output_Layer, "", "NEW_SELECTION")
        CatchmentsTouchingBoundingPolygon = gp.GetCount_management(Catchments_Output_Layer)
        print "CatchmentsTouchingBoundingPolygon = ", CatchmentsTouchingBoundingPolygon
        if CatchmentsCount == CatchmentsTouchingBoundingPolygon:
            print "\n\nAll Catchments intersect the convex hull."
            EndProgram(starttime, deletetempdata)
        gp.CalculateField_management(Catchments_Output_Layer, "IS_POUR_PT", "1", "VB")
        #Get a list of all the catchments touching the convex hull.
        ListOfPourPointCatchments = []
        rows2 = gp.SearchCursor(Catchments_Output_Layer)
        row2 = rows2.Next()
        while row2:
            if row2.IS_POUR_PT == 1:
                ListOfPourPointCatchments.append(row2.Catchments)
            row2 = rows2.next()
        del rows2, row2
        #Select only those catchment polygons that are interior polygons.
        gp.SelectLayerByLocation_management(Catchments_Output_Layer, "", "", "", "SWITCH_SELECTION")
        print "Select all edges that cross the boundary of the selected polygons."
        gp.SelectLayerByLocation_management(Output_Layer, "CROSSED_BY_THE_OUTLINE_OF", Catchments_Output_Layer, "", "NEW_SELECTION")
        print "Create feature class EdgesAccrossCatchmentBoundaries."
```

```
        gp.CopyFeatures_management(Output_Layer, EdgesAccrossCatchmentBoundaries, "", "0", "0", "0")
        del Output_Layer, Catchments_Output_Layer, EdgesAccrossCatchmentBoundaries, BoundingPolygon_Output_Layer
        print "Finished SelectEdgeLinesThatCrossCatchmentBoundaries()"
        return ListOfPourPointCatchments
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error with SelectEdgeLinesThatCrossCatchmentBoundaries()"
        sys.exit()

def AlterLineGeometryFlowsFrom2FlowsTo(EdgesAccrossCatchmentBoundaries, textfile):
    #ESRI feature class geometry holds a start node and an end node.  These nodes are independent of
    #poly z objects.  Because lines can have a start node with a lower elevation than a end node,
    #reassign the line direction so that the highest z value is that start node.
    #Feature geometry is read using cursors identifying the start and end node z values and written to a list.
    #The line-node coordinates are flipped if necessary so that the line start has the highest z value.
    #The resulting geometry is written to a text file and that text file is used to create a new feature class so
    #that line direction is the same as the flow direction.
    try:
        print "\nCall AlterLineGeometryFlowsFrom2FlowsTo()."
        FlowLinesAcrossCatchmentsWithFlowDirections = "FlowLinesAcrossCatchmentsWithFlowDirections"
        if os.path.exists(textfile):
            os.remove(textfile)
        print "Create Text File."
        f = open(textfile,'a')
        thestring = "polyline\n"
        f.writelines(thestring)
        f.close()
        print "Read Feature Geometry. Create Correct Flow Directions."
        desc = gp.Describe(EdgesAccrossCatchmentBoundaries)
        shapefieldname = desc.ShapeFieldName
        rows = gp.SearchCursor(EdgesAccrossCatchmentBoundaries)
        row = rows.Next()
        while row:
            feat = row.GetValue(shapefieldname)
            FeatureID = str(row.getvalue(desc.OIDFieldName))
            partnum = 0
            partcount = feat.PartCount
            while partnum < partcount:
```

