This is to certify that the
thesis entitled

QUANTIFYING AND PREDICTING ERROR IN DEM-DERIVED FLOW-DIRECTION

presented by

GLENN ALEXANDER O'NEIL

has been accepted towards fulfillment
of the requirements for the

M.S. degree in Geography

Major Professor's Signature

4-27-2010

Date

QUANTIFYING AND PREDICTING ERROR IN DEM-DERIVED FLOW-DIRECTION

By

Glenn Alexander O'Neil

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Geography

2010

ABSTRACT

QUANTIFYING AND PREDICTING ERROR IN
DEM-DERIVED FLOW-DIRECTION

By

Glenn Alexander O'Neil

In this thesis, I present a new method for quantifying errors in flow-direction rasters derived from USGS 10-meter digital elevation models (DEMs). The method utilizes flow-direction rasters derived from finer resolution 2.5-foot LiDAR DEMs as a reference dataset, and performs cell-by-cell comparisons between the two datasets to quantify error in the USGS product. I applied this method to a sample of 80 7.5-minute USGS quadrangles, stratified by land cover and topography, across Ohio. I employed quadrangle-level measures of agricultural concentration, total relief, and contour topology error as independent variables in a multiple linear regression model to predict mean flow-direction error.

DEDICATION

I dedicate this thesis to my wife Jamie and son Rowan. Though this work cannot measure my appreciation for their love and support, I would be remiss not to acknowledge my greatest sources for inspiration.

ACKNOWLEDGMENTS

I would not have been able to complete this thesis without the steadfast support and advice offered by my advisor, Ashton Shortridge. His insightful feedback helped me resolve the theoretical and technical hurdles this research presented.

I would also not have been able to complete this thesis without the support of Jon Bartholic of the Institute of Water Research. His flexibility as a director provided me with the time and resources needed to generate and analyze the detailed and extensive datasets of this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Images in this thesis are presented in color

KEY TO SYMBOLS AND ABBREVIATIONS

ASCII – American Standard Code for Information Interchange

ATtILA – Analytical Tools Interface for Landscape Assessments

D8 – single direction flow-direction algorithm

DEM – digital elevation model

DLG – digital line graph

EPA – Environmental Protection Agency

ESRI – Environmental Systems Research Institute

FDE – Flow-direction error

GDAL – Geospatial Data Abstraction Library

GIS – geographical information systems/science

HRHA – high relief high agricultural concentration

HRLA – high relief low agricultural concentration

LiDAR – Light Detection and Ranging

LRHA – low relief high agricultural concentration

LRLA – low relief low agricultural concentration

NED – National Elevation Dataset

NHD – National Hydrography Dataset

NLCD – National Land Cover Dataset

OGRIP – Ohio Geographically Referenced Information Program

OSIP – Ohio State-wide Imagery Program

RMSE – root mean square error

SAR – simultaneous auto-regression

USGS – United States Geological Survey

UTM – Universal Transverse Mercator

Chapter 1 Introduction

1.1 Statement of Problem

In the field of hydrology, digital elevation models (DEMs) are frequently used to predict surface water flow direction and derive digital stream networks. The accuracy of these predictions and networks relies on the fidelity of the source DEMs. In flatter areas inaccuracies in a DEM can be more pronounced, particularly if the interpolation of contours with whole number values (e.g. 315 versus 315.7 feet) was employed to build the DEM, which is the standard of most DEMs produced by the United States Geological Survey (USGS) (Carter 1988, Carter 1992, Wilson and Gallant 2000). Agricultural areas are also prone to DEM inaccuracy. The errors in both flat and agricultural areas are typically due to issues with the source contours that yield a DEM. In flat areas, sparse contours provide too little information to the interpolating algorithm that produces the DEM. In agricultural areas, the source contours can contain topological errors, such as intersections, around drainage ditches that confuse the interpolation algorithm. The results are digital surfaces that do not truly reflect the Earth's surface and therefore inaccurately predict surface water flow. These errors can pose a significant problem for researchers and conservationists who rely on models of surface water flow-direction, such as soil conservationists concerned with water-borne soil erosion and nutrient runoff from agricultural lands into streams and lakes. If they cannot reference accurate

predictions of how water flows over the surface, then efforts to identify and remediate sedimentation-prone areas will be misguided and waste valuable time and resources.

There is a rich literature documenting absolute elevation error in DEMs, but research assessing error in DEM-derived flow-direction is lacking. In this thesis I will help fill this gap by exploring the following questions:

1. How can finer-resolution DEMs be employed to evaluate flow-direction error in coarser DEMs?
2. What are the primary challenges in such an evaluation?
3. What landscape characteristics best predict flow-direction error?

To address these questions, I developed a method for calculating flow-direction error, and implemented it across a large sample of USGS 7.5-minute quadrangles in Ohio. I then conducted a multivariate analysis to predict flow-direction error in terms of landscape characteristics, accounting for residual spatial autocorrelation.

This research contributes to the understanding of uncertainty in digital elevation products, particularly how error propagates into flow-direction. The methods I developed also aid in the analysis of high-resolution DEMs, specifically in the extraction of hydrologic features from such data. At a more practical level, these results inform hydrological analyses in engineering, agriculture, and any other disciplines that rely on simulations of surface water flow. For example, with a better understanding of erosion model uncertainty, soil conservationists can more effectively target efforts to reduce sediment loading to streams and lakes.

The remainder of this introduction will preview the datasets and methods employed in this research and layout the organization of the thesis.

1.2 Digital Elevation Models and Flow-direction

A DEM is a raster (grid) dataset of elevation values sampled at regularly spaced intervals (resolution). They can be produced in a number of ways, but most published products have been generated through interpolation from topographic sheet contours, remote sensing from airplanes or satellites, or stereoscopic interpretation of aerial photographs. For this thesis, I evaluated 10-meter resolution USGS DEMs, produced through contour interpolation, with 2.5-foot resolution LiDAR (Light Detection and Ranging) DEMs, produced through remote sensing. The USGS DEMs were interpolated from contours of USGS topographic maps at a scale of 1:24,000. The LiDAR DEMs were interpolated from points captured by an airplane-mounted sensor at an altitude of 7,300 feet above ground level.

The USGS is the leading source for DEMs in the U.S. The freely accessible National Elevation Dataset (NED) includes 30-meter resolution DEMs for the contiguous U.S., 10-meter DEMs for the majority of the nation, and finer resolutions in select locations. Not all of the DEMs in the NED were produced by the USGS. The NED is described as "the best available raster elevation data for the coterminous United States," (USGS 2006). If DEMs meeting National Map Accuracy Standards and of resolutions finer than those produced by the USGS exist, they may be included in the NED. This situation applied to the study site of this thesis. Ohio's 2.5-foot resolution LiDAR DEMs

are accessible through the NED, even though they were generated for the state by a private contractor.

The organization of DEMs as neighborhoods of elevation values allows for the calculation of numerous derivative datasets, including slope, aspect, line of sight, and surface water flow-direction. This last derivative is the horizontal direction of surface water flow out of a particular area. In the case of flow-direction derived from a DEM, it is calculated for each grid cell in the original DEM, and can be determined through multiple methods. Some approaches represent flow-direction as an angle and disperse it in multiple directions (Quinn et al. 1991, Freeman 1991, Tarboton 1997). Other approaches are more restrictive and represent it as a single value assigned to one of eight compass directions (O'Callaghan and Mark 1984, Greenlee 1987, Jenson and Domingue 1988). Nearly all the methods are driven by calculating slope angles between a particular cell and its surrounding neighbors. For this research, I employed a single-direction method limited by the eight compass directions. I will discuss this selection in greater detail in Chapter 5.

1.3 Flow-direction Error

I developed a novel method to calculate the difference in flow-direction between 10-meter USGS DEMs and 2.5-foot LiDAR DEMs. This method compared a flow-direction value of a USGS DEM cell to an aggregated value from LiDAR DEM cells contained within the coarser USGS cell. I coded the comparison into a weighted error score based on the extent of agreement (or disagreement) between the two values. I then

determined an error score for the majority of USGS DEM cells in eighty 7.5-minute quadrangles in Ohio (roughly 1 million cells per quadrangle).

I then calculated the mean flow-direction error for each quadrangle, and used this statistic as the dependent variable in a multiple linear regression model. I sought to explain how flow-direction error varied in relation to changes in total relief, agricultural concentration, and contour topology error.

The next chapter will discuss previous efforts to evaluate DEMs and their derivatives, and establish this thesis as a new and unique contribution to that effort. Chapter 3 will describe how I gathered and processed the independent variables of the proposed multiple linear regression model. Chapter 4 will describe the pre-processing steps of the USGS 10-meter and LiDAR DEMs, including a new approach for stream extraction from high-resolution digital elevation data, and the calculation of flow-direction error. Chapter 5 will describe statistical measures of how flow-direction error varied across different landscapes, provide an evaluation of the proposed and alternative regression models, and present an analysis of the results. Chapter 6 will summarize the findings, discuss their significance, and describe additional research needed to extend this effort.

Chapter 2 Background and Hypotheses

2.1 Background

Due to the numerous applications that DEMs support, their ubiquity, and the fact that they are freely accessible, DEM error has been a well studied topic within geographic information science. Wilson and Gallant (2000) and Fisher and Tate (2006) provide thorough overviews of existing research on this topic. The USGS classifies error in its DEMs as either blunders (major elevation discrepancies), systematic (errors that follow a fixed pattern and are predictable), and random (beyond the control of the observer). It acknowledges that the errors cannot be completely resolved (USGS 1998). In addition to these errors, numerous studies have shown error in DEMs and their derivatives to be spatially autocorrelated (Goodchild et al. 1992, Fisher 1993, Heuvelink 1998, Oksanen and Sarjakoski 2005). To inform users of the uncertainty in the data, the USGS publishes each DEM with estimates of its root-mean-square error (RMSE), a measure of vertical accuracy. RMSE is calculated by comparing a minimum of 28 ground-truth elevations to those of the DEM. USGS standards for most DEMs hold RMSE to no more than one-half of a contour interval (USGS 1998). Most of the contour datasets used to produce USGS DEMs have intervals of 5 or 10 feet, implying maximum RMSE values of 2.5 or five feet. Fisher (1998) argued that in addition to RMSE, DEMs should be published with measures of spatial autocorrelation to allow users to appropriately visualize and simulate the error and its propagation into derivative products.

Knowing the RMSE for a DEM is useful to researchers concerned with absolute elevation accuracy, such as Carson et al. (1997) and Shortridge (2006), to name a few. However, absolute elevation accuracy is not necessarily informative to users studying DEM derivative products such as slope, curvature, catchment basins, aspect, and flow-direction. Even DEMs with low RMSEs can yield significantly variable derivatives (Garbrecht and Starks 1995, Wise 1998, Holmes et al. 2000, Endreny and Wood 2001, Barber and Shortridge 2005). Elevations for DEMs of particularly flat regions of Ohio can have value ranges less than 50 feet over areas as large as 64 square miles. These estimated elevations may be well within the RMSE standards; but derivative products dependent upon an elevation's relationship to neighboring values, such as slope, can be greatly affected by even subtle changes in elevation.

Martinoni and Bernhard (1998) argued that the accuracy of DEM derivatives warrant greater attention. Ziadat (2007) studied the impact of cell-size and contour interval on DEM-derived slope, in addition to absolute elevation values. Venteris and Slater (2005) evaluated multiple DEM derivatives in terms of their ability to predict soil carbon. Wu et al. (2005, 2007) evaluated the effects of DEM resolution on the estimation of soil erosion. Riggs and Dean (2007) explored errors in DEM-derived view-sheds. Desmet (1997) evaluated differences in slope and aspect across different interpolation methods, through mean and standard deviation comparisons, on small, arable fields in Belgium. Chang and Tsai (1991) studied slope and aspect differences between different DEM resolutions in a more spatially explicit manner, on a 1 square kilometer region of Taiwan's eastern coast.

Other studies have looked at DEM predictions of hydrology. Endreny and Wood (2001) studied differences between flow-direction methods in terms of predicted flow-paths on three 64 hectare study sites in western Oklahoma. They compared simulated flow-path networks to control networks and recorded the percent of overlap. Walker and Willgoose (1999) demonstrated the inaccuracies in derivatives of published DEMs through quantitative comparisons of up-stream flow-accumulation networks, slope-area relationship, and normalized width function within a 1.4 square kilometer surveyed reference dataset in southwest Australia. Wise (2000) similarly evaluated DEMs through catchment basins and up-stream flow accumulation, though he focused on a small catchment in southwest England. Kienzle (2004) explored actual directional changes in flow measured in degrees on four study sites (roughly 6 square miles each) in Canada. Wise (2007), Wilson et al. (2007), Barber and Shortridge (2005), and Thompson et al. (2001) studied DEM-derived surface flow through comparisons to derived catchment boundaries. Clarke and Lee (2007) evaluated DEM estimates of surface flow through a comparison to known stream features in the National Hydrography Dataset (NHD).

While these analyses have demonstrated the propagation of DEM error into hydrological derivatives, gaps remain in fully documenting this issue. The previous research has either focused on relatively small and, in some cases, homogenous study sites, or employed coarse metrics. To better describe error in these derivatives and identify landscape characteristics that potentially cause it, an analysis on a large and diverse study area is needed. Such an effort would generate findings that could be better generalized to other locations than the findings of previous research. Furthermore, while

evaluating simulated catchment boundaries is an accepted method of measuring error in DEM, it does not capture field-level variation in DEM-derived surface water flow-direction. To understand how errors in DEM hydrological derivatives differ at the finest scales, evaluations must be made on a cell-by-cell basis. Though Clark and Lee (2007) approach this specificity in their comparison of DEM-derived flow to NHD features, their approach would not allow for a direct comparison where stream features do not exist, such as the middle of a farm field.

For this thesis, I sought to quantify and predict error in flow-direction rasters derived from USGS 10-meter DEMs (RMSE 12.45 feet – Ohio Office of Information Technology 2005) by comparing them to flow-direction rasters derived from finer, more vertically and horizontally accurate LiDAR rasters (RMSE 0.5 feet -Woolpert Inc. 2008). The approach of evaluating less precise spatial data with more precise data has been utilized before (Chang and Tsai, 1991; Fisher, 1998; Kyriakidis et al., 1999; Walker and Willgoose, 1999; Holmes et al., 2000; Kienzle, 2004) and is endorsed by the USGS (Digital Cartographic Data Standards Task Force, 1988). Numerous studies have explored performance differences between DEMs of different resolutions and found finer resolutions superior, depending on the application (Chang and Tsai, 1991; Wolock and McCabe, 2000; Wu et al., 2005; Mouton, 2005; Wu et al., 2007; Deng et al., 2007; Aziz and Steward, 2007; Riggs and Dean, 2007). My study site comprised 80 USGS 7.5-minute quadrangles (over 5,000 square miles) of varying topography and land cover in Ohio. By focusing at such a fine scale over such a large and diverse study area, I sought to evaluate DEM error in a novel and insightful manner.

2.2 Hypotheses

I expected areas with low relief, high concentrations of agriculture, and represented by DEMs produced from contours that contained large numbers of topological errors (intersections) to produce higher values of flow-direction error. More formally:

Equation: $FDE = a + \beta_{PA}PA + \beta_{TR}TR + \beta_{CI}CI$

Where:
FDE = flow-direction error
PA = % agriculture
TR = total relief
CI = contour intersections

Hypotheses:
Ho: $\beta_{PA} = 0$ Ha: $\beta_{PA} > 0$
Ho: $\beta_{TR} = 0$ Ha: $\beta_{TR} < 0$
Ho: $\beta_{CI} = 0$ Ha: $\beta_{CI} > 0$

Figure 2-1: Proposed multiple linear regression model of flow-direction error.

As established earlier, derivatives from relatively flat DEMs are highly sensitive to propagated error. Digital contour datasets are a common source for creating DEMs (Florinsky, 1998), including all of the Ohio 10-meter DEMs employed in this research (Ohio Office of Information Technology 2005). Flat areas tend to have fewer and sparser contours as input to the interpolation algorithms that produce DEMs. The wider the geographic space between contours, the greater the uncertainty in the characterization of

surface water flow over that space. The interpolation process is known to introduce errors into the resulting DEMs, sometimes significantly (Faintich 1996, Carson and Reutebuch, 1997, Carrara et al. 1997, Meyer 2004).

The presence of agricultural drainage ditches can add to this error. These features tend to concentrate a flat area's relatively few contours around them and create topological errors (intersections, specifically) in the contours, confusing the DEM interpolation algorithm. These errors are propagated into the DEMs and their derivative products. Theoretically, contours should never cross; but when digital contours are derived from scanning paper or mylar maps, such as USGS 7.5-minute digital line graph (DLG) hypsography, artifacts in the source map or scanning resolution can cause dense contours to merge together and create a topological violation (Greenlee, 1987). Additionally, cartographers may force contours to merge to indicate steep features such as cliffs, identified as carrying contours in the DLG attributes. The USGS standards for DLGs require that intersecting contours be identified through node topology (described in greater detail in Chapter 3) (USGS, 1999). When these topologically errant contours are interpolated to yield a DEM, the elevation values around an area of intersection can be inaccurately estimated. For example, if a 700 and 710 foot contour intersect, the DEM interpolation algorithm would have difficulty accurately resolving the true surface at that location. The 700 foot contour would, in essence, be exposed to the area that cartographically represents the 710 to 720 foot elevation range, and would subsequently draw down surrounding values in the resulting DEM. In most cases, the resulting error in terms of absolute elevation values in this area would be small, as the error would be

mitigated by the neighboring cells that factored into the interpolation. Nevertheless, some DEM derivative products, such as surface water flow-direction, are more sensitive to this error, as discussed earlier.

Given the effect that low relief, agricultural ditches, and contour intersections can have on error in interpolated DEMs, I expected that their impacts would be more substantial on the particularly error-sensitive flow-direction derivative. This expectation drove the design my hypothesis.

The next chapter provides a detailed description of how the 80 quadrangles were sampled and processed to extract the dependent variables for the proposed regression model in Figure 2-1.

Chapter 3 Methods – The Sample and Independent Variables

The general steps I took to analyze flow-direction error in this research were as follows:

1. Select a sample of USGS 7.5-minute quadrangles, stratified by total relief and agricultural concentration.
2. Gather USGS 10-meter DEMs, 2.5-foot LiDAR DEMs, DLG hypsography, and DLG hydrography for each quadrangle.
3. Process the 10-meter and LiDAR DEMs by enforcing hydrography presence, filling spurious sinks, and calculating flow-direction rasters for each quadrangle.
4. Calculate the extent of contour intersection for DLG hypsography in each quadrangle.
5. Quantify mean flow-direction error for each quadrangle through cell-by-cell comparison of 10-meter and LiDAR flow-direction rasters.
6. Generate a multiple linear regression model predicting flow-direction error in terms of total relief, agricultural concentration, and contour intersection.

I calculated flow-direction error as a single mean value for each quadrangle in a sample of 80 USGS 7.5-minute (1:24,000) quadrangles. I will detail this calculation in the next chapter, but for now I establish it as a single numeric value that will serve as the dependent variable in a multiple linear regression model. This chapter focuses on the sampling and pre-processing of the proposed regression model's independent variables: percentage agriculture, relief, and contour intersections (steps 1, 2, and 4, of the process described above).

3.1 The Sample – Stratified by Agricultural Concentration and Relief

This research focused exclusively on the state of Ohio. The state was an ideal study location because it contained the variability in land cover and topography through which I attempted to explain flow-direction error. Additionally, the Ohio Geographically Referenced Information Program (OGRIP - http://ogrip.oit.ohio.gov/Home.aspx) provided nearly all of the data needed to conduct the analysis for free. The OGRIP website offered USGS 10-meter DEMs, LiDAR point elevations, interpolated LiDAR DEMs, USGS 7.5-minute DLG hypsography (contours) and DLG hydrography (streams).

The USGS 7.5-minute quadrangle served as the study's sample unit, as this is the spatial scale and extent at which USGS DEMs are typically generated, including all of the Ohio 10-meter DEMs provided through the NED. Ohio is covered by 793 quadrangles. Due to the intensive pre-processing needed to calculate flow-direction error (detailed in Chapter 4), I selected a sub-sample of 80 quadrangles. To avoid oversampling the proposed regression model's independent variables, I stratified the samples by their respective total relief and percentages of row-crop agriculture. I calculated these attributes for each quadrangle with the Analytical Tools Interface for Landscape Assessment (ATtILA) tool from EPA (US EPA, 2007), an extension for ESRI's ArcView 3.3.

I provided ATtILA with the quadrangle boundaries for Ohio, a 10-meter DEM for the entire state (described in greater detail in Chapter 4), and a state-wide land cover

raster. I downloaded the 2001 National Land Cover Database (NLCD) from the website of the Multi-Resolution Land Characteristics Consortium (http://www.mrlc.gov/nlcd.php). The NLCD was derived from 30-meter resolution Landsat data and reported an overall accuracy of 84% (Homer et al., 2007). I obtained a copy of the dataset that covered the entire Midwest region and clipped it by Ohio's state boundary. With these state-wide inputs of elevation and land cover, ATtILA was able to output the elevation range and the percentage of various land cover classes in each quadrangle, including row-crop agriculture (2001 NLCD class 82) (Figures 3-1 and 3-2). I then utilized these quadrangle attributes as independent variables in the proposed regression model (Figure 2-1).

Next, in order to avoid potential issues in deriving accurate hydrology from sampled DEMs, I removed quadrangles that contained large amounts of open water (e.g., quadrangles on the Lake Erie border), quadrangles where more than 10% of the area was classified as urban (because it is difficult to predict surface water flow in downtown Cleveland), quadrangles that straddled the state border (because the LiDAR tiles would not cover the entire areas of these quadrangles), and quadrangles that straddled the boundary of the Ohio State Plane Northern and Southern coordinate systems. OGRIP distributes the LiDAR DEMs in appropriate State Plane projections. It proved to be programmatically difficult to analyze a quadrangle that contained LiDAR data in two different projections. Since LiDAR rasters served as the more accurate reference dataset, it was important to not introduce additional error into them; therefore, I avoided re-projecting between State Plane zones. This filtering process reduced the quadrangle

population from 793 to 538. I then divided the 538 quadrangles into the following four bins: high percentage agriculture and high relief (HAHR), high percentage agriculture and low relief (HALR), low percentage agriculture and high relief (LAHR), low percentage agriculture and low relief (LALR). The breakpoints for these bins corresponded to the median values for each class, classifying half of the quadrangles as high agricultural concentration and half as low; they were similarly classified by relief. I then randomly selected 20 quadrangles from each of these bins in order to yield the final sample of 80 quadrangles (Table 3-1 and Figure 3-3).

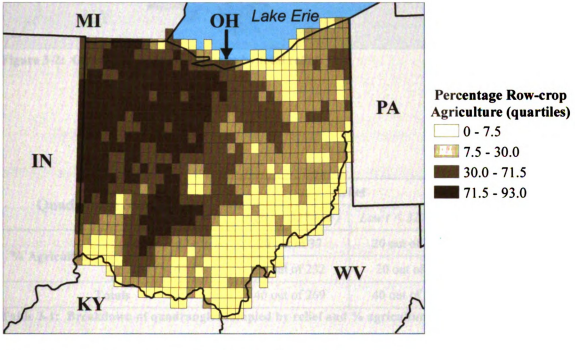


Figure 3-1: Ohio 7.5-minute quadrangles by agricultural concentration.

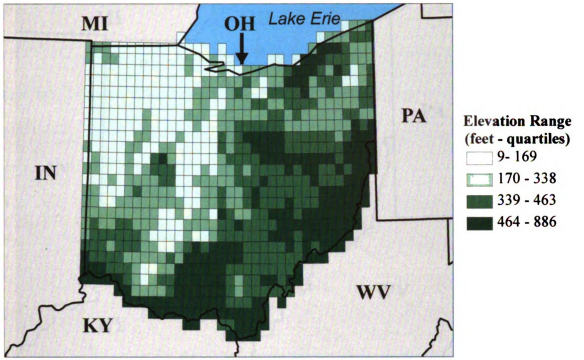


Figure 3-2: Ohio 7.5-minute quadrangles by total relief.

Quadrangle Sampling		Relief		Totals
		High (> 327 feet)	*Low (< 327 feet)*	
% Agriculture	*High (> 67.8%)*	20 out of 37	20 out of 232	40 out of 269
	Low (< 67.8%)	20 out of 232	20 out of 37	40 out of 269
Totals		40 out of 269	40 out of 269	**80 out of 538**

Table 3-1: Breakdown of quadrangles sampled by relief and % agriculture.

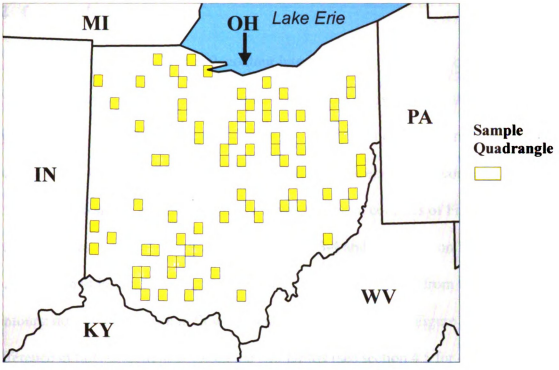


Figure 3-3: Sample 7.5-minute quadrangles.

As shown in Figure 3-3, the sampled quadrangles tended to cluster along Ohio's SW-NE axis. This trend was a byproduct of the sample stratification. Northwest Ohio is dominated by agriculture and low relief, so the 20 quadrangles that constituted this sub-sample had a large geographic area from which they could have been selected. The situation was similar for the high relief and low percentage agriculture sub-sample of southeast Ohio. These two situations naturally clustered the remaining half of the sample along the southwest-northeast axis.

3.2 Contour Topology Error

As established in Chapter 2, contour intersections are suspected of contributing to flow-direction error. Figures 3-4 through 3-10 illustrate the impact that these errors of topology can have on flow-direction. Figure 3-4 identifies topological errors in a small section of a USGS hypsography DLG (Alma, MI). Those errors were corrected in Figure 3-5. Figure 3-6 shows a DEM interpolated from the errant contours of Figure 3-4; note the large depression created at location 4 of Figure 3-4 and its impact on flow concentration in Figure 3-8. Figure 3-7 shows a DEM interpolated from the corrected contours; note the difference in flow concentration in Figure 3-9. Figure 3-10 maps the difference in flow-directions between the two DEMs (see section 4.5 for details). The blue areas indicate agreement, while the red areas indicate dramatic disagreement (180 degrees) and are found in the areas where contour intersections were identified in Figure 3-4.

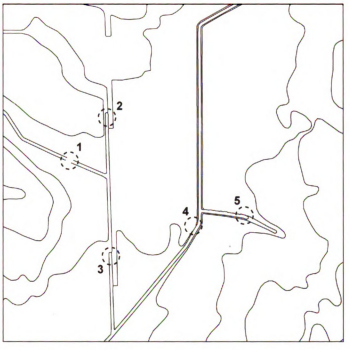


Figure 3-4: Sample contours from a 7.5-minute quadrangle. Note the topological errors - dangles at 1 and intersections at 2-5.

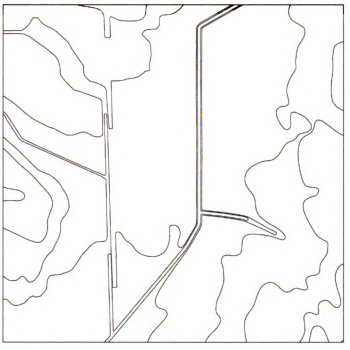


Figure 3-5: Corrected contours. Though it is difficult to tell, the intersecting contours have been manually separated through digitizing.

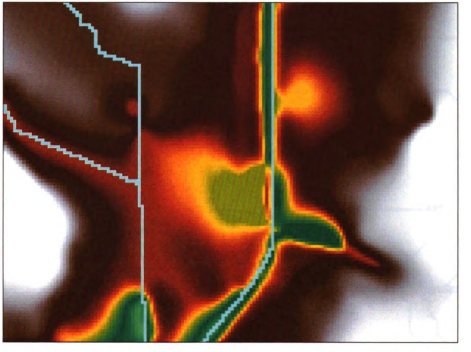


Figure 3-6: DEM derived from topologically incorrect contours.

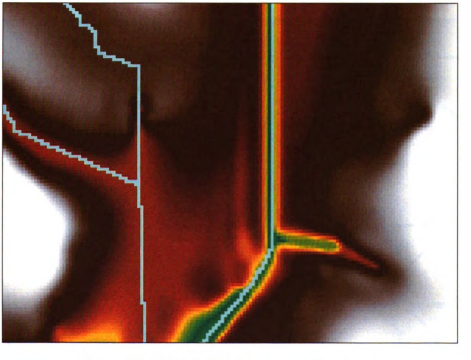


Figure 3-7: DEM derived from corrected contours.

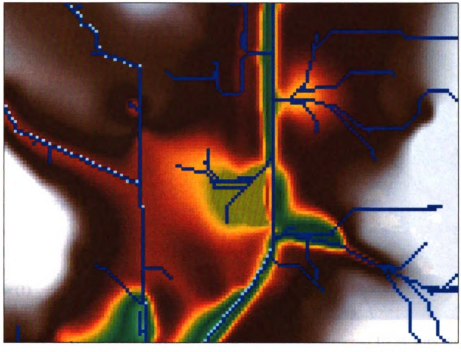


Figure 3-8: Surface flow concentrations (flow-accumulation) derived from topologically incorrect contours.

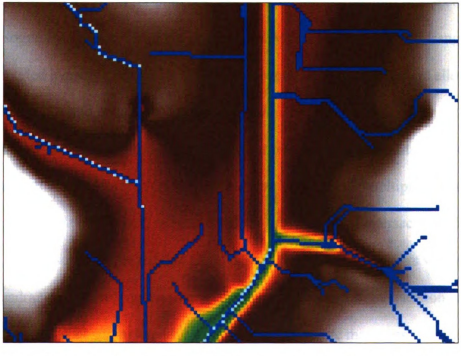


Figure 3-9: Surface flow concentrations derived from corrected contours.

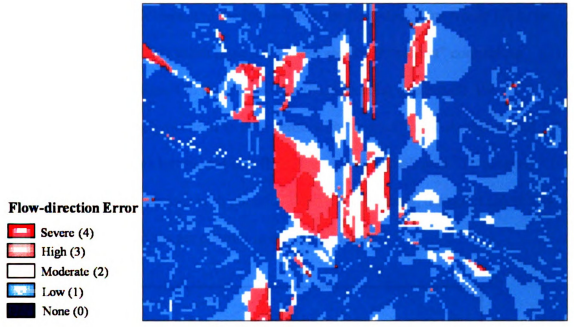


Figure 3-10: Flow-direction error calculated for the topologically incorrect DEM, with the corrected DEM as the reference. Error was greatest around the intersection points. (See Figure 4-35 for legend reference)

I developed a Python script to determine the number of contour intersections in DLG hypsography (the source for the 10-meter DEMs) in each sample quadrangle. The script converted hypsography DLGs to ArcInfo coverages. This conversion allowed for the building of line and node topology for each new coverage. The topology included *FromNode* and *ToNode* attributes for each contour feature, which constituted a logic within the contours by which intersections could be identified. If multiple features shared the same *FromNode* or *ToNode*, this indicated the presence of an intersection (Figure 3-11). The Python script employed this logic, iterating through the contours within each sample quadrangle, identifying the intersections, and recording an intersection count for each quadrangle. A quadrangle with high relief, and therefore many contours, could

potentially have more intersections than a flat quadrangle with comparatively fewer contours. However, the impact of the intersections on flow-directions would likely be more dramatic in the flatter quadrangle since there are fewer "correct" contours to mitigate the error (location 4 in Figure 3-4). In the high relief quadrangle, the intersections would likely occur in an area dense with "correct" contours that would confine any of the errors introduced into flow-direction to a small area. To properly associate contour intersections with potential flow-direction error, the script calculated the number of intersections per contour in each quadrangle.

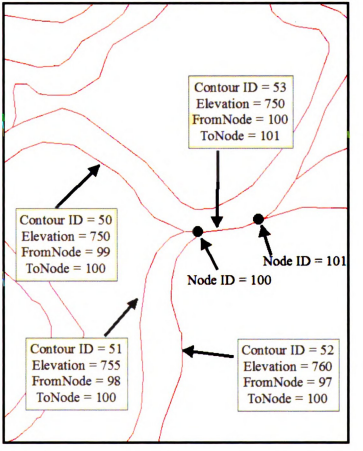


Figure 3-11: Contours 50, 51, and 52 all have Node 100 as their ToNode, indicating an intersection at that node.

In the next chapter, I will describe the steps I took to process the USGS and LiDAR DEMs, and detail how I calculated flow-direction error.

Chapter 4 Methods – Calculating the Dependent Variable: Flow-direction error

The previous chapter covered how I selected the study's 80 sample quadrangles, and calculated the dependent variables of the proposed regression model. In this chapter I discuss how I calculated flow-direction error. I first describe the software and hardware used in that calculation. I then describe the data sources of the calculation's inputs, the pre-processing of those inputs, and the specific steps for calculating flow-direction error.

The process I implemented for this thesis, outside of its design and programming, required a substantial quantity of person-hours and computer-hours. It took one month to complete the analysis for the selected sample (80 7.5-minute quadrangles), with six machines processing data the entire time. The basic steps were as follows: 1, pre-process the 10-meter DEMs; 2, pre-process the LiDAR DEMs; 3, calculate the flow-direction error in the 10-meter DEMs; 4, conduct the statistical analysis.

4.1 Software and Hardware

I performed the data processing almost entirely within ArcGIS 9.3 and the Python programming language. ArcGIS handled all of the data pre-processing including clipping, re-projecting, and map algebra functions. It also performed much of the hydrologic processing of DEMs, including the filling of spurious sinks and calculating surface water flow-directions. Python is a flexible and open-source scripting language that interfaces with ArcGIS's geo-processor. This integration allowed me to automate

much of the ArcGIS pre-processing functions through Python scripts. I employed Python outside of ArcGIS to perform some of the pre-processing and all of the data post-processing. Its list data structure is well-suited for analyzing raster datasets, such as LIDAR DEMs, as ASCII text files. It is also easy to create text files with Python in order to export analysis results and convert them back to rasters for further analysis in ArcGIS. Python's flexibility enables it to easily incorporate 3rd party tools, such as the Geospatial Data Abstraction Library (GDAL - http://www.gdal.org/), an open source project for raster analysis. Utilizing the list data structure and GDAL raster analysis tools, I developed custom Python scripts to identify stream locations, carve through artificial barriers in LiDAR DEMs, and calculate flow-direction error. I have included all of the Python scripts used in this research in Appendix B.

The automated data pre-processing and post-processing was conducted on multiple machines. Four desktop PCs of varying capabilities (from Pentium 4 with 1GB of RAM to Dual Core with 3GB) running Windows XP handled the pre-processing. On average, each of the 80 sample quadrangles took about 12 hours to pre-process. This entailed re-projecting inputs and clipping the state-wide 10-meter DEM by quadrangle boundaries. It also entailed burning stream locations into, filling, and calculating flow-direction for the 10-meter DEMs. The LiDAR pre-processing took up the significant majority of the processing time. The stream identification and carving of LiDAR DEMs took probably 95 – 99% of the total pre-processing time for each quadrangle. Three additional machines handled the post-processing (calculation of flow-direction error). Since I scripted the post-processing step entirely in Python (outside of ArcGIS) it could be performed on Linux machines. Two older PCs (a Pentium III and a Celeron, each

with 512 MB of RAM) installed with Linux (Ubuntu Jaunty 9.03) calculated flow-direction error at an average rate of four hours per quadrangle. A Lenovo Thinkpad (Dual-core with 3GB of RAM) also aided in the post-processing, at a rate of 1 hour per quadrangle.

4.2 Data Acquisition

I generated 10-meter DEMs for each sample quadrangle by clipping a mosaicked state-wide DEM by the respective quadrangle boundaries. The state-wide DEM was obtained from OGRIP, and is the same data accessible through the NED. The statewide DEM was created by OGRIP in piecemeal by merging 20-30 1:24,000 7.5-minute DLG hypsography ESRI shapefiles into a larger contour GIS layer, using a finite difference interpolation technique through ESRI ArcInfo's TopoToRaster command to convert the contours to a DEM, and finally mosaicking the new DEMs into a state-wide dataset (Ohio Office of Information Technology 2005).