```
            ThePart = str(partnum)
            part = feat.GetPart(partnum)
            pnt = part.Next()
            pntcount = 0
            Thecurrentpart = []
            while pnt:
               Thecurrentpart.append(pnt.x)
               Thecurrentpart.append(pnt.y)
               Thecurrentpart.append(pnt.z)
               pnt = part.Next()
               pntcount += 1
               if not pnt:
                  pnt = part.Next()
                  if pnt:
                     print "Interior Ring:"
            partnum += 1
         #If the from z is lower than the to z, flip them, write the results to a text file...
         if Thecurrentpart[2]<Thecurrentpart[5]:
            thestring = FeatureID +" 0" + "\n" + "0 " + str(Thecurrentpart[3])+ " "+ str(Thecurrentpart[4])+ " "+ str(Thecurrentpart[5])+ "\n" +"1 "+ str(Thecurrentpart[0])+ " "+
str(Thecurrentpart[1])+ " "+ str(Thecurrentpart[2])+"\n"
         else:
            thestring = FeatureID +" 0" + "\n" + "0 " + str(Thecurrentpart[0])+ " "+ str(Thecurrentpart[1])+ " "+ str(Thecurrentpart[2])+ "\n" +"1 "+ str(Thecurrentpart[3])+ " "+
str(Thecurrentpart[4])+ " "+ str(Thecurrentpart[5])+"\n"
         f = open(textfile,'a')
         f.writelines(thestring)
         f.close()
         row = rows.Next()
      f = open(textfile,'a')
      thestring = "END"
      f.writelines(thestring)
      f.close()
      del row, rows
      #Create a new feature class that has the correct flow directions.
      print "Create Features from Text File."
      #Process: Create Features From Text File...
      gp.CreateFeaturesFromTextFile_samples(textfile, ".", FlowLinesAcrossCatchmentsWithFlowDirections, "")
      print "Finished AlterLineGeometryFlowsFrom2FlowsTo()."
   except arcgisscripting.ExecuteError:
      print gp.GetMessages(2)
      sys.exit()
   except Exception, ErrorDesc:
```

```
        print ErrorDesc.message
        print "Error with AlterLineGeometryFlowsFrom2FlowsTo()."
        sys.exit()

def AddCatchmentIDsToFlowLines(FlowLinesAcrossCatchmentsWithFlowDirections, Catchments, ListOfPourPointCatchments):
    # This function appends to the attribute table of FlowLinesAcrossCatchmentsWithFlowDirections the
    #catchment ID for the catchment the line originates in and the catchment it ends in by exporting line nodes
    #to a new feature class, using a spatial join to append the catchment ID to the nodes.  The resuting nodes
    #catchment values are read with a cursor and stored in a Python list.  Finally, this list is iterated and the
    #catchment IDs are written to the attribute table of the lines.  The FlowLinesAcrossCatchmentsWithFlowDirections
    #feature class will store the catchment IDs in  attributes called From_ID, and To_ID.
    #Any line that originates in catchment x and flow back into it that same catchment is removed from the feature class
    #because this will cause closed loops which are not- reconsilable with the spanning tree funtion.
    try:
        print "\nCall AddCatchmentIDsToFlowLines()"
        SpatialJoinOutput = "SpatialJoinOutput"
        FeatureVerticiesToPoints1 = "FeatureVerticiesToPoints1"
        print "Adding Fields"
        gp.AddField_management(FlowLinesAcrossCatchmentsWithFlowDirections, "From_ID", "SHORT", "", "", "", "", "", "NON_REQUIRED", "")
        gp.AddField_management(FlowLinesAcrossCatchmentsWithFlowDirections, "To_ID", "SHORT", "", "", "", "", "", "NON_REQUIRED", "")
        gp.AddField_management(Catchments, "Catchment1", "SHORT", "", "", "", "", "", "NON_REQUIRED", "")
        #Get the catchment IDs for the lines that cross the catchment boundaries.
        print "Export Vertices to Points."
        gp.FeatureVerticesToPoints_management(FlowLinesAcrossCatchmentsWithFlowDirections, FeatureVerticiesToPoints1, "BOTH_ENDS")
        print "Join Vertices to Catchment IDs"
        fieldmappings = gp.CreateObject("FieldMappings")
        fieldmappings.AddTable(Catchments)
        fieldmap = fieldmappings.GetFieldMap(fieldmappings.FindFieldMapIndex("Catchments"))
        field = fieldmap.OutputField
        field.Name = "Catchments"
        fieldmap.OutputField = field
        fieldmappings.ReplaceFieldMap(fieldmappings.FindFieldMapIndex("Catchments"), fieldmap)
        gp.SpatialJoin_analysis(FeatureVerticiesToPoints1, Catchments, SpatialJoinOutput, "JOIN_ONE_TO_ONE", "KEEP_ALL",fieldmappings )
        #Enummerate the points and get the values of the catchments
        #Write the catchments to a list....
        #Because the FeatureVerticiesToPoints1 points are in order (from to) and
        #by original line FID, you can read the points and append to the lines.
        print "Searching Feature Attributes."
        rows2 = gp.SearchCursor(SpatialJoinOutput)
        row2 = rows2.Next()
        CatchmentNumbers = []
```

```
        while row2:
            catchment = row2.Catchments
            CatchmentNumbers.append(catchment)
            row2 = rows2.next()
        del rows2, row2
        #Each line flow from one catchment to another.
        #Add the from-catchment-id and the to-catchment-id
        #to the attribute table of the lines across catchments.
        #Remove any lines that flow back into themselved causing loop
        print "Writing Feature Attributes."
        counter = 0
        rows = gp.UpdateCursor(FlowLinesAcrossCatchmentsWithFlowDirections)
        row = rows.Next()
        while row:
            try:
                From_ID = CatchmentNumbers[counter]
                To_ID = CatchmentNumbers[counter + 1]
            except:
                pass
            counter += 2
            #Edges can cross catchment boundaries but from-to the same catchment
            #causes a closed loop that chokes the spanning tree.
            #If this happens, delete that row to avoid sinks in sinks.
            if From_ID == To_ID or From_ID in ListOfPourPointCatchments:
                    rows.DeleteRow(row)
            else:
                row.From_ID = From_ID
                row.To_ID = To_ID
                rows.UpdateRow(row)
            row = rows.Next()
        del row, rows
        print "Finished with AddCatchmentIDsToFlowLines()."
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error in AddCatchmentIDsToFlowLines()."
        sys.exit()

def FeatureZGeometryFromAPolylineZToList(InputFeatureClass):
```

```
    #Gets the feature geometry from a polyline z and
    #write the xyz values of the from nodes and to nodes
    #to a Python list.
    try:
        print "\nCall FeatureZGeometryFromAPolylineZToList()"
        desc = gp.Describe(InputFeatureClass)
        shapefieldname = desc.ShapeFieldName
        print "Start search cursor."
        rows = gp.SearchCursor(InputFeatureClass)
        row = rows.Next()
        Alltheparts = []
        while row:
            feat = row.GetValue(shapefieldname)
            partnum = 0
            partcount = feat.PartCount
            while partnum < partcount:
                part = feat.GetPart(partnum)
                pnt = part.Next()
                pntcount = 0
                Thecurrentpart = []
                while pnt:
                    Thecurrentpart.append(pnt.z)
                    pnt = part.Next()
                    pntcount += 1
                    if not pnt:
                        pnt = part.Next()
                partnum += 1
            Alltheparts.append(Thecurrentpart)
            row = rows.Next()
        return Alltheparts
        del row, rows, Thecurrentpart
        print "Finished FeatureZGeometryFromAPolylineZToList()"
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error writing feature z values to a list."
        sys.exit()

def WriteFeatureGeometryToTheAttributeTableLines(InputFeatureClass, ListOfZValues):
```

```
    #Read the feature geometry from the Python list generated by
    #FeatureZGeometryFromAPolylineZToList and writes that feature geometry
    #to the line file's attribute table.
    try:
        print "\nCall WriteFeatureGeometryToTheAttributeTableLines()."
        try:
            print "Add  fields"
            gp.AddField_management(InputFeatureClass, "FROM_Z", "DOUBLE", "", "", "", "", "", "NON_REQUIRED", "")
            gp.AddField_management(InputFeatureClass, "TO_Z", "DOUBLE", "", "", "", "", "", "NON_REQUIRED", "")
            #gp.AddField_management(InputFeatureClass, "FLOW_LINE", "DOUBLE", "", "", "", "", "", "NON_REQUIRED", "")
            gp.AddField_management(InputFeatureClass, "Per_Slope", "Float", "", "", "", "", "", "NON_REQUIRED", "")
        except:
            print "Fields already exist, passing."
            pass
        counter = 0
        print "Start UpdateCursor."
        rows = gp.UpdateCursor(InputFeatureClass)
        row = rows.Next()
        while row:
            row.FROM_Z = ListOfZValues[counter][0]
            row.TO_Z = ListOfZValues[counter][1]
            therise = ListOfZValues[counter][0]- ListOfZValues[counter][1]
            percentslope = abs(therise/row.Shape_Length *100)
            row.Per_Slope = percentslope
            counter += 1
            rows.UpdateRow(row)
            row = rows.Next()
        del row, rows, ListOfZValues, counter
        print "Finished with WriteFeatureGeometryToTheAttributeTableLines()."
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error with WriteFeatureGeometryToTheAttributeTableLines()."
        sys.exit()

def IdentifyFlowLineOutOfSinkPolygons(InputFeatureClass1):