I downloaded LiDAR 2.5-foot DEMs from OGRIP's OSIP (Ohio Statewide Imagery Program) site in zipped county-wide files (over 400GB). Each zipped file contained multiple 5,000 x 5,000 foot DEM tiles covering the particular county's extent. The naming scheme for the tiles was based on a tile's lower-left X and Y coordinate in the Ohio State Plane coordinate system. For example, the DEM for tile *n2180445* had its lower left coordinate at 218000, 445000 (X,Y) of the Ohio State Plane North coordinate system. I utilized this naming scheme to identify the LiDAR tiles that intersected each sample quadrangle. These DEMs were derived from LiDAR point data (in LAS format –

a standard format for LiDAR storage) gathered mainly during leaf-off conditions in spring 2006 by Woolpert Inc. for the State of Ohio (Woolpert, 2007).

I downloaded stream data, quadrangle boundaries, and hypsography from OGRIP in the form of 1:24,000 DLGs for each sample quadrangle. I then wrote custom Python scripts to convert these DLGs to ESRI shapefiles for use in the analysis.

4.3 10-meter DEM Pre-processing

Pre-processing took place one sample quadrangle at a time. To generate a flow-direction raster from a 10-meter DEM for a selected quadrangle I followed steps typically taken in hydrological GIS analyses to ensure proper surface water drainage. Once the state-wide 10-meter DEM had been clipped by the boundary of a sampled quadrangle, I set the analysis mask for the subsequent 10-meter processing to the clipped DEM; this ensured that all 10-meter outputs aligned spatially. Next, I re-projected the DLG stream vector dataset from a UTM projection to the appropriate Ohio State Plane Projection (either Ohio North – International Feet, or Ohio South – International Feet, depending on the sampled quadrangle) and converted it to a binary raster (stream cell = 1, all others = 0). I then burned the resulting stream raster into the DEM by raising all non-stream cell elevations in the DEM by 10 feet (Figure 4-1). Stream-burning can improve a DEM's ability to characterize the flow-direction of surface water by emphasizing the locations of streams (Hutchinson, 1989; Saunders, 1999). Next, the ArcGIS Fill command removed potential spurious sinks in the DEM. Sinks can appear in DEM data as artifacts of the production method and confound algorithms designed to route surface-water flow over

the landscape (Hutchison, 1989; Tarboton et al., 1991). Finally, the ArcGIS Flow-direction command created a flow-direction raster for the 10-meter DEM. The Flow-direction tool determines the direction of surface-water flow for each cell by identifying the cell in a 3x3 neighborhood with the steepest descent from the center cell. The result is a numeric value for the center cell corresponding to the direction of this steepest neighbor (Figure 4-2). This method is a single-flow algorithm, it cannot partition flow among multiple directions. It is commonly referred to as D8 (8 possible directions) (O'Callaghan and Mark, 1984; Greenlee, 1987; Jenson and Domingue, 1988).

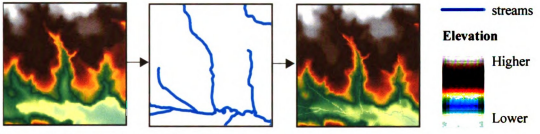


Figure 4-1: Burning streams into a DEM.

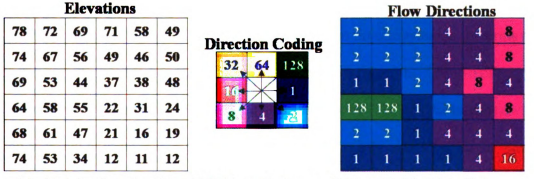


Figure 4-2: Deriving flow-direction with D8 (graphic source: ESRI ArcGIS 9.3 Help files)

While D8 has been shown to perform adequately in some studies (Skidmore 1989, Mouton 2007), it has been frequently maligned in literature as being too rigid and prone to introducing artifacts into the data, including long runs of parallel water flows (Wilson

et al. 2007, Raaflaub and Collins 2006, Venteris and Slater 2005, Schmidt and Persson 2003, Burrough et al. 2000, Jones 1998, Mitasova et al. 1995). However, for this analysis it was the logical choice for several reasons. First, the calculation of upstream flow accumulations (i.e., the number of cells flowing into a particular cell) in the LiDAR DEMs was essential for the comparison to the 10-meter DEM, and required a single-value flow-direction raster as input. Second, and most importantly, the D8 algorithm is the only available flow-direction tool in the ArcGIS Toolbox. Since ESRI is the world-leader in GIS market share (Daratech Inc., 2009), D8 is, by default, the most common flow-direction algorithm implemented in desktop GIS installations today, including the one used for this research. This popularity of D8, though arguably undeserved, makes the results of this research more applicable and relevant to the majority of current GIS users.

4.4 LiDAR DEM Processing

The pre-processing of LiDAR DEMs was the most time-consuming, in terms of person-hours and machine-hours, and complex piece of the research. The two primary challenges were developing a means to burn streams into the LiDAR DEMs and carving through artificial barriers that impede surface-water flow. Only after these issues were resolved could an adequate LiDAR flow-direction raster be generated and used to evaluate a 10-meter flow-direction raster.

The first step in pre-processing LiDAR DEMs for a sampled quadrangle was to identify which LiDAR tiles intersected the quadrangle. Within Python, I was able to use

the bounding Ohio State Plane coordinates of each 7.5-minute quadrangle and the coordinate-based nomenclature of the LiDAR tiles to determine which ones were contained within the quadrangle's extent. To avoid edge issues, I excluded tiles that overlapped the quadrangle boundaries. This choice sometimes led to a selection of 40 LiDAR tiles, but the majority of the time it selected 48 tiles (Figure 4-3).

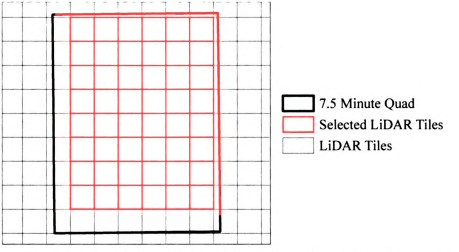


Figure 4-3: LiDAR tiles selected within a 7.5 minute quadrangle for pre-processing.

Once the LiDAR tiles had been identified, Python scripts began the process of deriving flow-direction rasters for each tile. Generally, the procedure to accomplish this was the same as that for the 10-meter DEMs. The code burned streams into a DEM to enforce appropriate surface-water drainage, filled spurious sinks, and calculated flow-direction with the D8 algorithm. However, LiDAR's positional accuracy and means of acquisition made stream burning difficult. The LiDAR DEM's map scales ranged from 1:100 to 1:1,000, far superior to the 1:24,000 map scales for the DLG hydrography. Though both datasets were developed to adhere to National Map Accuracy Standards, the difference in dataset map scales meant that the positions of the LiDAR elevation values

were more accurate than the positions of the DLG stream features. The LiDAR positional accuracy varied little about the 7-foot postings of the source LAS points (Woolpert Inc., 2007), whereas the DLG hydrography's position could be in error by as much as 40 feet or more (USGS, 1999), making stream burning by incorporating vector features impractical. It would be possible, if not probable, that a burned stream feature could run through a farm field, and not in the drainage ditch represented in the DEM (Figure 4-4).

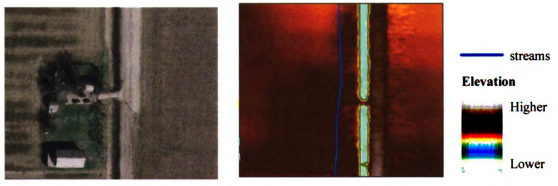


Figure 4-4: Left – aerial photograph. Right – LiDAR DEM and DLG stream feature in blue. DLG stream vector fails to align with LiDAR's drainage ditch due to differences in positional accuracy.

Figure 4-4 also illustrates how LiDAR DEMs can contain artificial barriers within streams. Notice how the drainage ditch is bisected in the middle of the figure. In this instance this feature indicates the presence of a culvert running under a driveway. This barrier is a byproduct of how LiDAR data is acquired. While the 10-meter DEMs were interpolated from contours, the LiDAR DEMs were interpolated from numerous tightly spaced points recorded by an airplane-mounted sensor. Each point represented a location where the sensor's beam struck the surface (or an object) and returned, allowing for a determination of elevation at that point. There is no way for the points and elevations

under the driveway bridge to be evaluated, creating a virtual barrier within the stream. Any effort to define surface-water flow for Figure 4-4 would err and assume that flow halted at this barrier; or worse, the entire ditch might be identified as a sink and filled by the ArcGIS filling algorithm. To resolve these two issues, I developed a method for identifying stream cells using only the LiDAR elevation values. I also implemented a method for carving through artificial barriers, like the bridge over the ditch.

4.4.1 LiDAR Stream Identification

The extraction of surface features from DEMs has been studied extensively. In terms of using LiDAR to identify stream features, five previous efforts stand out. MacMillan et al. (2003) described the capabilities and problems that can arise when trying to identify stream networks from high-resolution digital elevation data in Alberta, Canada. The authors used an approach similar to the one I implemented here; however, their design was more complex, appeared to have failed to resolve the in-stream barriers posed by bridges and overpasses, and focused on a single location. Leckie et al. (2005) used spectral reflectance, not elevation, of water and substrate material to identify stream features in Vancouver, British Columbia. Though they were able to achieve high accuracy rates in certain environments, spectral reflectance LiDAR data is less accessible than elevation data, making their method harder to implement elsewhere. Mason et al. (2006) developed an identification method based on edge-detection, similar to my approach. However, they focused on delta regions in Italy and Germany, whereas I implemented stream identification on areas of varying topography and land cover. Vogt

et al. (2003) attempted to identify ideal upslope accumulation thresholds for stream identification in different landscapes, but concluded that flat areas still required manual stream digitizing. The automated approach I developed performed best in flat areas. James et al. (2007) looked at contour crenulations as a means for locating streams and areas of concentrated surface flow. Despite their success, their method required that contour lines be adequately crenulated (i.e., the study-area was not flat), which would pose problems for flat agricultural areas dominated by ditch drainage, which characterizes most of northwest Ohio. Contours for this area are sparse and relatively un-crenulated. The stream identification method that I developed is tailored for such an environment.

A popular approach for deriving stream networks from DEMs is to identify a threshold for upland contributing cells, and code any cell that exceeds that threshold as a stream cell. Due to the presence of artificial barriers (discussed in 4.4.2) in LiDAR DEMs, this approach would not have worked well for much this study's area. Surface water flow would be halted as these barrier locations, limiting appropriate flow-accumulation tabulations.

I developed two methods for identifying stream features. One method relied on analyzing cell-neighborhoods around a center cell, and the other focused on transects of cells. In designing the two methods, I considered what distinguished stream locations in terms of topography. As discussed previously, I could not use vector features for streams; all I had were elevation values in the LiDAR DEMs. The basic assumption with both methods was that stream cells should be opposed by higher elevations in one direction (e.g., north to south), and similar elevations in the perpendicular direction (e.g.,

east to west). The higher elevations should represent the stream banks, while the similar elevations should represent the stream itself. I developed each method as a Python script.

The first method, called the Neighborhood Method, performed a neighborhood analysis on each cell in the DEM. The script iterated through each cell and analyzed a square 121-cell neighborhood (11 by 11) around the selected cell (call it *center).* It identified the maximum elevation value (call the cell with that value *max*) within that neighborhood and calculated the difference between that value and the elevation value of the selected cell (call that difference *max_minus_center).* If *max_minus_center* was greater than a user-specified minimum difference in elevation from a stream bank to the stream (call this value *min_elev_diff*; I typically employed values of 1 and 3 feet) the script then checked the value of the opposing cell (call it *opposite_max*) to see if the difference between its value and *center*'s also exceeded *min_elev_diff.* If both of these conditions were met, then the script deemed it possible that the center cell was a stream cell opposed by two stream banks. It then looked at opposing cells (call them *perpendicular_1* and *perpendicular_2*) in the direction perpendicular to the direction of *max* and *opposite_max* to see if their elevations were below *min_elev_diff.* If they were, then the script deemed *center* and its entire neighborhood as stream cells. See Figure 4-5 for a conceptual visual. Essentially, if a cell had two relatively tall points opposing it, these may have been stream banks. If the same cell had relatively flat slopes in the direction perpendicular to the tall points, then these may have been stream cells. This approach captured the basic topographic relationship I considered in defining stream locations. More formally:

IF *(max – center) > min_elev_diff*

AND *(opposite_max – center) > min_elev_diff*

AND *(perpendicular_1 – center) < min_elev_diff*

AND *(perpendicular_2 – center) < min_elev_diff*

THEN *neighborhood* = **stream cell**

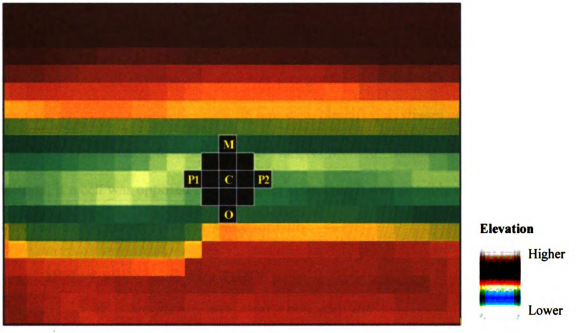


Figure 4-5: Cell positions evaluated by the Neighborhood Method. C = center, M = max, O = opposite max, R1 = perpendicular 1, R2 = perpendicular 2.

The second method for stream identification was called the Transect Method. Like the Neighborhood Method, it also performed a neighborhood analysis on each cell; but it was a much simpler technique, based on transects. The script began by moving through the DEM in a north-south direction looking for decreases in elevation. At each cell (call it *starter_cell)* the script checked *starter_cell*'s southern neighbor (call it *south_cell*) to see if its elevation value (call it *south_elev*) was less than that of *starter_cell*'s (call it *start_elev*). If it was, the script then evaluated a 10-cell transect

south of *starter_cell*. The script then identified the minimum elevation value (call it *min_elev*) and its cell location (call it *min_cell*) in the transect. If the difference between *start_elev* and *min_elev* was greater than a user-specified elevation difference (same as *min_elev_diff* in the Neighborhood Method, typically 1-3 feet) the script deemed the transect as potential stream cells. The script then continued to analyze the transect by looking for the maximum elevation value south of *min_cell* (call the value *max_elev_south_of_min* and the cell *max_cell_south_of_min)*. If the difference between *max_elev_south_of_min* and *min_elev* was also greater than *min_elev_diff*, then the cells from *starter_cell* to *max_cell_south_of_min* were classified as stream cells. Figure 4-6 provides a visual conceptualization of the Transect Method. Similar to the Neighborhood Method, if a cell was opposed by two higher cells, then it was possible that these cells were part of the stream network. More formally:

IF *start_elev* > *south_elev* **THEN**

IF (*start_elev* – *min_elev*) > *min_elev_diff*

AND (*max_elev_south_of_min* – *min_elev*) > *min_elev_diff*

THEN

start_cell **TO** *max_cell_south_of_min* = **stream cell**

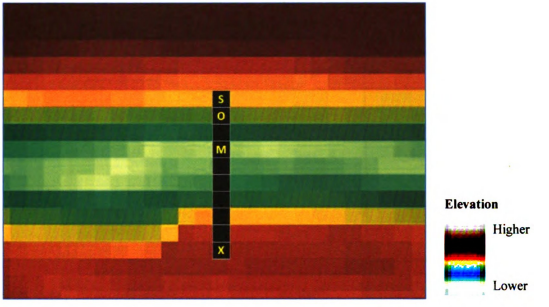


Figure 4-6: Transect analysis cells (in black); S = starter_cell, O = south_cell, M = min_cell, X = max_cell_south_of_min.

The Transect Method repeated this approach in three other directions: east to west (*west_cell* instead of *south_cell*), northwest to southeast, and southwest to northeast.

I evaluated the Neighborhood and Transect methods on ten LiDAR tiles of varying relief, drainage, and land-cover (Table 4-1 and Figure 4-7).

Site #	USGS 7.5 Minute Quadrangle	Change in Elevation (ft.)	Drainage	Land Cover
1	Warsaw	310	Natural	Forest
2	Cleveland South	132	Natural, urban	Urban
3	Columbus SE, NE	56	Natural, urban	Urban
4	Arlington	30	Natural	Agriculture
5	McClure	15	Ditch	Agriculture
6	Knoxville	485	Natural	Forest, residential
7	Huntsville	57	Ditch	Agriculture, forest
8	Genoa	24	Ditch, natural	Agriculture
9	Latty, Oakwood	30	Natural, ditch	Agriculture, forest
10	Stonecreek	320	Natural	Forest, agriculture

Table 4-1: Quadrangles where the LiDAR stream identification methods were tested.

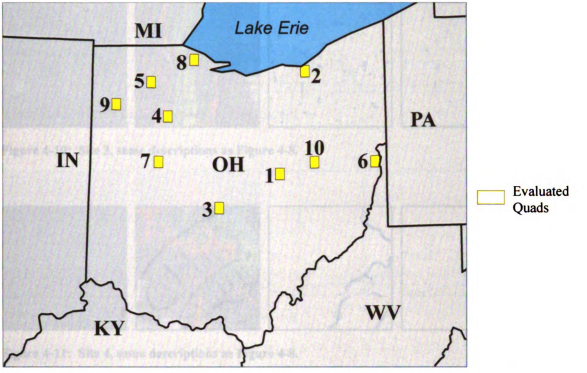


Figure 4-7: Quadrangles where the LiDAR stream identification methods were tested. Numbers refer to Site # in Table 4-1.

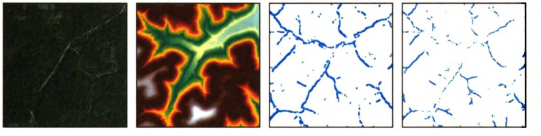


Figure 4-8: Site 1. Left – aerial; Left Center – LiDAR DEM; Right Center – streams identified by the Neighborhood Method; Right – by the Transect Method.

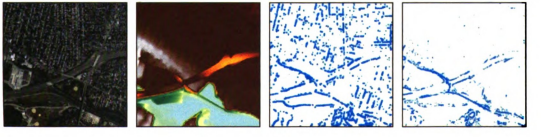


Figure 4-9: Site 2, same descriptions as Figure 4-8.

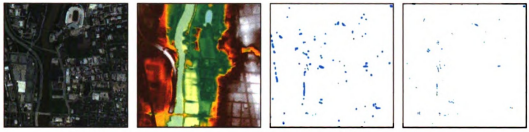


Figure 4-10: Site 3, same descriptions as Figure 4-8.

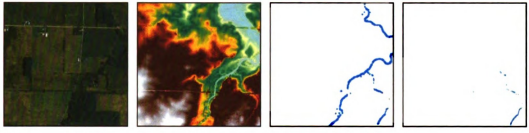


Figure 4-11: Site 4, same descriptions as Figure 4-8.

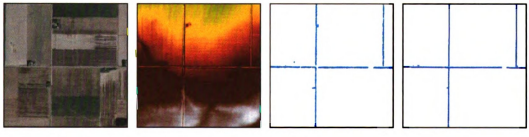


Figure 4-12: Site 5, same descriptions as Figure 4-8.

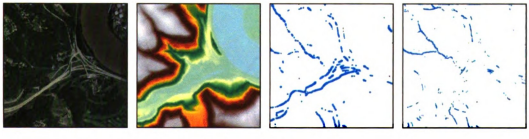


Figure 4-13: Site 6, same descriptions as Figure 4-8.

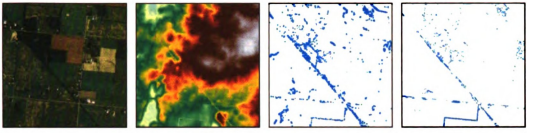


Figure 4-14: Site 7, same descriptions as Figure 4-8.

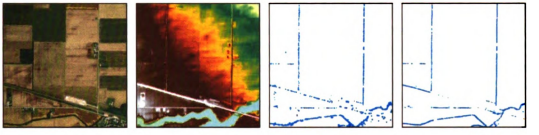


Figure 4-15: Site 8, same descriptions as Figure 4-8.

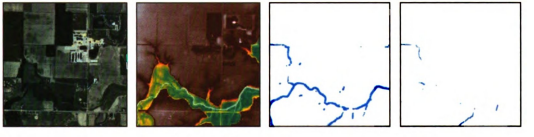


Figure 4-16: Site 9, same descriptions as Figure 4-8.

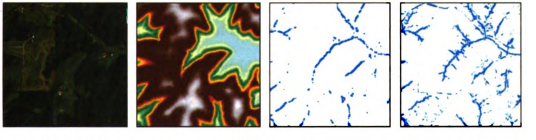


Figure 4-17: Site 10, same descriptions as Figure 4-8.

The evaluation of the ten sites indicated that the Neighborhood and Transect stream identification methods' performance varied over different landscapes. The Neighborhood Method performed adequately in the flatter, ditch-dominated study sites like 4, 5, 8, and 9. The Transect Method's best performance was in sites 5, 8, and 10. Neither method performed very well at sites 2, 3, 6, or 7. The main problem for the methods at sites 2 and 3 was the topographic complexity of an urban landscape. Site 2 was in the heart of Cleveland, and its streets and buildings posed too complex a surface for the Neighborhood and Transect methods to resolve. Site 3 captured the campus of Ohio State University, which also was too complex a surface for the methods to adequately identify stream locations. This difficulty in urban environments is the reason that study's sample of 80 quadrangles were filtered to avoid quadrangles that had even marginal percentages of urban land cover (> 10%). Neither method could process the width of the main rivers, such as the Olentangy in site 3 and the Ohio in site 6. Both methods expected the stream network not to exceed 10 cells (25 feet), which grossly underestimated the widths of the Olentangy and Ohio rivers, 261 and 1,100 feet respectively. The problem for the methods at site 7 was that they picked up too many non-stream cells. These errors were due to the forested landscape, which has been shown to pose problems in the analysis of LiDAR and other remotely sensed elevation products (Barber and Shortridge, 2005; Shortridge, 2006). Despite the fact that the LiDAR data was pre-processed by OGRIP to generate a bare-earth condition, artifact tree canopies remained in many LiDAR tiles and distorted the stream-to-bank relationship or hid a stream altogether. The adjacency of an errant elevation value of a tree canopy next to a

bare earth value created the illusion of a steep feature and, in the eyes of the two methods, a potential stream bank.

In comparison to each other, the Neighborhood Method performed better in areas where the drainage network was more sinuous (Figures 4-11 and 4-16), while the Transect Method better handled straight agricultural drainage ditches (Figures 4-12 and 4-15). The Transect Method's poorer performance on sinuous streams was due to its focus on the basic cardinal (N-S-E-W) and intermediate (NW-SW-SE-NE) directions. The directions of transects along a sinuous stream would include much more than the eight considered by the Transect Method; therefore it missed many transects along such streams and failed to identify them as part of the stream network. This limitation could potentially be resolved by incorporating all possible transect directions into the method, but this might render it computationally inefficient. However, the Transect Method's focus on those eight basic directions made it well-suited for identifying straight agricultural ditches. While the Neighborhood Method also did well in identifying these locations, the Transect Method outputs were cleaner and generated more quickly. The Neighborhood Method output tended to yield diamond-shaped stream neighborhoods, which in some instances gave the stream network messy and irregular boundaries. The boundaries of ditches identified by the Transect Method tended to be smooth and consistent, since its East-West analysis of a North-South trending ditch would essentially yield horizontal transects stacked neatly on top of each other (Figure 4-18). The processing of the Transect Method was also faster (1 min. 29 sec.) than the Neighborhood Method for site 5 (5 min. 42 sec.).

Figure 4-18: Left - rough edges of Neighborhood Method stream cells. Right – smoother edges of Transect Method.

4.4.2 Carving Through Artificial Barriers

A useful method to resolve artificial barriers in DEMs is to carve through them (Soille et al. 2003, Soille 2004a, Soille, 2004b). In this process, the elevations of the barrier cells are essentially reclassified in order to route the flow through them. However, in order to ensure that this step is only applied within stream cells, the stream network must first be identified. Soille's method looked at all sinks in a DEM; I was only interested in removing barriers within streams. I developed a process, which I refer to as the Carving Method, that searched within the stream cells identified by one of the previous methods (Neighborhood or Transect), and looked for sinks (cells in which each of the 8 adjacent neighbors have higher elevations – as would be the case at the base of the artificial barrier in Figure 4-4). The method then searched an expanding square neighborhood, within a user-specified maximum neighborhood size (100 cells as a default) for an elevation value smaller than that of the sink cell in question. If it found such a cell (presumably on the other side of the barrier), a path was charted from the sink to the lower cell along which the elevation of each cell was incrementally lowered to create a slope for water to flow along, essentially removing the barrier (Figure 4-19).

The method could be iterated over a user-specified number, as each carving could yield new sinks.

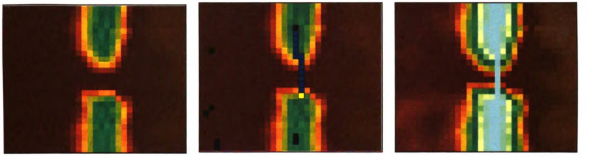


Figure 4-19: Left - the barrier in the DEM. Center – sinks in stream identified (in black), lower elevation found (in yellow), path charted (in blue). Right – carved DEM with elevation values incrementally lowered along path.

The implementation of the Carving Method yielded desired results in some locations, but introduced errors in others. The method was implemented in all ten of the stream identification study locations, but was most applicable in the agricultural drainage ditch-dominated sites (5, 7, and 8). The method required a prior identification of the stream network, rendering it essentially useless for the urban sites. The sites with natural drainage did not have many, if any, in-stream barriers. The method worked as designed for the site 5 location shown in Figure 4-19. Figure 4-20 illustrates how the method created clear and uninhibited paths within drainage ditches by removing even small barriers. The left side of Figure 4-21 shows how the Carving Method successfully routed water flow under a road (N-S direction); but it also shows how the method introduced undesired carvings. It is not likely that a NW-SE ditch flows under the road intersection, and aerial photography did not indicate the presence of a culvert running N-S along the right-edge of Figure 4-21 (in green).

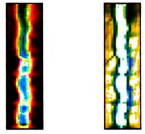


Figure 4-20: Carving freed up flow within stream features. Left - drainage ditch in pre-carved DEM. Right - carved drainage ditch.

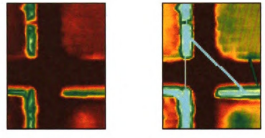


Figure 4-21: Carving resolved some barriers (created north-south connection) but also introduced potential errors (NW-SE connection, N-S connection at right).

The Carving Method was sensitive to even slight differences in elevation. When it encountered an artificial barrier in the stream, its primary goal was to find a neighbor with an elevation lower than that sink. As its search neighborhood expanded, it may have encountered cells with lower elevations that were not connected to the original sink, as illustrated by the right-side of Figure 4-21. A future, more "intelligent" Carving Method may be able to resolve this issue by considering the general direction of the stream in its neighborhood analysis. In the case of Figure 4-21, instead of searching in a neighborhood that incrementally expands in all directions, the expansion could occur in only the N-S direction, removing the possibility of the NW-SE carving that resulted in this analysis.

4.4.3 Implementing Stream Identification and Carving in LiDAR Processing

For each of the 80 sample quadrangles, I had to determine which stream identification method (Neighborhood or Transect) should be employed and what values should be chosen for identification parameters (minimum elevation difference and neighborhood size). As the results of the previous analysis indicate, the choices regarding the stream identification method and its parameters should be based upon the land-cover, relief, and the shape of the drainage features of a particular quadrangle. Even though these attributes may substantially vary across the sample of LiDAR tiles within a quadrangle (40 or 48), since the quadrangle pre-processing scripts operated on one quadrangle at a time the choices made for the stream identification parameters applied to all of a quadrangle's tiles. For example, the choice of minimum elevation difference in the stream identification method applied to all tiles, even if the method would have performed better with a different value in some of the tiles.

For the majority of quadrangles, I chose to use the Neighborhood Method with a minimum elevation difference of 3 feet, and a neighborhood size of 10 cells (25 feet by 25 feet). These choices held up well during post-processing inspection, particularly in quadrangles with high relief and little to no agriculture (Figure 4-22). For quadrangles dominated by agriculture, I still utilized the Neighborhood Method with a neighborhood size of 10 cells, but chose a more aggressive minimum elevation difference of 1 foot. In some instances these choices worked well, but in the flattest areas it still failed to adequately identify the stream network. I re-processed these quadrangles with the Transect Method, but results were still unsatisfactory. I then adjusted the pre-processing

Figu
well

Figur
areas

scripts so that both stream identification methods were integrated and employed on these quadrangles, which yielded more appropriate results (Figure 4-23).

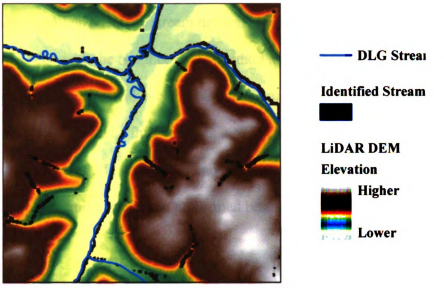


Figure 4-22: The Neighborhood Method with a minimum elevation difference of 3 feet performed well at identifying stream cells in the LiDAR DEM for areas of high relief and little to no agriculture.

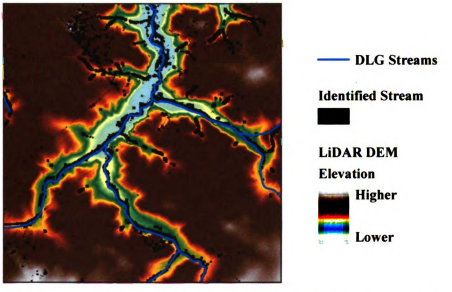


Figure 4-23: A combination of the Neighborhood and Transect methods performed best in flatter areas dominated by agriculture.

The LiDAR pre-processing was not a simple batch script that ran on its own for weeks. Many quadrangles were evaluated through the trial-and-error method described in the preceding paragraph. With a few quadrangles, I ran the stream identification algorithm three times until an adequate stream network was identified. For example, the first run might have been with the Neighborhood Method and a minimum elevation difference of 3 feet, a second run with a Neighborhood-Transect combination and an elevation difference of 1 foot, and a final run with a Neighborhood-Transect combination and an elevation difference of 2 feet. In these and all other similar instances, I inspected the results visually to determine which run performed best and should be utilized in the analysis.

Once the stream locations had been identified, the Carving Method removed artificial barriers within those streams. For all 80 of the sample quadrangles, I used a maximum neighborhood size of 100 cells and 3 iterations of carving, since the method's performance did not significantly vary across the 10-site evaluation.

4.4.4 Stream-burning and Sink-filling LiDAR DEMs

The last steps of the LiDAR pre-processing were to burn the identified stream locations into the carved DEM in the same manner used with the 10-meter DEMs: fill all sinks and calculate flow-direction through D8. It should be noted that the LiDAR pre-processing steps described above introduced artifacts into the final LiDAR DEM. Specifically, the stream identification and carving methods typically did not yield a perfectly contiguous stream network. Some of the identified cells were fragmented, as

seen in Figures 4-22 and 4-23. After the stream burning process these cells essentially became sinks that were subsequently filled in by the filling algorithm. The end results were flat table-like features in the final LiDAR DEM. In terms of the accuracy of elevation, this outcome was a clear error; however, it did not necessarily alter the general flow-direction of surface water over these cells, which was what I tried to measure in this research. The elevation values of the neighboring cells around those filled sinks usually remained unchanged after the fill process, meaning that a concentration of surface-water flow coming from the east still continued west along the flat cells towards the lower elevations west of those cells (Figure 4-24).

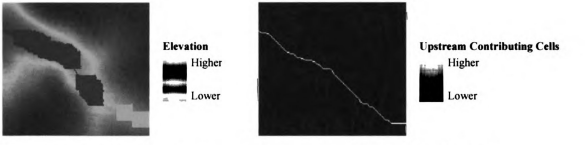


Figure 4-24: Left - the fill process created flat features in the LiDAR DEM. Right - Flow accumulation derived from the filled LiDAR DEM's flow-direction indicate that the flat features did not dramatically alter the flow-direction.

Once the LiDAR pre-processing steps described above had been completed for all of the LiDAR tiles for a sample quadrangle, the quadrangle was considered pre-processed and ready for the assessment of flow-direction error. See Appendix A for a model diagram of the quadrangle and LiDAR pre-processing.

4.5 Flow-direction Error

I calculated flow-direction error for each quadrangle by performing a cell-by-cell comparison of the 10-meter flow-direction raster to the LiDAR flow-direction rasters, assuming the LiDAR rasters as the reference condition. As with the pre-processing steps, I programmed the flow-direction error calculation as a Python script. The code iterated through each cell in the 10-meter flow-direction raster (around 1 million cells). The general approach was to compare the flow-direction value for each 10-meter cell to an aggregated flow-direction value for the neighborhood of LiDAR cells[1] that intersected the particular 10-meter cell (Figures 4-25 and 4-26). But how should the LiDAR flow be aggregated?

One method of aggregation would be to take an averaged direction from the LiDAR neighborhood cells, or calculate a single vector representing direction. But this would be misleading in instances where half of the cells drained in one direction and the other half drained in the opposite direction. If half of the cells drained north and the other half drained south, would an east or west vector of flow-direction be an accurate representation of surface-flow amongst the LiDAR cells?

Another method of aggregation would be to determine which flow-direction was most frequent amongst the intersecting LiDAR cells in Figure 4-26. However, this approach would not necessarily characterize where the majority of surface water flow

---

[1] The size of the intersecting LiDAR neighborhood was typically a 10x10 grid, though sometimes it was 9x9. The LiDAR cells did not perfectly align with 10-meter cells. This was due to the fact that the 10-meter width and height of the NED cell was not evenly divisible by the LiDAR resolution of 2.5 feet. Figures 4-26 through 4-34 illustrate the overlaying of LiDAR cells on a 10-meter cell and show them to be in perfect alignment for conceptual purposes only.

traveled amongst those LiDAR cells. It would be possible, for example, for 80% of the intersecting LiDAR cells to drain towards the northern edge of the 10-meter cell, only to be routed due east along that northern edge. In this instance, the predominant flow-direction value for the LiDAR cells would be north, even though the majority of the flow left the 10-meter cell area along the eastern edge.

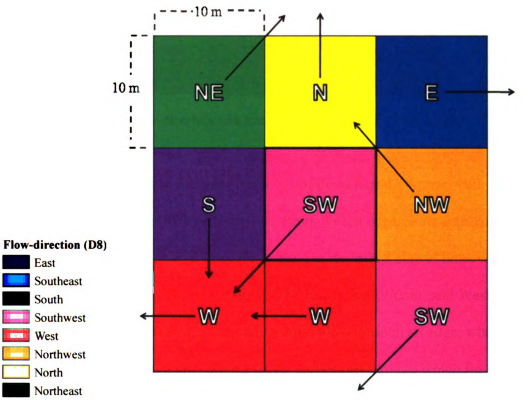


Figure 4-25: 3x3 neighborhood of 10-meter flow-direction cells. Cell to be analyzed for error at center.

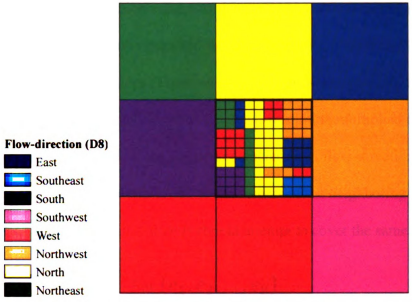


Figure 4-26: LiDAR flow-direction cells intersecting the selected 10-meter cell.

Another option would be to examine the edges of the selected 10-meter cell, and determine what percentage of LiDAR cells drained through each edge. This approach would avoid the misrepresentation of a frequency value, as described above; however, it would weight the cardinal directions too strongly. The North, East, South, and West edges all would have ten cells that could potentially route flow through an edge; whereas the diagonal directions would each only have a single corner cell through which flow could be routed. Furthermore, similar to the problem with looking solely at the mode of flow-direction values, if all of the LiDAR neighborhood cells flowed through the northern edge of the 10-meter cell is that a fair comparison to the 10-meter flow-direction value? It would still be possible for all of the LiDAR cells to drain through the northern edge, and then quickly divert due east as soon as they crossed the edge to ultimately drain out somewhere in the Northeast 10-meter cell.

The 10-meter flow-direction value was determined by calculating slopes between the center point of a cell and center points of that cell's eight immediate neighbors, and noting the cell with maximum slope (Equation 4-1). *Distance* was equal to the cell-resolution (10-meters) for cardinal neighbors, and cell-resolution multiplied by 1.414 for the diagonal neighbors. Therefore, the 10-meter calculation of flow-direction was a prediction from the center of the center cell to the centers of the neighboring cells. A comparative analysis with the LiDAR cells should attempt to cover the same geography.

$$\text{For } \{e_1, e_2, \ldots, e_8\}$$

$$\text{Flow-direction} = i, \text{ for } MAX\left[\left(\left|e_{center\ cell} - e_i\right|\right) \div distance \times 100\right]$$

Where e_i = elevation of the i-th neighbor

Equation 4-1: Formula for calculating flow-directions

The method I developed utilized an expanded neighborhood of LiDAR cells to determine the degree to which the 10-meter and LiDAR predictions for surface water flow agreed. This neighborhood extended to the center of each of the neighboring 10-meter cells, ensuring that the 10-meter flow direction value and aggregated LiDAR flow-direction value corresponded to the same geographic space. This expansion of the LiDAR neighborhood also gave each neighboring 10-meter cell an equal opportunity for flow to drain through its edges. As illustrated in Figure 4-27, each 10-meter neighboring cell had 10 LiDAR cells through which the LiDAR flow could have exited. For each of these neighboring cells, the method recorded the percentage of cells from the original 10 by 10 LiDAR neighborhood that drained through its edges (Figures 4-27 – 4-34). Since my goal was to evaluate the flow-direction value of the center 10-meter cell, I only

calculated the drainage percentage of LiDAR cells that intersected that center cell (Figure 4-26). The LiDAR cells in the expanded neighborhood (outside of the original LiDAR cells) were needed to trace the flow of the intersecting LiDAR cells out of the neighborhood, but these additional cells were not counted in the determination of aggregated LiDAR flow-direction value.

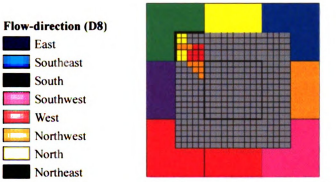


Figure 4-27: 0% of center LiDAR cells drain through the northwest edges of the neighborhood.

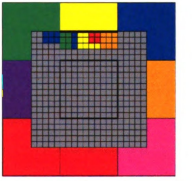


Figure 4-28: 0% of center LiDAR cells drain through the north edges of the neighborhood.

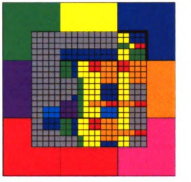


Figure 4-29: 89% of center LiDAR cells drain through the northeast edges of the neighborhood.

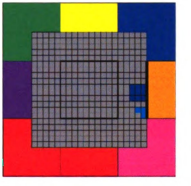


Figure 4-30: 0% of center LiDAR cells drain through the east edges of the neighborhood.

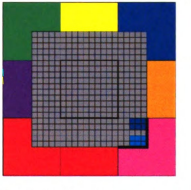


Figure 4-31: 0% of center LiDAR cells drain through the southeast edges of the neighborhood.

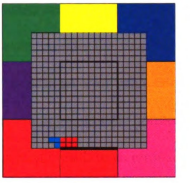


Figure 4-32: 0% of center LiDAR cells drain through the south edges of the neighborhood.

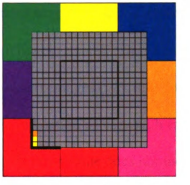


Figure 4-33: 0% of center LiDAR cells drain through the southwest edges of the neighborhood.

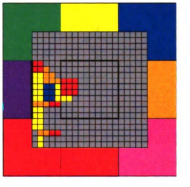


Figure 4-34: 11% percent of center LiDAR cells drain through the west edges of the neighborhood.

Next, the script used the flow percentages calculated for each 10-meter neighbor to weight an error score (Equation 4-2). Recall that in Figure 4-25 the 10-meter cell in question reported a flow-direction value of southwest. If all of the LiDAR cells that

intersected this 10-meter cell drained out of the southwest edges of the expanded LiDAR neighborhood, then the script would consider the 10-meter cell and LiDAR cells to be in perfect agreement. This would result in an error value of 0. If the result was the opposite condition, with the LiDAR cells draining out of the northeast edges of the neighborhood, this would be perfect disagreement and result in an error score of 4. Other scores would range between 1 and 3 increasing from agreement to error (Figure 4-35). In the case of the example that began with Figure 4-25, 89% of the center LiDAR cells drained through the northeast edges of the expanded neighborhood (a severe error: 4), while the remaining 11% drained through the western edges (a small error: 1). These error values were weighted by their respective flow percentages ((4 x 0.89) + (1 x 0.11)) to yield an aggregated flow error of 3.67, an indication of very high error in the 10-meter flow-direction estimation. Note that this error value does not indicate a specific direction of error. The number itself does not indicate whether the LiDAR flow would be closer to the North neighbor or the East neighbor in Figure 4-35, only that the error was high.

$$FDE = \sum_{i=1}^{8} p_i \times f_i$$

Where FDE = flow-direction error for the 10-meter cell

p_i = percentage of center LiDAR cells exiting out of the i-th neighbor 10-meter cell

f_i = error score (1-4) of the i-th neighbor as compared to the flow-direction of the 10-meter cell

Equation 4-2: Flow-direction error weighting.

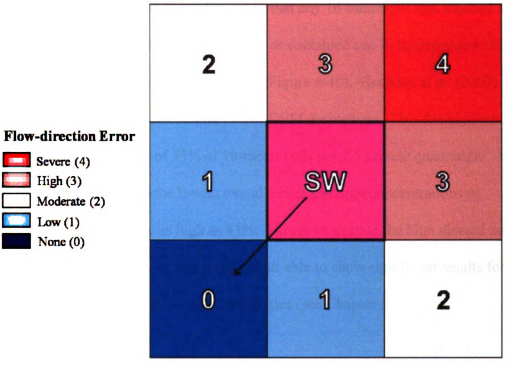


Figure 4-35: Potential flow error scores for assessment of 10-meter cell that flows southwest.

The script carried out the process described above for every 10-meter cell contained within a quadrangle's sampled LiDAR tiles. The process yielded maps illustrating the spatial distribution of flow-direction error for a particular quadrangle (Figure 4-36 – 4-39). Initial tests of this method revealed a spatial trend of high error in and along stream features (Figure 4-36). This trend was the result of the filling operation removing sinks in sections of stream within the LiDAR cells and modifying elevations across large sections to single elevation values, and subsequently single flow-direction values. In the flatter, agricultural areas entire fields were filled and reported the same flow-direction value for all their corresponding cells. These were clear distortions of reality and diminished the LiDAR DEM's utility as a reference dataset at these locations, despite its superior vertical and horizontal positional accuracy as compared to the 10-meter DEMs. To avoid the potential error that these flat cells could have introduced into

the analysis, I re-structured the Python script so that any 10-meter cell that touched a flat LiDAR cell directly (as an intersection of cells) or contained one in its expanded LiDAR neighborhood was discarded from the analysis (Figure 4-40). Esrskine et al. (2007) similarly removed flat cells from an analysis of DEM derivatives. This decision resulted in, on average, the discarding of 33% of 10-meter cells per 7.5 minute quadrangle. In some of the quadrangles with the lowest overall relief and high concentrations of agriculture, discard rates were as high as 83%. However, despite the high discard rates in these quadrangles, the statistical analysis was still able to show significant results for flow-direction error prediction in these quadrangles (see Chapter 5).

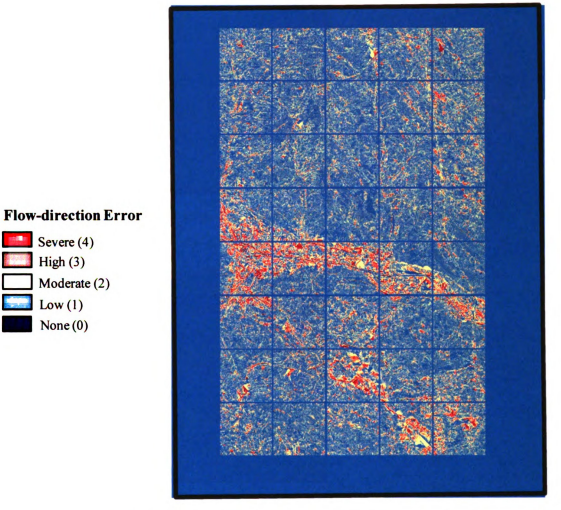


Figure 4-36: Flow-direction error map for a single 7.5-minute quadrangle. Blue grid lines are the result of avoiding edge issues in each LiDAR tile.

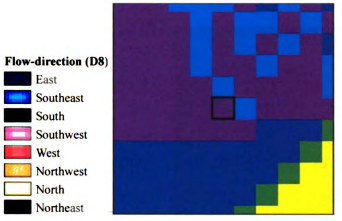


Figure 4-37: For the selected 10-meter cell, estimated flow-direction is to the south.

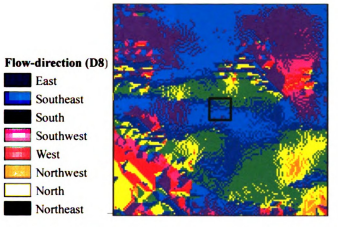


Figure 4-38: For the selected 10-meter cell, estimated LiDAR flow-direction trends mainly east and northeast.

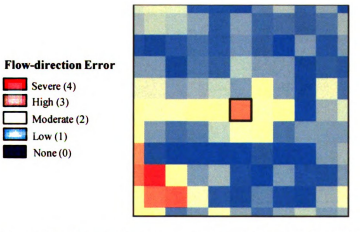


Figure 4-39: The flow-direction error is considered high, with a value of 2.98.

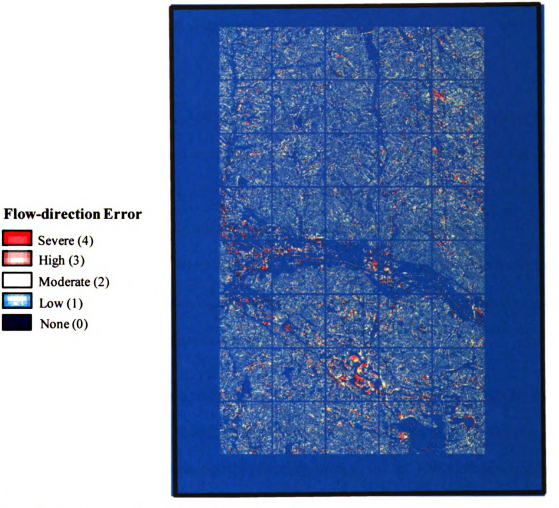


Figure 4-40: Flow-direction error map with flat cells discarded.

Chapter 5 Results and Analysis

The previous two chapters detailed this study's methodology for identifying the mean flow-direction error and its standard deviation. Using these data recorded for each of the 80 sample quadrangles, I conducted independent t-tests to assess whether the mean flow-direction error varied significantly across the agricultural and relief characterizations listed in Table 3-1. I also explored correlations between the mean flow-direction error and various quadrangle attributes, and evaluated linear regression models adjusted for residual spatial autocorrelation.

Overall flow-direction error among the 80 quadrangles was $M = 1.04$ ($SD = 0.85$). Figure 5-1 shows the spatial distribution of flow-direction error amongst the sample. There is a clear trend of increasing flow-direction error in the SE-NW direction amongst the quadrangles, mimicking the gradients for agriculture concentration and relief illustrated in Figures 3-1 and 3-2. These maps and the quantitative analysis I present here indicate that relief and agricultural concentration can be used to predict flow-direction error.

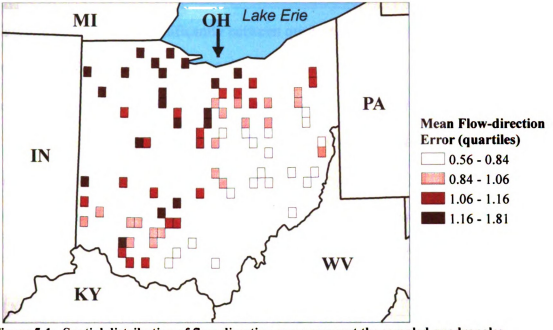


Figure 5-1: Spatial distribution of flow-direction error amongst the sampled quadrangles.

5.1 Independent T-tests

I expected that flow-direction error would be higher in areas of high agricultural concentration and low relief. More formally:

Hypotheses:
Ho: FDE High % Ag. $<$ FDE Low % Ag.
Ha: FDE High % Ag. $\geq$ FDE Low % Ag.

Ho: FDE High Relief $<$ FDE Low Relief
Ha: FDE High Relief $\geq$ FDE Low Relief

Where:
FDE = flow-direction error

Table 5-1 shows the overall mean flow-direction error for the sampled quadrangles and for the bins generated during the sampling process (see Table 3-1). Table 5-2 shows the

results of the independent t-tests calculated between these bins. As expected, mean flow-direction error differed significantly between quadrangles characterized as having a high percentage of agriculture ($M = 1.16$, $SD = 0.91$) and those with a relatively low percentage ($M = 0.93$, $SD = 0.79$), $t(78) = 4.33$, $p = < 0.001$. Similarly, quadrangles characterized as having high total relief ($M = 0.89$, $SD = 0.78$) versus those characterized as low in relief differed significantly ($M = 1.19$, $SD = 0.93$), $t(78) = 6.06$, $p = < 0.001$. On the basis of these results, both null hypotheses listed above were rejected.

Mean Flow-direction Error		**Relief**		**Overall**
		High (> 327 feet)	*Low (< 327 feet)*	
% Agriculture	*High (> 67.8%)*	*M*=1.04, *SD*=0.85, N=20	*M*=1.28, *SD*=0.97, N=20	*M*=1.16, *SD*=0.91, N=40
	Low (< 67.8%)	*M*=0.75, *SD*=0.70, N=20	*M*=1.11, *SD*=0.88, N=20	*M*=0.93, *SD*=0.79, N=40
Overall		*M*=0.89, *SD*=0.78, N=40	*M*=1.19, *SD*=0.93, N=40	***M*=1.04, *SD*=0.85, N=80**

Table 5-1: Flow-direction error results for the various sampling criteria combinations.

I also tested my expectations for the specific sampling bins, which I refer to as follows:

High relief, high % agriculture: HRHA
High relief, low % agriculture: HRLA
Low relief, high % agriculture: LRHA
Low relief, low % agriculture: LRLA

I expected that quadrangles with little relief and a high percentage of agriculture would contain the highest flow-direction error values, and vice versa. More formally:

Ho: FDE LRHA < FDE HRLA
Ha: FDE LRHA ≥ FDE HRLA

I also expected that when either relief or agricultural concentration was held constant the variation in the other would prove significant when compared to flow-direction error.

Ho: FDE LRHA < FDE LRLA
Ha: FDE LRHA ≥ FDE LRLA

Ho: FDE HRHA < FDE HRLA
Ha: FDE HRHA ≥ FDE HRLA

Ho: FDE LRHA < FDE HRHA
Ha: FDE LRHA ≥ FDE HRHA

Ho: FDE LRLA < FDE HRLA
Ha: FDE LRLA ≥ FDE HRLA

LRHA quadrangles reported the highest mean flow-direction error, $M = 1.28$, whereas the HRLA quadrangles reported the lowest value, $M = 0.75$. As shown in Table 5-2 these two means differed significantly, $t(38) = 7.16$, $p = < 0.001$ (a higher t score than any other pairing of mean flow-direction errors). These results show that when either relief or agricultural concentration was held constant, the change in the other corresponded significantly with changes in flow-direction error. There were not significant differences between HRHA and LRLA.

It appears that relief had a stronger impact on flow-direction error than the concentration of agriculture. The difference in overall mean-error between low relief quadrangles and high relief quadrangles was greater (1.19 – 0.89 = 0.3) than the difference between quadrangles with high concentrations of agriculture and those with low concentrations (1.16 – 0.93 = 0.23).

T-test results	HRHA	HRLA	LRHA	LRLA
HRHA				
HRLA	$t(38) = 6.29$, $p < 0.001$			
LRHA	$t(38) = 3.69$, $p < 0.001$	$t(38) = 7.16$, $p < 0.001$		
LRLA	$t(38) = 1.58$, $p = 0.12$	$t(38) = 5.64$, $p < 0.001$	$t(38) = 2.45$, $p = 0.02$	

Table 5-2: Independent t-test results between sampling criteria.

5.2 Correlations

Flow-direction error was highly correlated with agricultural concentration, relief, contour topology error, and the extent to which filling sinks in the LiDAR DEMs produced flat areas (Table 5-3). This last correlation was the strongest ($r = 0.85$). An initial reaction to such a strong correlation may be that these flat LiDAR cells distorted the true ground condition so dramatically that they yielded high values for flow-direction error. However, I discarded any 10-meter cell whose analysis neighborhood (Figure 4-27) contained even a single flat LiDAR cell. Therefore the error means used in calculating the correlation did not reflect these cells. The more likely explanation for the strong positive relationship is that the areas that contained a high number of flat cells in a depression-less DEM embodied the surface characteristics that confound flow-direction estimates. Flat cells in a filled DEM were more likely to occur in areas of high agricultural concentration and low relief, $r = 0.58$ and $r = -0.73$ respectively. These areas were dominated by agricultural ditches because the low relief provided little natural

drainage to route water away from fields and allow for crop production. Consequently, due to the presence of artificial barriers along them, these ditches had a greater chance of being identified as sinks than as natural drainage features and filled flat by the filling algorithm. A high percentage of flat cells for a quadrangle indicated low relief and a high concentration of agricultural ditches in that area. These areas essentially captured the "worst" of both worlds when trying to predict flow-directions; the low relief raised the uncertainty in flow-direction estimates, while the concentration of agricultural ditches increased the likelihood that drainage pathways were filled flat.

It is worth noting that all of the variables in Table 5-3 were significantly correlated. This fact raised issues of multicollinearity in the attempt to develop a linear regression model for flow-direction error.

Correlations	Flow-direction error	% row-crop agriculture	Relief	Intersections per contour	% of flat 10-meter cells discarded
Flow-direction error					
% row-crop agriculture	$r = 0.77$				
Relief	$r = -0.76$	$r = -0.68$			
Intersections per contour	$r = 0.70$	$r = 0.51$	$r = -0.60$		
% of 10-meter cells discarded due to flat LiDAR	$r = 0.85$	$r = 0.58$	$r = -0.73$	$r = 0.70$	

Table 5-3: Correlations between flow-direction error and other variables. N=80, all correlations were significantly different from zero ($p < 0.001$).

5.3 Spatial Regression

I attempted to identify the variables that best explained variability in flow-direction error through linear regression modeling using the R statistical package (ver. 2.7.1). Since flow-direction error was a spatial phenomenon, I adjusted the models to account for spatial autocorrelation in the model residuals.

Restating my general hypothesis, I expected that areas that were relatively flat, had high concentrations of agriculture, and whose DEMs were produced from contours that contained large numbers of topological errors (intersections) would have produced higher values of flow-direction error. More formally:

Equation: $FDE = a + \beta_{PA}PA + \beta_{TR}TR + \beta_{CI}CI$

Where:
FDE = flow-direction error
PA = % agriculture
TR = total relief
CI = contour intersections

Hypotheses:
Ho: $\beta_{PA} = 0$ Ha: $\beta_{PA} > 0$
Ho: $\beta_{TR} = 0$ Ha: $\beta_{TR} < 0$
Ho: $\beta_{CI} = 0$ Ha: $\beta_{CI} > 0$

Figure 5-2: Model 1 - initial linear regression model of flow-direction error.

I tested Model 1 (Figure 5-2) first, attempting to explain flow-direction error through relief, agriculture concentration, and contour intersections. Histograms of the model's variables indicated that a transformation was warranted for the contour

intersections variable; a logarithmic transformation produced the most normal distribution (Figure 5-3).

The results in Figure 5-4 confirmed the hypotheses of Model 1, with a solid overall model performance ($R^2 = 0.73$). However, the Moran's I value ($I = 0.3611, p = < 0.001$) for the model residuals indicated that they contained significant spatial autocorrelation (mapped in Figure 5-5 – note the clustering of shades), which was potentially inflating the R^2 and causing the horizontal cone shape (heteroscedasticity) of the residuals graphed in Figure 5-6.

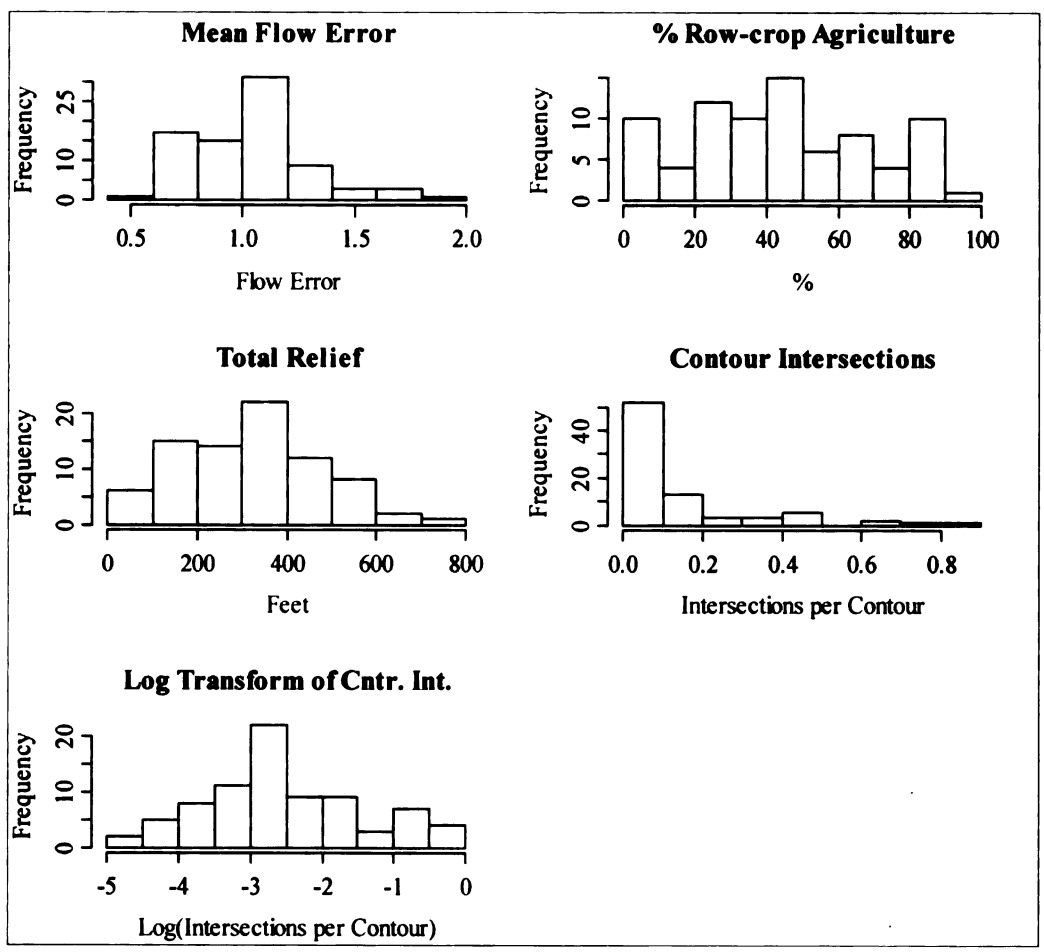


Figure 5-3: Histograms of regression variables from Model 1.

Significance of Terms:

Coefficients	Estimate	Std. Err.	*t* score	*p*-value
Intercept	1.1994774	0.0864874	13.869	< 2e-16
% row-crop agriculture	0.0043492	0.0008723	4.986	3.78e-06
Total relief (ft.)	-0.0005739	0.0001572	-3.652	0.000476
Log(int. per cntr.)	0.0621131	0.0194624	3.191	0.002059

Hypotheses:

β_{PA}: Reject Ho

β_{TR}: Reject Ho

β_{CI}: Reject Ho

Residuals:

Min.	25th pctl.	Median	75th pctl.	Max.
-0.309	-0.113	0.008	0.085	0.333

Residual standard error: 0.1426 on 76 degrees of freedom

Moran's I:

Observed	Expectation	Variance
0.3611	-0.0280	0.0080

p-value = 6.058e-06 (sampling test)

Model Performance:

R^2: 0.73

F-statistic: 71.01 on 3 and 76 DF, p-value: < 2.2e-16

Fitted values vs. Actual values correlation: 0.86

Aikaike Information Criterion (AIC): -78.743

Figure 5-4: Model 1 diagnostics.

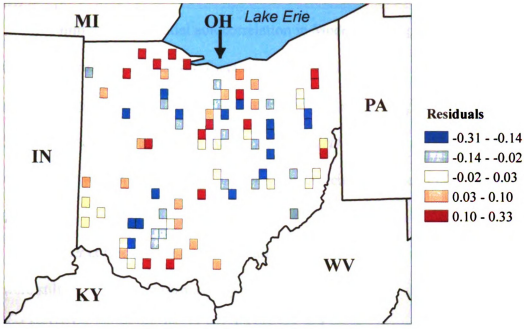


Figure 5-5: Mapped residuals of Model 1.

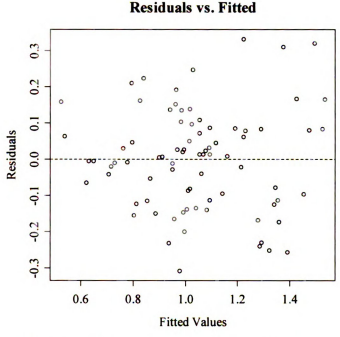


Figure 5-6: Residuals of Model 1, greater residual error of higher fitted values indicates potential heteroscedasticity.

I mitigated the spatial autocorrelation in Model 1 by implementing the SAR (simultaneous auto-regression) function of R's SPDEP (Spatial Dependence) module using each quadrangle's three nearest neighbors, inversely weighted by their distance. SAR corrected the estimates of the model coefficients, but was not designed to produce a new R^2. SAR corrected the residual heteroscedasticity illustrated in Figure 5-6 (Figure 5-7).

After correcting for residual spatial autocorrelation, the regression model's terms were still significant, confirming my hypotheses (Figure 5-9). The adjusted model's residual values were smaller in absolute terms and more concentrated than those for the original model; the range of residual values was smaller than in the original model (0.018 versus 0.024), as was the standard error (0.007 versus 0.143). Moran's I for the adjusted model's residuals was no longer statistically significant ($p = 0.28$). Additionally, the Lambda significance ($p < 0.001$), the slightly improved correlation of fitted values versus actual values (0.91 for SAR adjusted model versus 0.86 for original model), and the lower AIC value (-98.05 versus -78.4) all indicated that the coefficients of the new model successfully accounted for the presence of spatial autocorrelation in the residuals. Though not as clear as these numeric metrics of model improvement, the map of residuals from the adjusted model (Figure 5-8) shows a slight improvement as several of the darker clusters in the original residual map (Figure 5-5) were broken up.

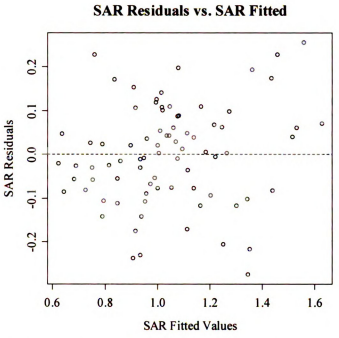


Figure 5-7: Residuals of Model 1, corrected for spatial autocorrelation, no apparent heteroscedasticity.

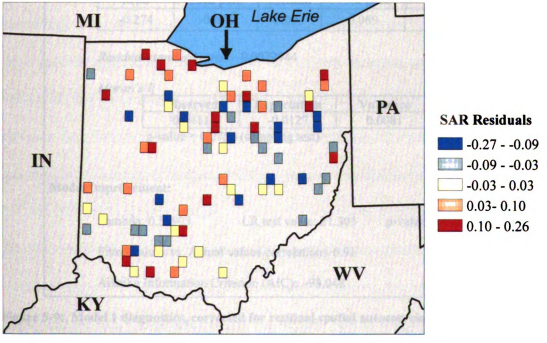


Figure 5-8: Mapped residuals, corrected for spatial autocorrelation.

Significance of Terms:

Coefficients	Estimate	Std. Err.	*t* score	*p*-value
Intercept	1.11910155	0.0832270	13.446	< 2e-16
% row-crop agriculture	0.00391785	0.0008740	4.482	7.384e-06
Total relief (ft.)	-0.00051557	0.0001362	-3.783	0.0001545
Log(int. per cntr.)	0.03246648	0.0151490	2.143	0.0321021

Hypotheses:

β_{PA}: Reject Ho

β_{TR}: Reject Ho

β_{CI}: Reject Ho

Residuals:

Min.	25th pctl.	Median	75th pctl.	Max.
-0.274	-0.078	0.003	0.069	0.256

Residual standard error: 0.0072944

Moran's I:

Observed	Expectation	Variance
0.0411	-0.0127	0.0081

p-value = 0.2755 (sampling test)

Model Improvement:

Lambda: 0.59923 LR test value: 21.305 p-value: 3.9180e-06

Fitted values vs. Actual values correlation: 0.91

Aikaike Information Criterion (AIC): -98.048

Figure 5-9: Model 1 diagnostics, corrected for residual spatial autocorrelation.

By most measures, Model 1 performed very well. However, multicollinearity in the independent variables may have inflated the model's performance. As indicated in

Table 5-3, there was a strong, negative correlation between agricultural concentration and relief (r = -0.68). Relief and the number of intersections per contour was also strongly negatively correlated (r = -0.60). Agricultural concentration and contour intersections were moderately positively correlated (r = 0.51). Therefore, I explored other regression models to see if flow-direction error could be described through other terms and combinations.

All of the independent variables in Table 5-3 were significantly correlated, so any regression model employing them would have had multicollinearity issues. To consider the power of each variable individually on flow-direction error, as well as to assess some additional variables, I developed six alternative regression models (Table 5-4). Models 2 through 4 each employed one of the terms in the original model (Model 1 – Figure 5-2): percentage row-cop agriculture, total relief, or intersections per contour. While all of these models explained significant variability in flow-direction error, the results confirmed that percentage row-crop agriculture and total relief explained more of the variance than intersections per contour. Model 5 employed the percentage of a quadrangle's 10-meter cells discarded from the analysis due to the presence of flat LiDAR cells (flat cell percentage). This model was the best single predictor of flow-direction error (R^2 = 0.72). This independent variable was most closely associated with elements that confound DEM prediction of flow-direction: drainage ditches (potential sinks) that are likely to be filled in and flat terrain. Model 6 was a modification of the original regression model (Model 1) and replaced total relief with flat cell percentage, since total relief was the independent variable most highly correlated with flat cell

percentage. While multicollinearity was still an issue, Model 6 performed much better than the original model on all measures. Model 7 employed a relatively simple combination of terms and performed nearly as well as the original model.

That all of these regression models yielded significant results begs the question of which model should those interested in predicting flow-direction error elsewhere employ. Solely by the numbers, Model 6 should be the predictor of choice. However, the terms of that model are relatively expensive to acquire. Specifically, the percentage of discarded flat cells requires access to LiDAR data, which is limited around the globe, and requires a significant amount of pre-processing and hard drive space. The most economical choice is Model 7, since land cover and elevation datasets are freely available for the entire U.S. and most of the world. Any of the models presented here are statistically significant predictors of flow-direction error. However, the results show that any implementation of these models requires a correction for spatial autocorrelation.

Model #	Model terms	Residual Heteroscedasticity	R^2	F Statistic	Corr. of SAR Fitted vs. Actual	SAR AIC	Sig. Terms
2	% ag.	no	0.59	114.0 ($p < 0.001$)	0.90	-83.0	all
3	total relief	yes	0.58	109.6 ($p < 0.001$)	0.89	-77.9	all
4	log(int. p. cnt.)	no	0.47	70.9 ($p < 0.001$)	0.88	-65.6	all
5	flat cell pct.	yes	0.72	208.4 ($p < 0.001$)	0.89	-88.9	all
6	% ag. flat cell pct. log(int. p. cnt.)	no	0.84	142.3 ($p < 0.001$)	0.93	-124.1	all
7	%ag total relief	yes	0.69	109.6 ($p < 0.001$)	0.91	-95.7	all

Table 5-4: Alternative linear regression models, and measures of performance.

Chapter 6 Conclusion

6.1 Summary

For this research, I sought to quantify error in estimates of surface water flow-direction derived from USGS 10-meter DEMs and explain how that error varied across different geographic landscapes. I developed a method by which finer resolution DEMs (LiDAR - 2.5-foot) can be utilized to evaluate estimates of flow-direction derived from coarser DEMs (USGS - 10-meter) on a cell-by-cell basis. I implemented this approach on 80 7.5-minute USGS quadrangles in Ohio, stratified by relief and agricultural concentration. As I expected, statistical analyses revealed that flow-direction error was strongly, negatively correlated with total relief, and strongly, positively correlated with agricultural concentration. The number of intersections per feature in the source contours that produced the USGS DEMs, and the percentage of sinks filled flat by the ArcGIS filling algorithm were also strongly, positively correlated with flow-direction error. A regression model employing agricultural concentration, contour intersections, and sink filling as independent variables explained 84% of the variance in flow-direction error. A simpler model employing only agricultural concentration and total relief still explained 69% of flow-direction error variance. Both of these regression models had to be adjusted to correct for residual spatial auto-correlation, and both contained some degree of multicollinearity.

The pre-processing of the LiDAR data was costly and time-consuming, requiring nearly one terabyte of hard drive space and four microcomputers running constantly for over a month. The most challenging and time-consuming part of pre-processing was identifying stream locations within the LiDAR DEMs in order to "burn" a hydrological network into the surfaces and enforce a proper drainage network. This step required the development of custom Python scripts that read LiDAR elevation data into Python lists, utilized neighborhood analyses to locate stream cells, and carved through artificial barriers within the identified stream networks. Qualitative tests indicated that these custom methods performed best in flat agricultural areas, where the enforcement of proper drainage networks was most needed.

6.2 Significance

While the relationship between DEM error, relief, and agriculture has been studied before, the specific attention to flow-direction at field scales and over such a large geographic sample make this research unique. Previous efforts studied the effect of resolution or interpolation methods on DEM derivatives such as slope, aspect, and soil moisture. Other research has evaluated surface flow from DEMs by comparing catchment boundaries derived from upland flow accumulations in relatively small study sites. In this thesis, however, I have presented a new method for evaluating flow-direction, conducted this evaluation at the DEM cell level, and described its variation across the Ohio landscape, which exhibits a diversity of topography and land cover.

This thesis makes several contributions to the field of geographic information science. The quantification of flow-direction error adds a new measure by which DEMs can be evaluated. The cell-by-cell evaluation of coarser data with finer data could be applied to any number of raster datasets. The new method for evaluating surface water flow-direction could be extended to other directional datasets, such as aspect, wind-direction, or groundwater flow-direction. Furthermore, the methods I developed and tested to locate stream features in LiDAR data, and my implementation of a sink carving method could be utilized in other hydrological studies of DEMs. This contribution in particular may prove to be the most significant as LiDAR DEMs replace contour-interpolated DEMs as the standard in spatial analysis.