    #The feature class called FlowLinesAcrossCatchmentsWithFlowDirections
    #represents all flow lines extending out of each polygon.
```

```
#This functions analyses each line for each polygon and identifies that line which that
#has the lowest 'flows from' z value.  Because this line represent the most likely
#path water would take if the polygon was filled, this line identifies the connective
#route between two sink polygons.  If there is more than one line that share the same
#lowest z value, then the line with the steepest slope is selected.  If there are
#more that one line with the same lowest z out value, and the same slope, the last
#item returned by the Python sort method is selected.
#A new feature class called FlowLinesAcrossCatchmentsWithFlowDirectionsedited is created.
#FlowLinesAcrossCatchmentsWithFlowDirectionsedited are those lines that define the path
#water would take if filled and flowed into its neighbor.
try:
    print "\nCall IdentifyFlowLineOutOfSinkPolygons()"
    InputFeatureClass1 = "FlowLinesAcrossCatchmentsWithFlowDirections"
    InputFeatureClass = "FlowLinesAcrossCatchmentsWithFlowDirectionsedited"
    print "copy features"
    gp.Copy_management(InputFeatureClass1, InputFeatureClass)
    print "Write all objectid, from catchment ids, fromz, and slopes to a list"
    rows = gp.SearchCursor(InputFeatureClass)
    row = rows.Next()
    SuperList = []
    while row:
        templist = []
        templist.append(row.OBJECTID)
        templist.append(row.From_ID)
        templist.append(row.FROM_Z)
        templist.append(-1 * row.Per_Slope)
        SuperList.append(templist)
        row = rows.next()
    del rows, row
    #Sort ascending order fromid, fromz, and descending perslope.
    print "Sort list by ascending catchment id, ascending from z, and descending slope values."
    SuperList = sorted(SuperList, key=operator.itemgetter(1,2,3))
    CatchmentList = []
    ObjectIDList = []
    #The first item in superlist is the line in that catchment with the lowest fromz and
    #the steepest slope if more than one, save this line and purge the rest.
    for item in SuperList:
        if item[1]not in  CatchmentList:
            #The first from catchment returned is that line with the lowest z value out, and the steepest slope.
            #Save that from catchment id to a list and save that object id, this identifies the flow out lines.
            CatchmentList.append(item[1])
```

```
            ObjectIDList.append(item[0])
        #Now delete any lines not the line out.
        rows = gp.UpdateCursor(InputFeatureClass)
        row = rows.Next()
        while row:
            #If the objectid is in the object id list, this is a flow out line, keep it.
            #Otherwise, remove it from the feature class.
            if row.OBJECTID in ObjectIDList:
                pass
            else:
                rows.DeleteRow(row)
            row = rows.next()
        del rows, row#, SuperList, templist, CatchmentList, ObjectIDList
        print "Finished IdentifyFlowLineOutOfSinkPolygons()"
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error with IdentifyFlowLineOutOfSinkPolygons()."
        sys.exit()

def CreateListOfFromAndToCatchmentValues(InputFeatureClass):
    #The lines in FlowLinesAcrossCatchmentsWithFlowDirectionsedited store all the catchment id that they
    #flow from, flow into, and the z value of the flows to end.  This function reads that feature
    #class and writes these values to a Python list including a new 'aggregated catchment ID value.  The
    #resulting list is formatted for use the SpanTheTree().
    try:
        InputFeatureClass = "FlowLinesAcrossCatchmentsWithFlowDirectionsedited"
        print "\nCall CreateListOfFromAndToCatchmentValues()"
        ToFromList = []
        rows = gp.SearchCursor(InputFeatureClass)
        row = rows.Next()
        counter = 0
        while row:
            templist = []
            templist.append(row.From_ID)
            templist.append(row.To_ID)
            templist.append(row.To_Z)
            templist.append(0)
            ToFromList.append(templist)
```