At a more practical level, the cell-level specificity of this thesis informs field-level analyses that would be impractical through previously described methods. Additionally, its broad scope informs analysis across varying topographic and land cover environments, making it applicable to a large audience of users. One group of DEM users that will benefit from this research are soil conservationists. Predictions of surface water flow-direction are critical to identifying specific field-level locations experiencing soil erosion. Inaccurate estimates will focus conservation efforts on the wrong locations. A conservationist may look for an erosive gully on the eastern edge of a field, when in actuality it may be on the western edge. Having a quantified value of flow-direction uncertainty for a particular location will aid soil conservationists in efficiently marshaling their time and resources. They will be able to gauge how much field verification is needed prior to targeting soil conservation practices. For example, if staff for the Lucas

County Soil and Water Conservation District utilized a USGS 10-meter DEM in an erosion model, as can be done with the Revised Universal Soil Loss Equation (Wu et al. 2005, Ouyang et al. 2005), to identify at-risk locations in a hilly area with little agricultural presence, they could be sufficiently confident in assuming that surface water flow was appropriately simulated by the model. If, on the other hand, district staff utilized the same approach to identify an at-risk location in a flat agricultural area (typical for that region of Ohio) they should exercise caution in assuming that the location was significantly eroding, and conduct a thorough field evaluation prior to the installation of remediation measures.

Soil conservationists are just one specific group that could benefit from this research. Environmental engineers deriving stream networks, stream biologists exploring non-point source pollution loading from small local catchments, civil engineers re-evaluating floodplain maps, or any user employing DEMs for field-level hydrological analyses will benefit from knowing the uncertainty in simulated flow-direction at a particular location.

Martinoni and Bernhard (1998) argued that greater documentation of error in DEMs, including assessment of derivative error, is needed to ensure their proper use and analysis. The quantification of flow-direction error could be a new metadata standard for DEMs, and support Martinoni's and Bernhard's goal.

6.3 Future Research

While I was able to successfully quantify and explain flow-direction error in terms of topographic and land cover characteristics, the process generated additional topics that warrant further research. Stream locations could potentially be better identified and processed; smoothed LiDAR DEMs may prove to be better reference datasets; additional flow-direction algorithms should be utilized in the assessment of flow-direction error; and the scale at which flow-direction between 10-meter and LiDAR DEMs changes from error to agreement should be evaluated.

The methods I developed to identify and burn stream locations in the LiDAR DEMs has several limitations. First, they cannot identify wide cross-sections of rivers. The analysis neighborhood is restricted to a 10-cell (25-foot by 25-foot) neighborhood and cannot locate the necessary stream bank elevations for large rivers in such a small window (Figures 4-10 and 4-13). Second, they require the user to specify single thresholds for stream bank elevation and neighborhood size that in actuality should vary depending on the relief and land cover of a particular area. In this research, I had to process several quadrangles three times with different threshold inputs until an acceptable output was realized. Third, the carving process to remove artificial barriers in the streams sometimes carved in inappropriate directions. For example, instead of carving through an east-west running bridge to connect a north-south drainage ditch, it sometimes carved in a northwest-southeast direction to connect the north-south ditch to a section of an east-west ditch on the opposite side of the road (Figure 4-21). Revised methods could accommodate wider rivers, automate input selections by analyzing total relief and land

cover

discer

metho

repres

numb

an ave

data. E

evalua

DEMs

direct

impac

preser

DEM

preser

hypot

evalu

servi

well

alter

cover percentages for given locations, and carve in a more intelligent manner by discerning the direction of the stream features that need to be connected. These revised methods would likely improve the precision of flow-direction error estimates.

LiDAR DEMs can contain significant noise that distorts the surface representation. These distortions can be limited by "smoothing" the DEM through any number of filtering methods. A common approach is to change each cell's elevation to an average of its nearest neighbors. This method reduces any large discrepancies in the data, but can also diminish valid features on the surface. I was particularly interested in evaluating flow-direction in flat areas; therefore, I was worried that smoothing LiDAR DEMs in these areas may distort the subtle surface features critical to calculating flow-direction. As mentioned in Chapter 3, subtle changes in elevation can have a significant impact on flow-direction if that area is flat. I chose to honor the original DEM and preserve these features, at the risk of noise affecting the model outputs. The LiDAR DEMs utilized in this thesis were unaltered from their downloaded format. Despite the presence of noise in some of the LiDAR DEMs, I was still able to confirm my hypotheses. However, since smoothing is a common procedure, it would be worth evaluating how the results of this study may change with smoothed LiDAR DEMs serving as the reference dataset.

While the D8 flow-direction algorithm allowed me to confirm my hypotheses, its well documented limitations warrant an evaluation of flow-direction error utilizing alternative and more accepted methods. Multi-direction methods such as MF by Quinn et

al. (1991) or multi/single direction hybrids like Tarboton's D-Infinity (1997) should be utilized in a cell-by-cell fashion as used here.

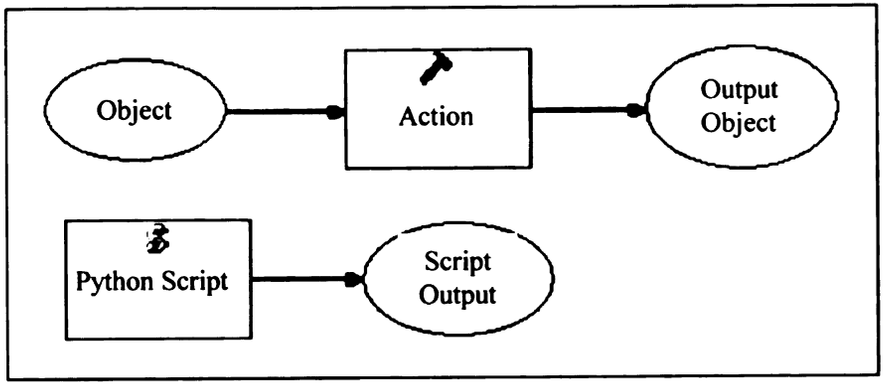
I calculated flow-direction error on a cell-by-cell basis, but at what extent, if ever, does that error change to agreement? The 10-meter cell and its corresponding LiDAR cells may be in disagreement at the immediate neighborhood level illustrated in 4-35, but perhaps the predicted flows converge at the next larger neighborhood. It is possible that flow-direction error is generally confined to a small local neighborhood of cells. It is also possible that simulated flows travel in different direction over large areas (Figures 3-8 through 3-10). This relationship should be explored to better describe the impact flow-direction error may have on hydrological analyses.

The availability of high-resolution LiDAR DEMs will continue to grow as GIS users demand finer data and as digital storage space becomes cheaper. LiDAR DEMs will likely replace the coarser, contour-interpolated and stereoscopic DEMs produced by the USGS as the standard digital elevation products for spatial analysis. As that transition takes place, research should document how existing DEM-based analyses may change with the incorporation of the finer resolution data. This research is a contribution to that effort.

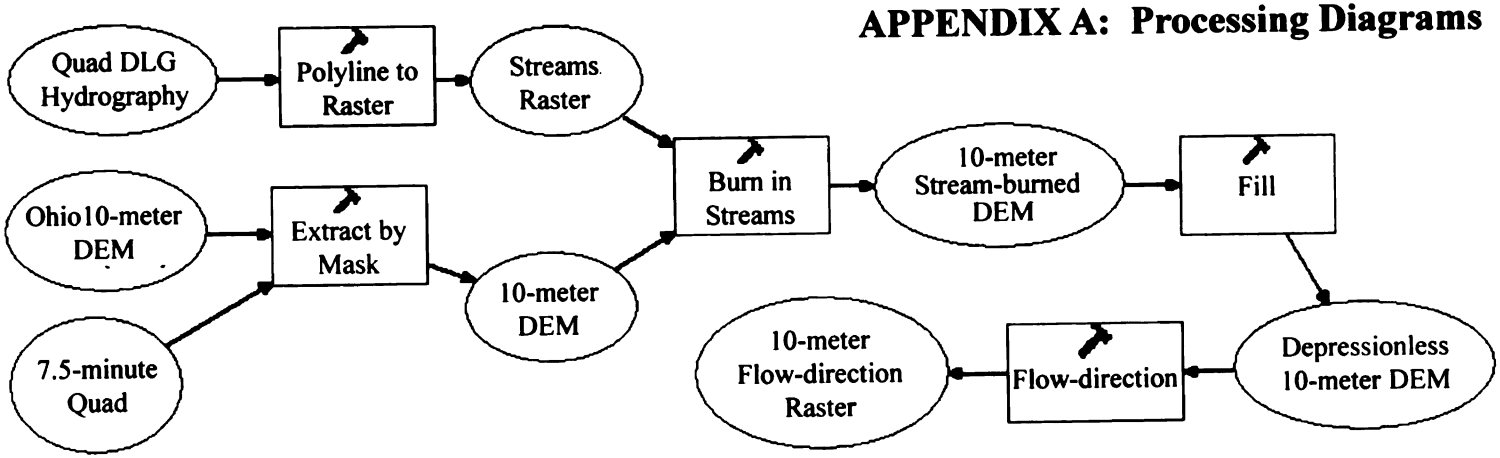
APPENDICES

APPENDIX A: Processing Diagrams

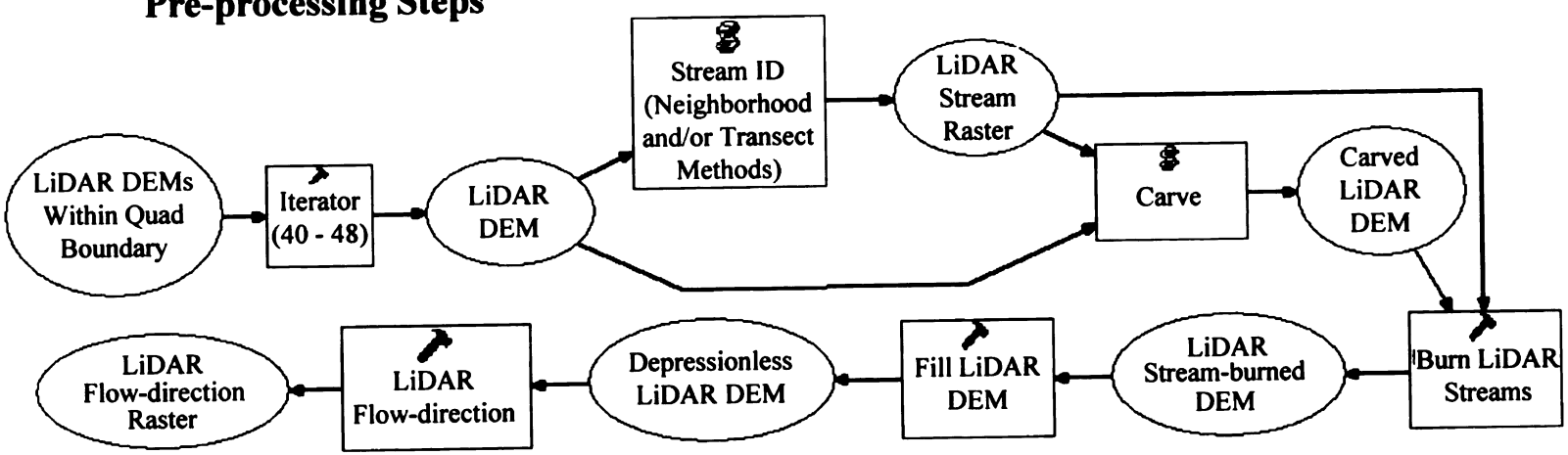
Diagram Legend



APPENDIX A: Processing Diagrams

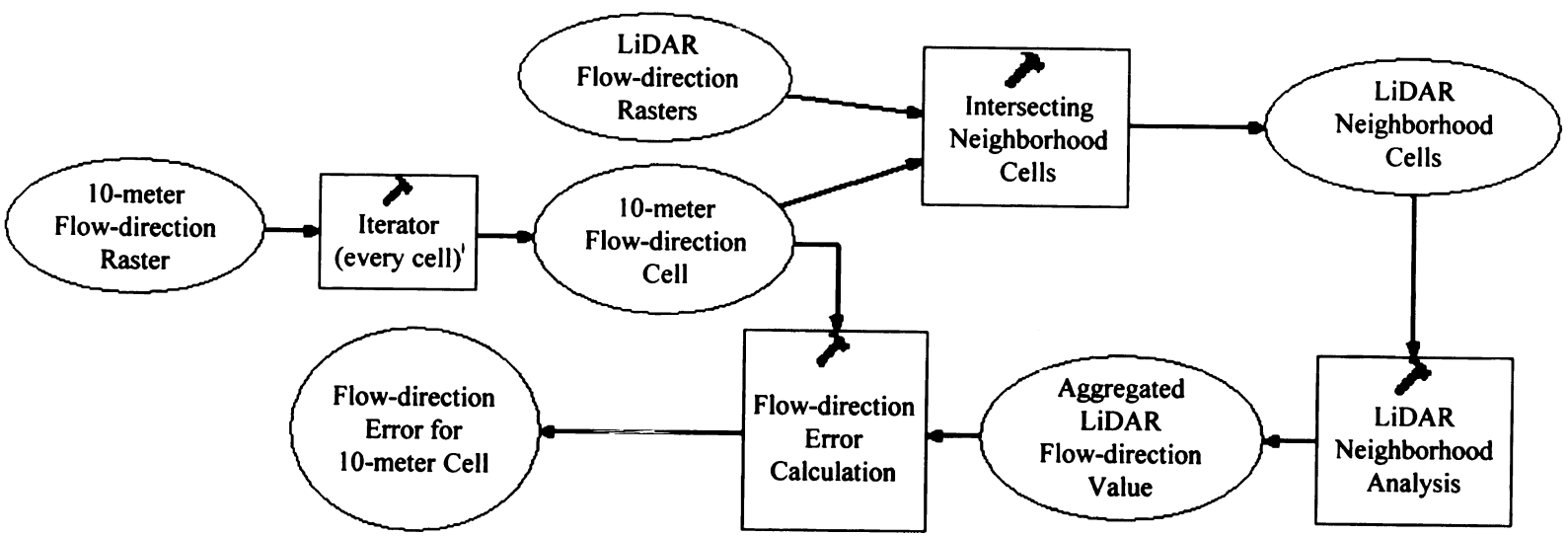


Pre-processing Steps



APPENDIX A: Processing Diagrams

Post-processing Steps



APPENDIX B – Python Code

raster_analysis.py

```
1  __all__ = ['raster2array', 'rastertext2list', 'extract_header']
2  #raster_analysis.py
3  #Author: Glenn O'Neil
4  #Date: September 2009
5  #This Python module provides tools for converting ArcGIS rasters
6  #between multiple formats, including NumPy arrays for analysis with GDAL
7  #and Python lists. It also includes a function for extracting a neighborhood
8  #from a larger list or array of values.
9
10 def raster2array (raster):
11    """Takes a raster path and converts it to a NumPy array using GDAL"""
12
13    #check to see if gdal and gdalconst modules have been imported
14    if 'gdal' not in globals():
15       from osgeo import gdal
16    if 'gdalconst' not in globals():
17       from osgeo import gdalconst
18
19    raster_ds = gdal.Open(raster, gdalconst.GA_ReadOnly)
20    raster_band = raster_ds.GetRasterBand(1)
21    raster_array = raster_band.ReadAsArray()
22    del raster_ds, raster_band
23    return raster_array
24
25
26 def array2raster (raster_array, raster_ds_template, raster_ds_out_path, raster_ds_out_format,
   scratch_folder):
27    """
28    Takes a NumPy array and converts it to an ArcINFO raster using GDAL
29          raster_array = the NumPy array to be converted
          raster_ds_template = the raster dataset that will serve as a template for the output raster
30 dataset
31          raster_ds_out_path = the full path of the output raster dataset
32          raster_ds_out_format = string of the the pixel type of the output raster dataset:
33                - 1_BIT: A 1-bit unsigned interger. Values can be 0 or 1.
34                - 2_BIT: A 2-bit unsigned integer. The values supported can be from 0 to 3.
35                - 4_BIT: A 4-bit nsigned integer. The values supported can be from 0 to 15.
36                - 8_BIT_UNSIGNED: An unsigned 8-bit data type. The values supported can be
   from 0 to 255.
37                - 8_BIT_SIGNED: An signed 8-bit data type. The values supported can be from -
   128 to 127.
```

```
38              - 16_BIT_UNSIGNED: An unsigned 16-bit data type. The values supported can
    be from 0 to 65,535.
39              - 16_BIT_SIGNED: An signed 16-bit data type. The values supported can be
    from -32,768 to 32,767.
40              - 32_BIT_UNSIGNED: An unsigned 32-bit data type. The values supported can
    be from 0 to 4,294,967,295.
41              - 32_BIT_SIGNED: An signed 32-bit data type. The values supported can be
    from -2,147,483,648 to 2,147,483,647.
42              - 32_BIT_FLOAT: A 32-bit data type supporting decimals.
43              - 64_BIT: A 64-bit data type supporting decimals.
44          scratch_folder = the folder where the temporary datasets will be stored
45
46      """
47
48      #check to see if the necessary modules have been imported
49      if 'gdal' not in globals():
50          from osgeo import gdal
51      if 'gdalconst' not in globals():
52          from osgeo import gdalconst
53      if 'arcgisscripting' not in globals():
54          import arcgisscripting
55      if 'gp' not in globals():
56          gp = arcgisscripting.create(9.3)
57      elif str(type(gp)) != "<type 'geoprocessing object'>":
58          del gp
59          gp = arcgisscripting.create(9.3)
60
61      #must first convert the template dataset to a GeoTIFF b/c GDAL can't writ to ESRI Grid
    format
62      template_copy = scratch_folder + "\\temp.tif"
63      gp.CopyRaster_management(raster_ds_template, template_copy, "#", "#", "#", "NONE",
    "NONE", raster_ds_out_format)
64
65      #read the copied TIFF file into a GDAL dataset and write the raster_array contents into it
66      raster_ds = gdal.Open(template_copy, gdalconst.GA_Update)
67      raster_band = raster_ds.GetRasterBand(1)
68      raster_band.WriteArray(raster_array)
69      raster_band.FlushCache()
70      raster_ds.FlushCache()
71
72      del raster_ds, raster_band
73
74      #copy the modified template_copy back to an ESRI Grid
75      gp.CopyRaster_management(template_copy, raster_ds_out_path)
76      gp.delete_management(template_copy)
77      del template_copy
```

```
78
79
80
81  def rastertext2list (raster_ascii_path, output_datatype='int'):
82      """Takes a raster as a text file and converts it to a list
83          raster_ascii_path = full path of an ASCII text file converted from a raster
84          output_datatype = the datatype of the values in the output list.
85                      Integer is the default, user can specify 'float' or 'string'
86
87      """
88      #read in the elevation ascii file
89      raster_ascii = open(raster_ascii_path, 'r')
90
91      #initiate the list that will store all of the text values
92      raster_list = []
93      raster_ascii.seek(0,0)
94      for eachLine in raster_ascii:
95          raster_list.append(eachLine.split(" "))
96      raster_ascii.close()
97
98      #get rid of the descriptive info (the first six rows)
99      for i in range(6):
100         raster_list.remove(raster_list[0])
101
102     #get rid of the newline character at the end of each row
103     for row in raster_list:
104         row.remove(row[len(row) - 1])
105
106     #convert the textfile values (string) to whatever numeric type (integer or floating type) was
    specified
107     if output_datatype == 'float' or output_datatype == 'Float':
108         for row in raster_list:
109             for i in range(len(row)):
110                 row[i] = float(row[i])
111     elif output_datatype == 'string' or output_datatype == 'String':
112         pass
113     else:  #the default will be integer
114         for row in raster_list:
115             for i in range(len(row)):
116                 row[i] = int(row[i])
117
118     raster_ascii.close()
119     del raster_ascii
```

```
120     return raster_list
121
122
123 def extract_header (raster_ascii_path):
124     """ Extracts the header information from a raster converted to a text file
125         and returns it as a list """
126     raster_ascii = open(raster_ascii_path, 'r')
127     header_list = []
128     for i in range(6):
129         header_list.append(raster_ascii.readline())
130
131     raster_ascii.close()
132     del raster_ascii
133     return header_list
134
135
136 def nhood(input_list, row_index, col_index, nsize):
137     """Takes a list, row index, and column index and extracts a neighborhood
138     of nsize around those indexes."""
139     nhood_list = []
140
141     #get the boundary indexes
142     upper_left_row = row_index - nsize
143     if upper_left_row < 0:  upper_left_row = 0
144     upper_left_col = col_index - nsize
145     if upper_left_col < 0:  upper_left_col = 0
146     lower_right_row = row_index + nsize
147     if lower_right_row > (len(input_list) - 1):  lower_right_row = len(input_list) - 1
148     lower_right_col = col_index + nsize
149     if lower_right_col > (len(input_list[0]) - 1):  lower_right_col = len(input_list[0]) - 1
150
151
152     #loop through the input_list for the specified boundaries and record the values
153     row_counter = 0
154     for i in range(upper_left_row, lower_right_row + 1, 1):
155         nhood_list.append([])
156         for j in range(upper_left_col, lower_right_col + 1, 1):
157             nhood_list[row_counter].append(input_list[i][j])
158         row_counter += 1
159
160     return nhood_list
```

```
1  # ------
2  = analy
3  = Creat
4  = Auth
5  =
6  = Descr
7  =
8  =
9  =
10 =
11 =
12 =
13 =
14 =
15 =
16 =
17 =
18 =
19 =
20 =
21 =
22 =
23 =
24 =
25 =
26 =
27 =
28 = Inp
29 =
30 =
31 =
32 =
33 =
34 =
35 # file
36 = file
37 = folder
38 =
```

APPENDIX B – Python Code

analysis_prep.py

```
1  # ---------------------------------------------------------------------------
2  # analysis_prep.py
3  # Created September 2009
4  # Author:  Glenn O'Neil
5  #
6  # Description:  This script generates the datasets and folders necessary for an
7  #               analysis of NED (Naitonal Elevation Dataset) DEM flow-direction
8  #               error by comparison to tiles of finer resolution LiDAR DEMS.
9  #               The preparation includes a stream-burning process to force surface-
10 #               water flow to streams. The NED DEMs are burned with USGS Hydrography
11 #               1:24K DLG vector features. LiDAR tile datasets are identified based
12 #               on the coordinates of the selected USGS 7.5 minute quadrangle.
13 #               This results in a sub-folder of around 40-48 LiDAR DEMs, each to be
14 #               prepped separately. The LiDAR DEMs' horizontal positional
15 #               accuracy are too precise to burn with 1:24K stream features.
16 #               Therefore, this script calls one of two custom scripts (specified
17 #               by the user) that identify stream locations in LiDAR datasets by
18 #               analyzing only elevation values. Once stream identification has been
19 #               completed for the LiDAR DEMs, another custom script is called to
20 #               attempt to carve through artificial barriers in the LiDAR DEMs,
21 #               such as foot-bridges and tile culverts. This script identifies
22 #               sinks and carves to the closest low-point within a user-specified
23 #               maximum neighborhood size. It continues to identify sinks and carve
24 #               for a user-specified number of iterations. Once the final carve
25 #               is complete, the LiDAR DEM is stream-burned, sinks filled, and
26 #               flow-direction calculated with the D8 method.
27 #
28 # Inputs:     1. quads - an ESRI shapefile of 7.5 Minute USGS Quadrangles
29 #             2. quad_id_field - unique quad ID field of the 'quads' shapefile
30 #             3. quad_id - the unique ID of the selected quad
31 #             4. state_plane_field - field in the 'quads' shapefile that denotes
32 #                                    whether the quad lies in the North or Southern
33 #                                    state planes.
34 #             5. state_plane_n_sr_file - path to the Ohio State Plane North NAD 83 Harn projection
   file
35 #             6. state_plane_s_sr_file - path to the Ohio State Plane South NAD 83 Harn projection
   file
36 #             7. workspace - the workspace where the quad folder containing all output files and
   folders
37 #                                    will be created.
38 #             8. streams_wspace - workspace containing a shapefile of stream features for each Ohio
```

```
quad
39 #           9. ned_dem - a single DEM raster for all of Ohio, generated by contour interpolation
   from
40 #                      DLG hypsography and submitted to NED.
41 #           10. lidar_master_wspace - workspace containing all of the LiDAR rasters used in this
   study.
42 #           11. stream_id_method - the method of stream identification for the LiDAR datsaets;
43 #                      'neighborhood': see script stream_id_nhood.py
44 #                      'neighborhood transect combo': a combination of the neighborhood and
45 #                                 trasect methods. See scripts stream_id_nhood.py
46 #                                 and stream_id_transect.py
47 #           12. elev_diff - parameter of the neighborhood stream ID method. The minimum
   difference in
48 #                      elevation from a stream bank to the stream center.
49 #           13. neigh_maxsize - parameter of neighborhood stream ID method. The maximum size
   in cells
50 #                      to search around a given cell for cells that exceed elev_diff.
51 #           14. elev_diff_trans - parameter of the transect stream ID method. The minimum
   difference in
52 #                      elevation from a stream bank to the stream center.
53 #           15. trans_length - parameter of the transect stream ID method. The transect length over
   which
54 #                      the code will search for elevation changes that exceed elev_diff_trans
55 #           16. max_cavre_length - parameter of the carving script. The maximum distance in cells
   to
56 #                      search about a sink for a lower point.
57 #           17. iterations - parameter of the carving script. The number of times to identify sinks
   and carve.
58 #           18. stream_id_nhood_script - the path to the neighborhood stream identification script.
59 #           19. stream_id_transect_script - the path to the transect stream identification script.
60 #           20. carve_script - the path to the carving script.
61 #           21. lidar_sample - boolean value whether to sample a lattice from the LiDAR tiles,
   reducing the
62 #                      total number of rasters that must be processed.
63 #           22. del_int_data - boolean value whether to delete intermediate datasets
64 #
65 # Outputs:    .
66 #           1. Folder containing output datasets
67 #           2. ned - 10-meter DEM clipped by quadrangle boundary
68 #           3. ned_b = clipped 10-meter DEM stream-burned with stream_g
69 #           4. ned_bf - filled (depressionless) version of ned_b
70 #           5. ned_fd - flow-direction of 10-meter DEM, derived from ned_bf
71 #           6. ned_fid - raster of unique ID values for each 10-meter cell
72 #           7. quad_sel.shp - shapefile of selected quadrangle boundary
73 #           8. streams.shp - shapefile of quadrangle streams (from DLG hydrography)
74 #           9. streams_g - binary raster of stream locations, from streams.shp
75 #           10. lidar - directory containing all of the quad's LiDAR data, organized by tile
```

```
76  #               10-1. carved_dem - LiDAR DEM with artificial barriers in streams carved through
77  #               10-2. flat_areas - raster of locations with a slope of zero, later ignored in error
    calculation
78  #               10-3. lidar_burn - LiDAR raster with lidar_stream locations burned in.
79  #               10-5. lidar_ned_fid -LiDAR-resolution raster of ned_fid, used in locating LiDAR
    cell neighborhoods for each NED cell.
80  #               10-6. lidar_stream - binary raster of LiDAR stream locations.
81  #           11. template_raster - TIFF copy of the ned_fd raster, used to write in flow_error results.
82  # ----------------------------------------------------------------------------
83
84
85  #import the necessary modules
86  import sys, os, time, arcgisscripting, math, random
87  from subprocess import call
88  gp = arcgisscripting.create(9.3)
89  gp.CheckOutExtension("Spatial")
90  ls = os.linesep
91
92  #function to convert seconds to hours and minutes (borrowed from:
93  #http://mail.python.org/pipermail/python-list/2003-January.181366.html
94  def sec_to_h_min(s):
95      temp = float()
96      temp = float(s) / (60*60*24)
97      d = int(temp)
98      temp = (temp - d) * 24
99      h = int(temp)
100     temp = (temp - h) * 60
101     m = int(temp)
102     temp = (temp - m) * 60
103     sec = int(temp)
104     return h,m,sec
105
106
107 #inputs
108 quads = sys.argv[1]
109 quad_id_field = sys.argv[2]
110 quad_id = sys.argv[3]
111 state_plane_field = sys.argv[4]
112 state_plane_n_sr_file = sys.argv[5]
113 state_plane_n_sr = gp.CreateObject("SpatialReference")
114 state_plane_n_sr.CreateFromFile(state_plane_n_sr_file)
115 state_plane_s_sr_file = sys.argv[6]
116 state_plane_s_sr = gp.CreateObject("SpatialReference")
117 state_plane_s_sr.CreateFromFile(state_plane_s_sr_file)
```

```
118 workspace = sys.argv[7]
119 streams_wspace = sys.argv[8] + "\\"
120 ned_dem = sys.argv[9]
121 lidar_master_wspace = sys.argv[10]
122 stream_id_method = sys.argv[11]
123 elev_diff = sys.argv[12]
124 neigh_maxsize = sys.argv[13]
125 elev_diff_trans = sys.argv[14]
126 trans_length = sys.argv[15]
127 max_carve_length = sys.argv[16]
128 iterations = sys.argv[17]
129 stream_id_nhood_script = sys.argv[18]
130 stream_id_transect_script = sys.argv[19]
131 carve_script = sys.argv[20]
132 lidar_sample = sys.argv[21]
133 del_int_data = sys.argv[22]
134
135
136 #get the path to the Python executable, for running certain parts of the script as more memory-
    efficient sub-processes.
137 pythonexe = os.environ.get("PYTHONEXE") #will be used to generate sub-processes to prevent
    memory leaks
138 pythonexe += "\\python.exe"
139
140 #check to see if a new workspace must be created
141 if not os.path.exists(workspace + "\\" + quad_id):
142     os.mkdir(workspace + "\\" + quad_id)
143
144 workspace += "\\" + quad_id + "\\"
145
146 #create the log textfile
147 log_file_path = workspace + "\\log.txt"
148 if not os.path.exists(log_file_path):
149     log_file = open(log_file_path, 'w')
150     log_file.write("QUAD_ID: " + quad_id + ls + ls)
151     log_file.write("Parameters:" + ls)
152     log_file.write("    - quads: " + quads + ls)
153     log_file.write("    - quad_id_field: " + quad_id_field + ls)
154     log_file.write("    - quad_id: " + quad_id + ls)
155     log_file.write("    - state_plane_field: " + state_plane_field + ls)
156     log_file.write("    - state_plane_n_sr_file: " + state_plane_n_sr_file + ls)
157     log_file.write("    - state_plane_s_sr_file: " + state_plane_s_sr_file + ls)
158     log_file.write("    - workspace: " + workspace + ls)
159     log_file.write("    - streams_wspace: " + streams_wspace + ls)
```

```
160     log_file.write("    - ned_dem: " + ned_dem + ls)
161     log_file.write("    - lidar_master_wspace: " + lidar_master_wspace + ls)
162     log_file.write("    - stream_id_method: " + stream_id_method + ls)
163     log_file.write("    - elev_diff: " + elev_diff + ls)
164     log_file.write("    - neigh_maxsize: " + neigh_maxsize + ls)
165     log_file.write("    - upslope_stream_threshhold: " + upslope_stream_threshhold + ls)
166     log_file.write("    - max_carve_length: " + max_carve_length + ls)
167     log_file.write("    - iterations: " + iterations + ls)
168     log_file.write("    - stream_id_nhood_script: " + stream_id_nhood_script + ls)
169     log_file.write("    - stream_id_transect_script: " + stream_id_transect_script + ls)
170     log_file.write("    - stream_id_flowacc_script: " + stream_id_flowacc_script + ls)
171     log_file.write("    - carve_script: " + carve_script + ls)
172     log_file.write("    - lidar_sample: " + lidar_sample + ls)
173     log_file.write("    - delete intermediate data: " + del_int_data + ls + ls)
174 else:
175     log_file = open(log_file_path, 'w')
176
177
178 ######################################NED analysis prep############################
179 #determine whether to skip the NED section because it had already been processed
180 ned_fid = workspace + "ned_fid"
181 ned_fd = workspace + "ned_fd"
182 quad_sel = workspace + "quad_sel.shp"
183
184 #Determine the state plane of the selected quad.
185 cursor = gp.searchcursor(quads, ' "'+ quad_id_field +'" = \'' + quad_id + '\' ')
186 row = cursor.next()
187 state_plane = row.getvalue(state_plane_field)
188
189
190 if not gp.exists(ned_fid):
191     log_file.write(time.ctime() + "    Prepping the NED DEM:" + ls)
192
193     #Get the quad
194     gp.AddMessage(" - Selecting Quad " + quad_id)
195     log_file.write(time.ctime() + "        - Selecting Quad " + quad_id + ls)
196     gp.select_analysis(quads, quad_sel, ' "'+ quad_id_field +'" = \'' + quad_id + '\' ')
197
198     #reproject the quad_boundary if necessary
199     quad_sel_desc = gp.describe(quad_sel)
200     if quad_sel_desc.SpatialReference.Name == "Unknown":
201         gp.AddError('The selected quad does not have a defined projection.')
202         log_file.write(time.ctime() + "        - ERROR: Selecting Quad The selected quad does not
```

```
have a defined projection.")
203         log_file.close()
204         del log_file
205         sys.exit()
206
207     quad_pcs = quad_sel_desc.SpatialReference.PCSCode
208     if state_plane == "N":
209         if quad_pcs != state_plane_n_sr.PCSCode: #Must re-project to State Plane North
210             gp.AddMessage(" - Projecting quad boundary from " + quad_sel_desc.Name + " to " +
state_plane_n_sr.Name)
211             log_file.write(time.ctime() + "        - Projecting quad boundary from " +
quad_sel_desc.Name + " to " + state_plane_n_sr.Name + ls)
212             quad_proj = workspace + "quad_proj.shp"
213             gp.project_management(quad_sel, quad_proj, state_plane_n_sr)
214             gp.delete_management(quad_sel)
215             gp.rename_management(quad_proj, quad_sel)
216     elif state_plane == "S":
217         if quad_pcs != state_plane_s_sr.PCSCode: #Must re-project to State Plane South
218             gp.AddMessage(" - Projecting quad boundary from " + quad_sel_desc.Name + " to " +
state_plane_s_sr.Name)
219             log_file.write(time.ctime() + "        - Projecting quad boundary from " +
quad_sel_desc.Name + " to " + state_plane_s_sr.Name + ls)
220             quad_proj = workspace + "quad_proj.shp"
221             gp.project_management(quad_sel, quad_proj, state_plane_s_sr)
222             gp.delete_management(quad_sel)
223             gp.rename_management(quad_proj, quad_sel)
224
225     #Clip the NED DEM by the quad boundary
226     ned_fd = workspace + "ned_fd"
227     if not gp.exists(ned_fd):
228         gp.AddMessage(" - Clipping the NED DEM by the quad boundary")
229         log_file.write(time.ctime() + "        - Clipping the NED DEM by the quad boundary" + ls)
230         ned = workspace + "ned"
231         gp.ExtractByMask_sa(ned_dem, quad_sel, ned)
232
233
234     #Get the streams
235     streams_input = streams_wspace + quad_id + "ohy.shp"
236     streams = workspace + "streams.shp"
237     #Check to see if the stream dataset must be re-projected
238     streams_desc = gp.describe(streams_input)
239     if streams_desc.SpatialReference.Name == "Unknown":
240         gp.AddError('The selected stream dataset does not have a defined projection.')
241         log_file.write(time.ctime() + "        - ERROR: The selected stream dataset does not have a
defined projection." + ls)
```

```
242         log_file.close()
243         del log_file
244         sys.exit()
245
246     streams_pcs = streams_desc.SpatialReference.PCSCode
247     if state_plane == "N":
248         if streams_pcs != state_plane_n_sr.PCSCode: #Must re-project to State Plane North
249             gp.AddMessage(" - Projecting streams from " + streams_desc.SpatialReference.Name + " to
" + state_plane_n_sr.Name)
250             log_file.write(time.ctime() + "        - Projecting streams from " +
streams_desc.SpatialReference.Name + " to " + state_plane_n_sr.Name + ls)
251             #select the appropriate transformation
252             if '16N' in streams_desc.SpatialReference.Name:
253                 transformation =
"Thesis_NAD1927_UTM16N_to_NAD1983_HARN_Ohio_State_Plane_North_fe"
254             elif '17N' in streams_desc.SpatialReference.Name:
255                 transformation =
"Thesis_NAD1927_UTM17N_to_NAD1983_HARN_Ohio_State_Plane_North_fe"
256             gp.project_management(streams_input, streams, state_plane_n_sr, transformation)
257         else:  #The streams projection is already in the appropriate projection
258             gp.AddMessage(" - Stream re-projection not needed.  Copying streams over to workspace.")
259             log_file.write(time.ctime() + "        - Stream re-projection not needed.  Copying streams
over to workspace." + ls)
260             gp.copy_management(streams_input, streams)
261     elif state_plane == "S":
262         if streams_pcs != state_plane_s_sr.PCSCode: #Must re-project to State Plane South
263             gp.AddMessage(" - Projecting streams from " + streams_desc.SpatialReference.Name + " to
" + state_plane_s_sr.Name)
264             log_file.write(time.ctime() + "        - Projecting streams from " +
streams_desc.SpatialReference.Name + " to " + state_plane_s_sr.Name + ls)
265             #select the appropriate transformation
266             if '16N' in streams_desc.SpatialReference.Name:
267                 transformation =
"Thesis_NAD1927_UTM16N_to_NAD1983_HARN_Ohio_State_Plane_South_fe"
268             elif '17N' in streams_desc.SpatialReference.Name:
269                 transformation =
"Thesis_NAD1927_UTM17N_to_NAD1983_HARN_Ohio_State_Plane_South_fe"
270             gp.project_management(streams_input, streams, state_plane_s_sr, transformation)
271         else:  #The streams projection is already in the appropriate projection
272             gp.AddMessage(" - Stream re-projection not needed.  Copying streams over to workspace.")
273             log_file.write(time.ctime() + "        - Stream re-projection not needed.  Copying streams
over to workspace." + ls)
274             gp.copy_management(streams_input, streams)
275
276
277     #Convert the streams to a raster for DEM burning
278     gp.AddMessage(" - Converting streams to raster")
```

```
279     log_file.write(time.ctime() + "        - Converting streams to raster." + ls)
280     gp.ResetEnvironments()
281     gp.Extent = ned
282     gp.SnapRaster = ned
283     gp.Mask = ned
284     gp.CellSize = ned
285     streams_g_temp = workspace + "streams_g_tmp"
286     gp.PolylineToRaster_conversion(streams, "Stream", streams_g_temp)
287     streams_g = workspace + "streams_g"
288     gp.SingleOutputMapAlgebra_sa("con(IsNull(" + streams_g_temp +"), 0, 1)", streams_g)
289     gp.delete_management(streams_g_temp)
290
291     #Burn the streams into the NED DEM
292     gp.AddMessage(" - Burning streams into NED DEM")
293     log_file.write(time.ctime() + "        - Burning streams into NED DEM." + ls)
294     ned_b = workspace + "ned_b"
295     gp.SingleOutputMapAlgebra_sa("con(" + streams_g + " == 1, " + ned + ", " + ned + " + 10)",
    ned_b)
296
297     #Fill the NED DEMs
298     gp.AddMessage(" - Filling NED DEM")
299     log_file.write(time.ctime() + "        - Filling NED DEM." + ls)
300     ned_bf = workspace + "ned_bf"
301     gp.fill_sa(ned_b, ned_bf)
302
303     #Calculate NED Flowdirection
304     print "Calculating NED flowdirection"
305     gp.AddMessage(" - Calculating NED flow-direction")
306     log_file.write(time.ctime() + "        - Calculating NED flow-direction." + ls)
307     ned_fd = workspace + "ned_fd"
308     gp.flowdirection_sa(ned_bf, ned_fd)
309
310     #determine if ned_fd needs to be re-projected
311     gp.ResetEnvironments()
312     ned_fd_desc = gp.describe(ned_fd)
313     ned_fd_pcs = ned_fd_desc.SpatialReference.PCSCode
314     if state_plane == "N":
315         if ned_fd_pcs != state_plane_n_sr.PCSCode: #Must re-project to State Plane North
316             gp.AddMessage(" - Projecting ned_fd boundary from " +
    ned_fd_desc.SpatialReference.Name + " to " + state_plane_n_sr.Name)
317             log_file.write(time.ctime() + "        - Projecting ned_fd boundary from " +
    ned_fd_desc.SpatialReference.Name + " to " + state_plane_n_sr.Name + ls)
318             ned_fd_proj = workspace + "ned_fd_proj"
319             gp.ProjectRaster_management(ned_fd, ned_fd_proj, state_plane_n_sr, "NEAREST", "#",
    "NAD_1983_To_HARN_Ohio")
```

```
320             gp.delete_management(ned_fd)
321             gp.rename_management(ned_fd_proj, ned_fd)
322         elif state_plane == "S":
323             if ned_fd_pcs != state_plane_s_sr.PCSCode: #Must re-project to State Plane South
324                 gp.AddMessage(" - Projecting ned_fd boundary from " + ned_fd_desc.Name + " to " +
    state_plane_s_sr.Name)
325                 log_file.write(time.ctime() + "        - Projecting ned_fd boundary from " +
    ned_fd_desc.SpatialReference.Name + " to " + state_plane_s_sr.Name + ls)
326                 ned_fd_proj = workspace + "ned_fd_proj"
327                 gp.ProjectRaster_management(ned_fd, ned_fd_proj, state_plane_s_sr, "NEAREST", "#",
    "NAD_1983_To_HARN_Ohio")
328                 gp.delete_management(ned_fd)
329                 gp.rename_management(ned_fd_proj, ned_fd)
330
331
332     #Copy the NED flowdirection raster to a TIFF format for use as a template dataset for writing
    error output raster with GDAL later
333     gp.AddMessage(" - Creating template raster for flow error output")
334     log_file.write(time.ctime() + "        - Creating template raster for flow error output." + ls)
335     ned_fd_float = workspace + "\\ned_fd_float"
336     gp.Float_sa(ned_fd, ned_fd_float)
337     template_raster = workspace + "\\template_raster.tif"
338     gp.CopyRaster_management(ned_fd_float, template_raster, "#", "#", "#", "NONE", "NONE",
    "32_BIT_FLOAT")
339     gp.delete_management(ned_fd_float)
340
341     #convert the ned_fd raster to a point feature dataset
342     gp.AddMessage(" - Converting NED flow-direction to a point feature dataset")
343     log_file.write(time.ctime() + "        - Converting NED flow-direction to a point feature dataset."
    + ls)
344     ned_fd_pt = workspace + "ned_fd_pt.shp"
345     gp.RasterToPoint_conversion(ned_fd, ned_fd_pt, "VALUE")
346
347     #convert back to a raster, but with the FID values as the raster values
348     gp.AddMessage(" - Converting NED flow-direction point feature dataset back to a raster with
    the FID Values as raster values")
349     log_file.write(time.ctime() + "        - Converting NED flow-direction point feature dataset back
    to a raster with the FID Values as raster values." + ls)
350     gp.ResetEnvironments()
351     gp.Extent = ned_fd
352     gp.Cellsize = ned_fd
353     gp.Mask = ned_fd
354     gp.SnapRaster = ned_fd
355     gp.PointToRaster_conversion(ned_fd_pt, "FID", ned_fid, "MOST_FREQUENT", "NONE")
356     gp.delete_management(ned_fd_pt)
357     #re-generate the raster using Map Algebra, this prevents the addition of NoDATA cells around
    the permiter
```

```
358     #of ned_fid, which PointToRaster can do.
359     gp.ResetEnvironments()
360     gp.Extent = ned_fd
361     gp.Cellsize = ned_fd
362     gp.Mask = ned_fd
363     gp.SnapRaster = ned_fd
364     ned_fid2 = ned_fid + "2"
365     gp.SingleOutputMapAlgebra_sa(ned_fid, ned_fid2)
366     gp.delete_management(ned_fid)
367     gp.rename_management(ned_fid2, ned_fid)
368
369     #clean up
370     if del_int_data == "True":
371        gp.delete_management(streams)
372        gp.delete_management(streams_g)
373        gp.delete_management(ned_b)
374        gp.delete_management(ned_bf)
375
376
377 #get the width of the NED flowdirection raster, for storing in the parameters text file
378 gp.AddMessage(" - Reading in NED raster cell size")
379 log_file.write(time.ctime() + "        - Reading in NED raster cell size." + ls)
380 ned_descr = gp.describe(ned_fd)
381 ned_height = ned_descr.meancellheight
382 ned_width = ned_descr.meancellheight
383
384 ###############################################################################
385
386
387 ########################LiDDAR analysis prep#####################################
388
389 if not os.path.isdir(workspace + "lidar"):
390    os.mkdir(workspace + "lidar")
391
392    #determine the LiDAR datasets that intersect the selected quad
393    log_file.write(ls + "LiDAR DEM prep:" + ls)
394    gp.AddMessage(" - Identifying LiDAR Rasters that intersect Quad " + quad_id)
395    log_file.write(time.ctime() + "        - Identifying LiDAR Rasters that intersect Quad " + quad_id
    + ls)
396
397 quad_sel_desc = gp.describe(quad_sel)
398 quad_sel_ll_x = quad_sel_desc.extent.xmin
399 quad_sel_ll_y = quad_sel_desc.extent.ymin
```

```
400 quad_sel_height = quad_sel_desc.extent.Height
401 quad_sel_width = quad_sel_desc.extent.Width
402
403 #determine the name of the LiDAR dataset that contains the lower_left corner of the quad
404 #first get the rasters
405 lidar_rasters = []
406 lidar_master_wspace_contents = os.listdir(lidar_master_wspace)
407 for i in lidar_master_wspace_contents:
408     if os.path.isdir(lidar_master_wspace + "\\" + i):
409         lidar_rasters.append(i)
410 del lidar_master_wspace_contents
411
412 #determine the size of a LiDAR raster
413 lidar_descr = gp.describe(lidar_master_wspace + "\\" + lidar_rasters[0] + "\\" + lidar_rasters[0])
414 lidar_height = lidar_descr.extent.ymax - lidar_descr.extent.ymin
415 lidar_width = lidar_descr.extent.xmax - lidar_descr.extent.xmin
416 lidar_rasters_per_quad_height = int(math.ceil(quad_sel_height / lidar_height))
417 lidar_rasters_per_quad_width = int(math.ceil(quad_sel_width / lidar_width))
418
419 for i in range(len(lidar_rasters)):
420     #get the lidar lower left coordinates from the LiDAR name (first is state plane system, 2-5 are x
    and last 3 are y)
421     lidar_ll_x = int(lidar_rasters[i][1:5] + "000")
422     lidar_ll_y = int(lidar_rasters[i][5:] + "000")
423
424     ll_x_diff = quad_sel_ll_x - lidar_ll_x
425     ll_y_diff = quad_sel_ll_y - lidar_ll_y
426
427     if ll_x_diff < lidar_width and ll_x_diff >=0 and ll_y_diff < lidar_height and ll_y_diff >=0:
    #we've found the lower left LiDAR Dataset
428         ll_lidar_raster = lidar_rasters[i]
429
430         #get the other defining rasters
431         ul_lidar_raster = lidar_rasters[i][:5] + str(lidar_ll_y + ((lidar_rasters_per_quad_height - 1) *
    lidar_height))[:3]
432         ur_lidar_raster = lidar_rasters[i][:1] + str(lidar_ll_x + ((lidar_rasters_per_quad_width - 1) *
    lidar_width))[:4] + str(lidar_ll_y + ((lidar_rasters_per_quad_height - 1) * lidar_height))[:3]
433         lr_lidar_raster = lidar_rasters[i][:1] + str(lidar_ll_x + ((lidar_rasters_per_quad_width - 1) *
    lidar_width))[:4] + lidar_rasters[i][5:]
434
435         break
436
437 #use the corner rasters to define a list of rasters for the entire quad
438 lidar_raster_list = [[]] * lidar_rasters_per_quad_height
439 for i in range(lidar_rasters_per_quad_height):
```

440
441
442 x
443 y
444 p
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
47(
47
47
47
47
47
47
47
47
47
48
48
48

```
440     lidar_raster_list[i] = [[]] * lidar_rasters_per_quad_width
441
442  x_location = int(ll_lidar_raster[1:5] + "000")
443  y_location = int(ll_lidar_raster[5:] + "000")
444  prefix = state_plane.lower()
445
446  for i in range(lidar_rasters_per_quad_height):
447     for j in range(lidar_rasters_per_quad_width):
448        lidar_raster_list[i][j] = prefix + str(x_location)[:4] + str(y_location)[:3]
449        x_location += lidar_width
450     y_location += lidar_height
451     #reset the x_location
452     x_location = int(ul_lidar_raster[1:5] + "000")
453
454  #Trim the boundaries off of lidar_raster_list to avoid edge issues during the analysis.
455  #There will be some areas where the NED dem does not cover the entire LiDAR dem.
456  lidar_raster_list2 = []
457  for i in range(1, len(lidar_raster_list) - 1):
458     lidar_raster_list2.append(lidar_raster_list[i][1:len(lidar_raster_list[i]) - 1])
459  del lidar_raster_list
460
461
462  if lidar_sample == "True":
463     #remove half of the rasters in the form of staggered lattice
464     lidar_raster_list3 = []
465     for i in range(len(lidar_raster_list2)):
466        lidar_raster_list3.append([])
467        #determine whether i is odd or even
468        if i % 2 == 0: # it's even, start at the first index of the row
469           for j in range(0, len(lidar_raster_list2[i]), 2):
470              lidar_raster_list3[i].append(lidar_raster_list2[i][j])
471        else: # its odd, start at the second index of the row
472           for j in range(1, len(lidar_raster_list2[i]), 2):
473              lidar_raster_list3[i].append(lidar_raster_list2[i][j])
474  else:
475     lidar_raster_list3 = lidar_raster_list2
476
477
478  #move the list into a simpler to handle single dimension
479  lidar_raster_list = []
480  for i in range(len(lidar_raster_list3)):
481     for j in range(len(lidar_raster_list3[i])):
482        lidar_raster_list.append(lidar_raster_list3[i][j])
```

```
483
484 del lidar_raster_list2, lidar_raster_list3
485
486
487 #call code to prep the LiDAR tiles
488 for i in range(len(lidar_raster_list)):
489     if not os.path.exists(workspace + "lidar\\" + lidar_raster_list[i]):
490         os.mkdir(workspace + "lidar\\" + lidar_raster_list[i])
491
492     gp.AddMessage("    - Prepping LiDAR tile " + str(lidar_raster_list[i]) + " (" + str(i) + " of " +
    str(len(lidar_raster_list)) + ")")
493     log_file.write(time.ctime() + "        - Prepping LiDAR tile " + str(lidar_raster_list[i]) + " (" +
    str(i) + " of " + str(len(lidar_raster_list)) + ")" + ls)
494     lidar_prep_time_start = time.clock()
495
496     #get the lidar raster
497     lidar_dem = lidar_master_wspace + "\\" + lidar_raster_list[i] + "\\" + lidar_raster_list[i]
498     lidar_wspace = workspace + "lidar\\" + lidar_raster_list[i]
499
500     #identify the stream network
501     lidar_fd = lidar_wspace + "\\lidar_fd"
502     if not os.path.exists(lidar_fd):  #then the code did not complete for this raster
503
504         lidar_stream = lidar_wspace + "\\lidar_stream"
505         if not os.path.exists(lidar_stream):
506             stream_id_time_start = time.clock()
507             gp.AddMessage("        - Identifying stream cells with the " + stream_id_method + "
    method")
508             log_file.write(time.ctime() + "            - Identifying stream cells with the " +
    stream_id_method + " method" + ls)
509
510             if stream_id_method == "neighborhood":
511                 call([pythonexe, stream_id_nhood_script, lidar_dem, elev_diff, neigh_maxsize,
    lidar_stream, lidar_wspace])
512
513             elif stream_id_method == "transect":
514                 call([pythonexe, stream_id_transect_script, lidar_dem, neigh_maxsize, elev_diff,
    trans_stream, lidar_wspace])
515
516             elif stream_id_method == "neighborhood transect combo":
517                 #run the neighborhood method
518                 gp.AddMessage("            - First identifying stream cells with the neighborhood
    method")
519                 log_file.write(time.ctime() + "                - First identifying stream cells with the
    neighborhood method" + ls)
520                 nhood_stream = lidar_wspace + "\\nhood_stream"
```

```
521             call([pythonexe, stream_id_nhood_script, lidar_dem, elev_diff, neigh_maxsize,
    nhood_stream, lidar_wspace])
522
523             #run the transect method
524             gp.AddMessage("            - Next identifying stream cells with the transect method")
525             log_file.write(time.ctime() + "                - Next identifying stream cells with the
    transect method" + ls)
526             trans_stream = lidar_wspace + "\\trans_stream"
527             call([pythonexe, stream_id_transect_script, lidar_dem, neigh_maxsize, elev_diff,
    trans_stream, lidar_wspace])
528
529             #combine the two results
530             gp.AddMessage("            - Combining the neighborhood and transect outputs")
531             log_file.write(time.ctime() + "                - Combining the neighborhood and transect
    outputs" + ls)
532             gp.ResetEnvironments()
533             gp.Extent = lidar_dem
534             gp.Mask = lidar_dem
535             gp.SnapRaster = lidar_dem
536             gp.SingleOutputMapAlgebra_sa("con(" + nhood_stream + " == 1 | " + trans_stream + "
    == 1, 1, 0)", lidar_stream)
537
538
539         else:
540             gp.AddError("UNRECOGNIZED STREAM IDENTIFICATION METHOD")
541             log_file.write(time.ctime() + "            - ERROR: UNRECOGNIZED STREAM
    IDENTIFICATION METHOD")
542             log_file.close()
543             del log_file
544             sys.exit()
545
546         stream_id_time_stop = time.clock()
547         h,m,s = sec_to_h_min(stream_id_time_stop - stream_id_time_start)
548         gp.AddMessage("          - ID took " + str(m) + " minutes " + str(s) + " seconds")
549         log_file.write(time.ctime() + "                - ID took " + str(m) + " minutes " + str(s) + "
    seconds" + ls)
550
551     #use the identified stream cells to burn through artificial barriers in the LiDAR DEM
552     carved_dem = lidar_wspace + "\\carved_dem"
553     if not os.path.exists(carved_dem):
554         stream_carve_time_start = time.clock()
555         gp.AddMessage("        - Carving stream cells through artificial barriers")
556         log_file.write(time.ctime() + "                - Carving stream cells through artificial barriers"
    + ls)
557         call([pythonexe, carve_script, lidar_dem, lidar_stream, max_carve_length, iterations,
    carve_iteration_script, "true", carved_dem, lidar_wspace])
558         stream_carve_time_stop = time.clock()
```

```
559         h,m,s = sec_to_h_min(stream_carve_time_stop - stream_carve_time_start)
560         gp.AddMessage("         - Carve took " + str(m) + " minutes " + str(s) + " seconds")
561         log_file.write(time.ctime() + "         - Carve took " + str(m) + " minutes " + str(s) + "
    seconds" + ls)
562
563       #burn in the identified streams into the LiDAR DEM
564       gp.ResetEnvironments()
565       lidar_burn = lidar_wspace + "\\lidar_burn"
566       if not os.path.exists(lidar_burn):
567         gp.AddMessage("       - Burning streams into LiDAR DEM")
568         log_file.write(time.ctime() + "         - Burning streams into LiDAR DEM" + ls)
569         gp.SingleOutputMapAlgebra_sa("con(" + lidar_stream + " == 1, " + carved_dem + ", " +
    carved_dem + " + 10)", lidar_burn)
570
571       #fill the carved and stream_burned LiDAR DEM
572       lidar_fill = lidar_wspace + "\\lidar_fill"
573       if not os.path.exists(lidar_fill):
574         gp.AddMessage("       - Filling the LiDAR DEM")
575         log_file.write(time.ctime() + "         - Filling the LiDAR DEM" + ls)
576         gp.fill_sa(lidar_burn, lidar_fill)
577
578         #calculate flow-direction on the filled LiDAR DEM
579         gp.AddMessage("       - Calculating LiDAR DEM flow-direction")
580         log_file.write(time.ctime() + "         - Calculating LiDAR DEM flow-direction" + ls)
581         gp.flowdirection_sa(lidar_fill, lidar_fd)
582
583       #create a version of the LiDAR dataset with the NED FID values
584       lidar_ned_fid = lidar_wspace + "\\lidar_ned_fid"
585       if not os.path.exists(lidar_ned_fid):
586         gp.AddMessage("       - Creating LiDAR scale raster of NED FID values")
587         log_file.write(time.ctime() + "         - Creating LiDAR scale raster of NED FID values"
    + ls)
588         gp.ResetEnvironments()
589         gp.Extent = lidar_fd
590         gp.Cellsize = lidar_fd
591         gp.Mask = lidar_fd
592         gp.SnapRaster = lidar_fd
593         gp.SingleOutputMapAlgebra_sa(ned_fid, lidar_ned_fid)
594
595
596       #Getting the LiDAR cell size for the parameters text file that will inform the
597       #selection of LiDAR tiles based on the NED DEM coordinate
598
599       gp.AddMessage("       - Recording LiDAR cellsize info for parameters text file")
          log_file.write(time.ctime() + "         - Recording LiDAR cellsize info for parameters text
    file" + ls)
```

```
600     lidar_descr = gp.describe(lidar_fd)
601     lidar_height = lidar_descr.meancellheight
602     lidar_width = lidar_descr.meancellheight
603     lidar_cells_per_ned_cell = ((ned_height / lidar_height) + (ned_width / lidar_width)) * 0.5
604     buffer_size = int(lidar_cells_per_ned_cell * 0.5)
605
606     #write cell size information to a parameters textfile for processing
607     parameters_file_path = lidar_wspace + "\\flow_analysis_parameters.txt"
608     parameters_file = open(parameters_file_path, 'w')
609     parameters_file.write("ned_height;" + str(ned_height) + ls)
610     parameters_file.write("ned_width;" + str(ned_width) + ls)
611     parameters_file.write("lidar_height;" + str(lidar_height) + ls)
612     parameters_file.write("lidar_width;" + str(lidar_width) + ls)
613     parameters_file.write("lidar_cells_per_ned_cell;" + str(lidar_cells_per_ned_cell) + ls)
614     parameters_file.write("buffer_size;" + str(buffer_size) + ls)
615     parameters_file.close()
616     del parameters_file
617
618
619     #create a binary raster of flat areas, so these areas may be discarded
620     #in the calculation of flow-direction error
621     gp.AddMessage("      - Creating a mask of flat areas")
622     log_file.write(time.ctime() + "          - Creating a mask of flat areas" + ls)
623     gp.ResetEnvironments()
624     gp.Extent = lidar_fill
625     gp.SnapRaster = lidar_fill
626     gp.Mask = lidar_fill
627     gp.CellSize = lidar_fill
628     #calculate slope
629     slope_temp = lidar_wspace + "\\slope_temp"
630     if not gp.Exists(slope_temp):
631         gp.Slope_sa(lidar_fill, slope_temp, "PERCENT_RISE")
632     #convert slope = 0% to binary grid
633     flat_areas = lidar_wspace + "\\flat_areas"
634     if not gp.Exists(flat_areas):
635         gp.SingleOutputMapAlgebra_sa("con(" + slope_temp + " == 0, 1, 0)", flat_areas)
636         gp.delete_management(slope_temp)
637
638     #clean up
639     if del_int_data == "True":
640         gp.AddMessage("      - Deleting intermediate data")
641         log_file.write(time.ctime() + "          - Deleting intermediate data" + ls)
642         if os.path.exists(carved_dem):
```

```
643             gp.delete_management(carved_dem)
644         if os.path.exists(lidar_stream):
645             gp.delete_management(lidar_stream)
646         if os.path.exists(lidar_burn):
647             gp.delete_management(lidar_burn)
648         if os.path.exists(lidar_fill):
649             gp.delete_management(lidar_fill)
650
651
652     gp.ResetEnvironments()
653
654     seconds = time.clock() - lidar_prep_time_start
655     h,m,s = sec_to_h_min(seconds)
656
657     gp.AddMessage("       - Tile " + str(lidar_raster_list[i]) + " took " + str(m) + " minutes " +
str(s) + " seconds")
658     log_file.write(time.ctime() + "          - Tile " + str(lidar_raster_list[i]) + " took " + str(m) + "
minutes " + str(s) + " seconds" + ls)
659
660 log_file.close()
661 del log_file
```

APPENDIX B – Python Code

stream_dlg_conversion.py

```
1  # ---------------------------------------------------------------------------
2  # stream_dlg_conversion.py
3  # Created September 2009
4  # Author:  Glenn O'Neil
5  #
6  # Description:  Takes a directory of DLG hydrography files and converts each
7  #               files relevant features to ESRI shapefiles.  Crashes due to an
8  #               ESRI bug if it tries to process more than 70 files.
9  #
10 # Inputs:       1. dlg_wspace - directory containing DLG hypsography files.
11 #               2. scratch_wspace - directory for temporary files.
12 #               3. out_wspace - directory where the output shapefiles are written.
13 #
14 # Outputs:      1. <quad_name>_ohy.shp - shapefile of quadrangle contours.
15 # ---------------------------------------------------------------------------
16 import os, arcgisscripting, sys
17 gp = arcgisscripting.create(9.3)
18
19 dlg_wspace = sys.argv[1]
20 scratch_wspace = sys.argv[2] + "\\"
21 out_wspace = sys.argv[3] + "\\"
22
23 #get the dlg files
24 dlg_wspace_list = os.listdir(dlg_wspace)
25 dlg_list = []
26 #filter the list so it only contains .dlg files
27 for i in range(len(dlg_wspace_list)):
28     if '.dlg' in (dlg_wspace_list[i]):
29         dlg_list.append(dlg_wspace_list[i])
30 del dlg_wspace_list
31
32 dlg_list_length = len(dlg_list)
33 for i in range(dlg_list_length):
34     gp.AddMessage("Processing " + str(i + 1) + " of " + str(dlg_list_length) + ":")
35
36     #convert the dlg to a coverage
37     gp.AddMessage("     - converting DLG to coverage")
38     dlg_cov = scratch_wspace + "dlg_cov"
39     gp.dlgarc(dlg_wspace + "\\" + dlg_list[i], dlg_cov)
```

```
40
41      #convert the coverage to a shapefile
42      gp.AddMessage("    - converting coverage to shapefile")
43      dlg_shp = dlg_cov + "_arc.shp"
44      gp.FeatureClassToShapefile(dlg_cov + "\\arc", scratch_wspace)
45
46      #join the .acode file to the shapefile table on the ID field
47      gp.AddMessage("    - joining acode table to shapefile")
48      acode = dlg_cov + ".acode"
49      gp.joinfield (dlg_shp, "ID", acode, "DLG_COV-ID")
50
51      #extract the appropriate stream features
52      gp.AddMessage("    - extracting stream features")
53      #get the appropriate fields from dlg_shp
54      code_fields = gp.listfields(dlg_shp, 'MINOR*')
55
56      select_query = '"MAJOR1" = 50 AND ('
57      for j in range(len(code_fields)):
58          if j != (len(code_fields) - 1):
59              select_query += '"' + code_fields[j].Name + '" IN (400,401,412,413,414,605,606) OR
    '
60          else:  #treat the last record differently
61              select_query += '"' + code_fields[j].Name + '" IN (400,401,412,413,414,605,606))'
62
63      dlg_out = out_wspace + dlg_list[i][:8] + '.shp'
64      gp.select_analysis(dlg_shp, dlg_out, select_query)
65
66      gp.AddMessage("    - deleting intermediate data")
67      gp.delete_management(dlg_cov)
68      gp.delete_management(dlg_shp)
69
70  del dlg_list, select_query
```