```
            row = rows.next()
            counter +=1
        print "Finished with CreateListOfFromAndToCatchmentValues()"
        del rows, row
        return ToFromList
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error in CreateListOfFromAndToCatchmentValues()."
        sys.exit()
def SpanTheTree(ToFromList, From_ID, counter):
    #The ToFromList is sort by increasing flows to z values. This
    #funtions iterates the list and identifies any connected catchments by walking up
    #the connected graph and checking flows from -flows to values.
    #The variable counter is used to store a nominal value used to
    #identify which sinks are connected.
    try:
        for item in ToFromList:
            currentCatchment = From_ID
            for item2 in ToFromList:
                if item2[3] == 0 and item2[1] == From_ID:
                    item2[3] = counter
                    From_ID = item2[0]
                    ToFromList = SpanTheTree(ToFromList, From_ID, counter)
        return ToFromList
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error in SpanTheTree()."
        sys.exit()
def AggregateCatchmentSinksToNewCatchments(Catchments, ToFromList):
    #Dissolves catchments together that share connected flow by iterating the ToFromList.
    #If the catchment value exists in the ToFromList, it is assigned that new catchment ID value
    #from an item in the ToFromList which is writen to a new attribute called Catchments1.
    #If the catchment is not found in the ToFromList that polygon is asigned an arbitrary unique
    #nominal value.
    try:
```

```
CatchmentsCount = gp.GetCount_management(Catchments)
print "\nStart AggregateCatchmentSinksToNewCatchments()"
rows = gp.UpdateCursor(Catchments)
row = rows.Next()
while row:
    #Identify any catchment polygon without flow in or out (edge polygons)
    noflowpathcatchment = 0
    for item in ToFromList:
        if item[0] == row.Catchments or item[1] == row.Catchments:
            row.Catchment1 = item[3]
            #This row has flow in or out, so assign noflowpathcatchment a value of 1 and break the iteration.
            noflowpathcatchment = 1
            break
    #The ToFromList was iterated and no connective flow found, give the catchment
    #a unique catchment1 id.  The CatchmentCount is used to assign a values that will not conflict
    #with the catchment1 IDs defined earlier in the code.
    if noflowpathcatchment == 0:
        row.Catchment1 = int(CatchmentsCount)
        CatchmentsCount -=1
    rows.UpdateRow(row)
    row = rows.Next()
print  "Create Catchments featureclass."
CatchmentsFirstFill = "CatchmentsFirstFill"
gp.Dissolve_management(Catchments, CatchmentsFirstFill, "Catchment1", "", "MULTI_PART", "DISSOLVE_LINES")
gp.AddField_management(CatchmentsFirstFill, "Catchments", "SHORT", "", "", "", "", "", "NON_REQUIRED", "")
gp.CalculateField_management(CatchmentsFirstFill, "Catchments", "[OBJECTID]", "VB", "")
gp.CalculateField_management(CatchmentsFirstFill, "Catchment1", "[OBJECTID]", "VB", "")
print "RenameFiles and Proceed."
dummy = 1
counter = 1
while dummy == 1:
    newcatchment = "Catchments" + str(counter)
    print "Check for " + newcatchment
    if gp.Exists(newcatchment):
        counter +=1
    else:
        print "rename Catchments to ", newcatchment
        gp.Rename_management(Catchments, newcatchment, "FeatureClass")
        print "Rename CatchmentsFirstFill"
        gp.Rename_management(CatchmentsFirstFill, Catchments, "FeatureClass")
        print "reset dummy"
```