APPENDIX B – Python Code

stream_id_nhood.py

```
#----------------------------------------------------------------------------
# stream_id_nhood.py
# Created September 2009
# Author:  Glenn O'Neil
#
# Description:  Takes an elevation raster and identifies potential stream cells
#               based on their relationship to cell neighborhood of user-specified
#               size.  The basic premise is that stream cells will be opposed
#               by higher elevation in one direction (stream banks) and similar
#               elevations in the direction perpendicular to the bank cells.
#
# Inputs:       1. elev_raster - the elevation raster.
#               2. elev_diff - the difference in elevation between a cell and a
#                  neighbor that, when exceeded, could potentially represent
#                  a stream bank.
#               3. neigh_maxsize - the maximum neighborhood size (in cells) to
#                  search for the stream cell relationship defined above.
#               4. stream_raster - the output binary stream raster.
#               5. workspace - the directory where temporary files will be written.
#
# Outputs:      1. stream_raster - the output binary stream raster.
#----------------------------------------------------------------------------
from decimal import *
from math import floor
import os, sys, time, arcgisscripting
import raster_analysis as raster
ls = os.linesep
gp = arcgisscripting.create(9.3)

arcgis_home = os.environ.get("ARCGISHOME")
gp.AddToolbox(arcgis_home + "ArcToolbox\\Toolboxes\\Conversion Tools.tbx")

#function to convert seconds to hours and minutes (borrowed from:
#http://mail.python.org/pipermail/python-list/2003-January.181366.html
def sec_to_h_min(s):
    temp = float()
    temp = float(s) / (60*60*24)
    d = int(temp)
    temp = (temp - d) * 24
```

```
    h = int(temp)
    temp = (temp - h) * 60
    m = int(temp)
    temp = (temp - m) * 60
    sec = int(temp)
    return h,m,sec


#get the necessary files from the user
elev_raster = sys.argv[1]
#get the elevation difference that will identify streambed cells\
elev_diff = sys.argv[2]
elev_diff = float(elev_diff)
#get the max neighborhood size\
neigh_maxsize = sys.argv[3]
#get the file name and path for the output stream raster
stream_raster = sys.argv[4]
#get the scratch workspace
workspace = sys.argv[5] + "\\"

#start the timer
start_time = time.clock()

#convert the raster to a text file
elev_ascii_path = workspace + "elev_ascii.txt"
gp.AddMessage("- Converting raster to ASCII")
gp.RasterToASCII_conversion(elev_raster, elev_ascii_path)

#record the header information for writing in the output stream file
header_list = raster.extract_header(elev_ascii_path)

#read the elevation text raster into a list
elev_list = raster.rastertext2list(elev_ascii_path, 'float')
os.remove(elev_ascii_path)

#intiate a list to represent the resulting stream ascii, set initial values to 0.
stream_list = [[]] * len(elev_list)
for i in range(len(stream_list)):
    stream_list[i] = [0] * len(elev_list[0])

#Move through elev_list and look at the surrounding neighbors for
#values that exceed the user-specified elevation difference.  If one is found,
#check the opposite cell.  For example, if a southeast cell is above the
```

```
#threshold, check the corresponding northwest cell. If both are above the
#threshold, then the center cell may be the stream. To check, check the opposite
#set of cells (in the example above this would be the southwest and northwest
#cells. If they are below the threshold, then that direction may be the stream.
#Code those cells to 1 in the stream list.



    #counters for output
    streambed_cells = 0

    #a function for analyzing neighborhoods
    def nhood (thelist, row_index, col_index, nsize):
        #thelist is the list to operate on
        #row_index is the row index number of the center cell
        #col_index is the column index number of the center cell
        #nsize is the size of the neighborhood buffer (nsize of 1 = 3x3 neighborhood)

        ubound_index = row_index - nsize #the upper index
        bbound_index = row_index + nsize #the bottom index
        lbound_index = col_index - nsize #the left index
        rbound_index = col_index + nsize #the right index

        global streambed_cells

        #check to see if we're near the boundary of the dataset, and adjust accordingly
        if ubound_index < 0: ubound_index = 0
        if bbound_index > (len(thelist) - 1): bbound_index = (len(thelist) - 1)
        if lbound_index < 0: lbound_index = 0
        if rbound_index > (len(thelist[0]) - 1): rbound_index = (len(thelist[0]) - 1)

        #find the cell in the neighborhood with the maximum steepness
        center_value = thelist[row_index][col_index]
        max_value = center_value
        max_location = [row_index,col_index]
        for k in range(ubound_index, bbound_index):
            for l in range(lbound_index, rbound_index):
                if thelist[k][l] > max_value:
                    max_value = thelist[k][l]
                    max_row_index = k
                    max_col_index = l

        if (max_value - center_value) > elev_diff: #There was a cell in the neighborhod
            #higher than the center cell and above the elevation difference threshold.
```

```
126         #Now look for another steep cell on the opposite side of the center cell
127         #first, find the distance between the max location and center cell
128         row_distance = row_index - max_row_index
129         col_distance = col_index - max_col_index
130         abs_row_distance = abs(row_distance) if abs(row_distance) > 0 else 1
131         abs_col_distance = abs(col_distance) if abs(col_distance) > 0 else 1
132
133         opposite_row_index = row_index + row_distance
134         opposite_col_index = col_index + col_distance
135
136         #check for the boundaries
137         if opposite_row_index < 0: opposite_row_index = 0
138         elif opposite_row_index > bbound_index: opposite_row_index = bbound_index
139
140         if opposite_col_index < 0: opposite_col_index = 0
141         elif opposite_col_index > rbound_index: opposite_col_index = rbound_index
142
143
144         if (thelist[opposite_row_index][opposite_col_index] - center_value) > elev_diff:
145           #Then the opposite side is also above the threshold.  The center of
146           #the neighborhood may be a the stream.
147
148           #Get the indexes of the recipricol diagonal (e.g. the peaks were northeast and southwest of
149           #the center cell.  Now check northwest and southwest to see if it's less than the threshold.
150           recip_row_index1 = row_index - col_distance
151           recip_col_index1 = col_index + row_distance
152           recip_row_index2 = row_index + col_distance
153           recip_col_index2 = col_index - row_distance
154
155           if recip_row_index1 < 0: recip_row_index1 = 0
156           elif recip_row_index1 > bbound_index: recip_row_index1 = bbound_index
157
158           if recip_row_index2 < 0: recip_row_index2 = 0
159           elif recip_row_index2 > bbound_index: recip_row_index2 = bbound_index
160
161           if recip_col_index1 < 0: recip_col_index1 = 0
162           elif recip_col_index1 > rbound_index: recip_col_index1 = rbound_index
163
164           if recip_col_index2 < 0: recip_col_index2 = 0
165           elif recip_col_index2 > rbound_index: recip_col_index2 = rbound_index
166
167           if (thelist[recip_row_index1][recip_col_index1] - center_value) < elev_diff and
    (thelist[recip_row_index2][recip_col_index2] - center_value) < elev_diff:
```

```
168
169             streambed_cells += 1
170             #it might be the stream.
171
172             #We now begin to define the stream basin.
173             #In this instance, we will define a rectangular region, with the max value location,
174             #it's opposite index, and the two reciprocal indidecs as the corners.
175             #The area of the rectangle will be coded 1, defining it as part of the streambed.
176             #To do this we must determine the slope of the rectangles edges. This will
177             #enable us to define the area within the edges. Though, we must determine the
178             #path from one corner to the other. Here is an example.
179             #If the slope was 15/7, it would make a big L shaped edge.
180             #We need to find the relatively straight path from each corner to the next.
181             #We do this by reducing the smaller number of the
182             #slope to 1. For example, a slope of 15/7 would reduce to a horizontal (H)
183             #movment of 1 for every 2.5 vertical movements (V). Of course we can't move
184             #2.5 cells. So we will construct a list of vertical movements for the path from
185             # the max cell to the first reciprocal cell (one edge of the rectangle). In our
186             #example above, half of the vertical movements in that list will be a value of 3,
187             #and half will be 2. Because, along the path, for every horizontal movement of 1,
188             #50% of the time the subsequent vertical movement will be 2, and 50% of the time 3.
189
190             #We had previously determined the absolute row and col distance from the max cell
191             #to the center cell. The slope of the edge from the max cell to the first reciprocal
192             #cell (one edge of the rectangle) is as follows:
193
194             #Calculate new row and col distances. This time from the max index to the first
    reciprocal
195
196             #Watch out for the edges. They could throw off these calculations
197             #using absolute values can help avoid the problem. Just swap the values
198             orig_max_row_index = max_row_index
199             orig_max_col_index = max_col_index
200             orig_recip_row_index1 = recip_row_index1
201             orig_recip_col_index1 = recip_col_index1
202
203             if (abs(row_index - max_row_index) < abs(row_index - opposite_row_index)) or
    (abs(col_index - max_col_index) < abs(col_index - opposite_col_index)):
204                 #you need to switch the max and opposite indexes
205                 max_row_index = opposite_row_index
206                 max_col_index = opposite_col_index
207                 opposite_row_index = orig_max_row_index
208                 opposite_col_index = orig_max_col_index
209
```

```
210             if (abs(row_index - recip_row_index1) < abs(row_index - recip_row_index2)) or
    (abs(col_index - recip_col_index1) < abs(col_index - recip_col_index2)):
211                 #you need to switch the reciprocal indexes
212                 recip_row_index1 = recip_row_index2
213                 recip_col_index1 = recip_col_index2
214                 recip_col_index2 = orig_recip_col_index1
215                 recip_row_index2 = orig_recip_row_index1
216
217
218             row_distance2 = max_row_index - recip_row_index1
219             col_distance2 = max_col_index - recip_col_index1
220
221             abs_row_distance2 = float(abs(row_distance2))
222             abs_col_distance2 = float(abs(col_distance2))
223
224             #Initiate lists that will store the h_move and v_move lists.
225             #The first list will contain the h_move and v_move values for the path
226             #from the max index to the first reciprocal index.
227             #The second list will contain the h_move and v_move values for the path
228             #from the max index to the second reciprocal index.
229             #The stream basin will be defined by looping throught the first list
230             #within the second list.
231             #The list sizes will be the same as the minimum value between abs_row_distance2
232             #and abs_col_distance2.
233             list_size = min(abs_row_distance2, abs_col_distance2)
234             if list_size == 0: list_size = 1
235
236             abs_difference = abs(abs_row_distance2 - abs_col_distance2)
237
238
239             move_quotient = 0
240             #Determine the h_move and v_move
241             if abs_row_distance2 < abs_col_distance2 and abs_row_distance2 != 0: #we're stepping
    horizontally
242                 primary_move = "horizontal"
243                 #recod the maximum H distance for a V distance of 1
244                 if col_distance2 > 0: #we're stepping left
245                     if abs_row_distance2 > 0:
246                         move_quotient = (abs_col_distance2/abs_row_distance2) * -1
247                         min_h_move = int(floor(move_quotient)) + 1
248                         if abs(abs_col_distance2 % abs_row_distance2) > 0:
249                             max_h_move = min_h_move - 1
250                         else:
251                             max_h_move = min_h_move
```

```
252                 else: #it's a straight line down to the destination index
253                     min_h_move = col_distance2
254                     max_h_move = min_h_move
255                 elif col_distance2 < 0: #we're stepping right
256                     if abs_row_distance2 > 0:
257                         move_quotient = abs_col_distance2/abs_row_distance2
258                         min_h_move = int(floor(move_quotient))
259                         if abs(abs_col_distance2 % abs_row_distance2) > 0:
260                             max_h_move = min_h_move + 1
261                         else:
262                             max_h_move = min_h_move
263                     else: #it's a straight line down to the destination index
264                         min_h_move = col_distance2 * -1
265                         max_h_move = min_h_move
266
267                 if row_distance2 < 0: v_move = 1
268                 elif row_distance2 > 0: v_move = -1
269                 else: v_move = 0
270
271                 if min_h_move == max_h_move:
272                     h_move_list1 = [min_h_move] * int(list_size)
273                 else:
274                     h_move_list1 = move_list(list_size, min_h_move, max_h_move, move_quotient,
    row_index, col_index)
275
276                 v_move_list1 = [v_move] * int(list_size)
277
278
279             elif abs_col_distance2 < abs_row_distance2 and abs_col_distance2 != 0: #we're
    stepping vertically
280                 primary_move = "vertical"
281                 #recod the maximum V distance for an H distance of 1
282                 if row_distance2 > 0:  #we're stepping up (visually, not in terms of row index
    numbers)
283                     if abs_col_distance2 > 0:
284                         move_quotient = (abs_row_distance2/abs_col_distance2) * -1
285                         min_v_move = int(floor(move_quotient)) + 1
286                         if abs(abs_row_distance2 % abs_col_distance2) > 0:
287                             max_v_move = min_v_move - 1
288                         else:
289                             max_v_move = min_v_move
290                     else:  #it's a straight line down to the destination index
291                         min_v_move = row_distance2
292                         max_v_move = min_v_move
```

```
293             elif row_distance2 < 0:  #we're stepping down (visually, not in terms of row index
    numbers)
294                 if abs_col_distance2 > 0:
295                     move_quotient = abs_row_distance2/abs_col_distance2
296                     min_v_move = int(floor(move_quotient))
297                     if abs(abs_row_distance2 % abs_col_distance2) > 0:
298                         max_v_move = min_v_move + 1
299                     else:
300                         max_v_move = min_v_move
301                 else:
302                     min_v_move = row_distance2 * -1
303                     max_v_move = min_v_move
304
305             if col_distance2 < 0: h_move = 1
306             elif col_distance2 > 0: h_move = -1
307             else: h_move = 0
308
309             if min_v_move == max_v_move:
310                 v_move_list1 = [min_v_move] * int(list_size)
311             else:
312                 v_move_list1 = move_list(list_size, min_v_move, max_v_move, move_quotient,
    row_index, col_index)
313
314             h_move_list1 = [h_move] * int(list_size)
315
316         else: #the're equal, it's a square
317             primary_move = "either"
318             if row_distance2 < 0: v_move = 1
319             elif row_distance2 > 0: v_move = -1
320             else: v_move = 0
321             if col_distance2 < 0: h_move = 1
322             elif col_distance2 > 0: h_move = -1
323             else: h_move = 0
324
325
326             v_move_list1 = [v_move] * int(list_size)
327             h_move_list1 = [h_move] * int(list_size)
328
329             #check to see if it's a perfect square
330             if v_move == 0:
331                 h_move_list1 = [h_move] * int(abs_col_distance2)
332                 v_move_list1 = [0] * int(abs_col_distance2)
333             if h_move == 0:
334                 h_move_list1 = [0] * int(abs_row_distance2)
```

```
335                 v_move_list1 = [v_move] * int(abs_row_distance2)
336
337
338
339             #figure out the distances from the max index to recipricol index2.
340             row_distance3 = max_row_index - recip_row_index2
341             col_distance3 = max_col_index - recip_col_index2
342             h_move_list2 = []
343             v_move_list2 = []
344
345             if col_distance3 < 0: #we're stepping to the right
346               for k in range(len(v_move_list1)):
347                 h_move_list2.append(abs(v_move_list1[k]))
348             elif col_distance3 > 0:  #we're stepping to the left
349               for k in range(len(v_move_list1)):
350                 h_move_list2.append(abs(v_move_list1[k])* -1)
351             else:
352               h_move_list2 = [0] * len(v_move_list1)
353
354             if row_distance3 < 0: #we're stepping down (visually, not in terms of row index
    numbers)
355               for k in range(len(h_move_list1)):
356                 v_move_list2.append(abs(h_move_list1[k]))
357             elif row_distance3 > 0:  #we're stepping up
358               for k in range(len(h_move_list1)):
359                 v_move_list2.append(abs(h_move_list1[k])* -1)
360             else:
361               v_move_list2 = [0] * len(h_move_list1)
362
363
364             #Call a function that takes the four corners of the stream basin (the max, its opposite
    cell,
365             #and the two recipracols) and constructs a square using those points as the corners.
366             #Then set all of the values in that area to 1 for the stream_list.
367             river_bed(max_row_index, max_col_index, v_move_list1, h_move_list1, v_move_list2,
    h_move_list2, primary_move, ubound_index, bbound_index, lbound_index, rbound_index)
368
369
370
371 def move_list(list_length, min_move, max_move, quotient, row, col):
372   #This function constructs the move lists from one cell to another.
373
374   the_list = [min_move] * int(list_length)
375   max_index_list = []
```

```
376     move_difference = quotient - min_move #gives us the percentage of the time that max_move
      will be used.
377     max_index_places = abs(list_length * move_difference)
378     max_index_step = list_length / max_index_places
379
380
      if int(abs(move_difference) * 10) >= 5: # then the max move will dominate the list or split it
381   evenly, and should be first
382       max_index_holder = 0
383     else: #it should start later in the list
384       max_index_holder = int(round(max_index_step)) - 1 #minus 1 to put it in proper zero-based
      index mode
385
386     max_index_step_counter = 0
387     while max_index_holder < len(the_list):
388       max_index_list.append(max_index_holder)
389       max_index_step_counter += max_index_step
390       max_index_holder = int(round(max_index_step_counter))
391
392
393     for i in range(len(the_list)):
394       if i in max_index_list:
395         the_list[i] = max_move
396
397     return the_list
398
399
400   def river_bed(max_row_index, max_col_index, v_move_list1, h_move_list1, v_move_list2,
      h_move_list2, primary_move, ubound_index, bbound_index, lbound_index, rbound_index):
401     #this function will loop through the v_move_list1 and h_move_list1, while looping though a
402     #reciprocal version of each list, v_move_list2 and h_move_list2.
403
404     current_row_index1, current_row_index2 = max_row_index, max_row_index
405     current_col_index1, current_col_index2 = max_col_index, max_col_index
406
407     stream_list[current_row_index1][current_col_index1] = 1
408
409     #need to make an initial run through v_move_list1 and h_move_list1
410     if primary_move == "vertical":
411       for p in range(len(v_move_list1)):
412         for q in range(abs(v_move_list1[p])):
413           if v_move_list1[p] > 0 : current_row_index1 += 1
414           elif v_move_list1[p] < 0: current_row_index1 -= 1
415
416           if current_row_index1 > bbound_index:  current_row_index1 = bbound_index
```

```
417             elif current_row_index1 < 0: current_row_index1 = 0
418
419             stream_list[current_row_index1][current_col_index1] = 1
420
421         if h_move_list1[p] > 0 : current_col_index1 += 1
422         elif h_move_list1[p] < 0: current_col_index1 -= 1
423
424         if current_col_index1 > rbound_index: current_col_index1 = rbound_index
425         elif current_col_index1 < 0: current_col_index1 = 0
426
427         stream_list[current_row_index1][current_col_index1] = 1
428
429     elif primary_move == "horizontal":
430       for p in range(len(v_move_list1)):
431         for q in range(abs(h_move_list1[p])):
432           if h_move_list1[p] > 0 : current_col_index1 += 1
433           elif h_move_list1[p] < 0: current_col_index1 -= 1
434
435           if current_col_index1 > rbound_index: current_col_index1 = rbound_index
436           elif current_col_index1 < 0: current_col_index1 = 0
437
438           stream_list[current_row_index1][current_col_index1] = 1
439
440         if v_move_list1[p] > 0 : current_row_index1 += 1
441         elif v_move_list1[p] < 0: current_row_index1 -= 1
442
443         if current_row_index1 > bbound_index:  current_row_index1 = bbound_index
444         elif current_row_index1 < 0: current_row_index1 = 0
445
446         stream_list[current_row_index1][current_col_index1] = 1
447
448     else:
449       for p in range(len(v_move_list1)):
450         if h_move_list1[0] > 0 : current_col_index1 += 1
451         elif h_move_list1[0] < 0: current_col_index1 -= 1
452         if current_col_index1 > rbound_index: current_col_index1 = rbound_index
453         elif current_col_index1 < 0: current_col_index1 = 0
454
455         if v_move_list1[0] > 0 : current_row_index1 += 1
456         elif v_move_list1[0] < 0: current_row_index1 -= 1
457         if current_row_index1 > bbound_index:  current_row_index1 = bbound_index
458         elif current_row_index1 < 0: current_row_index1 = 0
459
```

```
460             stream_list[current_row_index1][current_col_index1] = 1
461
462
463     if h_move_list2[0] > 0: current_col_index2 += 1
464     elif h_move_list2[0] < 0: current_col_index2 -= 1
465     if current_col_index2 > rbound_index: current_col_index2 = rbound_index
466     elif current_col_index2 < 0: current_col_index2 = 0
467
468     if v_move_list2[0] > 0: current_row_index2 += 1
469     elif v_move_list2[0] < 0: current_row_index2 -= 1
470     if current_row_index2 > bbound_index: current_row_index2 = bbound_index
471     elif current_row_index2 < 0: current_row_index2 = 0
472
473     current_row_index1, current_col_index1 = current_row_index2, current_col_index2
474
475     #now loop through both sets of lists
476     for m in range(len(v_move_list2)):
477       #need to caputre loop of v_move_list2
478       if primary_move == "vertical":
479         for n in range(abs(h_move_list2[m])):
480
481           stream_list[current_row_index2][current_col_index2] = 1
482
483           for p in range(len(v_move_list1)):
484             for q in range(abs(v_move_list1[p])):
485               if v_move_list1[p] > 0 : current_row_index1 += 1
486               elif v_move_list1[p] < 0: current_row_index1 -= 1
487
488               if current_row_index1 > bbound_index:  current_row_index1 = bbound_index
489               elif current_row_index1 < 0: current_row_index1 = 0
490
491               stream_list[current_row_index1][current_col_index1] = 1
492
493             if h_move_list1[p] > 0 : current_col_index1 += 1
494             elif h_move_list1[p] < 0: current_col_index1 -= 1
495
496             if current_col_index1 > rbound_index: current_col_index1 = rbound_index
497             elif current_col_index1 < 0: current_col_index1 = 0
498
499             stream_list[current_row_index1][current_col_index1] = 1
500
501           if h_move_list2[0] > 0:  current_col_index2 += 1
502           elif h_move_list2[0] < 0:  current_col_index2 -= 1
```

```
503
504             if current_col_index2 > rbound_index: current_col_index2 = rbound_index
505             elif current_col_index2 < 0: current_col_index2 = 0
506
507             stream_list[current_row_index2][current_col_index2] = 1
508
509             current_col_index1 = current_col_index2
510             current_row_index1 = current_row_index2
511
512         if v_move_list2[0] > 0:  current_row_index2 += 1
513         elif v_move_list2[0] < 0:  current_row_index2 -= 1
514
515         if current_row_index2 > bbound_index:  current_row_index2 = bbound_index
516         elif current_row_index2 < 0: current_row_index2 = 0
517
518         current_row_index1 = current_row_index2
519
520     elif primary_move == "horizontal":
521         for n in range(abs(v_move_list2[m])):
522
523             stream_list[current_row_index2][current_col_index2] = 1
524
525             for p in range(len(v_move_list1)):
526                 for q in range(abs(h_move_list1[p])):
527                     if h_move_list1[p] > 0 : current_col_index1 += 1
528                     elif h_move_list1[p] < 0: current_col_index1 -= 1
529
530                     if current_col_index1 > rbound_index: current_col_index1 = rbound_index
531                     elif current_col_index1 < 0: current_col_index1 = 0
532
533                     stream_list[current_row_index1][current_col_index1] = 1
534
535                 if v_move_list1[p] > 0 : current_row_index1 += 1
536                 elif v_move_list1[p] < 0: current_row_index1 -= 1
537
538                 if current_row_index1 > bbound_index:  current_row_index1 = bbound_index
539                 elif current_row_index1 < 0: current_row_index1 = 0
540
541                 stream_list[current_row_index1][current_col_index1] = 1
542
543             if v_move_list2[0] > 0:  current_row_index2 += 1
544             elif v_move_list2[0] < 0:  current_row_index2 -= 1
545
```

```
546             if current_row_index2 > bbound_index:  current_row_index2 = bbound_index
547             elif current_row_index2 < 0: current_row_index2 = 0
548
549             stream_list[current_row_index2][current_col_index2] = 1
550
551             current_row_index1 = current_row_index2
552             current_col_index1 = current_col_index2
553
554         if h_move_list2[0] > 0:  current_col_index2 += 1
555         elif h_move_list2[0] < 0:  current_col_index2 -= 1
556
557         if current_col_index2 > rbound_index: current_col_index2 = rbound_index
558         elif current_col_index2 < 0: current_col_index2 = 0
559
560
561         current_col_index1 = current_col_index2
562
563     else:
564
565         stream_list[current_row_index2][current_col_index2] = 1
566
567         for p in range(len(v_move_list1)):
568             if h_move_list1[0] > 0 : current_col_index1 += 1
569             elif h_move_list1[0] < 0: current_col_index1 -= 1
570             if current_col_index1 > rbound_index: current_col_index1 = rbound_index
571             elif current_col_index1 < 0: current_col_index1 = 0
572
573             stream_list[current_row_index1][current_col_index1] = 1
574
575             if v_move_list1[0] > 0 : current_row_index1 += 1
576             elif v_move_list1[0] < 0: current_row_index1 -= 1
577             if current_row_index1 > bbound_index:  current_row_index1 = bbound_index
578             elif current_row_index1 < 0: current_row_index1 = 0
579
580             stream_list[current_row_index1][current_col_index1] = 1
581
582
583         if h_move_list2[0] > 0: current_col_index2 += 1
584         elif h_move_list2[0] < 0: current_col_index2 -= 1
585         if current_col_index2 > rbound_index: current_col_index2 = rbound_index
586         elif current_col_index2 < 0: current_col_index2 = 0
587
588
```

```
589             if v_move_list2[0] > 0: current_row_index2 += 1
590             elif v_move_list2[0] < 0: current_row_index2 -= 1
591             if current_row_index2 > bbound_index: current_row_index2 = bbound_index
592             elif current_row_index2 < 0: current_row_index2 = 0
593
594             current_row_index1, current_col_index1 = current_row_index2, current_col_index2
595
596     percent_done = 0
597     for i in range(len(elev_list)):
598         for j in range(len(elev_list[i])):
599             nhood(elev_list, i, j, int(neigh_maxsize))
600         percent = int((float(i) / (len(stream_list) - 1) * 100))
601         if percent > percent_done:
602             gp.AddMessage(' - Analyzing:  '+ str(percent) + '% completed')
603             percent_done = percent
604
605
606     #write the output stream file
607     stream_file_text_name = stream_raster + ".txt"
608     stream_file = open(stream_file_text_name, 'w')
609     for i in range(len(header_list)):
610         stream_file.write(header_list[i])
611
612     percent_done = 0
613     for i in range(len(elev_list)):
614         for j in range(len(elev_list[i])):
615             stream_file.write('%s ' % (stream_list[i][j]))
616         stream_file.write(ls)
617         percent = int((float(i) / (len(stream_list) - 1) * 100))
618         if percent > percent_done:
619             percent_done = percent
620     stream_file.close()
621
622     gp.ASCIIToRaster_conversion(stream_file_text_name, stream_raster, "INTEGER")
623     os.remove(stream_file_text_name)
624
625     gp.AddMessage(" - There were " + str(streambed_cells) + " recip_small_slopes")
626
627     stop_time = time.clock() - start_time
628
629     seconds = stop_time - start_time
630     h,m,s = sec_to_h_min(seconds)
631     gp.AddMessage(' - Took ' + str(h) + ' hours ' + str(m) + ' minutes ' + str(s) + ' seconds')
```

APPENDIX B – Python Code

stream_id_transect.py

```
1  # ---------------------------------------------------------------------------
2  # stream_id_transect.py
3  # Created September 2009
4  # Author:  Glenn O'Neil
5  #
6  # Description:  Takes an elevation raster and identifies potential stream cells
7  #               based on their relationship within directional transects.
8  #               The basic premise is that stream cells in the middle of a
9  #               transect will be opposed by higher elevations towards the ends
10 #               of the transect.
11 #
12 # Inputs:       1. elev_raster_path - the elevation raster.
13 #               2. stream_width - the maximum width of the stream, and therefore
14 #                  the maximum width of the transect.
15 #               3. bank_depth - the theshold elevation difference defining a
16 #                  stream cell from a stream bank cell.
17 #               4. stream_output_raster - the output binary stream raster.
18 #
19 # Outputs:      1. stream_output_raster - the output binary stream raster.
20 # ---------------------------------------------------------------------------
21
22 from decimal import *
23 from math import floor
24 import time, os, sys, arcgisscripting
25 import raster_analysis as raster
26 ls = os.linesep
27 gp = arcgisscripting.create(9.3)
28
29
30 #function to convert seconds to hours and minutes (borrowed from:
31 #http://mail.python.org/pipermail/python-list/2003-January.181366.html
32 def sec_to_h_min(s):
33     temp = float()
34     temp = float(s) / (60*60*24)
35     d = int(temp)
36     temp = (temp - d) * 24
37     h = int(temp)
38     temp = (temp - h) * 60
39     m = int(temp)
```

```
40      temp = (temp - m) * 60
41      sec = int(temp)
42      return h,m,sec
43
44
45  #Get the necessary files from the user
46  elev_raster_path = sys.argv[1]
47
48  #Get the maximum width of the stream
49  stream_width = sys.argv[2]
50  stream_width = float(stream_width)
51
52  #Get the minimum depth of the stream bed to the stream bank
53  bank_depth = sys.argv[3]
54  bank_depth = float(bank_depth)
55
56  #Output raster
57  stream_output_raster = sys.argv[4]
58
59  workspace = stream_output_raster[:stream_output_raster.rfind('\\')] + '\\'
60
61  #Get the file name and path for the output stream file
    stream_file_name = workspace + 'strtrant' + str(int(stream_width)) + 'e' + str(int(bank_depth))
62  + '.txt'
63
64  #Start the timer
65  start_time = time.clock()
66
67  #Convert the raster to a text file
68  elev_ascii_path = workspace + "elev_ascii.txt"
69  gp.AddMessage("- Converting raster to ASCII")
70  gp.RasterToASCII_conversion(elev_raster_path, elev_ascii_path)
71
72  #record the header information for writing in the output stream file
73  header_list = raster.extract_header(elev_ascii_path)
74
75  #read the elevation text raster into a list
76  elev_list = raster.rastertext2list(elev_ascii_path, 'float')
77  os.remove(elev_ascii_path)
78
79  #Intiate a list to represent the resulting stream ascii, set initial values to 0.
80  stream_list = [[]] * len(elev_list)
81  for i in range(len(stream_list)):
```

```
82      stream_list[i] = [0] * len(elev_list[0])
83
84
85  #Move through the list along transects to identify stream cells.  Start by moving
86  #vertically from the top cells to the bottom cells, beginning with the upper-left
87  #cell.  Move south, and note any locations where elevation decreases, and then increases
88  #within the user-specified maximum stream width.  You may have to set a minimum
89  #stream width threshold if this approach picks up too many non-stream features.
90
91  #Counters for output
92  streambed_cells = 0
93
94
95  #A function to analyze a transect of cells. The number of cells is equal to the
96  #user specified maximum stream width
97  def transect(thelist):
98      #thelist is the list to operate on
99
100     global bank_similarity, bank_depth
101
102     #Identify the minimum elevation along the transect
103     min_elev = min(thelist)
104     #Note its index in the list
105     for m in range(len(thelist)):
106         if thelist[m] == min_elev:
107             min_index = m
108             break
109
110     #Check the cells after the minimum_index to see if elevation increases significantly
111     max_elev = max(thelist[min_index:len(thelist)])
112     if (thelist[0] - min_elev) > bank_depth and (max_elev - min_elev) > bank_depth:
113         transect_is_stream = True
114     else:
115         transect_is_stream = False
116
117     return transect_is_stream
118
119
120 #initiate a transect list to send to the transect function
121 transect_list = [0] * int(stream_width)
122
123 transect_called = 0
124
```

```
125 #Move through the list vertically, from northernmost cells to sourthernmost,
126 #looking for decreases in elevation
127 percent_done = 0
128 for i in range(len(elev_list[0])):
129   for j in range(len(elev_list)):
130     elev = elev_list[j][i]
131     #Check the next elevation, watch out for the boundary
132     if (j + 1) < len(elev_list):
133       elev_next = elev_list[j + 1][i]
134     else:
135       break
136     if elev > elev_next: #We might have the beginning of a stream, send the next K cells to
    the transect function
137       for k in range(int(stream_width)):
138         #Make sure you're not at the dataset boundary
139         if (j + k) < len(elev_list) and i < len(elev_list[j]):
140           transect_list[k] = elev_list[j + k][i]
141
142       is_stream = transect(transect_list)
143       transect_called += 1
144
145       if is_stream:
146         for k in range(int(stream_width)):
147           if (j + k) < len(elev_list) and i < len(elev_list[j]):
148             stream_list[j + k][i] = 1
149
150   percent = int((float(i) / (len(elev_list[0]) - 1) * 100))
151   if percent > percent_done:
152     gp.AddMessage(' - Analyzing vertical transects:  ' + str(percent) + '% completed')
153     percent_done = percent
154
155
156 #Move through the list horizontally, from westernmost cells to easternmost,
157 #looking for decreases in elevation
158 percent_done = 0
159 for i in range(len(elev_list)):
160   for j in range(len(elev_list[i])):
161     elev = elev_list[i][j]
162     #Check the next elevation, watch out for the boundary
163     if (j + 1) < len(elev_list[0]):
164       elev_next = elev_list[i][j + 1]
165     else:
166       break
```

```
167         if elev > elev_next: #We might have the beginning of a stream, send the next K cells to
    the transect function
168             for k in range(int(stream_width)):
169                 #Make sure you're not at the dataset boundary
170                 if (j + k) < len(elev_list[j]) and i < len(elev_list):
171                     transect_list[k] = elev_list[i][j + k]
172
173             is_stream = transect(transect_list)
174             transect_called += 1
175
176             if is_stream:
177                 for k in range(int(stream_width)):
178                     if (j + k) < len(elev_list[j]) and i < len(elev_list):
179                         stream_list[i][j + k] = 1
180
181     percent = int((float(i) / (len(elev_list[0]) - 1) * 100))
182     if percent > percent_done:
183         gp.AddMessage(' - Analyzing horizontal transects:  ' + str(percent) + '% completed')
184         percent_done = percent
185
186
187 #Move through the list diagnollay, from northwest cells to southeast, looking for decreases in
    elevation
188 percent_done = 0
189 for i in range(len(elev_list)):
190     for j in range(len(elev_list[i])):
191         elev = elev_list[i][j]
192         #Check the next elevation, watch out for the boundary
193         if (j + 1) < len(elev_list[0]) and (i + 1) < len(elev_list):
194             elev_next = elev_list[i + 1][j + 1]
195         else:
196             break
197         if elev > elev_next: #We might have the beginning of a stream, send the next K cells to
    the transect function
198             for k in range(int(stream_width)):
199                 #Make sure you're not at the dataset boundary
200                 if (j + k) < len(elev_list[j]) and (i + k) < len(elev_list):
201                     transect_list[k] = elev_list[i + k][j + k]
202
203             is_stream = transect(transect_list)
204             transect_called += 1
205
206             if is_stream:
207                 for k in range(int(stream_width)):
```

```
208                 if (j + k) < len(elev_list[j]) and (i + k) < len(elev_list):
209                     stream_list[i + k][j + k] = 1
210
211     percent = int((float(i) / (len(elev_list[0]) - 1) * 100))
212     if percent > percent_done:
213         gp.AddMessage(' - Analyzing diagonal (NW-SE) transects:  ' + str(percent) + '% completed')
214         percent_done = percent
215
216
217 #Move through the list diagnollay, from southwest cells to northeast,
218 #looking for decreases in elevation
219 percent_done = 0
220 for i in range(len(elev_list) - 1, -1, -1):
221     for j in range(len(elev_list[i])):
222         elev = elev_list[i][j]
223         #Check the next elevation, watch out for the boundary
224         if (j + 1) < len(elev_list[0]) and (i - 1) >= 0:
225             elev_next = elev_list[i - 1][j + 1]
226         else:
227             break
228         if elev > elev_next: #We might have the beginning of a stream, send the next K cells to the transect function
229             for k in range(int(stream_width)):
230                 #Make sure you're not at the dataset boundary
231                 if (j + k) < len(elev_list[j]) and (i - k) >= 0:
232                     transect_list[k] = elev_list[i - k][j + k]
233
234             is_stream = transect(transect_list)
235             transect_called += 1
236
237             if is_stream:
238                 for k in range(int(stream_width)):
239                     if (j + k) < len(elev_list[j]) and i >= 0:
240                         stream_list[i - k][j + k] = 1
241
242     percent = int(float((len(elev_list[0]) - i + 1) / float(len(elev_list[0]) - 1)) * 100)
243     if percent > percent_done:
244         gp.AddMessage(' - Analyzing diagonal (SW-NE) transects:  ' + str(percent) + '% completed')
245         percent_done = percent
246
247 #Write the output stream file
248 stream_file = open(stream_file_name, 'w')
```

```
249 for i in range(len(header_list)):
250     stream_file.write(header_list[i])
251
252 percent_done = 0
253 for i in range(len(elev_list)):
254     for j in range(len(elev_list[i])):
255         stream_file.write('%s ' % (stream_list[i][j]))
256     stream_file.write(ls)
257 stream_file.close()
258
259 gp.AddMessage(' - Converting stream output to raster.')
260 gp.ASCIIToRaster_conversion(stream_file_name, stream_output_raster, "INTEGER")
261
262 os.remove(stream_file_name)
263 os.remove(elev_ascii_path)
264
265 print "%i transects were analyzed " % (transect_called)
266
267 seconds = time.clock() - start_time
268
269 h,m,s = sec_to_h_min(seconds)
270 print 'Took ' + str(h) + ' hours ' + str(m) + ' minutes ' + str(s) + ' seconds'
```

APPENDIX B – Python Code

carve.py

```
1  # ---------------------------------------------------------------------------
2  # carve.py
3  # Created September 2009
4  # Author:  Glenn O'Neil
5  #
6  # Description:  Takes a DEM and binary raster of stream locations, identifies
7  #               sinks in the DEM, and carves paths through artificial barriers
8  #               within the stream locations of the DEM.  Based on work by
9  #               Soille.
10 #
11 # Inputs:       1. elev_raster - full path to the DEM.
12 #               2. stream_raster - binary raster of stream locations, must be
13 #                  the same row and column size as the DEM.
14 #               3. max_carve_length - the maximum distance (in cells)
15 #                  that a carving can cover.
16 #               4. iterations - the number of times the script should iterate
17 #                  through the DEM to carve through barriers.  Each loop may
18 #                  yield new sinks that need to be carved.
19 #               5. delete_intermediate - boolean value indicating whether to
20 #                  to delete the temporary folders for each iteration.
21 #               6. output_raster - the final carved DEM.
22 #               7. workspace - folder where intermediate data is stored.
23 #
24 # Outputs:      1. output_raster - the final carved DEM.
25 #
26 # ---------------------------------------------------------------------------
27
28
29 from decimal import *
30 from math import floor
31 import os, sys, copy, arcgisscripting, shutil
32 ls = os.linesep
33 import raster_analysis as raster
34 gp = arcgisscripting.create(9.3)
35
36
37 # Load required toolboxes...
38 arcgis_home = os.environ.get("ARCGISHOME")
39 gp.AddToolbox(arcgis_home + "ArcToolbox\\Toolboxes\\Data Management Tools.tbx")
```

```
40  gp.AddToolbox(arcgis_home + "ArcToolbox\\Toolboxes\\Conversion Tools.tbx")
41
42  gp.CheckOutExtension("Spatial")
43
44
45  #get the necessary files from the user
46  elev_raster = sys.argv[1]
47  stream_raster = sys.argv[2]
48  max_carve_length = sys.argv[3]
49  max_carve_length = int(max_carve_length)
50  #number of times to iterate
51  iterations = sys.argv[4]
52  iterations = int(iterations)
53  delete_intermediate = sys.argv[5]
54  output_raster = sys.argv[6]
55  workspace = sys.argv[7] + "\\"
56
57
58  #Move through elev_list and look for cells that are sinks AND stream cells.
59  #If that condition is met, begin a radial search looking for an elevation lower
60  #than the selected cell.  If/once that lower cell is found, carve a path to the
61  #cell by manually lowering elevation values along the path incrementally,
62  #yielding a slope through the potential barrier that created the sink.
63
64
65
66  def move_list(list_length, min_move, max_move, quotient):
67      #This function constructs the move lists from one cell to another.
68
69      the_list = [min_move] * int(list_length)
70      max_index_list = []
71      move_difference = quotient - min_move #gives us the percentage of the time that max_move
    will be used.
72      max_index_places = abs(list_length * move_difference)
73      max_index_step = list_length / max_index_places
74
75      if int(abs(move_difference) * 10) >= 5: # then the max move will dominate the list or split it
    evenly, and should be first
76          max_index_holder = 0
77      else: #it should start later in the list
78          max_index_holder = int(round(max_index_step)) - 1 #minus 1 to put it in proper zero-
    based index mode
79
80      max_index_step_counter = 0
```

```
81      while max_index_holder < len(the_list):
82          max_index_list.append(max_index_holder)
83          max_index_step_counter += max_index_step
84          max_index_holder = int(round(max_index_step_counter))
85
86      for i in range(len(the_list)):
87          if i in max_index_list:
88              the_list[i] = max_move
89
90      return the_list
91
92
93  def carve(row_index, col_index):
94      global elev_list, elev_out_list, max_carve_length, rbound_index, bbound_index,
    lbound_index, ubound_index
95
96      sink_elev = elev_list[row_index][col_index]
97      carve_to_elev = sink_elev
98
99      #initiate the initial neighborhood size to search.  You don't start with 1
100     #because the adjacent neighbors were searched in the sink identification.
101     nhood_size = 2
102     while not carve_to_elev < sink_elev and nhood_size <= max_carve_length:
103
104         #get the cell's neighbors
105         nhood_list = raster.nhood(elev_list, row_index, col_index, nhood_size)
106
107         #get the minimum elevation value from the returned nhood_list
108         #only check the outer edges since the others had already been checked
109         row_length = len(nhood_list)
110         col_length = len(nhood_list[0])
111         #check the first row
112         min_elev_top = min(nhood_list[0])
113         min_elev_top_index = nhood_list[0].index(min_elev_top)
114
115         #check the bottom row
116         min_elev_bottom = min(nhood_list[row_length - 1])
117         min_elev_bottom_index = nhood_list[row_length - 1].index(min_elev_bottom)
118
119         #check the left column
120         min_elev_left = nhood_list[0][0]
121         min_elev_left_index = 0
122         for i in range(row_length):
```

```
123             if nhood_list[i][0] < min_elev_left:
124                 min_elev_left = nhood_list[i][0]
125                 min_elev_left_index = i
126
127         #check the right column
128         min_elev_right = nhood_list[0][col_length - 1]
129         min_elev_right_index = 0
130         for i in range(row_length):
131             if nhood_list[i][col_length - 1] < min_elev_right:
132                 min_elev_right = nhood_list[i][col_length - 1]
133                 min_elev_right_index = i
134
135         #determine which elevation and index has the lowest elevation
136         min_elev = min(min_elev_top, min_elev_bottom, min_elev_left, min_elev_right)
137
138         #check to see if we've found an elevation less than the sink's
139         if min_elev < sink_elev:
140             #note the indexes of the lower sink
141             if min_elev == min_elev_top:
142                 lower_sink_row = row_index - (int(row_length / 2)) #easy since its the top row
143                 lower_sink_col = (col_index - (int(col_length / 2))) + min_elev_top_index
144
145             elif min_elev == min_elev_bottom:
146                 lower_sink_row = row_index + (int(row_length / 2)) #easy since its the bottom row
147                 lower_sink_col = (col_index - (int(col_length / 2))) + min_elev_bottom_index
148
149             elif min_elev == min_elev_left:
150                 lower_sink_row = (row_index - (int(row_length / 2))) + min_elev_left_index
151                 lower_sink_col = col_index - (int(row_length / 2)) #easy since its the left column
152
153             elif min_elev == min_elev_right:
154                 lower_sink_row = (row_index - (int(row_length / 2))) + min_elev_right_index
155                 lower_sink_col = col_index + (int(row_length / 2)) #easy since its the left column
156
157         nhood_size += 1
158         carve_to_elev = min_elev
159
160     #check to see if the search was successful
161     if nhood_size > max_carve_length:
162         #couldn't find a lower elevation within the max neighborhood size
163         no_lower_sink_found.append('row: ' + str(row_index) + '  col: ' + str(col_index))
164         return
165
```

```
166     #carve from row_index and col_index to lower_sink_row and lower_sink_col
167     row_dist = row_index - lower_sink_row
168     col_dist = col_index - lower_sink_col
169
170     abs_row_dist = float(abs(row_dist))
171     abs_col_dist = float(abs(col_dist))
172
173     #Initiate lists that will store the h_move and v_move lists.
174     #The first list will contain the h_move and v_move values for the path
175     #from the row_index and col_index to the lower_sink_row and lower_sink_col.
176     #The list sizes will be the same as the minimum value between abs_row_dist
177     #and abs_col_dist.
178     list_size = min(abs_row_dist, abs_col_dist)
179     if list_size == 0: list_size = 1
180
181     move_quotient = 0
182     #Determine the h_move and v_move
183     if abs_row_dist < abs_col_dist and abs_row_dist != 0: #we're stepping horizontally
184       primary_move = "horizontal"
185       #recod the maximum H distance for a V distance of 1
186       if col_dist > 0: #we're stepping left
187         if abs_row_dist > 0:
188           move_quotient = (abs_col_dist/abs_row_dist) * -1
189           min_h_move = int(floor(move_quotient)) + 1
190           if abs(abs_col_dist % abs_row_dist) > 0:
191             max_h_move = min_h_move - 1
192           else:
193             max_h_move = min_h_move
194         else: #it's a straight line down to the destination index
195           min_h_move = col_dist
196           max_h_move = min_h_move
197       elif col_dist < 0: #we're stepping right
198         if abs_row_dist > 0:
199           move_quotient = abs_col_dist/abs_row_dist
200           min_h_move = int(floor(move_quotient))
201           if abs(abs_col_dist % abs_row_dist) > 0:
202             max_h_move = min_h_move + 1
203           else:
204             max_h_move = min_h_move
205         else: #it's a straight line down to the destination index
206           min_h_move = col_dist * -1
207           max_h_move = min_h_move
208
```

```
209         if row_dist < 0: v_move = 1
210         elif row_dist > 0: v_move = -1
211         else: v_move = 0
212
213         if min_h_move == max_h_move:
214           h_move_list = [min_h_move] * int(list_size)
215         else:
216           h_move_list = move_list(list_size, min_h_move, max_h_move, move_quotient)
217
218         v_move_list = [v_move] * int(list_size)
219
220
221       elif abs_col_dist < abs_row_dist and abs_col_dist != 0: #we're stepping vertically
222         primary_move = "vertical"
223         #recod the maximum V distance for an H distance of 1
224         if row_dist > 0:  #we're stepping up (visually, not in terms of row index numbers)
225           if abs_col_dist > 0:
226             move_quotient = (abs_row_dist/abs_col_dist) * -1
227             min_v_move = int(floor(move_quotient)) + 1
228             if abs(abs_row_dist % abs_col_dist) > 0:
229               max_v_move = min_v_move - 1
230             else:
231               max_v_move = min_v_move
232           else:  #it's a straight line down to the destination index
233             min_v_move = row_dist
234             max_v_move = min_v_move
235         elif row_dist < 0:  #we're stepping down (visually, not in terms of row index numbers)
236           if abs_col_dist > 0:
237             move_quotient = abs_row_dist/abs_col_dist
238             min_v_move = int(floor(move_quotient))
239             if abs(abs_row_dist % abs_col_dist) > 0:
240               max_v_move = min_v_move + 1
241             else:
242               max_v_move = min_v_move
243           else:
244             min_v_move = row_dist * -1
245             max_v_move = min_v_move
246
247         if col_dist < 0: h_move = 1
248         elif col_dist > 0: h_move = -1
249         else: h_move = 0
250
251         if min_v_move == max_v_move:
```

```
252             v_move_list = [min_v_move] * int(list_size)
253         else:
254             v_move_list = move_list(list_size, min_v_move, max_v_move, move_quotient)
255
256         h_move_list = [h_move] * int(list_size)
257
258     else: #the're equal, it's a square
259         primary_move = "either"
260         if row_dist < 0: v_move = 1
261         elif row_dist > 0: v_move = -1
262         else: v_move = 0
263         if col_dist < 0: h_move = 1
264         elif col_dist > 0: h_move = -1
265         else: h_move = 0
266
267
268         v_move_list = [v_move] * int(list_size)
269         h_move_list = [h_move] * int(list_size)
270
271         #check to see if it's a perfect square
272         if v_move == 0:
273             h_move_list = [h_move] * int(abs_col_dist)
274             v_move_list = [0] * int(abs_col_dist)
275         if h_move == 0:
276             h_move_list = [0] * int(abs_row_dist)
277             v_move_list = [v_move] * int(abs_row_dist)
278
279
280     #we now need to determine the length of the path, and how much the incremental elevation
    decline will be
281
282     if primary_move == 'either':
283         path_length = len(h_move_list)
284     else:
285         path_length = abs(sum(h_move_list)) + abs(sum(v_move_list))
286     elev_diff = sink_elev - min_elev
287     elev_decline_increment = elev_diff / float(path_length)
288
289     current_elev = sink_elev
290
291     #variables to keep track of the indexes the path moves through
292     current_row = row_index
293     current_col = col_index
```

```
294
295     if primary_move == 'horizontal':
296         #iterate through the h_move_list
297         for m in range(len(h_move_list)):
298             h_move = h_move_list[m]
299             for n in range(abs(h_move)):
300                 if h_move_list[m] > 0:  current_col += 1
301                 elif h_move_list[m] < 0:  current_col -= 1
302
303                 current_elev -= elev_decline_increment
304                 elev_out_list[current_row][current_col] = current_elev
305
306             if v_move_list[m] > 0:  current_row += 1
307             elif v_move_list[m] < 0:  current_row -= 1
308
309             #watch for the dataset boundaries
310             if current_row < 0 or current_row > bbound_index or current_col > rbound_index or
    current_col < lbound_index:
311                 carve_to_elev = sink_elev
312                 break
313
314
315             current_elev -= elev_decline_increment
316             elev_out_list[current_row][current_col] = current_elev
317
318
319     elif primary_move == 'vertical':
320         #iterate through the v_move_list
321         for m in range(len(v_move_list)):
322             v_move = v_move_list[m]
323             for n in range(abs(v_move)):
324                 if v_move_list[m] > 0:  current_row += 1
325                 elif v_move_list[m] < 0:  current_row -= 1
326
327                 current_elev -= elev_decline_increment
328                 elev_out_list[current_row][current_col] = current_elev
329
330             if h_move_list[m] > 0:  current_col += 1
331             elif h_move_list[m] < 0:  current_col -= 1
332
333             #watch for the dataset boundaries
334             if current_row < 0 or current_row > bbound_index or current_col > rbound_index or
    current_col < lbound_index:
335                 carve_to_elev = sink_elev
```

```
336             break
337
338         current_elev -= elev_decline_increment
339         elev_out_list[current_row][current_col] = current_elev
340
341
342     else:  #its a straight diagonal path
343       for p in range(len(v_move_list)):
344         if h_move_list[0] > 0 : current_col += 1
345         elif h_move_list[0] < 0: current_col -= 1
346
347         if v_move_list[0] > 0 : current_row += 1
348         elif v_move_list[0] < 0: current_row -= 1
349
350         #watch for the dataset boundaries
351         if current_row < 0 or current_row > bbound_index or current_col > rbound_index or
    current_col < lbound_index:
352             carve_to_elev = sink_elev
353             break
354
355         current_elev -= elev_decline_increment
356         elev_out_list[current_row][current_col] = current_elev
357
358
359
360 for h in range(iterations):
361     gp.AddMessage("Beginning Iteration " + str(h + 1) + ":")
362     #create a workspace for the current iteration
363     sub_workspace = workspace + 'iteration' + str(h + 1)
364     os.mkdir(sub_workspace)
365
366     #Calculating flow direction
367     gp.AddMessage("   - Calculating flow direction of elevation raster")
368     flowdir_raster = sub_workspace + "\\fd_temp"
369     if h > 0:
370       #get the previous iteration's output
371       elev_raster = workspace + 'iteration' + str(h) + "\\elev_out" + str(h)
372     gp.FlowDirection_sa(elev_raster, flowdir_raster)
373
374     #Calculate sinks
375     gp.AddMessage("   - Calculating sinks")
376     sinks_raster = sub_workspace + "\\sinks" + str(h + 1)
377     gp.Sink_sa(flowdir_raster, sinks_raster)
```

```
378
379     #convert the rasters to text files
380     if h == 0:
381         gp.AddMessage("   - Converting elevation raster to a text file")
382         elev_ascii_path = sub_workspace + "\\elev_temp.txt"
383         gp.RasterToASCII_conversion(elev_raster, elev_ascii_path)
384     else:  #get the previous iteration's elevation text file
385         elev_ascii_path = workspace + 'iteration' + str(h) + '\\elev_out' + str(h) + '.txt'
386
387     stream_ascii_path = workspace + "stream_temp.txt"
388     if h == 0:
389         gp.AddMessage("   - Converting stream raster to a text file")
390         gp.RasterToASCII_conversion(stream_raster, stream_ascii_path)
391
392     gp.AddMessage("   - Converting sinks raster to a text file")
393     sink_ascii_path = sub_workspace + "\\sinks" + str(h + 1) + ".txt"
394     gp.RasterToASCII_conversion(sinks_raster, sink_ascii_path)
395
396     #convert the raster text files to arrays
397     gp.AddMessage("   - Reading elevation text file into a list")
398     elev_list = raster.rastertext2list(elev_ascii_path, 'float')
399     if h == 0:
400         gp.AddMessage("   - Reading stream text file into an array")
401         stream_ary = raster.raster2array(stream_raster)
402     gp.AddMessage("   - Reading sinks text file into an array")
403     sink_ary = raster.raster2array(sinks_raster)
404
405     #record the header information for writing in the output stream file
406     raster_header = raster.extract_header(elev_ascii_path)
407
408     #note the indexes of the dataset boundaries
409     rbound_index = len(elev_list[0]) - 1
410     lbound_index = 0
411     bbound_index = len(elev_list) - 1
412     ubound_index = 0
413
414     #intiate a list to represent the resulting elevation ascii, set initial values to elev_list's.
415     elev_out_list = copy.deepcopy(elev_list)
416
417     #initiate a list to keep track of carving attempts that could not find a lower cell (i.e., the
    maximum
418     #carve length was exceeded before a lower elevation was found
419     no_lower_sink_found = []
```

```
420
421     #counters for output
422     sinks_carved = 0
423     percent_done = 0
424     stream_ary_length = len(stream_ary)
425
426     gp.AddMessage('   - Analyzing')
427     for i in range(len(elev_list)):
428        for j in range(len(elev_list[i])):
429           if stream_ary[i,j] == 1 and sink_ary[i,j] > 0: #we may have a barrier to carve through
430              carve(i, j)
431              sinks_carved += 1
432        percent = int((float(i) / (stream_ary_length - 1) * 100))
433        if percent > percent_done:
434           gp.AddMessage('   - Analyzing:  ' + str(percent) + '% completed')
435           percent_done = percent
436
437     del elev_list, sink_ary
438
439     #write the output stream file
440     elev_out_file_name = sub_workspace + "\\elev_out" + str(h + 1) + ".txt"
441     elev_out_file = open(elev_out_file_name, 'w')
442     for i in range(len(raster_header)):
443        elev_out_file.write(raster_header[i])
444
445     percent_done = 0
446     gp.AddMessage('   - Writing outputs')
447     for i in range(len(elev_out_list)):
448        for j in range(len(elev_out_list[i])):
449           elev_out_file.write('%s ' % (elev_out_list[i][j]))
450        elev_out_file.write(ls)
451        percent = int((float(i) / (len(elev_out_list) - 1) * 100))
452        if percent > percent_done:
453           gp.AddMessage('   - Writing:  ' + str(percent) + '% completed')
454           percent_done = percent
455     elev_out_file.close()
456     del elev_out_file
457
458
459     gp.AddMessage('   - There were ' + str(sinks_carved) + ' sinks carved.')
460     gp.AddMessage('   - There were ' + str(len(no_lower_sink_found)) + ' sinks left unresolved')
461     del no_lower_sink_found
462
```

```
463     #convert the elev_out_file to a raster
464     gp.AddMessage('   - Converting the output elevation file to a raster')
465     elev_out_raster = sub_workspace + "\\elev_out" + str(h + 1)
466     gp.ASCIIToRaster_conversion(elev_out_file_name, elev_out_raster, "FLOAT")
467
468
469
470     #clean up
471     del elev_out_list
472     gp.AddMessage('   - Cleaning up')
473     gp.Delete_management(flowdir_raster)
474     os.remove(sink_ascii_path)
475
476     if h == 0:  os.remove(elev_ascii_path)
477
478
479 if delete_intermediate == 'True':
480     #clean up some more
481     gp.AddMessage('Deleting intermediate data')
482
483     for i in range(iterations):
484         sub_workspace = workspace + 'iteration' + str(i + 1)
485         if i < (iterations - 1):
486             shutil.rmtree(sub_workspace)
487         else:
488             gp.CopyRaster_management(sub_workspace + "\\elev_out" + str(i + 1), output_raster)
489             shutil.rmtree(sub_workspace)
490             os.remove(stream_ascii_path)
491             #del stream_list
492             del stream_ary
493 else:
494     os.remove(stream_ascii_path)
```

APPENDIX B – Python Code

flow_error.py

```
1  # ---------------------------------------------------------------------------
2  # flow_error.py
3  # Created September 2009
4  # Author:  Glenn O'Neil
5  #
6  # Description:  Uses the 10-meter flow-direction raster (ned_fd) and the LiDAR
7  #                 tile flow-direction rasters (lidar_fd) in the lidar directory
8  #                 to calculate flow-direction error for each 10-meter cell.
9  #
10 # Inputs:      1. quad_id - quadrangle ID (e.g. CW210).
11 #              2. workspace - directory for quadrangle containing all of the
12 #                datasets (ned_fd and lidar_fd).
13 #
14 # Outputs:     1. flow_error.tif - TIFF raster of flow-direction raster at the
15 #              extent and cell-size of the ned_fd (10-meter) with a value
16 #                of 0-4 calculated for each cell.
17 # ---------------------------------------------------------------------------
18 from osgeo import gdal, gdalconst
19 import time, os, sys, numpy, copy
20 import raster_analysis as raster
21 ls = os.linesep
22
23 #get the quad name
24 quad_id = sys.argv[1]
25 workspace = sys.argv[2]
26
27 #function to convert seconds to hours and minutes (borrowed from:
28 #http://mail.python.org/pipermail/python-list/2003-January.181366.html
29 def sec_to_h_min(s):
30     temp = float()
31     temp = float(s) / (60*60*24)
32     d = int(temp)
33     temp = (temp - d) * 24
34     h = int(temp)
35     temp = (temp - h) * 60
36     m = int(temp)
37     temp = (temp - m) * 60
38     sec = int(temp)
39     return h,m,sec
```

```
40
41
42  #start the timer
43  script_start_time = time.clock()
44
45  #create the log textfile
46  flow_results_file_path = workspace + "\\log.txt"
47  flow_results_file = open(flow_results_file_path, 'w')
48  flow_results_file.write("QUAD_ID;" + quad_id + ls)
49  flow_results_file.close()
50  del flow_results_file
51
52  #read the lidar_fd into a list, by way of a less efficient array
53  ned_fd = workspace + "\\ned_fd"
54  ned_fd_ary = raster.raster2array(ned_fd)
55  ned_fd_list = ned_fd_ary.tolist()
56  del ned_fd_ary
57
58  #read the ned_fid raster into an array
59  ned_fid = workspace + "\\ned_fid"
60  ned_fid_ary = raster.raster2array(ned_fid)
61  ned_fid_list = ned_fid_ary.tolist()
62  del ned_fid_ary
63
64  #initiate an array to represent the resulting flow direction evaluation, with initial values of 0.
65  #make it the same size as the fd_list
66  fd_eval_ary = numpy.zeros([len(ned_fd_list),len(ned_fd_list[0])])
67
68  #initiate an array to keep track of cells that have been processed (to facilitate assignment of
    NoData values later)
69  cells_processed_ary = numpy.zeros([len(ned_fd_list),len(ned_fd_list[0])])
70
71  #iterate through the folders of the workspace, and conduct the flow analysis
72  lidar_wspace_contents = os.listdir(workspace + "\\lidar")
73  raster_list = []
74  for i in lidar_wspace_contents:
75    if os.path.isdir(workspace + "\\lidar\\" + i) and i != 'info':
76      raster_list.append(i)
77
78
79  raster_list_length = len(raster_list)
80  for z in range(len(raster_list)):
81    raster_folder_start_time = time.clock()
```

```
82    check_timer_start = time.clock()
83
84    raster_folder = workspace + "\\lidar\\" + raster_list[z]
85    print ' - Processing raster ' + raster_folder + '(' + str(z + 1) + ' of ' + str(raster_list_length) + ')'
86    #get the LiDAR dataset
87    lidar_fd = raster_folder + "\\lidar_fd"
88
89
90    #get the cell size parameters from the flow_analysis_parameters.txt file and store them in a
    dictionary
91    parameters_list = []
92    parameters_dict = {}
93    parameters_file_path = raster_folder + "\\flow_analysis_parameters.txt"
94    parameters_file = open(parameters_file_path, 'r')
95    for eachLine in parameters_file:
96      parameters_list.append(eachLine.split(";"))
97    parameters_file.close()
98
99    #get rid of the newline character at the end of each row, and add it to the dictionary
100   for i in range(len(parameters_list)):
101     parameters_list[i][1] = parameters_list[i][1][:len(parameters_list[i][1]) - 2]
102     parameters_dict[parameters_list[i][0]] = parameters_list[i][1]
103
104   #indices for performing neighborhood analyses
105   buffer_size = int(parameters_dict['buffer_size'])
106   start_center_row = buffer_size
107   start_center_col = buffer_size
108   end_center_row = 0
109   end_center_col = 0
110
111   #read the lidar_fd into a list by way of an array
112   lidar_fd_ary = raster.raster2array(lidar_fd)
113   lidar_fd_list = lidar_fd_ary.tolist()
114   del lidar_fd_ary
115
116   #read the LiDAR dataset with the NED FID values into a list by way of an array
117   lidar_ned_fid = raster_folder + "\\lidar_ned_fid"
118   lidar_ned_fid_ary = raster.raster2array(lidar_ned_fid)
119   lidar_ned_fid_list = lidar_ned_fid_ary.tolist()
120   del lidar_ned_fid_ary
121
122   #read the binary mask for flat areas (derived from LiDAR-scale dataset 'carved_dem')
123   flat_areas = raster_folder + "\\flat_areas"
```

```
124     flat_areas_ary = raster.raster2array(flat_areas)
125     flat_areas_list = flat_areas_ary.tolist()
126     del flat_areas_ary
127
128     #Index counters for keeping track of lidar_ned_id indices as the code loops through the
    ned_fid
129     #e.g. We will extract the lidar cells that correspond to the selected NED ID. To do that
130     #we must loop through the array to find the appropriate cells, and we don't want to start the
    search
131     #at the beginning each time. Since we're skipping the edge cells of the NED, we'll set the
    starting
132     #values to two less than the number of lidar cells per NED cell.
133     lidar_cells_per_ned_cell = int(float(parameters_dict["lidar_cells_per_ned_cell"]))
134     lidar_ned_fid_row_start = int(lidar_cells_per_ned_cell) - 2
135     lidar_ned_fid_row_end = lidar_ned_fid_row_start + 1
136     lidar_ned_fid_col_start = int(lidar_cells_per_ned_cell) - 2
137     lidar_ned_fid_col_end = lidar_ned_fid_row_end
138
139     script_start_time = time.clock()
140
141     #identify the NED FID of the first and last LiDAR cells
142     lidar_ned_fid_start = lidar_ned_fid_list[0][0]
143     lidar_ned_fid_end = lidar_ned_fid_list[len(lidar_ned_fid_list) - 1][len(lidar_ned_fid_list[0]) -
    1]
144     #In a few tiles, NoData values crept into the dataset at the corners, so we must check for those
145     #and grab the first positive ned_fid value, by moving diagonally from the corners
146     row = 0
147     col = 0
148     while lidar_ned_fid_start < 0:
149         lidar_ned_fid_start = lidar_ned_fid_list[row][col]
150         row += 1
151         col += 1
152
153     row = len(lidar_ned_fid_list) - 1
154     col = len(lidar_ned_fid_list[row]) - 1
155     while lidar_ned_fid_end < 0:
156         lidar_ned_fid_end = lidar_ned_fid_list[row][col]
157         row -= 1
158         col -= 1
159
160     #determine the index in ned_fid_list where the lidar_ned_fid_start and lidar_ned_fid_end
    values are located
161     ned_fid_list_start_row = 0
162     ned_fid_list_start_col = 0
163     ned_fid_list_end_row = 0
```

```
164     ned_fid_list_end_col = 0
165     for i in range(len(ned_fid_list)):
166       try:
167         ned_fid_list_start_col = ned_fid_list[i].index(lidar_ned_fid_start)
168         ned_fid_list_start_row = i
169         break
170       except ValueError:
171         pass
172
173     for i in range(ned_fid_list_start_row, len(ned_fid_list)):
174       try:
175         ned_fid_list_end_col = ned_fid_list[i].index(lidar_ned_fid_end)
176         ned_fid_list_end_row = i
177         break
178       except ValueError:
179         pass
180
181     #move through ned_fid_list performing a neighborhood assessment of flow-direction
182     percent_done = 0
183
184     #variable for determining if a flat_cell was encountered, breaking the loop
185     flat_area_found = False
186
187
188
189     for i in range(ned_fid_list_start_row + 2, ned_fid_list_end_row - 2): #+2 and -2 to avoid the
    top and bottom edges
190
191       #reset the lidar_ned_fid_col_start =
192       lidar_ned_fid_col_start = int(lidar_cells_per_ned_cell) - 2
193
194       for j in range(ned_fid_list_start_col + 2, ned_fid_list_end_col - 2): #+2 and -2 to avoid the
    left and right edges
195
196         ned_fid_value = ned_fid_list[i][j]
197
198         #extract the LiDAR cells of the selected NED cell and the neighborhood around the
    selected NED cell
199         #first, identify where the LiDAR cells are
200         end_search = False
201
202         #skip no data cells, e.g. ned_fid_value < 0 for a 32-bit unsigned integer, and cells that
    have already been processed
203         if ned_fid_value < 0 or cells_processed_ary[i,j] == 1:
204           break
```

```
205
206         for k in range(lidar_ned_fid_row_start, len(lidar_ned_fid_list)):
207             for m in range(lidar_ned_fid_col_start, len(lidar_ned_fid_list[k])):
208                 if lidar_ned_fid_list[k][m] == ned_fid_value:  #we've found the start of the NED
    area within the LiDAR cells
209                     lidar_ned_fid_row_start = k
210                     lidar_ned_fid_col_start = m
211                     #find the col_end
212                     for n in range(lidar_ned_fid_col_start, len(lidar_ned_fid_list[k])):
213                         if lidar_ned_fid_list[k][n] != ned_fid_value:  #we've found the end of the NED
    area (the previous column)
214                             lidar_ned_fid_col_end = n - 1
215                             break
216                     #find the row_end
217                     for n in range(lidar_ned_fid_row_start, len(lidar_ned_fid_list)):
218                         if lidar_ned_fid_list[n][lidar_ned_fid_col_start] != ned_fid_value: #we've
    found the end of the NED area (the previous row)
219                             lidar_ned_fid_row_end = n - 1
220                             end_search = True
221                             break
222                     break
223             if end_search:
224                 break
225
226         #determine the boundaries for extracting the LiDAR flow-direction values for the
    defined area and a buffer-area equal to half a NED cell.
227         lidar_nhood_row_start = lidar_ned_fid_row_start - buffer_size
228         lidar_nhood_row_end = lidar_ned_fid_row_end + buffer_size
229         lidar_nhood_col_start = lidar_ned_fid_col_start - buffer_size
230         lidar_nhood_col_end = lidar_ned_fid_col_end + buffer_size
231
232         #create lists to store the values of neighborhood cells
233         lidar_nhood_fd_list = []
234         flat_areas_nhood_list = []
235         for k in range(lidar_nhood_row_start, lidar_nhood_row_end + 1):

236 lidar_nhood_fd_list.append(lidar_fd_list[k][lidar_nhood_col_start:lidar_nhood_col_end + 1])
237 flat_areas_nhood_list.append(flat_areas_list[k][lidar_nhood_col_start:lidar_nhood_col_end +
    1])
238
239         nhood_rows = len(lidar_nhood_fd_list)
240         nhood_cols = len(lidar_nhood_fd_list[0])
241
242         end_center_row = nhood_rows - buffer_size
243         end_center_col = nhood_cols - buffer_size
244
```

```
245         #create a list that will mark the selected NED area LiDAR cells (center cells) as 1 and the
    buffer cells as 0
246         lidar_nhood_ary = numpy.zeros_like(lidar_nhood_fd_list)
247         lidar_nhood_list = lidar_nhood_ary.tolist()
248         del lidar_nhood_ary
249
250         #lidar_nhood_list[start_center_row:end_center_row][start_center_col:end_center_col] =
    1
251         for k in range(start_center_row, end_center_row + 1):
252           for n in range(start_center_col, end_center_col + 1):
253             lidar_nhood_list[k][n] = 1
254
255         #create a list that will store flow-accumulation of center cells within the neighborhood
256         lidar_nhood_fa_ary = numpy.zeros_like(lidar_nhood_fd_list)
257         lidar_nhood_fa_list = lidar_nhood_fa_ary.tolist()
258         del lidar_nhood_fa_ary
259
260         #Calculate flow_accumulation in the LiDAR neighborhood.
261         #Visit each center cell (i.e. lidar_nhood_list == 1), trace its flow path using
    lidar_nhood_fd_list until it exits the neighborhood
262         counter = 0
263         for k in range(start_center_row, end_center_row):
264           for m in range(start_center_col, end_center_col):
265             current_row = k
266             current_col = m
267             #while current_row < nhood_rows - 1 and current_row >= 0 and current_col <
    nhood_cols - 1 and current_col >= 0:
268             while current_row < nhood_rows and current_row >= 0 and current_col <
    nhood_cols and current_col >= 0:
269
270               #check for flat_area
271               if flat_areas_nhood_list[current_row][current_col] == 1:
272                 flat_area_found = True
273                 break
274
275               #determine what direction flow exits the current cell
276               lidar_fd_value = lidar_nhood_fd_list[current_row][current_col]
277               if lidar_fd_value == 1: #flow east
278                 current_col += 1
279               elif lidar_fd_value == 2: #flow southeast
280                 current_col += 1
281                 current_row += 1
282               elif lidar_fd_value == 4: #flow south
283                 current_row += 1
284               elif lidar_fd_value == 8: #flow southwest
```

```
285                     current_col -= 1
286                     current_row += 1
287                 elif lidar_fd_value == 16: #flow west
288                     current_col -= 1
289                 elif lidar_fd_value == 32: #flow northwest
290                     current_col -= 1
291                     current_row -= 1
292                 elif lidar_fd_value == 64: #flow north
293                     current_row -= 1
294                 elif lidar_fd_value == 128: #flow northeast
295                     current_col += 1
296                     current_row -= 1
297
298                 #update the flowaccumulation list
299                 if current_row < nhood_rows and current_row >= 0 and current_col <
    nhood_cols and current_col >= 0:
300                     lidar_nhood_fa_list[current_row][current_col] += 1
301
302             if flat_area_found:
303                 break
304
305         if flat_area_found:
306             break
307
308     if flat_area_found:
309         fd_eval_ary[i,j] = -9999
310         flat_area_found = False
311
312     else:
313
314         #re-adjust end_center_row and end_center_col to zero-based indexes, they currently
    just
315         #hold the difference in size between the neighborhood size and buffer size
316         end_center_row -= 1
317         end_center_col -= 1
318
319         #create a dictionary that will store the proportion of edge cells in each neighborhood
    quad
320         nhood_proportions_dict = {}
321         nhood_proportions_dict["NW"] = start_center_row + start_center_col
322         nhood_proportions_dict["N"] = end_center_col - start_center_col + 1
323         nhood_proportions_dict["NE"] = nhood_proportions_dict["NW"]
324         nhood_proportions_dict["E"] = nhood_proportions_dict["N"]
325         nhood_proportions_dict["SE"] = nhood_proportions_dict["NW"]
```

```
326             nhood_proportions_dict["S"] = nhood_proportions_dict["N"]
327             nhood_proportions_dict["SW"] = nhood_proportions_dict["NW"]
328             nhood_proportions_dict["W"] = nhood_proportions_dict["N"]
329             num_edge_cells = float(sum(nhood_proportions_dict.values()))
330             for key in nhood_proportions_dict:
331                 nhood_proportions_dict[key] = nhood_proportions_dict[key] / num_edge_cells
332
333
334             #create a dictionary that will store the total exiting flow for each neighborhood quad
335             flow_dict = {}
336             flow_dict["NW"] = 0
337             flow_dict["N"] = 0
338             flow_dict["NE"] = 0
339             flow_dict["E"] = 0
340             flow_dict["SE"] = 0
341             flow_dict["S"] = 0
342             flow_dict["SW"] = 0
343             flow_dict["W"] = 0
344
345
346             #trace the outer edge of the boundary cells, and count the flowaccumulation cells that
    exit the study area
347             #aggregate the results by direction (E,SE,S,SW,W,NW,N,NE)
348
349             #trace the northern edge
350             for k in range(nhood_cols):
351                 if k < start_center_col:  #we're in the northewest quad
352                     if lidar_nhood_fd_list[0][k] in [32,64,128]: #then the flow exits the quad NW, N,
    or NE
353                         flow_dict["NW"] += lidar_nhood_fa_list[0][k]
354                 elif k >= start_center_col and k <= end_center_col: #we're in the northern quad
355                     if lidar_nhood_fd_list[0][k] in [32,64,128]: #then the flow exits the quad NW, N,
    or NE
356                         flow_dict["N"] += lidar_nhood_fa_list[0][k]
357                 else:  #we're in the northeast quad
358                     if lidar_nhood_fd_list[0][k] in [32,64,128]: #then the flow exits the quad NW, N,
    or NE
359                         flow_dict["NE"] += lidar_nhood_fa_list[0][k]
360
361             #trace the southern edge
362             last_row_index = nhood_rows - 1
363             for k in range(len(lidar_nhood_fd_list[last_row_index])):
364                 if k < start_center_col:  #we're in the southwest quad
365                     if lidar_nhood_fd_list[last_row_index][k] in [2,4,8]: #then the flow exits the quad
    SE, S, or SW
```

```
366                     flow_dict["SW"] += lidar_nhood_fa_list[last_row_index][k]
367                 elif k >= start_center_col and k <= end_center_col: #we're in the south quad
368                     if lidar_nhood_fd_list[last_row_index][k] in [2,4,8]: #then the flow exits the quad
    SE, S, or SW
369                         flow_dict["S"] += lidar_nhood_fa_list[last_row_index][k]
370                 else:  #we're in the southeast quad
371                     if lidar_nhood_fd_list[last_row_index][k] in [2,4,8]: #then the flow exits the quad
    SE, S, or SW
372                         flow_dict["SE"] += lidar_nhood_fa_list[last_row_index][k]
373
374             #trace the western edge
375             for k in range(nhood_rows):
376                 if k < start_center_row:  #we're in the northewest quad
377                     if lidar_nhood_fd_list[k][0] in [8,16,32]: #then the flow exits the quad SW, W, or
    NW
378                         #don't double count the northwesternmost cell, it may have been counted in the
    trace of the northern edge
379                         if k == 0 and lidar_nhood_fd_list[0][0] == 32:
380                             pass
381                         else:
382                             flow_dict["NW"] += lidar_nhood_fa_list[k][0]
383                 elif k >= start_center_row and k <= end_center_row: #we're in the west quad
384                     if lidar_nhood_fd_list[k][0] in [8,16,32]: #then the flow exits the quad SW, W, or
    NW
385                         flow_dict["W"] += lidar_nhood_fa_list[k][0]
386                 else:  #we're in the southwest quad
387                     if lidar_nhood_fd_list[k][0] in [8,16,32]: #then the flow exits the quad SW, W, or
    NW
388                         flow_dict["SW"] += lidar_nhood_fa_list[k][0]
389
390             #trace the eastern edge
391             last_col_index = nhood_cols - 1
392             for k in range(nhood_rows):
393                 if k < start_center_row:  #we're in the northeast quad
394                     if lidar_nhood_fd_list[k][last_col_index] in [128,1,2]: #then the flow exits the
    quad NE, E, or SE
395                         #don't double count the northeasternmost cell, it may have been counted in the
    trace of the northern edge
396                         if k == 0 and lidar_nhood_fd_list[0][last_col_index] == 128:
397                             pass
398                         else:
399                             flow_dict["NE"] += lidar_nhood_fa_list[k][last_col_index]
400                 elif k >= start_center_row and k <= end_center_row: #we're in the east quad
401                     if lidar_nhood_fd_list[k][last_col_index] in [128,1,2]: #then the flow exits the
    quad NE, E, or SE
402                         flow_dict["E"] += lidar_nhood_fa_list[k][last_col_index]
403                 else:  #we're in the southeast quad
```

```
404                 if lidar_nhood_fd_list[k][last_col_index] in [128,1,2]: #then the flow exits the
    quad NE, E, or SE
405                     #don't double count the southeasternmost cell, it may have been counted in the
    trace of the southern edge
406                     if k == last_row_index and
    lidar_nhood_fd_list[last_row_index][last_col_index] == 2:
407                         pass
408                     else:
409                         flow_dict["SE"] += lidar_nhood_fa_list[k][last_col_index]
410
411
412
413             #create a dictionary that will store the percentage of total exiting flow for each
    neighborhood quad
414             sum_flow_exit_cells = float(sum(flow_dict.values()))
415             if sum_flow_exit_cells == 0:
416                 print '0 value at ' + str(i) + ' , ' + str(j) + '; ned_id = ' + str(ned_fid_value)
417             flow_pct_dict = {}
418             flow_pct_dict["NW"] = flow_dict["NW"] / sum_flow_exit_cells
419             flow_pct_dict["N"] = flow_dict["N"] / sum_flow_exit_cells
420             flow_pct_dict["NE"] = flow_dict["NE"] / sum_flow_exit_cells
421             flow_pct_dict["E"] = flow_dict["E"] / sum_flow_exit_cells
422             flow_pct_dict["SE"] = flow_dict["SE"] / sum_flow_exit_cells
423             flow_pct_dict["S"] = flow_dict["S"] / sum_flow_exit_cells
424             flow_pct_dict["SW"] = flow_dict["SW"] / sum_flow_exit_cells
425             flow_pct_dict["W"] = flow_dict["W"] / sum_flow_exit_cells
426
427
428             #evaluate the difference in flow-direction from the center cells to the NED flow-
    direction
429             ned_fd_value = ned_fd_list[i][j]
430             flow_error_dict = {}
431             if ned_fd_value == 1: #east
432                 flow_error_dict["NW"] = 3
433                 flow_error_dict["N"] = 2
434                 flow_error_dict["NE"] = 1
435                 flow_error_dict["E"] = 0
436                 flow_error_dict["SE"] = 1
437                 flow_error_dict["S"] = 2
438                 flow_error_dict["SW"] = 3
439                 flow_error_dict["W"] = 4
440             elif ned_fd_value == 2: #southeast
441                 flow_error_dict["NW"] = 4
442                 flow_error_dict["N"] = 3
443                 flow_error_dict["NE"] = 2
```

```
444             flow_error_dict["E"] = 1
445             flow_error_dict["SE"] = 0
446             flow_error_dict["S"] = 1
447             flow_error_dict["SW"] = 2
448             flow_error_dict["W"] = 3
449         elif ned_fd_value == 4: #south
450             flow_error_dict["NW"] = 3
451             flow_error_dict["N"] = 4
452             flow_error_dict["NE"] = 3
453             flow_error_dict["E"] = 2
454             flow_error_dict["SE"] = 1
455             flow_error_dict["S"] = 0
456             flow_error_dict["SW"] = 1
457             flow_error_dict["W"] = 2
458         elif ned_fd_value == 8: #southwest
459             flow_error_dict["NW"] = 2
460             flow_error_dict["N"] = 3
461             flow_error_dict["NE"] = 4
462             flow_error_dict["E"] = 3
463             flow_error_dict["SE"] = 2
464             flow_error_dict["S"] = 1
465             flow_error_dict["SW"] = 0
466             flow_error_dict["W"] = 1
467         elif ned_fd_value == 16: #west
468             flow_error_dict["NW"] = 1
469             flow_error_dict["N"] = 2
470             flow_error_dict["NE"] = 3
471             flow_error_dict["E"] = 4
472             flow_error_dict["SE"] = 3
473             flow_error_dict["S"] = 2
474             flow_error_dict["SW"] = 1
475             flow_error_dict["W"] = 0
476         elif ned_fd_value == 32: #northwest
477             flow_error_dict["NW"] = 0
478             flow_error_dict["N"] = 1
479             flow_error_dict["NE"] = 2
480             flow_error_dict["E"] = 3
481             flow_error_dict["SE"] = 4
482             flow_error_dict["S"] = 3
483             flow_error_dict["SW"] = 2
484             flow_error_dict["W"] = 1
485         elif ned_fd_value == 64: #north
486             flow_error_dict["NW"] = 1
```

```
487                 flow_error_dict["N"] = 0
488                 flow_error_dict["NE"] = 1
489                 flow_error_dict["E"] = 2
490                 flow_error_dict["SE"] = 3
491                 flow_error_dict["S"] = 4
492                 flow_error_dict["SW"] = 3
493                 flow_error_dict["W"] = 2
494               elif ned_fd_value == 128: #northeast
495                 flow_error_dict["NW"] = 2
496                 flow_error_dict["N"] = 1
497                 flow_error_dict["NE"] = 0
498                 flow_error_dict["E"] = 1
499                 flow_error_dict["SE"] = 2
500                 flow_error_dict["S"] = 3
501                 flow_error_dict["SW"] = 4
502                 flow_error_dict["W"] = 3
503
504               weighted_flow_error = 0
505               for direction in flow_pct_dict:
506                 weighted_flow_error += flow_pct_dict[direction] * flow_error_dict[direction]
507
508               #record the weighted flow error in the NED sized raster list
509               fd_eval_ary[i,j] = weighted_flow_error
510
511
512               del nhood_proportions_dict, flow_dict, flow_pct_dict, flow_error_dict
513
514
515             cells_processed_ary[i,j] = 1
516
517             #delete used lists and dictionaries
518             del lidar_nhood_list, lidar_nhood_fd_list, flat_areas_nhood_list, lidar_nhood_fa_list
519
520
521         percent = int(float(i - ned_fid_list_start_row) / (ned_fid_list_end_row -
    ned_fid_list_start_row) * 100)
522         if percent > percent_done:
523             print '      - ' + str(percent) + "% done (ned_fid : " + str(ned_fid_value) + ")"
524             percent_done = percent
525
526       del lidar_fd_list, lidar_ned_fid_list
527
528
```

```
529     raster_folder_seconds = time.clock() - raster_folder_start_time
530     h,m,s = sec_to_h_min(raster_folder_seconds)
531     print '    - raster ' + raster_folder + ' took ' + str(h) + ' hours ' + str(m) + ' minutes ' + str(s) + '
    seconds'
532
533
534 #loop through the cells_processed_ary and set any cells with a value of 0 to NoData (-9999)
535 for i in range(len(cells_processed_ary)):
536     for j in range(len(cells_processed_ary[i])):
537         if cells_processed_ary[i,j] == 0:
538             fd_eval_ary[i,j] = -9999
539
540
541 #write the cumulative error array to a raster
542 #use the TIFF file as a template, since GDAL cannot write ArcINFO binary rasters
543 template_raster = workspace + "\\template_raster.tif"
544 template_raster_ds = gdal.Open(template_raster, gdalconst.GA_Update)
545 template_raster_band = template_raster_ds.GetRasterBand(1)
546 template_raster_band.WriteArray(fd_eval_ary)
547 template_raster_band.FlushCache()
548 template_raster_ds.FlushCache()
549
550 del template_raster_ds, template_raster_band, fd_eval_ary, ned_fd_list, ned_fid_list,
    lidar_wspace_contents, raster_list, cells_processed_ary
551 os.rename(template_raster, template_raster.replace('template_raster', 'flow_error'))
552
553 seconds = time.clock() - script_start_time
554 h,m,s = sec_to_h_min(seconds)
555 print 'Whole thing Took ' + str(h) + ' hours ' + str(m) + ' minutes ' + str(s) + ' seconds'
```