```
            dummy = 0
        print "Finished with AggregateCatchmentSinksToNewCatchments()."
    except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
        sys.exit()
    except Exception, ErrorDesc:
        print ErrorDesc.message
        print "Error with AggregateCatchmentSinksToNewCatchments()."
        sys.exit()
##############
try:
    try:
        print "Create geoprocessor."
        gp = arcgisscripting.create()
        print "Set product type to ArcInfo."
        gp.SetProduct("ArcInfo")
        gp.overwriteoutput = overwriteoutput
        print "Check out 3D and SA extentions."
        gp.CheckOutExtension("3D")
        gp.CheckOutExtension("sa")
        print "Set workspace directory."
        gp.workspace = TheWorkingDirectory
        gp.RefreshCatalog(TheWorkingDirectory)
        starttime = time.time()
        t = time.localtime(starttime)
        print "Start Time = " + GetTime(t)
        #Script defined (required) variables. Do not alter these variable names...
        print "Define variables."
        Edges = "Edges"
        Catchments = "Catchments"
        FlowLinesAcrossCatchments = "FlowLinesAcrossCatchments"
        Nodes = "Nodes"
        BoundingPolygon = "BoundingPolygon"
        BoundingPolygonLine = "BoundingPolygonLine"
        EdgesAccrossCatchmentBoundaries = "EdgesAccrossCatchmentBoundaries"
        FlowLinesAcrossCatchmentsWithFlowDirections = "FlowLinesAcrossCatchmentsWithFlowDirections"
        FlowLinesAcrossCatchmentsWithFlowDirectionsLayer = "FlowLinesAcrossCatchmentsWithFlowDirectionsLayer"
        CatchmentsFirstFill = "CatchmentsFirstFill"
        NodesFeatureLayer = "NodesFeatureLayer"
        FlowLinesAcrossCatchmentsFeatureLayer = "FlowLinesAcrossCatchmentsFeatureLayer"
        startnodes = "startnodes"
```