APPENDIX B – Python Code

contour_dlg_conversion.py

```
1  # ---------------------------------------------------------------------------
2  # contour_dlg_conversion.py
3  # Created September 2009
4  # Author:  Glenn O'Neil
5  #
6  # Description:  Takes a directory of DLG hypsography files and converts each
7  #               files relevant features to ESRI shapefiles.  Crashes due to an
8  #               ESRI bug if it tries to process more than 70 files.
9  #
10 # Inputs:       1. dlg_wspace - directory containing DLG hypsography files.
11 #               2. scratch_wspace - directory for temporary files.
12 #               3. out_wspace - directory where the output shapefiles are written.
13 #
14 # Outputs:      1. <quad_name>_ohp.shp - shapefile of quadrangle contours.
15 # ---------------------------------------------------------------------------
16
17 import os, arcgisscripting, sys
18 gp = arcgisscripting.create(9.3)
19
20 dlg_wspace = sys.argv[1]
21 scratch_wspace = sys.argv[2] + "\\"
22 out_wspace = sys.argv[3] + "\\"
23
24 #get the dlg files
25 dlg_wspace_list = os.listdir(dlg_wspace)
26 dlg_list = []
27 #filter the list so it only contains .dlg files
28 for i in range(len(dlg_wspace_list)):
29     if '.dlg' in (dlg_wspace_list[i]):
30         dlg_list.append(dlg_wspace_list[i])
31 del dlg_wspace_list
32
33 dlg_list_length = len(dlg_list)
34 for i in range(dlg_list_length):
35     gp.AddMessage("Processing " + str(i + 1) + " of " + str(dlg_list_length) + ":")
36
37     #convert the dlg to a coverage
38     gp.AddMessage("    - converting DLG to coverage")
39     dlg_cov = scratch_wspace + "dlg_cov"
```

```
40      gp.dlgarc(dlg_wspace + "\\" + dlg_list[i], dlg_cov)
41      #convert the coverage to a shapefile
42      gp.AddMessage("    - converting coverage to shapefile")
43      dlg_shp = dlg_cov + "_arc.shp"
44      gp.FeatureClassToShapefile(dlg_cov + "\\arc", scratch_wspace)
45      #join the .acode file to the shapefile table on the ID field
46      gp.AddMessage("    - joining acode table to shapefile")
47      acode = dlg_cov + ".acode"
48      gp.joinfield (dlg_shp, "ID", acode, "DLG_COV-ID")
49      #extract the appropriate stream features
50      gp.AddMessage("    - extracting features")
51      #get the appropriate fields from dlg_shp
52      code_fields = gp.listfields(dlg_shp, 'MINOR*')
53      select_query = '"MAJOR1" = 20 AND ('
54      for j in range(len(code_fields)):
55          if j != (len(code_fields) - 1):
56              select_query += '"' + code_fields[j].Name + '" NOT IN
    (202,205,206,207,208,209,210,299) OR '
57          else:  #treat the last record differently
              select_query += '"' + code_fields[j].Name + '" NOT IN
58  (202,205,206,207,208,209,210,299))'
59
60      dlg_out = out_wspace + dlg_list[i][:8] + '.shp'
61      gp.select_analysis(dlg_shp, dlg_out, select_query)
62
63      gp.AddMessage("    - deleting intermediate data")
64      gp.delete_management(dlg_cov)
65      gp.delete_management(dlg_shp)
66
67  del dlg_list, select_query
```

APPENDIX B – Python Code

contour_toppology_analysis.py

```
1  # ---------------------------------------------------------------------------
2  # contour_topology_analysis.py
3  # Created September 2009
4  # Author:  Glenn O'Neil
5  #
6  # Description:  This script identifies topological error in contour datasets.
7  #               It analyzes node ID information to determine how many contour
8  #               intersections exists in a dataset.  It uses quad boundary and
9  #               featre vertices to determine how many contour dangles exist.
10 #
11 # Inputs:       1. quad_ds - an ESRI shapefile of 7.5 Minute USGS Quadrangles
12 #               2. quad_id_field - unique ID field of the 'quads' shapefile
13 #               3. contour_wspace - workspace where the contour datasets are stored
14 #               4. workspace - a folder where temporary datasets will be written
15 #
16 # Outputs:      1. Inter - a field in quad_ds that represents the number of
17 #               intersections in that quad.
18 #               2. Dangle - a field in quad_ds that represents contour dangle
19 #               3. Contours - a field in quad_ds that represents the number of contours
20 #               4. Int_p_cnt - a field in quad_ds that represents the number of
21 #               intersections per contour.
22 #               5. Dan_p_cnt - a field in quad_ds that represents the number of
23 #               dangles per contour.
24 # ---------------------------------------------------------------------------
25
26 #import the necessary modules
27 import sys, os, time, arcgisscripting
28 gp = arcgisscripting.create(9.3)
29
30 #inputs
31 quad_ds = sys.argv[1]
32 quad_id_field = sys.argv[2]
33 contour_wspace = sys.argv[3]
34 workspace = sys.argv[4]
35
36 gp.OverWriteOutput = 1
37
38 #insert a field into the quad dataset that will store the number of intersections
39 quad_ds_fields = gp.listfields(quad_ds)
```

```
40  field_present = False
41  for i in range(len(quad_ds_fields)):
42      if quad_ds_fields[i].name == "Inter":
43          field_present = True
44  if not field_present:
45      gp.addfield(quad_ds, "Inter", "short", "8")
46      gp.addfield(quad_ds, "Dangle", "short", "8")
47      gp.addfield(quad_ds, "Contours", "short", "8")
48      gp.addfield(quad_ds, "Int_p_cnt", "FLOAT")
49      gp.addfield(quad_ds, "Dan_p_cnt", "FLOAT")
50
51
52  #determine the number of quads to process
53  rows = gp.searchcursor(quad_ds)
54  quad_count = 0
55  row = rows.next()
56  while row:
57      quad_count += 1
58      row = rows.next()
59  del row, rows
60
61
62  #loop through the quad_ids
63  quad_rows = gp.updatecursor(quad_ds)
64  quad_row = quad_rows.next()
65  counter = 0
66  while quad_row:
67
68      quad_id = quad_row.getvalue(quad_id_field)
69      quad_contour_ds = contour_wspace + "\\" + quad_id + "ohp.shp"
70
71      gp.AddMessage("- Processing quad " + quad_id + "(" + str(counter + 1) + " of " +
    str(quad_count) + ")")
72      counter += 1
73
74      if not gp.exists(quad_contour_ds):
75          gp.AddMessage("   - could not find contours for quad " + quad_id)
76          quad_row = quad_rows.next()
77
78      else:
79          #check to see if intersections have already been identified
80          if quad_row.getvalue("Inter") > 0: #the quad has PROBABLY already been processed,
    possible that there were 0 intersections anyway
81              quad_row = quad_rows.next()
```

```
82            continue
83
84        #create a table of unique node IDs in the contours
85        #read the fnode field and store unique results in an array, repeat for tnode fields, then create a
    table
86        cursor = gp.searchcursor(quad_contour_ds)
87        node_ids = []
88        row = cursor.next()
89        while row:
90            fnode_id = row.getvalue("FROMNODE")
91            tnode_id = row.getvalue("TONODE")
92            if not fnode_id in node_ids:
93                node_ids.append(fnode_id)
94            if not tnode_id in node_ids:
95                node_ids.append(tnode_id)
96
97            row = cursor.next()
98
99        node_ids.sort()
100
101       node_id_table = workspace + "\\node_id.dbf"
102       gp.CreateTable(workspace, "node_id.dbf")
103       gp.AddField(node_id_table, "ID", "SHORT")
104       rows = gp.InsertCursor(node_id_table)
105       for i in range(len(node_ids)):
106           row = rows.NewRow()
107           row.ID = node_ids[i]
108           rows.InsertRow(row)
109
110       del row, rows
111
112       #select only non-border contours (i.e. MAJOR1 > 0)
113       #make a feature layer first
114       quad_contour_fl = quad_id + "_contour_ds_fl"
115       gp.MakeFeatureLayer(quad_contour_ds, quad_contour_fl)
116       gp.SelectLayerByAttribute(quad_contour_fl, "NEW_SELECTION", "\"MAJOR1\" > 0")
117
118       #Calculate FNode and TNOde Frequency Tables
119       fnode_freq = workspace + "\\" + quad_id + "_fnode_freq.dbf"
120       tnode_freq = workspace + "\\" + quad_id + "_tnode_freq.dbf"
121       gp.Frequency(quad_contour_fl, fnode_freq, "FROMNODE")
122       gp.Frequency(quad_contour_fl, tnode_freq, "TONODE")
123
```

```
124     #Convert FNode and TNode Freq. to Table Views
125     fnode_freq_tv = quad_id + "_fnode_freq_tv"
126     tnode_freq_tv = quad_id + "_tnode_freq_tv"
127     gp.maketableview(fnode_freq, fnode_freq_tv)
128     gp.maketableview(tnode_freq, tnode_freq_tv)
129
130     #select intersections in the frequency tables
131     gp.SelectLayerByAttribute(fnode_freq_tv, "NEW_SELECTION", "\"FREQUENCY\" > 1")
132     gp.SelectLayerByAttribute(tnode_freq_tv, "NEW_SELECTION", "\"FREQUENCY\" > 1")
133
134     #join the selected frequency tables
135     gp.AddJoin_management(quad_contour_fl, "FROMNODE", fnode_freq_tv, "FROMNODE",
    "KEEP_ALL")
136     gp.AddJoin_management(quad_contour_fl, quad_id + "ohp.TONODE", tnode_freq_tv,
    "TONODE", "KEEP_ALL")
137
138     #select the contours that have intersections (fromnode and tnode frequencies > 1)
139     gp.SelectLayerByAttribute(quad_contour_fl, "NEW_SELECTION", "\"" + quad_id +
    "_fnode_freq.FREQUENCY\" > 1 OR \"" + quad_id + "_tnode_freq.FREQUENCY\" > 1")
140     #gp.SelectLayerByAttribute("quad_contour_ds_fl", "NEW_SELECTION",
    '"fnode_freq_tv:FREQUENCY" > 1')
141
142     #Subratct mid-contour nodes
143     gp.SelectLayerByAttribute(quad_contour_fl, "REMOVE_FROM_SELECTION", "(\"" +
    quad_id + "_fnode_freq.FREQUENCY\" = 2 AND \"" + quad_id + "_tnode_freq.FREQUENCY\"
    IS NULL) OR (\"" + quad_id + "_tnode_freq.FREQUENCY\" = 2 AND \"" + quad_id +
    "_fnode_freq.FREQUENCY\" IS NULL)")
144
145     intersection_count = gp.GetCount_management(quad_contour_fl)
146
147     quad_row.Inter = intersection_count.getOutput(0)
148
149
150     #the dangle process:
151     #use feature vertices to points with the "BOTH_ENDS"
152     contour_vertices = workspace + "\\" + quad_id + "_vertices.shp"
153     gp.FeatureVerticesToPoints(quad_contour_ds, contour_vertices, "BOTH_ENDS")
154
155     #use addXY on the new points layer
156     gp.AddXY_management(contour_vertices)
157
158     #create a new string field 30 characters long for X Y concatenation
159     gp.AddField(contour_vertices, "XY", "TEXT")
160
161     #concatenate the X field and the Y field with a ";"
162     gp.CalculateField(contour_vertices, "XY", "[POINT_X] & \";\" & [POINT_Y]")
```

```
163
164         #calculate a frequency table on the concatenated XY values
165         contour_vertices_fl = quad_id + "_contour_vertices_fl"
166         gp.MakeFeatureLayer(contour_vertices, contour_vertices_fl)
167         xy_freq = workspace + "\\" + quad_id + "_xy_freq.dbf"
168         gp.Frequency(contour_vertices_fl, xy_freq, "XY")
169         xy_freq_tv = quad_id + "_xy_freq_tv"
170         gp.maketableview(xy_freq, xy_freq_tv)
171
172         #join the frequency table to the contour_vertices
173         gp.AddJoin_management(contour_vertices_fl, "XY", xy_freq_tv, "XY", "KEEP_ALL")
174
175         #select the selected quad from the quad shapefile and create a 500 foot inside buffer
176         #This allows us to avoid selecting the dangling nodes on the quad boundary
177         quad_select = workspace + "\\" + quad_id + "_sel.shp"
178         gp.select_analysis(quad_ds, quad_select, "\"" + quad_id_field + "\" = '" + quad_id + "'")
179         quad_inside_buff = workspace + "\\" + quad_id + "_inside_buff.shp"
180         gp.buffer_analysis(quad_select, quad_inside_buff, "-200") #probably in meters, depends on
    quad_select units
181         quad_inside_buff_fl = quad_id + "_inside_buff_fl"
182         gp.MakeFeatureLayer(quad_inside_buff, quad_inside_buff_fl)
183
184         #select the vertices that intersect the new inside buffer
185         gp.SelectLayerByLocation(contour_vertices_fl, "intersect", quad_inside_buff)
186
187         #create a sub-selection the vertices that have a frequency = 1
188         gp.SelectLayerByAttribute(contour_vertices_fl, "SUBSET_SELECTION", "\"" + quad_id +
    "_xy_freq.FREQUENCY\" = 1")
189
190         #get the recourd count on the vertices
191         dangle_count = gp.GetCount_management(contour_vertices_fl)
192
193         quad_row.Dangle = dangle_count.getOutput(0)
194
195
196         #calculate the number of contours, to determine intersection and dangle ratios
197
198         #Calculate a contour frequency table
199         gp.RemoveJoin(quad_contour_fl, quad_id + "_fnode_freq", quad_id + "_tnode_freq")
200         gp.SelectLayerByAttribute(quad_contour_fl) #clears its current selection
201         contour_freq = workspace + "\\" + quad_id + "_contour_freq.dbf"
202         gp.Frequency(quad_contour_fl, contour_freq, "ID")
203         contour_count = gp.GetCount_management(quad_contour_fl)
204         quad_row.Contours = contour_count.getOutput(0)
```

```
205
206         intersections_per_contour = float(intersection_count.getOutput(0)) /
    float(contour_count.getOutput(0))
207         dangles_per_contour = float(dangle_count.getOutput(0)) / float(contour_count.getOutput(0))
208
209         quad_row.Int_p_cnt = str(intersections_per_contour)
210         quad_row.Dan_p_cnt = str(dangles_per_contour)
211
212         quad_rows.updaterow(quad_row)
213
214         gp.delete_management(quad_contour_fl)
215         gp.delete_management(fnode_freq)
216         gp.delete_management(tnode_freq)
217         gp.delete_management(node_id_table)
218         gp.delete_management(contour_vertices)
219         gp.delete_management(contour_vertices_fl)
220         gp.delete_management(xy_freq)
221         gp.delete_management(quad_select)
222         gp.delete_management(quad_inside_buff)
223         gp.delete_management(quad_inside_buff_fl)
224         gp.delete_management(contour_freq)
225
226         quad_row = quad_rows.Next()
227
228 del quad_row, quad_rows
```

APPENDIX B – Python Code

analysis_results.py

```
1  # ---------------------------------------------------------------------------
2  # analysis_results.py
3  # Created September 2009
4  # Author:  Glenn O'Neil
5  #
6  # Description:  Takes a directory of flow error rasters, reads it into a NumPy
7  #               Array through GDAL and records each raster's mean, standard
8  #               deviation and median flow error value into a text file.
9  #
10 # Inputs:       1. workspace - directory
11 #               2. output_text_file - text file containing the mean, median,
12 #               and standard deviation flow error values for each raster.
13 #
14 # Outputs:      1. output text file -
15 #
16 # ---------------------------------------------------------------------------
17
18
19 import sys, os, raster_analysis, numpy
20 from osgeo import gdal, gdalconst
21
22
23 #get the inputs
24 workspace = sys.argv[1]
25 output_text_file_path = sys.argv[2]
26 ls = os.linesep
27
28
29 output_text_file = open(output_text_file_path, 'w')
30
31 #get the folders of the workspace
32 quads_list = []
33 quads_wspace_contents = os.listdir(workspace)
34 for i in quads_wspace_contents:
35     if os.path.isdir(workspace + "\\" + i):
36         quads_list.append(i)
37 del quads_wspace_contents
38
39 quads_list.sort()
```

```
40
41  quads_list_length = len(quads_list)
42  for i in range(quads_list_length):
43  #for i in range(1):
44
45      #get the flow_error raster
46      flow_error_raster = workspace + "\\" + quads_list[i] + "\\" + quads_list[i] +
    "_flow_error.tif"
47
48      if os.path.exists(flow_error_raster):
49
50          #read the raster into an array
51          flow_error_ary = raster_analysis.raster2array(flow_error_raster)
52
53          #append non NoData (i.e. -9999) results to a list
54          error_values_list = []
55          for j in range(len(flow_error_ary)):
56              for k in range(len(flow_error_ary[j])):
57                  if flow_error_ary[j,k] != -9999.0:
58                      error_values_list.append(flow_error_ary[j,k])
59
60          del flow_error_ary
61
62          #convert the list back to an array for stat analysis
63          error_values_ary = numpy.array(error_values_list)
64          error_values_list.sort()
65          median = error_values_list[int(numpy.round(len(error_values_list) / 2)) - 1]
66
67          #capture the mean and standard deviation of the ary
68          mean = error_values_ary.mean()
69          stdev = error_values_ary.std()
70          del error_values_ary, error_values_list
71
72          #record the stats to the text file
73          output_text_file.write(quads_list[i] + "|" + str(mean) + "|" + str(stdev) + "|" +
    str(median) + ls)
74
75          print 'Processed ' + quads_list[i] + "(" + str(i + 1) + " of " + str(quads_list_length) + ")"
76
77
78  output_text_file.close()
79  del output_text_file, quads_list
```

REFERENCES

REFERENCES

Aziz, S. A.; Steward, Brian L. (2007). Development of Agricultural Field DEM Using Repeated GPS Measurements from Field Operations: Effects of Sampling Intensity and Pattern. ASABE paper No. 071089. St. Joseph, Mich.: ASABE.

Barber, C. P.; Shortridge, A. (2005). Lidar Elevation Data for Surface Hydrologic Modeling: Resolution and Representation Issues. *Cartography and Geographic Information Science 32.4*, pp. 401-410.

Burrough, P.; vanGans, P.F.M; MacMillan, R.A. (2000). High resolution landform classification using fuzzy k-means. *Journal of Fuzzy Sets and Systems 113*, pp. 37-52.

Carrara, A.; Bitelli, G.; Carla, R. (1997). Comparison of techniques for generating digital terrain models from contour lines. *International Journal of Geographic Information Science 11*, pp. 451-473.

Carson, W.; Reutebuch, S. (1997). A Rigorous Test of the Accuracy of USGS Digital Elevation Models in Forested Areas of Oregon and Washington. *ACSM/ASPRS Annual Convention & Exposition Technical Papers.* April 7-10, 1997.

Carter, J. R. (1988). Digital representations of topographic surfaces. *Photogrammetric Engineering and Remote* Sensing *54*, pp. 1577-1580.

Carter, J. R. (1992). The effect of data precision on the calculation of slope and aspect using gridded DEMs. *Cartographica 29.1*, pp. 22-34.

Chang, K.; Tsai, B. (1991). The Effect of DEM Resolution on Slope and Aspect Mapping. *Cartography and Geographic Information Systems 18.1*, pp. 69-77.

Clarke, K.C., Lee, S.J. (2007). Spatial Resolution and Algorithm Choice as Modifiers of Downslope Flow Computed from Digital Elevation Models. *Cartography and Geographic Information Science 34.3*, pp. 215-230.

Daratech Inc. (2009, August 20). Press Release – 'GIS/Geospatial Industry Worldwide Growth Slows to 1% in 2009." *Directions Magazine.* Retrieved February 16, 2010, from Directions Magazine: http://www.directionsmag.com/press.releases/?duty=Show&id=36318.

Deng, Y.; Wilson, J.; Bauer, B. (2007). DEM resolution dependencies of terrain attributes across a landscape. *International Journal of Geographical Information Science 21.2*, pp. 187-213.

Desmet, P. (1997). Effects of Interpolation Errors on the Analysis of DEMs. *Earth Surface Processes and Landforms 22*, pp. 563-580.

Digital Cartographic Data Standards Task Force. (1998). A proposed standard for digital cartographic data. *The American Cartographer 15.1*, pp. 129 – 136.

Endreny, T.A.; Wood, E.F. (2001). Representing elevation uncertainty in runoff modeling and flowpath mapping. *Hydrological Processes 15*, pp. 2223-2236.

Erskine, R.H.; Green, T.R.; Ramiriez, J.A.; MacDonald, L.H. (2007). Digital Elevation Accuracy and Grid Cell Size: Effects on Estimated Terrain Attributes. *Soil Science Society of America Journal 71.4*, pp. 1371-1380.

Faintich, M. (1996). Digital Elevation Models. *Cellular Business September*, pp. 46-58.

Fisher, P. (1993). Algorithm and implementation uncertainty in viewshed analysis. *International Journal of Geographical Information Science 7.4*, pp. 331-347.

Fisher, P. (1998). Improved Modeling of Elevation Error with Geostatistics. *GeoInformatica 2.3*, pp. 215-233.

Fisher, P.; Tate, N. (2006). Causes and consequences of error in digital elevation models. *Progress in Physical Geography 30.4*, pp. 467-489.

Florinsky, I. (1998). Accuracy of local topographic variables derived from digital elevation models. *International Journal of Geographical Information Science 12.1*, pp. 47-61.

Freeman, T. (1991). Calculating Catchment Area With Divergent Flow Based on a Regular Grid. *Computers & Geosciences 17.3*, pp. 413-422.

Garbrecht, J., Starks, P. (1995). Note on the Use of USGS Level 1 7.5-Minute DEM Coverages for Landscape Drainage Analyses. *Photogrammetric Engineering & Remote Sensing 61.5*, pp. 519-522.

Goodchild, M.; Guoqing, S.; Shiren, Y. (1992). Development and test of an error model for categorical data. *International Journal of Geographic Information Science 6.2*, pp. 87-104.

Greenlee, D. (1987). Raster and Vector Processing for Scanned Linework. *Photogrammetric Engineering and Remote Sensing 53.10*, pp. 1383-1387.

Heuvelink, G. (1998). *Error Propagation in Environmental Modelling*. Bristol, PA: Taylor and Francis, Inc.

Holmes, K.W.; Chadwick, O.A.; Kyriakidis, P.C. (2000). Error in a USGS 30-meter digital elevation model and its impact on terrain modeling. *Journal of Hydrology 233*, pp. 154-173.

Homer, C.; Dewitz, J.; Fry, J.; Coan, M.; Hossain, N.; Larson, C.; Herold, N.; McKerrow, A.; Van Driel, J.; Wickham, J. (2007). Completion of the 2001 National Land Cover Database for the Coterminous United States. *Photogrammetric Engineering and Remote Sensing April*, pp. 829-840.

Hutchinson, M. (1989). A New Procedure for Gridding Elevation and Stream Line Data with Automatic Removal of Spurious Pits. *Journal of Hydrology 106*, pp. 211-232.

James, L.; Watson, D.; Hansen, W. (2007). Using LiDAR data to map gullies and headwater streams under forest canopy. *CATENA 71.1*, pp. 132-144.

Jenson, S.; Domingue, J. (1988). Extracting Topographic Structure from Digital Elevation Data for Geographic Information System Analysis. *Photogrammetric Engineering and Remote Sensing 54.11*, pp. 1593-1600.

Jones, K.H. (1998). A Comparison of Algorithms used to Compute Hill Slope as a Property of the DEM. *Computers & Geosciences 24.4*, pp. 315-323.

Kienzle, S. (2004). The Effect of DEM Raster Resolution on First Order, Second Order and Compound Terrain Derivatives. *Transactions in GIS 8.1*, pp. 83-111.

Kyriakidis, P.C.; Shortridge, A.M.; Goodchild, M.F. (1999). Geostatistics for conflation and accuracy assessment of digital elevation models. *International Journal of Geographical Information Science 13.7*, pp. 677-707.

Leckie, D.; Cloney, E.; Jay, C. (2005). Automated mapping of stream features with high-resolution multispectral imagery: An example of the capabilities. *Photogrammetric Engineering and Remote Sensing 71.2*, pp. 145-155.

MacMillan, R.; Martin, T.; Earle, T.; McNabb, D. (2003). Automated analysis and classification of landforms using high-resolution digital elevation data: applications and issues. *Canadian Journal of Remote Sensing 29.5*, pp. 592-606.

Martinoni, D., Bernhard, L. (1998). A conceptual framework for reliable digital terrain modelling. In: *Proceedings of the Eighth Symposium on Spatial Data Handling, Vancouver, Canada*, pp. 737-750.

Mason, D.; Scott, T.; Wang, H. (2006). Extraction of tidal channel networks from airborne scanning laser altimetry. *ISPRS Journal of Photogrammetry and Remote Sensing 61.2*, pp. 67-83.

Meyer, T.H. (2004). The Discontinuous Nature of Kriging Interpolation for Digital Terrain Modeling. *Cartography and Geographic Information Science 31.4*, pp. 209-216.

Mitasova, H.; Hofierka, J.; Zlocha, M.; Iverson, L. (1995). Modeling Topographic Potential for Erosion and Deposition Using GIS. *International Journal of GIS January*, pp. 1-19.

Mouton, A. (2005). *Generating Stream Maps Using LiDAR Derived Digital Elevation Models and 10-m USGS DEM.* Thesis. Washington State University.

O'Callaghan, J.F., Mark, D.M. (1984). The Extraction of Drainage Networks From Digital Elevation Data. *Computer Vision, Graphics and Image Processing 28*, pp. 328-344.

Ohio Office of Information Technology, GIS Support Center. (2005). *Ohio 10-meter Digital Elevation Model (FGDC) / oh_dem10b (ISO).* Retrieved July 17, 2008, from http://metadataexplorer.gis.state.oh.us/metadataexplorer/full_metadata.jsp?docId={D0820264-6E96-4E1A-860F-F89F42C39F64}&loggedIn=false.

Oksanen, J.; Tapani, S. (2005). Error propagation of DEM-based surface derivatives. *Computers and Geosciences 31*, pp. 1015-1027.

Ouyang, D.; Bartholic, J.; Selegean, J. (2005). Assessing Sediment Loading from Agricultural Croplands in the Great Lakes Basin. *Journal of American Science 1.2*, pp. 14-21.

Quinn, P.; Beven, K.; Chevallier, P.; Planchon, O. (1991). The Prediction of Hillslope Flowpaths for Distributed Hydrological Modelling Using Digital Terrain Models. *Hydrological Processes 5*, pp. 59-80.

Raaflaub, L.D.; Collins, M.J. (2006). The effect of error in gridded digital elevation models on the estimation of topographic parameters. *Environmental Modelling and Software 21*, pp. 710-732.

Riggs, P.D.; Dean, D.J. (2007). An Investigation into the Causes of Errors and Inconsistencies in Predicted Viewsheds. *Transactions in GIS 11.2*, pp. 175-196.

Saunders, W. (1999). Preparations of DEMS for use in Environmental Modeling Analysis. *1999 ESRI User Conference. San Diego, CA, July 24-30, 1999.* http://proceedings.esri.com/library/userconf/proc99/proceed/papers/pap802/p802.Htm.

Schmidt, F.; Persson, A. (2003). Comparison of DEM Data Capture and Topographic Wetness Indices. *Precision Agriculture 4*, p. 179-192.

Shortridge, A. (2006). Shuttle Radar Topography Mission Elevation Data Error and Its Relationship to Land Cover. *Cartography and Geographic Information Science 33.1*, pp. 65-75.

Skidmore, A.K. (1989). A comparison of techniques for calculating gradient and aspect from a gridded digital elevation model. *International Journal of Geographical Information Systems 2.4*, pp. 323-334.

Soille, P.; Vogt, J.; Colombo, R. (2003). Carving and adaptive drainage enforcement of grid digital elevation models." *Water Resources Research 39.12*, pp. 10-1 to 10-3.

Soille, P. (2004a). Morphological carving. *Pattern Recognition Letters 25,* pp. 543-550.

Soille, P. (2004b). Optimal removal of spurious pits in grid digital elevation models. *Water Resources Research 40.12.*

Tarboton, D.G. (1991). On the Extraction of Channel Networks from Digital Elevation Data. *Hydrological Processes 5*, pp. 81-100.

Tarboton, D.G. (1997). A New Method for the Determination of Flow Directions and Upslope Areas in Grid Digital Elevation Models. *Water Resources Research 33.2*, pp. 309-319.

Thompson, J.A.; Bell, J.C.; Butler, C.A. (2001). Digital elevation model resolution: effects on terrain attribute calculation and quantitative soil-landscape modeling. *Geoderma 1*, pp. 67-89.

US EPA. (2007, July 17). Analytical Tools Interface for Landscape Assessments (ATtILA). *Environmental Protection Agency Website.* Retrieved August 2009, from http://www.epa.gov/esd/land-sci/attila/index.htm.

USGS. (2006, August). National Elevation Dataset. *United States Geological Survey.* Retrieved March 2009, from http://ned.usgs.gov.

USGS. (1998, January). Standards for Digital Elevation Models. *Rocky Mountain Mapping Center.* Retrieved March, 2009, from http://rockyweb.cr.usgs.gov/nmpstds/demstds.html.

USGS. (1999, September). Standards for Digital Line Graphs. *Rocky Mountain Mapping Center*. Retrieved March 2009, from http://rmmcweb.cr.usgs.gov/nmpstds/acrodocs/dlg-3/2dlg0999.pdf.

Venteris, E.R.; Slater, B.K. (2005). A Comparison Between Contour Elevation Data Sources for DEM Creation and Soil Carbon Prediction, Coshocton, Ohio. *Transactions in GIS 9.2*, pp. 179-198.

Vogt, J.; Colombo, R.; Bertolo, F. (2003). Deriving drainage networks and catchment boundaries: a new methodology combining digital elevation data and environmental characteristics. *Geomorphology 53*, pp. 281-298.

Walker, J.P., Willgoose, G.R. (1999). On the effect of digital elevation model accuracy on hydrology and geomorphology. *Water Resources Research 53.7*, pp. 2259-2268.

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

Wilson, J.P.; Lam, C.S.; Deng, Y. (2007) Comparison of the performance of flow-routing algorithms used in GIS-based hydrologic analysis. *Hydrological Processes February*, pp. 1026-1044.

Wise, S. (1998). The Effect of GIS Interpolation Errors on the Use of Digital Elevation Models in Geomorphology, In: Landform Monitoring, Modelling and Analysis, Edited by S. N. Lane, K. S. Richards and J. H. Chandler, John Wiley and Sons, 300 pp.

Wise, S. (2000). Assessing the quality for hydrological applications of digital elevation models derived from contours. *Hydrological Processes 14*, pp. 1909-1929.

Wise, S. (2007). Effect of differing DEM creation methods on the results from a hydrological model. *Computers & Geosciences 33*, pp. 1351-1365.

Wolock, D.M.; McCabe, G.J. Differences in topographic characteristics computed from 100- and 1000-m resolution digital elevation model data. *Hydrological Processes 14,* pp. 987-1002.

Woolpert Inc. (2007, October 11). OSIP DEM Tiles by County – GRID Format (FGDC) / N1440375 (ISO). *OGRIP Website*. Retrieved July 17, 2008, from http://metadataexplorer.gis.state.oh.us/metadataexplorer/full_metadata.jsp?docId={6FAB0C57-1A57-4142-BCF7-A38F4D11FCA3}&loggedIn=false.

Wu, S.; Li, J.; Huang, G. (2005). An evaluation of grid size uncertainty in empirical soil loss modeling with digital elevation models. *Environmental Modelling and Assessment 10*, pp. 33-42.

Wu, S.; Li, J., Huang, G. (2007). Modeling the effects of elevation data resolution on the performance of topography-based watershed runoff simulation. *Environmental Modelling & Software 22*, pp. 1250-1260.

Ziadat, F.M. (2007). Effect of Contour Intervals and Grid Cell Size on the Accuracy of DEMs and Slope Derivatives. *Transactions in GIS. 11.1*, pp. 67-81.