```
        startnodes_Output_Layer = "startnodes_Output_Layer"
        FlowLinesAcrossCatchmentsWithFlowDirectionsedited = "FlowLinesAcrossCatchmentsWithFlowDirectionsedited"
        EndNodesWithCatchmentIDs = "EndNodesWithCatchmentIDs"
        FeatureVerticiesToPoints1 = "FeatureVerticiesToPoints1"
        EndNodes = "EndNodes"
        SpatialJoinOutput = "SpatialJoinOutput"
        EndNodesTemp = "EndNodesTemp"
        EndNodes_Dissolve = "EndNodes_Dissolve"
        CatchmentsFirstFillTemp = "CatchmentsFirstFillTemp"
        ListOFZValues = []
    except:
        print "Error in creating gp or declaring variables."
        sys.exit()
    #Keep doing this until the EndProgram() is called.
    while True:
        FillDonut(Catchments)
        ListOfPourPointCatchments = SelectEdgeLinesThatCrossCatchmentBoundaries(Edges, Catchments, BoundingPolygonLine,starttime, deletetempdata)
        AlterLineGeometryFlowsFrom2FlowsTo(EdgesAccrossCatchmentBoundaries, textfile)
        AddCatchmentIDsToFlowLines(FlowLinesAcrossCatchmentsWithFlowDirections, Catchments, ListOfPourPointCatchments)
        ListOFZValues = FeatureZGeometryFromAPolylineZToList(FlowLinesAcrossCatchmentsWithFlowDirections)
        WriteFeatureGeometryToTheAttributeTableLines(FlowLinesAcrossCatchmentsWithFlowDirections, ListOFZValues)
        IdentifyFlowLineOutOfSinkPolygons(FlowLinesAcrossCatchmentsWithFlowDirections)
        ToFromList = CreateListOfFromAndToCatchmentValues(FlowLinesAcrossCatchmentsWithFlowDirectionsedited)
        ToFromList = sorted(ToFromList,key=operator.itemgetter(2))
        counter = 1
        for item in ToFromList:
            currentCatchment = item[1]
            for item2 in ToFromList:
                if item2[1] == currentCatchment and item2[3] == 0:
                    item2[3] = counter
                    From_ID = item2[0]
                    itemcounter = 0
                    FromToCatchmentIDs = SpanTheTree(ToFromList, From_ID, counter)
            counter +=1
        AggregateCatchmentSinksToNewCatchments(Catchments, ToFromList)
except arcgisscripting.ExecuteError:
        print gp.GetMessages(2)
except Exception, ErrorDesc:
    print ErrorDesc.message
    print "General Python Error."
```

Appendix F- Surveyed Sample Locations and Surface Model Elevation Values

Survey Point Location		Elevations in Feet Above NAVD88									
Longitude	Latitude	Surveyed Elevation	USGS 10 M	30 ft. Linear	30 ft. Nat. Neig.	6 ft. Linear	6 ft Nat. Neig.	3 ft. Linear	3 ft. Nat. Neig.	1 ft. Linear	1/2 ft Linear
-122.6250	48.8192	15.19	13.80	14.77	14.57	14.85	14.87	14.87	14.87	14.86	14.86
-122.6120	48.8191	21.32	13.80	20.55	20.46	20.55	20.50	20.55	20.55	20.55	20.55
-122.5840	48.8190	17.50	10.94	16.72	16.51	17.13	17.15	17.19	17.20	17.24	17.24
-122.6000	48.8192	14.36	13.81	13.48	13.31	13.97	14.05	14.06	14.06	14.04	14.04
-122.6280	48.8193	10.64	7.38	10.42	10.47	10.41	10.43	10.44	10.44	10.46	10.46
-122.6420	48.8194	10.88	11.17	9.04	8.62	10.49	10.50	10.56	10.53	10.52	10.52
-122.6830	48.8485	254.86	253.66	254.27	254.31	254.42	254.43	254.45	254.46	254.45	NA
-122.6850	48.8484	265.92	267.67	265.50	265.43	265.50	265.47	265.45	265.49	265.52	NA
-122.6500	48.8484	196.72	192.81	196.04	196.24	196.42	196.41	196.45	196.44	196.47	NA
-122.6460	48.8483	197.30	192.23	196.02	195.67	196.90	197.12	197.08	197.14	197.08	NA
-122.6350	48.8483	197.51	206.02	196.19	196.10	197.01	197.03	196.98	196.99	196.99	NA
-122.6230	48.8483	221.99	217.94	220.00	220.59	221.68	221.66	221.66	221.65	221.67	NA
-122.6160	48.8484	133.50	134.82	132.39	132.66	133.16	133.14	133.15	133.19	133.20	NA
-122.6670	48.7327	11.60	9.10	10.18	10.05	10.10	10.12	10.12	10.13	10.13	10.13
-122.6720	48.7321	11.80	5.60	10.19	10.34	10.30	10.29	10.29	10.29	10.33	10.33
-122.6630	48.7313	31.18	24.61	30.33	30.23	29.97	30.00	29.95	29.96	29.95	29.94
-122.6590	48.7263	23.73	17.39	22.63	22.49	22.64	22.60	22.63	22.65	22.66	22.66
-122.6570	48.7214	16.74	9.38	14.79	15.65	15.58	15.58	15.59	15.57	15.57	15.57
-122.6550	48.7222	58.39	50.17	57.41	57.08	56.74	56.67	56.65	56.67	56.75	56.75
-122.6420	48.7290	14.22	6.03	12.20	12.55	12.74	12.73	12.74	12.77	12.79	12.79
-122.6450	48.7245	12.13	6.19	10.94	10.74	11.11	11.09	11.11	11.11	11.13	11.13

-122.6470	48.7199	13.56	7.69	11.88	12.08	12.05	12.05	12.04	12.05	12.02	12.02
-122.6520	48.7173	31.64	27.84	30.34	29.80	30.76	30.82	30.84	30.87	30.87	30.87
-122.6510	48.7160	12.26	8.04	10.50	10.75	11.39	11.41	11.53	11.47	11.49	11.49
-122.6270	48.7452	27.63	25.89	25.11	25.18	25.91	25.86	25.82	26.00	26.03	26.05
-122.6360	48.7377	27.64	26.45	27.66	27.70	28.73	28.74	28.74	28.74	28.75	28.75
-122.6380	48.7332	26.28	25.96	23.47	23.50	25.35	25.34	25.36	25.38	25.37	25.37
-122.6380	48.7325	24.24	25.88	22.58	22.89	22.95	22.97	22.95	22.91	22.90	22.90
-122.6130	48.7556	53.23	48.63	50.35	50.42	48.07	48.06	47.97	48.01	48.03	48.03
-122.6180	48.7528	45.82	32.12	44.65	44.54	44.73	44.73	44.72	44.70	44.70	44.70
-122.6210	48.7499	69.23	65.53	64.76	64.80	64.87	64.85	64.87	64.86	64.89	64.89
-122.6250	48.7467	41.08	30.35	40.17	40.14	40.48	40.48	40.58	40.57	40.46	40.46
-122.6830	48.8045	47.00	56.13	46.53	46.57	46.32	46.38	46.27	46.38	46.42	46.42
-122.6240	48.7949	46.50	48.68	46.15	46.10	45.99	46.04	46.00	46.02	46.02	46.02
-122.5950	48.7960	18.40	7.24	17.95	18.19	18.58	18.51	18.55	18.55	18.56	18.56
-122.6510	48.7165	19.80	14.25	18.58	18.41	19.61	19.63	19.67	19.67	19.67	19.67
-122.6610	48.7468	147.05	155.39	145.14	145.39	145.60	145.63	145.66	145.66	145.67	145.67
-122.6390	48.7576	113.20	116.14	109.94	110.06	110.77	110.80	110.81	110.82	110.83	110.83
-122.6390	48.7468	84.65	85.93	82.98	83.20	83.30	83.27	83.26	83.28	83.27	83.27
-122.6040	48.7743	12.81	12.69	11.24	11.23	11.37	11.41	11.40	11.41	11.43	11.43
-122.6440	48.7762	34.65	18.36	32.72	32.74	32.88	32.86	32.82	32.83	32.82	32.82
-122.6280	48.7657	156.52	161.72	155.31	155.28	155.18	155.20	155.22	155.20	155.21	155.20
-122.6100	48.7619	35.13	35.11	32.51	32.93	33.64	33.62	33.66	33.64	33.66	33.66
-122.6230	48.7950	47.11	51.30	46.59	46.66	46.81	46.78	46.79	46.78	46.82	46.82
-122.6270	48.7699	119.15	112.10	118.95	119.32	119.39	119.31	119.40	119.42	119.45	119.45
-122.6490	48.7574	104.45	101.61	104.84	104.90	104.99	105.08	105.12	105.02	104.93	104.92
-122.6340	48.7578	110.36	116.31	112.89	113.45	113.85	113.88	113.87	113.90	114.09	114.09
-122.6220	48.7503	65.64	62.93	67.00	67.16	67.28	67.31	67.24	67.29	67.27	67.27

-122.6120	48.7609	92.71	98.88	93.29	93.05	92.99	92.71	92.77	92.66	92.49	92.48
-122.6450	48.7794	34.12	13.92	34.06	34.10	33.92	33.95	33.91	33.91	33.90	33.90
-122.6550	48.7591	27.53	25.57	27.34	27.18	27.45	27.48	27.46	27.50	27.51	27.52
-122.6580	48.7488	146.59	145.33	145.77	145.72	146.11	146.08	146.17	146.18	146.15	146.15
-122.6550	48.7245	56.39	51.39	56.25	56.19	56.26	56.16	56.21	56.18	56.20	56.20
-122.6670	48.7429	119.28	109.96	118.96	119.06	118.90	118.95	119.02	118.97	119.01	119.02
-122.6640	48.7395	126.03	128.37	126.07	126.26	126.05	126.04	126.03	126.04	126.00	126.00
-122.6610	48.7422	145.29	155.43	145.32	145.51	145.08	145.18	145.17	145.13	145.09	145.09
-122.6550	48.7202	57.26	53.00	59.73	60.08	60.35	60.45	60.58	60.56	60.63	60.64
-122.6620	48.7425	139.73	144.69	139.68	139.83	139.67	139.66	139.66	139.65	139.62	139.62
-122.6650	48.7448	144.64	145.80	144.13	144.18	144.21	144.21	144.20	144.21	144.21	144.21
-122.6420	48.7325	32.41	43.60	31.76	31.63	32.11	32.01	32.01	32.07	32.06	32.05
-122.6320	48.7633	153.11	160.39	153.41	153.67	153.34	153.37	153.33	153.35	153.35	153.35
-122.6490	48.7538	141.88	148.20	141.89	141.86	141.94	141.93	141.96	141.95	141.95	141.95
-122.6560	48.7358	74.94	82.07	75.01	75.17	75.07	75.05	75.13	75.15	75.16	75.16